Doctoral Thesis

Formal Foundations for Access and Inference Control in Databases

Author(s):
Guarnieri, Marco

Publication Date:
2017

Permalink:
https://doi.org/10.3929/ethz-b-000238984

Rights / License:
In Copyright - Non-Commercial Use Permitted
Formal Foundations for Access and Inference Control in Databases

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

MARCO GUARNIERI
Laurea Magistrale in Ingegneria Informatica, Università degli Studi di Bergamo
born on 06.03.1988
citizen of Italy

accepted on the recommendation of

Prof. Dr. David Basin, examiner
Prof. Dr. Stephen Chong, co-examiner
Prof. Dr. Heiko Mantel, co-examiner

2017
Abstract

Databases often store and manage sensitive data. Regulating the access to databases is, therefore, essential. To this end, database security researchers have developed both access control and inference control mechanisms. The former limit direct access to sensitive data, whereas the latter prevent leaks caused by combining query results with external information such as prior knowledge or data dependencies. Ideally, all these mechanisms should come with security proofs clearly stating what attacks they are designed to thwart, as with security mechanisms in other domains. Unfortunately, this is far from reality. Existing protection mechanisms are implemented in an ad hoc fashion, with neither precise security guarantees nor the means to verify them. This has immediate consequences as existing mechanisms are inadequate to secure modern databases and are susceptible to attacks.

In this thesis, we develop theoretical foundations for access and inference control in databases. We leverage these foundations to design provably secure and practical protection mechanisms for modern database systems. In more detail, our thesis makes the following contributions.

First, we formalize Security-Aware Query Processing, the task of computing answers to SELECT queries in the presence of access control policies, and we study the existence of optimal Security-Aware Query Processing algorithms, i.e., those algorithms that provide both confidentiality and maximal data availability. We prove that there are no optimal algorithms for the relational calculus, whereas optimal algorithms exist for some of its fragments. We also investigate the connections between two different semantics for database access control that have been previously presented in the literature. In this respect, we show that, for optimal algorithms, these semantics are distinct for non-boolean SELECT queries, but they coincide for boolean queries.

Second, we develop foundations for database inference control in the presence of probabilistic data dependencies. Our foundations are based on ProbLog, a state-of-the-art probabilistic logic programming language. We leverage these foundations to develop Angerona, a database inference control mechanism that secures databases against attackers exploiting probabilistic dependencies. Angerona provides precise security, completeness, and tractability guarantees by exploiting a tractable inference algorithm for a practically relevant fragment of ProbLog. We empirically evaluate Angerona’s performance and show that it scales to relevant security-critical problems.

Third, we develop a formal framework for reasoning about the security of database access control mechanisms. Our framework consists of (1) a formal operational semantics of databases supporting advanced features, such as triggers and views, (2) an attacker model formalizing how attackers infer information from the database system’s behavior, and (3) precise security conditions accounting for dynamic security policies. Our attacker model and security conditions subsume the standard, limited attacker models and conditions considered in previous works, where both the database state and the policy are fixed and attackers are restricted to SELECT queries. Guided by our operational semantics and attacker model, we develop a provably secure enforcement mechanism that thwart attacks confidentiality and integrity attacks that existing mechanisms fail to prevent.

Finally, we reconcile database access control and information-flow control, two seemingly disparate research areas that share the same goal: protecting the confidentiality and integrity of sensitive information. We develop WhileSql, an imperative language with querying capabilities that relies on our database operational semantics, and we provide a framework for reasoning about the security of WhileSql programs. We reduce database access control to determining whether WhileSql programs leak information, thereby providing a way of applying existing information-flow techniques to database access control. We also develop a provably secure enforcement mechanism for WhileSql programs, which secures programs that existing mechanisms fail to secure. Our mechanism combines dynamic information-flow tracking with concepts from database theory, such as disclosure lattices and query determinacy, to provide end-to-end security guarantees for programs interacting with modern databases.
Riassunto

I moderni sistemi informativi spesso processano informazioni sensibili, che vengono memorizzate all’interno di basi di dati (database). Prevenire possibili attacchi aventi come obiettivo tali informazioni richiede quindi di controllare e limitare l’accesso alle basi di dati. Per questo motivo, sono stati proposti e implementati vari meccanismi che permettono il controllo degli accessi (access control) e delle inferenze (inference control). I primi limitano l’accesso diretto alle informazioni sensibili. I secondi, invece, regolano l’accesso indiretto a tali informazioni, essi cioè limitano le informazioni che possono essere ottenute combinando i risultati estratti dalle basi di dati tramite interrogazioni con informazioni aggiuntive come, per esempio, informazioni pregresse o dipendenze tra i dati. Idealmente, tutti questi meccanismi dovrebbero essere accompagnati da dimostrazioni formali che specificano e certificano quali attacchi vengono bloccati. Sfortunatamente, la realtà è diversa. I meccanismi di sicurezza esistenti sono stati sviluppati senza precise garanzie di sicurezza. Questo ha conseguenze immediate: i meccanismi esistenti non sono adeguati per proteggere le moderne basi di dati e sono vulnerabili a vari attacchi.

In questa tesi presentiamo i fondamenti teorici per il controllo degli accessi e delle inferenze. Basandoci su questi fondamenti teorici, abbiamo progettato e implementato nuovi meccanismi per controllare gli accessi e le inferenze all’interno delle moderne basi di dati. Questi meccanismi forniscono precise garanzie di sicurezza (certificate da dimostrazioni formali) e sono al contempo utilizzabili in pratica.

Il nostro primo contributo studia come eseguire interrogazioni di tipo SELECT in presenza di politiche di controllo degli accessi. Tale problema è anche chiamato Security-Aware Query Processing. In particolare, studiamo l’esistenza di algoritmi ottimali per effettuare Security-Aware Query Processing, cioè quelli che garantiscono allo stesso tempo confidenzialità e disponibilità dei dati. In questa tesi, dimostriamo che non ci sono algoritmi ottimali in caso si utilizzi il calcolo relazionale come linguaggio di interrogazione, mentre tali algoritmi esistono per vari frammenti del calcolo relazionale. In aggiunta, investighiamo anche le relazioni tra due differenti semantiche per il controllo degli accessi nelle basi di dati. A questo proposito, dimostriamo che, per gli algoritmi ottimali, queste due semantiche sono distinte per le interrogazioni non booleane, mentre coincidono per le interrogazioni booleane.

Il nostro secondo contributo, invece, analizza il controllo delle inferenze in presenza di dipendenze probabilistiche tra i dati memorizzati nella base di dati. A questo proposito, sviluppiamo i fondamenti teorici per il controllo delle inferenze utilizzando ProbLog, un linguaggio di programmazione logica con aspetti probabilistici. Basandoci su tali fondamenti, sviluppiamo ANGERONA, un meccanismo per il controllo delle inferenze che protegge le basi di dati da attaccanti aventi accesso alle dipendenze probabilistiche tra i dati stessi. ANGERONA fornisce precise garanzie di sicurezza, completezza, e trattabilità utilizzando un nuovo algoritmo per effettuare inferenze probabilistiche per un sottoinsieme praticamente rilevante dei programmi esprimibili in ProbLog. Infine, valutiamo empiricamente le prestazioni di ANGERONA e mostriamo che scalano a vari problemi d’interesse.

Il nostro terzo contributo consiste nello sviluppare i fondamenti teorici per il controllo degli accessi nelle basi di dati. A questo proposito, presentiamo (1) una semantica operazionale delle basi di dati che supporta funzionalità avanzate come viste e trigger, (2) un modello dell’attaccante che formalizza come un attaccante estrae informazioni sul contenuto della base di dati osservando il comportamento del sistema, e (3) precise definizioni di sicurezza che tengono conto di politiche del controllo degli accessi dinamiche. Sia il nostro modello dell’attaccante che le nostre definizioni di sicurezza sono più generali di quelle considerate in lavori precedenti, dove sia il contenuto della base di dati che la politica di controllo degli accessi sono fissi e gli attaccanti possono utilizzare solo interrogazioni di tipo SELECT. Utilizzando questi fondamenti teorici, infine, sviluppiamo un nuovo meccanismo per il controllo degli accessi che previene una serie di attacchi all’integrità e confidenzialità dei dati che non sono bloccati dai meccanismi esistenti.

L’ultimo nostro contributo consiste nel riconcilire il controllo degli accessi all’interno delle basi di dati con il controllo del flusso delle informazioni (information-flow control) all’interno dei programmi. Queste due aree di ricerca sembrano distinte, ma hanno lo stesso obiettivo: proteggere la confidenzialità e l’integrità di informazioni sensibili. In questa tesi, sviluppiamo WhileSQL, un linguaggio di programmazione imperativo che permette anche di interrogare delle basi di dati, insieme con adeguate definizioni di sicurezza per i programmi espressi utilizzando WhileSQL. Riduciamo il
controllo degli accessi per le basi di dati al problema di determinare se un programma WHILESQL divulga informazioni sensibili. Questo permette di applicare varie tecniche già esistenti per il controllo del flusso delle informazioni al problema del controllo degli accessi nelle basi di dati. In aggiunta, sviluppiamo un meccanismo per controllare il flusso delle informazioni in programmi WHILESQL e dimostriamo la sua sicurezza. Il nostro meccanismo combina il tracciamento dinamico del flusso delle informazioni con concetti derivati dalla teoria delle basi di dati per fornire precise garanzie di sicurezza per programmi che interagiscono con le moderne basi di dati.
Acknowledgements

First and foremost, I wish to thank my advisor David Basin for his support, education, and guidance in developing this work. During my years at ETH, David has always provided prompt, precise, and insightful comments on all my ideas and writings, and he has always found time in his busy schedule to discuss the many issues associated with my research. David taught me a lot from a research perspective: how to formulate clean and abstract problem statements, how to effectively organize and develop my ideas, and how to concisely communicate them. Thanks to him, I am a better researcher and person. I also wish to thank him for his continued effort in reminding me that simplicity and conciseness (in thoughts, words, and notation) lay at the heart of clean and ultimately beautiful research.

I wish to express my appreciation and gratitude to my co-examiners—Stephen Chong and Heiko Mantel—for taking time to read through this thesis, providing insightful comments and criticisms, and for allowing me to defend my work.

The work presented in this thesis owes a large debt of gratitude to my collaborators—Srdjan Marinovic, Daniel Schoepe, Musard Balliu, and Andrei Sabelfeld—who shared greatly in its creation. It was a pleasure working with them and I enjoyed our discussions on database security.

I wish to express a special thank to Srdjan Marinovic, who, in addition to being a terrific collaborator, provided many creative ideas as well as insightful comments. Despite his busy schedule, Srdjan has always been available for lengthy discussions about our research as well as the pitfalls of academic life. These discussions allowed me to evaluate my research from new perspectives and, ultimately, greatly improved this thesis’ quality.

I wish to extend my heartfelt gratitude to all those who commented on drafts of my papers, chapters, and presentations from which this thesis is composed. In alphabetical order, these persons are: Carlos Cotrini, Mohammad Torabi Dashti, Jannik Dreier, Erwin Fang, Arthur Gervais, Nikolaos Karapanos, Andreas Lochbihler, Ognjen Maric, Esfaandiar Mohammadi, Hubert Ritzauf, Lara Schmid, David Sommer, Christoph Sprenger, Dmitriy Traytel, Petar Tsankov, Thilo Weghorn, Der-Yeuan Yu, and Eugen Zalinescu.

Our research group’s secretary Barbara Pfänder deserves a special praise for how she easily, efficiently, and effectively handled the many bureaucratic issues—both academic and governmental—that arose during my stay at ETH.

I wish to express a special thank to Arthur Gervais, Nikolaos Karapanos, David Sommer, and Der-Yeuan Yu, who shared the office with me during my stay at ETH and endured the mess on my desk (and other office’s furnitures) for so long. I also wish to thank Carlos Cotrini, Petar Tsankov, and Thilo Weghorn for the many discussions on various aspects of access control. I wish to thank all the members of the Institute of Information Security, who made my stay at ETH an intellectually stimulating experience. Furthermore, I would like to thank all my friends at ETH for all the after-work outings and movies, all the fun discussions during tea and coffee breaks, lunches, and dinners, as well as all the late nights spent together trying to meet a deadline. All these moments made my stay at ETH memorable and extremely pleasant.

I wish to thank my parents Angela and Ernesto and my brother Stefano for their support throughout my entire life. What I am now I owe to them, and this thesis is as much my achievement as theirs. They also deserve my gratitude for hosting me during my (rather frequent) visits to Italy, which represented an often much needed break from the stress of academic life as well as one of the few ways of getting access to delicious and reasonably priced food.

Finally, I would like to thank my girlfriend Letizia. She supported and encouraged me during the many ups and downs of my PhD for 5 years. She patiently waited me for all this time, and she put a lot of effort into making our relationship work even across the Alps. Without her support, this thesis would hardly exist.
Contents

Abstract iii
Riassunto v
Acknowledgements vii

I Introduction and Background 1

1 Introduction 3
  1.1 Regulating Access to Databases .............................. 4
  1.2 Provable Security for Databases – Challenges and Gaps ........ 5
  1.3 Contributions ..................................................... 6
  1.4 Publications ....................................................... 8
  1.5 Organization and Structure ..................................... 8

2 Notation and Background 11
  2.1 Notation ............................................................. 11
  2.2 Databases .......................................................... 12
    2.2.1 Modeling Databases ........................................... 12
    2.2.2 Relational calculus .......................................... 12
    2.2.3 Relevant Database Theory Problems ......................... 14
    2.2.4 Disclosure Lattices ........................................... 15
  2.3 Logic Programming ............................................... 15
  2.4 Bayesian Networks ............................................... 16

II Protecting databases against SELECT-only attackers 19

3 Database Access Control for SELECT-only attackers 21
  3.1 Introduction ......................................................... 21
  3.2 Fragments of the relational calculus ........................... 22
    3.2.1 Decision Problems .......................................... 23
  3.3 Security Policies .................................................. 23
  3.4 Security-Aware Query Processing ................................ 25
  3.5 Boolean Queries ................................................... 26
    3.5.1 Preliminaries ................................................ 26
    3.5.2 Impossibility Results ....................................... 27
    3.5.3 Possibility Results .......................................... 27
    3.5.4 Truman and Non-Truman models ............................... 29
  3.6 Non-boolean queries ............................................. 32
    3.6.1 Correctness Criteria ........................................ 32
    3.6.2 Impossibility Results ....................................... 33
    3.6.3 Possibility Results .......................................... 34
    3.6.4 Truman and Non-Truman models ............................... 34
  3.7 Related Work ..................................................... 37
  3.8 Conclusions ....................................................... 38

4 Securing Databases from Probabilistic Inference 39
  4.1 Motivating Example .............................................. 39
  4.2 System Model ..................................................... 40
  4.3 Formal Model ..................................................... 41
    4.3.1 Database Model .............................................. 41
    4.3.2 Security Policies .......................................... 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3</td>
<td>Formalized System Model</td>
<td>42</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Attacker Model</td>
<td>43</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Confidentiality</td>
<td>44</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Discussion</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>ATKLOG</td>
<td>45</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Probabilistic Logic Programming</td>
<td>45</td>
</tr>
<tr>
<td>4.4.2</td>
<td>ATKLOG’s Foundations</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Tractable Inference for ProbLog programs</td>
<td>47</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Preliminaries</td>
<td>48</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Annotations</td>
<td>50</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Acyclic ProbLog programs</td>
<td>51</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Inference Engine</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>ANGERONA</td>
<td>54</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Checking Query Security</td>
<td>54</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Implementation and Empirical Evaluation</td>
<td>56</td>
</tr>
<tr>
<td>4.7</td>
<td>Related Work</td>
<td>57</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusions</td>
<td>59</td>
</tr>
<tr>
<td>4.9</td>
<td>Technical Details, Extensions, and Additional Examples</td>
<td>59</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Relaxed acyclic programs</td>
<td>59</td>
</tr>
<tr>
<td>4.9.1.1</td>
<td>Rule Domination</td>
<td>60</td>
</tr>
<tr>
<td>4.9.1.2</td>
<td>Safe annotated disjunctions</td>
<td>60</td>
</tr>
<tr>
<td>4.9.1.3</td>
<td>Relaxed Acyclic ProbLog programs</td>
<td>61</td>
</tr>
<tr>
<td>4.9.1.4</td>
<td>Notation</td>
<td>62</td>
</tr>
<tr>
<td>4.9.1.5</td>
<td>Compilation to Bayesian Networks</td>
<td>62</td>
</tr>
<tr>
<td>4.9.2</td>
<td>From relational calculus to logic programs</td>
<td>62</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Acyclicity of the medical data example</td>
<td>66</td>
</tr>
<tr>
<td>4.9.4</td>
<td>Genomic Data Example</td>
<td>68</td>
</tr>
<tr>
<td>4.9.4.1</td>
<td>Overview</td>
<td>68</td>
</tr>
<tr>
<td>4.9.4.2</td>
<td>Encoding</td>
<td>68</td>
</tr>
<tr>
<td>4.9.4.3</td>
<td>Experiments</td>
<td>70</td>
</tr>
<tr>
<td>5.1.1</td>
<td>System Model</td>
<td>75</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Supported features</td>
<td>76</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Databases and Queries</td>
<td>77</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Views</td>
<td>77</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Security Policies</td>
<td>77</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Triggers</td>
<td>78</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Auxiliary functions</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>Illustrative Attacks</td>
<td>87</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Integrity Attacks</td>
<td>88</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Confidentiality Attacks</td>
<td>89</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Discussion</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>Attacker Model</td>
<td>90</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Formal Attacker Model</td>
<td>90</td>
</tr>
<tr>
<td>6.3</td>
<td>Database Integrity</td>
<td>92</td>
</tr>
<tr>
<td>6.4</td>
<td>Data Confidentiality</td>
<td>95</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Definition</td>
<td>95</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Indistinguishability</td>
<td>96</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Examples</td>
<td>98</td>
</tr>
<tr>
<td>6.5</td>
<td>A Provably Secure PDP</td>
<td>98</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Enforcing Database Integrity</td>
<td>99</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Enforcing Data Confidentiality</td>
<td>99</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Theoretical Evaluation</td>
<td>101</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Implementation</td>
<td>102</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>B.4                      Complexity of Inference</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>B.4.1                    Size of the encoding</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>B.4.2                    Complexity Proofs</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>B.5                      Expressiveness</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>B.6                      ANGERONA</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>B.6.1                    Security Proof</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>B.6.2                    Complexity Proof</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>B.6.3                    Completeness Proof</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>C.1                      Soundness of the attacker model</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>C.2                      Indistinguishability is an equivalence relation</td>
<td>197</td>
<td></td>
</tr>
<tr>
<td>C.3                      Database Integrity Proofs</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>C.3.1                    Extend function</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>C.3.2                    A sound under-approximation of query determinacy</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>C.3.3                    $\approx_{\text{query}}$ is a sound approximation of $\approx_{\text{auth}}$</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>C.3.4                    $f$ provides Database Integrity</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>C.4                      Data Confidentiality Proofs</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>C.4.1                    A sound under-approximation of query containment</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>C.4.2                    Data security is a sound under-approximation of judgment’s security</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>C.4.3                    Properties of $\phi^\wedge_{\text{auth}}$ and $\phi^\wedge_{\text{conf}}$</td>
<td>204</td>
<td></td>
</tr>
<tr>
<td>C.4.4                    The $\text{secure}$ function is a sound under-approximation of data security</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>C.4.5                    Auxiliary results about $f_{\text{conf}}$</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>C.4.6                    Equivalence class preservation</td>
<td>212</td>
<td></td>
</tr>
<tr>
<td>C.4.7                    Auxiliary results about $getInfoV$ and $getInfoS$</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>C.4.8                    Auxiliary results about $f$</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>C.4.9                    $f$ preserves the equivalence class</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>C.4.10                   $f$ provides Data Confidentiality</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>C.5                      Complexity Proofs</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>C.5.1                    Complexity of $f_{\text{int}}$</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>C.5.2                    Complexity of $f_{\text{conf}}$</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>C.5.3                    Complexity of the overall algorithm</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>D.1                      From Database Access Control to Information-flow Control</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>D.1.1                    Weak indistinguishability</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>D.1.2                    Auxiliary notation</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>D.1.3                    Proofs about weak indistinguishability</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>D.1.4                    Correctness of Reduction 7.1</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>D.1.5                    Equivalence-class preservation</td>
<td>237</td>
<td></td>
</tr>
<tr>
<td>D.1.6                    Proof of the main result</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>D.1.7                    Security in the Truman model</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>D.1.8                    From internal attackers to external attackers</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>D.1.9                    From progress-sensitive to progress-insensitive security</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>D.2                      Monitor’s transparency</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>D.2.1                    Local semantics</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>D.2.2                    Global semantics</td>
<td>264</td>
<td></td>
</tr>
<tr>
<td>D.3                      Monitor’s soundness</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>D.3.1                    Auxiliary notation</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>D.3.2                    Equivalence definitions</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>D.3.3                    Results about $\sqcup$</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td>D.3.4                    Results about $L_Q$</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>D.3.5                    Results about relaxed NSU checks</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>D.3.6                    Lemmas about the local semantics</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>D.3.7                    Lemmas about the global semantics</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>D.3.8                    Bisimulations</td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>D.3.9                    Proof of the main result</td>
<td>285</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography 296
Resume 297
List of Figures

1.1 System model for DBAC and DBIC. .................................................. 4

2.1 Portion of the disclosure lattice involving the queries $T(1) \land R(2)$, $T(1) \lor R(2)$, $T(1)$, and $R(2)$. .................................................. 15

3.1 Some states. .................................................................................. 24

3.2 An optimal SAQP for boolean queries. ........................................ 26

3.3 An optimal SAQP for non-boolean queries .................................. 33

3.4 A strongly-optimal SAQP for non-boolean queries ...................... 35

4.1 System model. ............................................................................ 40

4.2 The template for all database states, where the content of the cancer table is left unspecified. .................................................. 41

4.3 Probability distribution for the random variables $X_{\text{klies}}, X_{\text{bak}},$ and $X_{\text{early}}$ from Example 4.4. ........................................ 44

4.4 Probability distribution over all database states. Each state is denoted as $s_{C}$, where $C$ is the content of the cancer table. Here we denote the patients’ names with their initials. .................................................. 44

4.5 Evolution of Mallory’s beliefs in the secrets $\phi_{1}, \ldots, \phi_{4}$ for the run $r$ and the attacker model $\text{ATK}$ from Example 4.5. In the table, $X$ and $\checkmark$ denote that secrecy-preservation is violated and satisfied respectively, whereas $*$ denotes trivial secrets. .................................................. 45

4.6 Dependency graph for the program in Example 4.7. ................. 49

4.7 Ground graph for the program in Example 4.7. The ground rules $r'_{A}, r'_{B}, r_{A},$ and $r^{2}_{C}$ are as follows: $r'_{A} = B(1) \leftarrow A(1), D(1), r'_{B} = B(2) \leftarrow A(2), E(2), r'_{B} = B(2) \leftarrow B(1),$ $\neg F(1), O(1, 2),$ and $r_{C}^{2} = B(3) \leftarrow B(2), \neg F(2), O(2, 3)$. .................................................. 49

4.8 Ground graph for the program in Example 4.7 extended with the atom $E(1)$. The additional edges and nodes are represented using dashed lines. The ground rules $r'_{A}, r'_{B}, r_{A},$ and $r_{C}^{2}$ are as in Figure 4.7, and $r' = B(1) \leftarrow A(1), E(1)$, $\neg F(1), O(1, 2),$ and $r_{C}^{2} = B(3) \leftarrow B(2), \neg F(2), O(2, 3)$, together with the CPT encoding $r_{C}^{2}$’s semantics. .................................................. 53

4.9 Portion of the resulting BN for the atoms $B(2), F(2),$ and $O(2, 3)$, the rule $r_{C} = B(y) \leftarrow B(x), \neg F(x), O(x, y)$, and the ground rule $r_{C}^{2} = B(3) \leftarrow B(2), \neg F(2), O(2, 3)$, together with the CPT encoding $r_{C}^{2}$’s semantics. .................................................. 53

4.10 Angerona execution time in seconds. ........................................ 57

4.11 Dependency graph for the program capturing the motivating example in Section 4.1. .................................................. 67

4.12 Probability distribution for a child’s genome given his parents’ genome, for a fixed position. .................................................. 68

4.13 Dependency graph for the genomic data example. .................. 70

4.14 Angerona execution time in seconds. ........................................ 71

5.1 System model. ............................................................................ 75

5.2 Syntax of the supported commands. ........................................... 76

5.3 Rules for the ADD USER, SELECT, GRANT, and REVOKE commands. .................................................. 81

5.4 Rules for the INSERT and DELETE commands. ....................... 82

5.5 Rules for triggers. .................................................................... 83

5.6 Rules for the creation of triggers and views. ............................ 84

6.1 Summary of the integrity attacks. ............................................. 88

6.2 Summary of the confidentiality attacks. ..................................... 88

6.3 Example of attacker inference rules, where $r, i \vdash_{\text{w}} \phi$ denotes that this judgment holds in $\text{ATT}_{w}$. .................................................. 91

6.4 Template Derivation of Attack 6.5 (contains just selected subgoals) .................................................. 92

6.5 Definition of the $\vdash_{\text{auth}}$ relation. .......................................... 94

6.6 The runs $r(db_{1})$ and $r(db_{2})$ are indistinguishable, whereas $r(db_{1})$ and $r(db_{3})$ are not. .................................................. 96
6.7 The PDP $f$ uses the two subroutines $f_{out}$ and $f_{conf}$. The former provides database integrity and the latter provides data confidentiality with respect to the user $user(s, a)$, which denotes either the user issuing the action, when the system is not executing a trigger, or the trigger’s invoker.

6.8 Checking the security of the judgment $r. 1 \vdash \alpha (\exists y. S(2, y)) \land (\neg R(5) \lor \exists y. S(4, y))$ from Example 6.7.

6.9 Example 6.6 – PDP execution time

6.10 Example 6.7 – PDP execution time

6.11 Example 6.7 – $f_{conf}$’s execution time

6.12 Rules defining how the attacker propagates the knowledge.

6.13 Rules regulating how information propagates in case of successful INSERT and DELETE.

6.14 Rules defining how the attacker extracts knowledge from the run.

6.15 Rules regulating how information propagates in case of rollbacks.

6.16 Rules regulating the reasoning.

6.17 Rules describing how the attacker learns facts about INSERT and DELETE commands.

6.18 Rules regulating the propagation of information through disabled triggers.


6.20 Extracting knowledge from triggers – part 2.

6.21 Rules for propagating knowledge through triggers.

6.22 Extracting knowledge from disabled triggers.

6.23 Extracting knowledge from the PDP.

6.24 Extracting knowledge from trigger’s exceptions.

6.25 Definition of the $-\equiv_{\text{with}}$ relation.

6.26 Access control function $f_{\text{acC}}$.

6.27 Containment rules.

7.1 System model.

7.2 WHILESQL’s syntax.

7.3 WHILESQL’s local operational semantics.

7.4 WHILESQL’s global operational semantics.

7.5 Security monitor – Selected Rules.

7.6 Definition of the $toTrace$ function. The functions $res$ and $acC$ are defined in Chapter 5.

7.7 Definition of the $\upsilon(q, s)$ function – part 1.

7.8 Definition of the $\upsilon(q, s)$ function – part 2. Note that $q = \text{INSERT } T \text{ INTO } T$.

7.9 Definition of the $\upsilon(q, s)$ function – part 3. Note that $q = \text{DELETE } T \text{ FROM } T$.

7.10 Definition of the $\upsilon(q, s)$ function – part 4.

7.11 Security monitor – local operational semantics for assignments, print, and control flow statements.

7.12 Security monitor – local operational semantics for atomic statements.

7.13 Security monitor – local operational semantics for the new commands (set pc, asuser, and dbout).


7.15 Security monitor – global operational semantics.

7.16 allowed function for the expansion process.

7.17 apply function for the expansion process. Note that we are interested only in changes to the database configuration, not to the database state. Therefore, the function does not update the database on INSERT and DELETE commands.

7.18 Weakest precondition for sequences of instrumented commands.

7.19 Expansion process – 1.

7.20 Expansion process – 2.

A.1 Derivation for the $\psi_{S,q,s}$ formula.

A.2 Derivation for the $\theta_{q,s}(\mathbf{7}, \mathbf{7})$ formula.

C.1 Rules defining the gen relation.

D.1 Direct dependencies.
List of Tables

3.1 Summary of results. Additionally, Theorem 3.4 proves that the Non-Truman model is a special case of the Truman model for boolean queries, whereas Corollary 3.1 proves that the two models are distinct for non-boolean queries. . . . . . . . . . . . . . . . . . . . . . 22
3.2 Decidability of the $FINSAT^F$ and $FINVAL^F$ decision problems (taken from [37]). . . 23
Part I

Introduction and Background
Chapter 1

Introduction

Our society is interconnected and data-driven. According to a recent study [75], the digital universe, i.e., the amount of data generated, processed, and replicated, in the year 2020 will be about 40 zettabytes, which is $10^{21}$ bytes (this is equivalent approximately to 5,200 gigabytes for every human being on the planet). These data range from seemingly non-sensitive information, like public posts on social networks, to extremely sensitive ones, such as financial transactions or health-related information.

Protecting and regulating the access to this growing collection of data is one of the main challenges faced by information systems. On the one hand, many companies consider the data they process and collect as a critical asset. They are therefore interested in protecting data and preventing undesired leaks of sensitive information. On the other hand, these data can be used to infer sensitive information about individuals, and our society is therefore growing increasingly aware of the associated security and privacy risks. A recent survey [156] shows that security is one of the main concerns of European citizens. In particular, 88% of the interviewees say that data security is an important factor when choosing a company, and about 50% of them are willing to pay for better data protection. This increased interest in security is reflected, for example, by the recent European data protection regulation [130], which requires IT companies to properly secure customer data. For instance, “Personal data should be processed in a manner that ensures appropriate security and confidentiality of the personal data, including for preventing unauthorised access to or use of personal data and the equipment used for the processing” [130]. Satisfying such a requirement necessitates protection mechanisms that provide precise security guarantees and are secure against attacks.

Databases are one of the main components of information systems. Regulating the access to databases is, therefore, of utmost importance to prevent leaks of sensitive information. To this end, researchers have developed protection mechanisms offering access control and inference control capabilities. Existing mechanisms, however, are inadequate to secure modern systems. They are developed in ad hoc fashion, ignore many security-relevant database features, and do not provide precise security guarantees. It is, therefore, unclear what classes of attacks they prevent and in which sense they secure the database system. This has immediate consequences: attackers can exploit advanced database features to subvert protection mechanisms implemented in commercial databases and infer sensitive information. These mechanisms, therefore, cannot be used to fulfill the strict requirements, such as ensuring data confidentiality, imposed by recent regulations [130].

To address all these limitations, we propose the use of provably secure protection mechanisms. These mechanisms come with precise security guarantees and proofs specifying which classes of attacks they are designed to thwart. With provably secure mechanisms, securing a system amounts to (1) identifying the precise attacker model and the desired security guarantees, and (2) deploying and configuring the mechanisms that provide these guarantees. The accompanying proofs then certify that the desired guarantees are met and that the system is protected against specific classes of attacks.

In this thesis, we design provably secure mechanisms for access control and inference control. We first develop formal foundations for access and inference control in databases. We then leverage these foundations to construct provably secure and practical protection mechanisms for modern databases. The protection mechanisms we present in this thesis provide precise security guarantees, support advanced database features, and prevent attacks that existing mechanisms fail to stop. Observe that, in the context of this thesis, “provably secure” means that a protection mechanism’s specification provably provides the desired security guarantees. However, we do not cover the verification of protection mechanisms’ implementations, i.e., whether the implementation actually complies with the mechanism’s specification.

In the following, we first introduce access control and inference control for database systems. Afterwards, we illustrate the limitations of existing approaches. We then present our contributions and, finally, we discuss this thesis’ organization.
1.1 Regulating Access to Databases

It is essential to control the access to databases that store sensitive information. To this end, researchers have developed both database access control (DBAC) and database inference control (DBIC) mechanisms. We now summarize the key features of these approaches.

System Model. DBAC and DBIC share a common system model, depicted in Figure 1.1. Users interact with two components: a database system, which stores and manages the data, and a protection mechanism (be it a DBAC or DBIC mechanism), which intercepts the commands issued by the users and determines whether these commands are secure. The protection mechanism is parametrized by a security policy, which is designed by the security engineers and specifies which operations are secure. For simplicity, we assume that all communication between users and the components and between the components themselves is over secure channels.

When a user issues a command, the protection mechanism first checks whether the command is secure. If this is the case, it forwards the command to the database system. The database, then, executes the command and returns the command’s result to the mechanism, which forwards it to the user. If instead the command is not secure, the protection mechanism blocks the command’s execution, often returning an error message to the user.

Database Access Control. The main goal of DBAC is restricting the operations that each user can perform on the database. In general, a DBAC mechanism restricts both read and write access to the database. The former happens when a user acquires information about a portion of the database’s content, for instance using a SELECT query. The latter instead happens when a user modifies the database content, e.g., using INSERT and DELETE commands, or the database configuration, for instance by modifying the policy or the database schema.

A DBAC mechanism is parametrized by a security policy, which expresses, often using a formal language, the operations each user is authorized to perform on the database. In the case of commercial databases, the policy is formalized using SQL, and it specifies (1) the database’s content each user can read, and (2) the write operations each user can execute. A DBAC mechanism then intercepts all commands issued by the users and blocks all those commands that are not authorized, i.e., that do not comply with the given policy. Observe that in terms of read access, DBAC mechanisms mostly focus on restricting direct access [72]. Direct access to information stored in a database happens whenever a user issues a query, the database answers the query, and the user finally observes this query’s result.

Database Inference Control. In contrast to DBAC, inference control restricts indirect read access to the database [72]. Indirect access happens whenever users infer sensitive information by combining the observed query results with external information, such as data dependencies or prior knowledge. Note that DBIC mechanisms usually ignore write access to the database.

In this setting, a DBIC mechanism is usually parametrized by (1) a security policy specifying the sensitive information, and (2) a description of the source of external information. Similarly to DBAC mechanisms, an inference control mechanism intercepts all user queries and authorizes only those that do not violate the security policy. DBIC has attracted considerable attention in recent years, and current research considers different sources of external information, such as the database schema [45, 91, 95, 128, 151, 152], statistical information [11, 47, 62, 65, 67], error messages [101], user-defined functions [101], and data dependencies [36, 40, 120, 121, 157, 158, 168].

Securing databases. Protecting the confidentiality of the data stored in databases requires protection from both direct and indirect access. As a result, it is essential to deploy both DBAC and DBIC protection mechanisms. Observe that the distinction between direct and indirect access is not always clear-cut. For instance, a leak through an exception caused by an INSERT command could be interpreted as direct access, as a user may learn information by directly observing the exception’s result. However, it could also be classified as an indirect access since to infer information from the message a user would have to know which integrity constraint has been violated. Hence, some DBAC mechanisms may also include inference control aspects.
1.2 Provable Security for Databases – Challenges and Gaps

To secure a database system, security engineers first have to identify the attacker model and the desired security guarantees. Afterwards, they have to deploy some enforcement mechanisms that can provide these guarantees. Finally, they have to properly configure the deployed mechanisms to ensure that the guarantees are met.

Securing database systems is difficult. The aforementioned process may fail at any step, leading to insecure and vulnerable systems. For example, overseeing some aspects of the security analysis may lead to underestimating the attacker’s capabilities and settling for security guarantees that are weaker than those needed in practice. Similarly, a misconfigured enforcement mechanism may grant too many permissions to users, thereby giving them access to sensitive information. These kinds of failures are well understood and researchers have been studying how to prevent them for decades. In particular, methodologies [119,147,159] exist to help security engineers in eliciting the attacker model and the security requirements, thereby reducing the likelihood of mistakes during the first step. Similarly, access control researchers have been developing techniques and tools [74,100] for verifying the correctness of a given security policy with respect to a set of requirements, making it easier to identify misconfigured systems.

A third, subtler, way in which the above process may fail is in case the enforcement mechanisms do not offer the desired guarantees. This is particularly critical since security engineers often just re-use existing enforcement mechanisms; they do not design them. Hence, a mechanism that fails in delivering the desired guarantees would affect the security of all systems relying on it. Despite this, developing provably secure enforcement mechanisms is extremely hard, and even enforcement mechanisms implemented in commercial database systems fail in providing basic security guarantees, such as confidentiality. This is due to a lack of adequate and solid foundations for access and inference control in databases. In particular, existing formal models for DBAC and DBIC fail in meeting the following requirements, which are essential for reasoning about security of modern databases.

- **Realistic attacker models.** An attacker model specifies an attacker’s capabilities, how he interacts with the database system, how he infers information from the system’s behavior, and how he reasons about the acquired information. Reasoning about the security of an enforcement mechanism, therefore, requires an attacker model. In particular, the attacker model should be precise and formal to allow for security proofs, and it should capture the attacker’s capabilities. However, existing DBAC and DBIC mechanisms, such as [13,40,131,165], do not explicitly formalize their attacker models, thereby making it more difficult to understand the provided security guarantees. Attacker models should be first-class citizens in database security. The relevant models must be made explicit, just like when analyzing security in other domains. Solid foundations for DBAC and DBIC, therefore, must come with a family of precise and realistic attacker models.

- **Support for advanced, security-critical, database features.** Modern database systems provide users with much more than just SELECT queries to retrieve information. Indeed, users can modify the database content using INSERT and DELETE commands. They can also modify the database configuration or the security policy, for instance using GRANT commands to delegate permissions. Additionally, databases support advanced features like views, which allows the creation of virtual tables, or triggers, which automatically execute operations in response to user commands. All these features can be exploited by attackers to leak sensitive information. Reasoning about the security of modern database systems, therefore, requires formal models that account for all these advanced features. Unfortunately, existing formal models [131,165] mostly consider only SELECT commands, thereby ignoring more advanced features.

- **Adequate security conditions.** Reasoning about the security of enforcement mechanisms requires one to first understand and formalize what it means to be secure with respect to a given attacker model. Existing research [131,165] propose security conditions for SELECT queries based on well-known database theory problems. These conditions are, however, not adequate for a realistic setting involving modern databases, as they ignore attackers that interact with the database using more than just SELECT queries. As a result, existing conditions do not consider leaks involving features like INSERT and DELETE commands or triggers. They also ignore changes to the database and the security policy. All these features must be accounted for in any reasonable formal definition of security for modern databases.
Challenges

We identify four key challenges for DBAC and DBIC:

- **Developing foundations for DBAC and DBIC.** As explained above, existing formal models for DBAC and DBIC are not adequate for reasoning about the security of modern databases. Developing provably secure enforcement mechanisms, however, requires solid foundations. In more detail, we need (1) a formal operational semantics of database systems that accounts for advanced security-critical features, (2) precise attacker models capturing how malicious users may interact with the database, and (3) adequate security conditions.

- **Theoretical limitations.** A thorough understanding of DBAC and DBIC requires studying the theoretical limitations of these two tasks. This is important to drive the practical development of enforcement mechanisms. For instance, answering questions like “What can and cannot be done in terms of DBAC and DBIC?” or “What is the trade-off between security and data availability?” could clarify our understanding of these tasks as well as shed new light on settings where provably secure algorithms can be used efficiently without losing data availability.

- **Provably-secure enforcement mechanisms.** As mentioned before, existing protection mechanisms for databases fail in securing database systems. Attackers can infer sensitive information and modify the database’s content in unauthorized ways. An open problem for DBAC and DBIC is, therefore, developing novel enforcement mechanisms that are tractable and provide precise security guarantees. Observe that tractability is a critical feature for an interactive setting like databases, where hundreds or thousands of queries per second are processed at runtime. Providing precise guarantees is essential to ensure that the developed mechanisms work as expected and thwart attacks. In this respect, enforcement mechanisms should be accompanied with precise attacker models and proofs certifying what security guarantees are met.

- **Security beyond the database.** In addition to databases, information systems consist of applications that process the data stored in databases. To prevent leaks, one also has to reason about how these applications process the data and how they interact with the databases. To this end, security researchers have developed information-flow control (IFC) techniques which track how the data flows inside programs and detect possible leaks of sensitive information. Existing IFC techniques, such as [141,167], however, have severe limitations when it comes to dealing with databases. In particular, they consider only simplistic database models that ignore advanced security-critical features. They also ignore advances in DBAC and DBIC. Effectively securing modern information systems, however, requires combining information-flow control with DBAC and DBIC.

1.3 Contributions

This thesis makes the following contributions.

**Limitations of Database Access Control.** We characterize the limitations of database access control. In this respect, we formalize Security-Aware Query Processing, the task of computing answers to SELECT queries in the presence of access control policies, and we investigate the existence of optimal Security-Aware Query Processing algorithms, i.e., those algorithms that provide both confidentiality and maximal data availability. We present general impossibility results for the existence of optimal algorithms for Security-Aware Query Processing and classify query languages for which such algorithms exist. In particular, we show that for the relational calculus there are no optimal algorithms, whereas optimal algorithms exist for some of its fragments, such as the existential fragment. These results clarify the trade-offs between security and data availability and identify fragments of the relational calculus where it is possible to secure databases without reducing data availability due to false positives, i.e., without rejecting as insecure queries that are actually secure.

We also establish relationships between two different semantics for database access control, called Truman and Non-Truman models. These two models have different, and seemingly contradictory, goals, and their relationship was previously unclear. In the past, they were sometimes considered as distinct [131,165]. For optimal Security-Aware Query Processing, we show that the Non-Truman model is a special case of the Truman model for boolean queries in the relational calculus. Moreover, the two models coincide for sufficiently powerful query languages, such as the relational calculus with aggregation operators. In contrast, for non-boolean queries, we prove that the two models are distinct.

**Protecting databases from probabilistic inference.** We study DBIC in the presence of probabilistic data dependencies, such as those arising in genomics [97,105,109], social networks [92],
1.3. Contributions

and location tracking [117]. Attackers can exploit these dependencies to infer sensitive information with high confidence. Existing DBIC mechanisms, however, are inadequate to secure database systems in this settings. They either support very limited classes of probabilistic dependencies [44, 45, 91, 102, 120, 121, 166] or rely on intractable enforcement algorithms [116].

To address this, we develop foundations for DBIC in the presence of probabilistic data dependencies. We then use these foundations to build a tractable and practically useful DBIC mechanism based on probabilistic logic programming.

- We develop AtkLog, a language for formalizing attackers’ beliefs and how they evolve while interacting with the system. AtkLog builds on ProbLog [60, 61, 73], a state-of-the-art probabilistic extension of Datalog, and extends its semantics by building on three key ideas from [52, 103, 116]: (1) users’ beliefs can be represented as probability distributions, (2) belief revision can be performed by conditioning the probability distribution based on the users’ observations, and (3) rejecting queries as insecure may leak information. By combining Datalog with probabilistic models and belief revision based on users’ knowledge, AtkLog provides a natural and expressive language to model users’ beliefs and thereby serves as a foundation for DBIC in the presence of probabilistic inferences.

- We identify acyclic ProbLog programs, a class of programs where the data complexity of probabilistic inference is PTime. We precisely characterize this class and develop a dedicated inference engine. Since ProbLog’s inference is intractable in general, we see acyclic programs as an essential building block to effectively use AtkLog for DBIC.

- We present Angerona, a novel DBIC mechanism that secures databases against probabilistic inferences. We prove that Angerona is secure with respect to any AtkLog-attacker. In contrast to existing mechanisms, Angerona provides precise tractability and completeness guarantees for a practically relevant class of attackers. We empirically show that Angerona scales to relevant problems of interest.

Protecting modern database systems. We study the problem of protecting modern database systems that support advanced features like triggers and dynamic policies. We show that popular database systems are susceptible to two types of attacks. Integrity attacks allow an attacker to perform non-authorized changes to the database. Confidentiality attacks instead allow an attacker to learn sensitive data. These attacks exploit advanced SQL features, such as triggers, views, and integrity constraints, and they are easy to carry out.

Motivated by these attacks, we develop a comprehensive formal framework for the design and analysis of database access control. Our framework consists of a formal operational semantics of database systems and an attacker model formalizing how attackers infer information from the system’s behavior, complemented with adequate security conditions. We use this framework to design and verify an access control mechanism that prevents confidentiality and integrity attacks that existing mechanisms fail to stop.

- Our operational semantics supports SQL’s core features, as well as triggers, views, and integrity constraints. It models both the security-critical aspects of these features and the database’s dynamic behavior at the level needed to capture realistic attacks. Our semantics is substantially more detailed than those used in previous works [131, 165], which ignore the database’s dynamics and advanced database features.

- In addition to SQL’s core features, our attacker model incorporates advanced features such as triggers, views, and integrity constraints. It also accounts for how an attacker can infer information based on the semantics of these features. Our attacker model subsumes those considered in previous works [131, 165].

- Our security definitions—database integrity and data confidentiality—reflect two important security requirements for database access control. There is a natural and intuitive relationship between these definitions and the types of attacks that we identify. We thus argue that these definitions provide a strong measure of whether a given access control mechanism prevents attackers from exploiting modern SQL databases.

Guided by our operational semantics and attacker model, we build a database access control mechanism that is provably secure with respect to our attacker model and security definitions. In contrast to existing mechanisms, our solution prevents all the attacks that we identified.

Reconciling Database Access Control and Information-flow Control. Database access control and information-flow control (IFC) have the same objective: to protect the confidentiality and integrity of sensitive information. However, they have different foundations, security notions, and
enforcement mechanisms. We develop common foundations for IFC and DBAC. We leverage these foundations to (1) integrate advanced DBAC concepts into IFC, and (2) provide ways to re-use IFC techniques as building blocks for developing provably secure DBAC mechanisms.

- We develop common foundations for IFC and DBAC using WhileSQL, a simple imperative language extended with querying capabilities. WhileSQL is built on top of our database operational semantics and supports advanced database features like triggers and views. We propose two security conditions, one capturing internal attackers that directly interact with the database and the other capturing external attackers that interact with the database through an application. We use these conditions to clarify the differences between IFC and DBAC.

- We bridge IFC and DBAC. In particular, we reduce the latter to checking whether WhileSQL programs leak information. We prove that sound IFC mechanisms for internal attackers can be used to construct provably secure DBAC mechanisms. Our reduction enables the direct application of existing IFC techniques to DBAC.

- We develop a novel dynamic security monitor for WhileSQL programs and prove it sound with respect to our security condition for external attackers. Our monitor combines information-flow tracking with advanced DBAC concepts such as disclosure lattices [26] and query determinacy [124]. As a result, it can track fine-grained dependencies at the tuple-level. Additionally, our monitor supports advanced database features, such as triggers and policy changes, which are not supported by existing mechanisms [57, 59, 111, 141, 143, 167].

1.4 Publications

Much of the work presented in this thesis is based on the following co-authored articles:

- Marco Guarnieri and David Basin, “Optimal Security-Aware Query Processing”, in Proceedings of the 40th International Conference on Very Large Data Bases (VLDB 2014)
- Marco Guarnieri, Daniel Schoepe, Musard Balliu, David Basin, and Andrei Sabelfeld, “Reconciling Database Access Control and Information-flow Control” (technical report)

1.5 Organization and Structure

The thesis is organized in 5 parts, 8 chapters, and 4 appendices.

Part I consists of this introduction and Chapter 2, which presents background about databases, query languages, logic programming, and Bayesian Networks. Chapter 2 also introduces relevant problems from database theory, such as query containment and determinacy, and disclosure lattices.

Part II focuses on protecting databases against attackers that can execute only SELECT-queries, i.e., those attackers that can just retrieve information from a database, but cannot modify the database's content or configuration.

- Chapter 3 covers DBAC in the presence of SELECT-only attackers, i.e., those that can only execute SELECT queries. The chapter introduces Security-Aware Query Processing and investigates the existence of optimal Security-Aware Query Processing algorithms. It also investigates the connections between the Truman model and Non-Truman model semantics.

- Chapter 4 targets DBIC for SELECT-only attackers that, additionally, have access to the probabilistic dependencies among the database's data. It introduces ATKLOG as well as a dedicated inference engine for a fragment of PROBLOG. It also presents ANGERONA, a tractable and provably secure DBIC mechanism.

Part III focuses on securing databases against more powerful attacker models, which can exploit the advanced features provided by modern databases.

- Chapter 5 introduces our operational semantics for database systems supporting advanced features like triggers and views.
• Chapter 6 focuses on DBAC in the presence of attackers that can exploit advanced database features. The chapter presents the formal foundations for DBAC, which consist of a formal attacker model complemented with adequate security conditions, together with a provably secure enforcement mechanism.

• Chapter 7 moves beyond DBAC by focusing on end-to-end security for applications interacting with databases. It presents WhileSQL’s syntax and semantics. Additionally, it presents a construction for building DBAC mechanisms using IFC solutions as building blocks, as well as a dynamic IFC monitor that exploits advanced DBAC concepts.

Part IV, which consists only of Chapter 8, draws our conclusions and discusses future work. Finally, Part V contains the proofs of all our results.

• Appendix A contains proofs of the results in Chapter 3.
• Appendix B contains proofs of the results in Chapter 4.
• Appendix C contains proofs of the results in Chapter 6.
• Appendix D contains proofs of the results in Chapter 7.
Chapter 2

Notation and Background

In this chapter, we introduce background about the main topics covered in the thesis. We first introduce some notation that we use throughout this thesis. Afterwards, we provide an overview of databases, query languages, logic programming, and Bayesian Networks.

**Organization.** We present notation in Section 2.1. We overview database theory concepts in Section 2.2. In more detail, we formalize database theory in Section 2.2.1 and we introduce the relational calculus query language in Section 2.2.2. Furthermore, we introduce relevant database theory decision problems in Section 2.2.3, and in Section 2.2.4 we present Disclosure Lattices, which we use in Chapter 7. In Section 2.3, we introduce logic programs and we present an introduction to Bayesian Networks in Section 2.4, both of which are relevant for our treatment of probabilistic logic programs in Chapter 4.

2.1 Notation

Here, we introduce some notation used throughout the thesis.

**Sets.** Given a set $S$, we denote by $2^S$ its power-set, i.e., the set of all subsets of $S$. As is standard, $x \in S$ denotes that $x$ belongs to the set $S$ and $S' \subseteq S$ denotes that $S'$ is a subset of $S$. Furthermore, we write $x_1, \ldots, x_n \in S$ as a short-hand for $x_1 \in S, \ldots, x_n \in S$. Finally, we use standard notation for set comprehension. We write $\{x \in S \mid \phi\}$ to denote the set of all $x$ in $S$ that satisfies the predicate $\phi$. Furthermore, given $x, y \in \mathbb{N}$ such that $x \leq y$, $\{x, \ldots, y\}$ denotes the set $\{z \in \mathbb{N} \mid x \leq z \land z \leq y\}$. Finally, given two real numbers $x$ and $y$ such that $x \leq y$, $[x, y]$ denotes the set $\{z \in \mathbb{R} \mid x \leq z \land z \leq y\}$.

**Sequences.** Let $S$ be a set. We denote by $S^k$ the set of sequences over $S$ of length $k \in \mathbb{N}$. Furthermore, $S^* = \bigcup_{k \in \mathbb{N}} S^k$ is the set of all finite sequences over $S$, $S^\omega$ is the set of all infinite sequences over $S$, and, finally, $S^{\omega*} = S^* \cup S^\omega$ is the set of all finite and infinite sequences over $S$. Given a sequence $s$, we denote by $|s|$ the number of elements in it (note that $|s| = \infty$ if $s \in S^\omega$), by $s_j$, where $j \in \mathbb{N}$, its prefix of length $j$, and by $s(j)$ its $j$-th element (if it exists). We also denote by $\epsilon$ the empty sequence, by $s_1 \cdot s_2$, where $s_1 \in S^*$ and $s_2 \in S^{\omega*}$, the concatenation of $s_1$ and $s_2$, and by $s_1 \leq s_2$ that $s_1$ is a prefix of $s_2$.

**Tuples.** Let $n \in \mathbb{N}$ and $S_1, \ldots, S_n$ be sets. A $n$-tuple with domains $S_1, \ldots, S_n$ is of the form $(v_1, \ldots, v_n)$, where $v_i \in S_i$ for $1 \leq i \leq n$. We often overline an identifier to stress that it represents a tuple, e.g., we write $\overline{t}$ instead of $t$ to stress that $t$ represents a tuple.

Given an $n$-tuple $t$, we denote by $t(i)$, where $1 \leq i \leq n$, the $i$-th element in $t$ and by $|t|$ the number of elements in $t$. Namely, $(v_1, \ldots, v_n)(i) = v_i$ and $|(v_1, \ldots, v_n)| = n$. Moreover, given a tuple $t$ without repetitions and a value $v$ in $t$, we denote by $\text{pos}(t, v)$ the position of $v$ in $t$, i.e., $\text{pos}(t, v) = j$ iff $t(j) = v$. Finally, given two tuples $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_m)$, we denote by $(x_1, \ldots, x_n) \circ (y_1, \ldots, y_m)$ the tuple $(x_1, \ldots, x_n, y_1, \ldots, y_m)$.

The cartesian product of $S_1, \ldots, S_n$ is the set $S_1 \times \cdots \times S_n = \{(v_1, \ldots, v_n) \mid \bigwedge_{1 \leq i \leq n} v_i \in S_i\}$ containing all tuples with domains $S_1, \ldots, S_n$. Note that, given a set $D$, we denote by $D^n$ the cartesian product of $D$ by itself $n$ times, e.g., $D^3$ is $D \times D \times D$.

**Substitutions.** Let $A$ and $B$ be two sets. A substitution $\theta : A \to B$ is a (possibly partial) function $\theta : A \to B$. Given a tuple $(v_1, \ldots, v_n)$ and a substitution $\theta$, we denote by $(v_1, \ldots, v_n)\theta$ denoted also by $\theta((v_1, \ldots, v_n))$, the tuple $(v'_1, \ldots, v'_n)$ obtained by applying $\theta$, i.e., for each $1 \leq i \leq n$, $v'_i = \theta(v_i)$ if this is defined and $v'_i = v_i$ otherwise. Furthermore, given a $n$-tuple without repetitions $(a_1, \ldots, a_n) \in A^n$ and an $n$-tuple $(b_1, \ldots, b_n) \in B^n$, we denote by $[(a_1, \ldots, a_n) \mapsto (b_1, \ldots, b_n)]$ the substitution $A \to B$ assigning to each $a_i$ the corresponding $b_i$. Finally, we denote by $\theta[a \mapsto b]$, where $a \in A$ and $b \in B$, the assignment $\theta'$ obtained as follows: $\theta'(a') = \theta(a')$ for all $a' \neq a$, and $\theta'(a) = b$.

**Functions.** Given a function $f : A \to B$, we denote by $\text{dom}(f)$ its domain and by $\text{img}(f)$ its image. Given two functions $f : A \to B$ and $g : B \to C$, we denote by $g \circ f$ their composition, namely the function $g \circ f : A \to C$ such that $g \circ f(x) = g(f(x))$. Finally, given two functions $f : A \to B$ and
\[ g : C \rightarrow D \text{ such that } \text{dom}(f) \cap \text{dom}(g) = \emptyset, \text{ we denote by } f \bowtie g \text{ the following function:} \]

\[
f \bowtie g(x) = \begin{cases} f(x) & \text{if } x \in \text{dom}(f) \\ g(x) & \text{if } x \in \text{dom}(g) \end{cases}
\]

We sometimes adopt a notation inspired by functional programming for defining functions via \(\lambda\)-terms. For instance, we may write \(\lambda x. x + 3\) for the function \(f : \mathbb{N} \rightarrow \mathbb{N}\) such that \(f(x) = x + 3\) for all \(x \in \mathbb{N}\). Note that we often omit the typing information in case the latter is clear from the context, e.g., we write \(\lambda x. x + 3\) instead of \(\lambda x. x + 3 : \mathbb{N}\).

We also use standard (higher-order) functions from functional programming. Given a function \(f : \mathbb{N} \rightarrow M\) and a sequence \(x_1, \ldots, x_n \in \mathbb{N}\) of elements in \(\mathbb{N}\), we denote by \(\text{map}(f, x_1, \ldots, x_n)\) the sequence \(f(x_1), \ldots, f(x_n) \in M\). Given a function \(\oplus : (\mathbb{N} \times M) \rightarrow O\), a sequence \(x_1, \ldots, x_n \in \mathbb{N}\), and a sequence \(y_1, \ldots, y_n \in M\), \(\text{zipWith}(\oplus, x_1, \ldots, x_n, y_1, \ldots, y_n)\) denotes the sequence \((x_1 \oplus y_1), \ldots, (x_n \oplus y_n) \in O\).

**Binary relations.** Let \(A\) be a set. A binary relation over \(A\) is a set \(R \subseteq A \times A\). Given a binary relation \(R\), we denote by \(R^n\) the binary relation \(\{(a, b) \mid \exists c \in A. \langle a, c, b \rangle \in R^{n-1} \wedge \langle c, b \rangle \in R\}\). Observe that \(R^0\) is the reflexive relation \(\{(a, a) \mid a \in A\}\) and \(R^1\) is the relation \(R\) itself. Furthermore, we denote by \(R^*\) the reflexive and transitive closure of \(R\), i.e., \(R^* = \bigcup_{n \in \mathbb{N}} R^n\).

### 2.2 Databases

We now present background about databases, query languages, and database theory. We refer the reader to [10] for a detailed introduction to databases.

#### 2.2.1 Modeling Databases

Here, we formalize databases. Instead of relying on definitions based on the relational model, such as [10], we present a formulation based on standard concepts from first-order logic (over finite structures). While both approaches are equivalent for the purposes of this thesis, our formulation allows us to simplify various aspects, such as many-sorted tuples, that are not interesting from a security point of view.

Let \(\mathcal{R}\) be a countably infinite set representing identifiers of predicate symbols. A database schema \(D\) is a pair \(\langle \Sigma, \text{dom}\rangle\), where \(\Sigma\) is a first-order signature and \(\text{dom}\) is a fixed domain. For simplicity, we consider just a single domain. Extensions to the many-sorted case are straightforward [10]. The signature \(\Sigma\) consists of a set of predicate symbols \(R \in \mathcal{R}\) (also called relation schemas), each symbol with arity \(|R|\), and one constant symbol for each constant in \(\text{dom}\). We interpret constants by themselves in the semantics.

A database state \(db\) of \(D\), also called a state, is a finite \(\Sigma\)-structure with domain \(\text{dom}\) that interprets each predicate symbol \(R\) by a finite \(|R|\)-ary relation over \(\text{dom}\), called relation instance or table. We denote by \(\Omega_D\) the set of all states. Given a relation schema \(R\) in \(D\), \(db(R)\) denotes the set of tuples that belong to (the interpretation of) \(R\) in the state \(db\). Observe that if the domain \(\text{dom}\) is a finite set, then the set of all possible states \(\Omega_D\) is finite as well.

Given a database state \(db\), a relation schema \(R\), and a tuple \(\overline{t} \in \text{dom}^{|R|}\), we denote by \(db[R \oplus \overline{t}]\) the database \(db'\) obtained by adding \(\overline{t}\) to the relation defined by \(R\), i.e., \(db'(R) = db(R) \cup \{\overline{t}\}\) and \(db'(R') = db(R')\) for all \(R' \neq R\). Similarly, \(db[R \ominus \overline{t}]\) denotes the database \(db'\) obtained by removing \(\overline{t}\) from the relation defined by \(R\), i.e., \(db'(R) = db(R) \setminus \{\overline{t}\}\) and \(db'(R') = db(R')\) for all \(R' \neq R\).

#### 2.2.2 Relational calculus

The domain relational calculus, or simply the relational calculus (RC), is a query language defined by Codd [55] built on top of function-free first-order logic (FOL). The relational calculus is, historically, one of the classical query languages studied in database theory, together with the relational algebra and languages based on logic programming. We first introduce the relational calculus’ syntax and semantics. Afterwards, we formalize boolean and non-boolean relational calculus queries.

**Syntax.** We now formalize the relational calculus’ syntax.

**Definition 2.1.** Let \(D = \langle \Sigma, \text{dom}\rangle\) be a database schema and \(\text{Vars}\) be a countably infinite set of variables. An RC-formula over \(D\) is inductively defined as follows:

- \(R(x_1, \ldots, x_n)\) is an RC-formula over \(D\), where \(R\) is a relation schema in \(D\) and \(x_1, \ldots, x_n\) are variables in \(\text{Vars}\).
- \(x = y\) is an RC-formula over \(D\), where \(x\) and \(y\) are variables in \(\text{Vars}\) or constants in \(\text{dom}\).
- \(Q x. \phi\) is an RC-formula over \(D\), where \(\phi\) is an RC-formula over \(D\), \(x \in \text{Vars}\), and \(Q \in \{\exists, \forall\}\).
- \(\neg \phi\) is an RC-formula over \(D\), where \(\phi\) is an RC-formula over \(D\).
2.2. Databases

\[ \phi \text{ op } \psi \text{ is an RC-formula over } D, \text{ where } \phi \text{ and } \psi \text{ are RC-formulae over } D \text{ and } \text{op} \in \{\land, \lor\}. \]

As is standard, we consider only \( \text{free} \) variables. The extension to the general case is straightforward.

We also allow using constant symbols inside formulae of the form \( R(x_1, \ldots, x_n) \) with the standard meaning. For instance, \( R(1, 2, x) \) is equivalent to \( \exists x_1, x_2. (R(x_1, x_2, x) \land x_1 = 1 \land x_2 = 2) \). Finally, given two tuples \( \overline{x} \) and \( \overline{y} \) of variables and constants such that \( |\overline{x}| = |\overline{y}| \), we write \( \overline{y} \neq \overline{x} \) to denote \( \bigwedge_{i \leq |\overline{x}|} \overline{x}(i) = \overline{y}(i) \) and \( \overline{x} \neq \overline{y} \) to denote \( \bigvee_{i \leq |\overline{x}|} \overline{x}(i) \neq \overline{y}(i) \).

Let \( \phi \) be a relational calculus formula over \( D \). We denote by \( \text{free}(\phi) \) the set of free variables in \( \phi \), and we say that \( \phi \) is a \( \text{sentence} \) if \( \text{free}(\phi) = \emptyset \). We write \( \phi(\overline{x}) \), where \( \overline{x} \) is a tuple of variables without repetitions, to denote that the free variables in \( \phi \) are exactly those in \( \overline{x} \), i.e., \( \text{free}(\phi) = \{\overline{x}(1), \ldots, \overline{x}(n)\} \). Finally, given a tuple \( \overline{t} = (t_1, \ldots, t_n) \in (\text{Vars} \cup \text{dom})^n \) and a relational calculus formula \( \phi(\overline{x}, \overline{y}) \) with free variables \( \overline{x} \) and \( \overline{y} \), where \( \overline{x} = (x_1, \ldots, x_n) \), we denote by \( \phi(\overline{x} \rightarrow \overline{t}) \) the formula obtained by replacing all the free occurrences of \( x_i \) with \( t_i \) for each \( i \in \{1, \ldots, n\} \).

**Semantics.** The semantics of the relational calculus is similar to that of first-order logic.

**Definition 2.2.** Let \( D = (\Sigma, \text{dom}) \) be a database schema, \( db \) be a \( D \)-database state in \( \Omega_D \), and \( \phi \) be an \( RC \)-sentence. The satisfiability relation \( \models \) is inductively defined as follows:

- \( db \models R(v_1, \ldots, v_n) \), where \( R \) is a relation schema in \( D \) and \( v_1, \ldots, v_n \in \text{dom} \), holds iff \( (v_1, \ldots, v_n) \in db(R) \).
- \( db \models x_1 = x_2 \), where \( x_1, x_2 \in \text{dom} \), i.e., if \( db \models x_1 \mapsto x \) and \( db \models x_2 \mapsto x \) for some \( x \in \text{dom} \).
- \( db \models \forall \phi \) iff \( db \models \phi[x \rightarrow v] \) for all \( v \in \text{dom} \).
- \( db \models \exists \phi \) iff \( db \models \phi[x \rightarrow v] \) for some \( v \in \text{dom} \).
- \( db \models \neg \phi \) iff \( db \not\models \phi \).
- \( db \models \phi \land \psi \) iff \( db \models \phi \) and \( db \models \psi \).
- \( db \models \phi \lor \psi \) iff \( db \models \phi \) or \( db \models \psi \).

**Queries.** We are now ready to introduce boolean and non-boolean queries.

**Definition 2.3.** Let \( D \) be a database schema. A \( \text{non-boolean query over } D \) is of the form \( \{\overline{x} | \phi\} \), where \( \phi \) is a relational calculus sentence over \( D \), \( \overline{x} \) is a tuple of variables, \( \text{free}(\phi) = \{\overline{x}(1), \ldots, \overline{x}(n)\} \), and \( \text{free}(\phi) \neq \emptyset \). A boolean query over \( D \) is a query \( \{\phi\} \), where \( \phi \) is a relational calculus sentence.

For simplicity, we only consider non-boolean queries \( \{\overline{x} | \phi\} \) where the tuple \( \overline{x} \) does not contain repeated variables. The extension to the general case is straightforward.

We now define the result of a query over a database state.

**Definition 2.4.** Let \( D = (\Sigma, \text{dom}) \) be a database schema, \( db \in \Omega_D \) be a database state, and \( q \) be a query over \( D \). We denote the evaluation of \( q \) on the state \( db \) by \( [q]^{db} \). For a boolean query \( q = \{\phi\} \), \( [q]^{db} = \top \) if \( \phi \) is satisfied on the state \( db \), i.e., \( db \models \phi \), and otherwise \( [q]^{db} = \bot \). For a non-boolean query \( q = \{\overline{x} | \phi(\overline{x})\} \), \( [q]^{db} \) is the largest set \( T \subseteq \text{dom}^{\overline{x}} \) such that \( \{\{\overline{x} \mapsto t\} | t \in T \} \models \phi(\overline{x}) \).

With a slight abuse of notation, we refer to boolean queries \( \{\phi\} \) using only the sentence \( \phi \). We therefore write \([\phi]^{db}\) instead of \([\{\phi\}]^{db}\).

In the following, we denote by \( RC \) the set of all relational calculus queries and by \( RC_{\text{bool}} \) the set of all relational calculus boolean queries (i.e., all \( RC \)-sentences).

**Integrity Constraints.** An integrity constraint is an invariant that must be satisfied for a database state to be considered \( \text{valid} \). Formally, an \( \text{integrity constraint over } D \) is a relational calculus sentence \( \gamma \) over \( D \). Given a database state \( db \), we say that \( db \) \( \text{satisfies the constraint } \gamma \) iff \( [\gamma]^{db} = \top \). Given a set of constraints \( \Gamma \), \( \Omega_D^\Gamma \) denotes the set of all database states satisfying the constraints in \( \Gamma \), i.e., \( \Omega_D^\Gamma = \{db \in \Omega_D | \bigwedge_{\gamma \in \Gamma}[\gamma]^{db} = \top\} \). Note that if \( \Gamma = \emptyset \) (or if all sentences in \( \Gamma \) are valid), then \( \Omega_D^\Gamma = \Omega_D \), namely all database states comply with the integrity constraints.

Functional and inclusion dependencies are two well-known classes of integrity constraints [10]. The former capture primary-key constraints, whereas the latter capture foreign-key constraints. A \text{functional dependency} is a sentence of the form \( \forall \overline{x}. y, y', z. (R(\overline{x}, y, z) \land R(\overline{x}, y', z) \Rightarrow y = y') \). An \text{inclusion dependency}, instead, is a sentence of the form \( \forall \overline{x}. y. (R(\overline{x}, y) \Rightarrow \exists z. S(\overline{x}, z)) \).

**Restrictions.** As is standard, we consider only \text{domain independent} queries, which are queries that yield the same result on all possible underlying domains. Domain independent relational calculus is as expressive as relational algebra [10]. Although checking whether a query is domain independent is undecidable, there are various ways to obtain domain independent queries through syntactic
restrictions, such as safe-range queries and range-restricted queries [10]. Note that these syntactic approaches necessarily characterize proper subsets of the domain independent queries.

**Assignments.** An assignment is a substitution from the set of variables Vars to the domain dom. Given a relational calculus formula \( \phi \) and an assignment \( \nu \), we say that \( \nu \) is well-formed for \( \phi \) iff \( \nu \) is defined for all variables in free(\( \phi \)). Furthermore, given a formula \( \phi \) with free variables free(\( \phi \)) and an assignment \( \nu \), we denote by \( \phi \nu \) the formula \( \phi' \) obtained by replacing, for each free variable \( x \in \text{free}(\phi) \) such that \( \nu(x) \) is defined, all the free occurrences of \( x \) with \( \nu(x) \).

### 2.2.3 Relevant Database Theory Problems

We now review some relevant problems from database theory. These problems have a wide range of applications, such as query optimization and rewriting. They are relevant for this thesis as they are closely related to existing security conditions for DBAC and DBIC [131,165].

**Finite satisfiability and validity.** Finite satisfiability and finite validity are the variants of the usual satisfiability and validity problems for first-order logic but restricted to finite structures. These problems have been studied in finite model theory [113]. Formally, given a database schema \( D \), we say that a sentence \( \phi \) is finitely satisfiable iff there is a database state \( db \in \Omega_D \) such that \( [\phi]^{db} = T \).

Similarly, we say that a sentence \( \phi \) is finitely valid iff for all database states \( db \in \Omega_D \), \([\phi]^{db} = T \).

Hence, the finite satisfiability (respectively validity) problem is the task of checking whether a given sentence is finitely satisfiable (respectively valid).

**Query Containment.** Query containment [10] is the task of determining, given two queries \( q \) and \( q' \), whether the result of \( q' \) is always contained in the result of \( q \) in any possible database state. Query containment is a fundamental problem in query evaluation and optimization. In particular, it can be used to check whether different rewritings of the same query are equivalent, and this can be done efficiently for a practically relevant fragment of the relational calculus (called Conjunctive Queries) [48]. Formally, given a database schema \( D \) and a set of integrity constraints \( \Gamma \), the non-boolean query \( q \) contains the non-boolean query \( q' \) iff \([q']^{db} \subseteq [q]^{db}\) for all \( db \in \Omega_D \).

**Certain and Possible Answers.** The certain answer problem is the task of determining, given a query \( q \), a database state \( db \), and a tuple \( t \), whether \( t \) belongs to \( q \)'s results in all database states \( db' \) that agree with \( db \) on a certain criterion. The possible answer problem, instead, is its dual, i.e., it is the task of determining, given a query \( q \), a database state \( db \), and a tuple \( t \), whether there exists a database state \( db' \) that agrees with \( db \) on a certain criterion such that \( t \) belongs to \( q \)'s results.

These two problems have been thoroughly studied in the context of answering queries in the presence of incomplete information, for instance for incomplete databases [112] or for determining answers given a set of views [9].

We formalize here a general version of these problems inspired by [9]. Let \( D \) be a database schema, \( \Gamma \) be a set of integrity constraints over \( D \), \( Q \) be a set of queries over \( D \), \( q = \{ \exists \mid \phi \} \) be a non-boolean query over \( D \), \( t \in \text{dom}(\phi) \) be a tuple, and \( db \in \Omega_D \) be a database state. We say that \( t \) is a certain answer for \( q \) given \( db \) and \( Q \) iff for all database states \( db' \) in \( \Omega_D \), \([q]^{db'} = [q]^{db} \) for all \( q' \in Q \), then \( t \in [q]^{db} \).

Similarly, we say that \( t \) is a possible answer for \( q \) given \( db \) and \( Q \) iff there exists a database state \( db' \) in \( \Omega_D \) such that \([q]^{db'} = [q']^{db} \) for all \( q' \in Q \) and \( t \in [q']^{db} \).

**Query Determinacy.** Query determinacy [124] is the task of determining, given two sets of queries \( Q \) and \( Q' \), if the results of the queries in \( Q \) are always sufficient to determine the result of the queries in \( Q' \). This, therefore, means that the queries in \( Q \) contain more information than those in \( Q' \). Note that query determinacy has many applications to query rewriting [124].

Let \( D \) be a database schema, \( \Gamma \) be a set of integrity constraints over \( D \), and \( Q \) and \( Q' \) be sets of queries over \( D \). We say that \( Q \) determines \( Q' \), written \( D, \Gamma \models Q \Rightarrow Q' \), iff for all database states \( db, db' \) in \( \Omega_D \), if \([q']^{db} = [q']^{db'} \) for all \( q' \in Q \), then \([q]^db = [q']^{db} \) for all \( q' \in Q' \). For instance, the set \( Q = \{ T(1), \forall \} \) determines the query \( T(1) \lor R(2) \).

**Instance-based Determinacy.** Instance-based determinancy is a variant of the determinacy problem restricted to a fixed database state. Namely, the instance-based determinacy problem [107] consists of checking whether, given a database state \( db \), a set of queries \( Q \), and a query \( q \), the results of the queries in \( Q \) in the state \( db \) fully determine \( q \)'s result in \( db \).

Let \( D \) be a database schema, \( \Gamma \) be a set of integrity constraints over \( D \), \( Q \) be a set of queries over \( D \), \( q \) be a query over \( D \), and \( db \) be a database state in \( \Omega_D \). We say that \( Q \) determines \( q \) in the instance \( db \), written \( D, \Gamma, db \models Q \Rightarrow q \), iff for all database states \( db' \) in \( \Omega_D \), if \([q']^{db} = [q']^{db'} \) for all \( q' \in Q \), then \([q]^db = [q']^{db} \). Observe that if \( D, \Gamma, db \models Q \Rightarrow q \) holds for all \( db \in \Omega_D \), then \( D, \Gamma \models Q \Rightarrow q \) holds as well. Observe also that instance-based determinacy can be easily extended to sets of queries.

To illustrate, the set \( Q = \{ T(1) \} \) determines the query \( T(1) \lor R(2) \) in any database state \( db \) where \( T(1) \) is satisfied. Note, however, that \( Q = \{ T(1) \} \) does not determine the query \( T(1) \lor R(2) \).
2.3. Logic Programming

2.2.4 Disclosure Lattices

Disclosure lattices have been introduced by Bender et al. [26,27] to reason about the information disclosed by queries. Given two sets of queries Q₁ and Q₂, disclosure lattices provide a precise model for answering the following questions: (i) Does Q₁ reveal more information than Q₂? (ii) What combined information is revealed by Q₁ and Q₂? (iii) What common information is revealed by Q₁ alone and Q₂ alone?

Disclosure Orders. A disclosure order [26] is a binary relation \( \preceq \) over \( 2^{RC} \) such that: (1) For all \( Q, Q' \in 2^{RC} \), if \( Q \subseteq Q' \), then \( Q \preceq Q' \), (2) for all \( Q, Q', Q'' \in 2^{RC} \), if \( Q \preceq Q' \) and \( Q' \preceq Q'' \), then \( Q \preceq Q'' \), and (3) for all \( Q, Q', Q'' \in 2^{RC} \), if \( Q \preceq Q'' \) and \( Q' \preceq Q'' \), then \( Q \cup Q' \preceq Q'' \).

Disclosure Lattices. Given a set of queries \( Q \) and a disclosure order \( \preceq \), the closure of \( Q \), written \( cl(Q) \), is \( \{ q \in RC \mid [q] \preceq Q \} \). The \( \preceq \)-disclosure lattice [26] is a tuple \( (L, \subseteq, \cap, \cup, \bot, \top) \) where (1) \( L = \{ cl(Q) \mid Q \in 2^{RC} \} \), (2) \( cl(Q) \subseteq cl(Q') \) if \( Q \preceq Q' \), (3) \( cl(Q) \cap cl(Q') = cl(Q) \cap cl(Q') \), (4) \( cl(Q) \cup cl(Q') = cl(Q \cup Q') \), (5) \( \bot = cl(\emptyset) \), and (6) \( \top = cl(RC) \). Observe that \( (L, \subseteq, \cap, \cup, \bot, \top) \) is a complete lattice [26].

Disclosure Lattices based on determinacy. Determinacy induces an ordering on the information content of queries. Hence, it is a good candidate for defining disclosure lattices. Bender et al. [26] present also other disclosure lattices, such as those based on query containment.

We are ready to define the relation \( \preceq_{D, \Gamma} \): given \( Q, Q' \in 2^{RC} \), \( Q \preceq_{D, \Gamma} Q' \) iff \( D, \Gamma \vdash Q' \rightarrow Q \). Observe that \( Q \preceq_{D, \Gamma} Q' \) means that \( Q \) is less informative than \( Q' \) (since the results of the queries in \( Q' \) can be derived from the results of the queries in \( Q' \)). As shown in [26], \( \preceq_{D, \Gamma} \) is a disclosure order. As a result, the disclosure lattice defined by \( \preceq_{D, \Gamma} \) is a complete lattice [26].

To illustrate, consider the queries \( T(1) \land R(2) \), \( T(1) \lor R(2) \), \( T(1) \), and \( R(2) \). Figure 2.1 illustrates the portion of the lattice involving these queries, where arrows represent the relation \( \subseteq \). We can see that \( cl\{\{T(1)\}\} \subseteq cl\{\{T(1), R(2)\}\} \), \( cl\{\{T(1)\}\} \cup cl\{\{R(2)\}\} = cl\{\{T(1), R(2)\}\} \), \( cl\{\{T(1) \land R(2)\}\} \subseteq cl\{\{T(1), R(2)\}\} \) and \( cl\{\{T(1) \lor R(2)\}\} \subseteq cl\{\{T(1), R(2)\}\} \).

2.3 Logic Programming

We now introduce background on logic programs. In particular, we present DATALOG (extended with stratified negation), a declarative rule-based query language [10]. The key feature of DATALOG is that it supports the definition of recursive predicates.

Even though DATALOG is only a subset of more complex logic programming languages like PROLOG [43], with a slight abuse of notation in this thesis we use the term “logic programming” to refer to DATALOG with stratified negation. Note that DATALOG is relevant for our treatment of probabilistic logic programs in Chapter 4. Our treatment of logic programs follows [10]. We refer the reader to [10,43] for more details about DATALOG and PROLOG.

Syntax. Conventional DATALOG programs are constructed from atoms, literals, and rules. Let \( D = (\Sigma, \text{dom}) \) be a database schema and \( \text{Var}s \) be a countably infinite set of variable identifiers. A \((\Sigma, \text{dom})\)-atom \( R(v_1, \ldots, v_n) \) consists of a predicate symbol \( R \) in \( \Sigma \) and arguments \( v_1, \ldots, v_n \) such that \( n \) is the arity of \( R \) and each \( v_i \), for \( 1 \leq i \leq n \), is either a variable identifier in \( \text{Var}s \) or a constant in \( \text{dom} \). We denote by \( \mathcal{A}_\Sigma, \text{dom} \) the set \( \{ R(v_1, \ldots, v_n) \mid R \in \Sigma \wedge v_1, \ldots, v_n \in \text{dom} \} \) of all \((\Sigma, \text{dom})\)-atoms. A \((\Sigma, \text{dom})\)-literal \( l \) is either a \((\Sigma, \text{dom})\)-atom \( a \) or its negation \( \neg a \), where \( a \in \mathcal{A}_\Sigma, \text{dom} \). We denote by \( \mathcal{L}_\Sigma, \text{dom} \) the set \( \mathcal{A}_\Sigma, \text{dom} \cup \{ \neg a \mid a \in \mathcal{A}_\Sigma, \text{dom} \} \) of all \((\Sigma, \text{dom})\)-literals. Given a literal \( l \), \( \text{vars}(l) \) denotes the set of its variables, \( \text{args}(l) \) the tuple containing its arguments, and \( \text{pred}(l) \) the predicate symbol (i.e., the relation schema) used in \( l \). As is standard, we say that a literal \( l \) is positive if it is an atom in \( \mathcal{A}_\Sigma, \text{dom} \) and negative if it is the negation of an atom. Furthermore, we say that a literal \( l \) is ground if \( \text{vars}(l) = \emptyset \). Finally, we denote by \( \text{atom}(l) \) the atom used in \( l \), i.e., given \( a \in \mathcal{A}_\Sigma, \text{dom} \), \( \text{atom}(a) = a \) and \( \text{atom}(\neg a) = a \).

A \((\Sigma, \text{dom})\)-rule is of the form \( h \leftarrow l_1, \ldots, l_n, e_1, \ldots, e_m \), where \( h \in \mathcal{A}_\Sigma, \text{dom} \) is a \((\Sigma, \text{dom})\)-atom, \( l_1, \ldots, l_n \in \mathcal{L}_\Sigma, \text{dom} \) are \((\Sigma, \text{dom})\)-literals, and \( e_1, \ldots, e_m \) are equality and inequality constraints.

![Figure 2.1](image-url)
over the variables in \(h, l_1, \ldots, l_m\). Given a rule \(r\), we denote by \(\text{head}(r)\) the atom \(h\), by \(\text{body}(r)\) the literals \(l_1, \ldots, l_m\), by \(\text{strat}(r)\) the constraints \(e_1, \ldots, e_m\), and by \(\text{body}(r, i)\) the \(i\)-th literal in \(r\)'s body, i.e., \(\text{body}(r, i) = l_i\). Furthermore, we denote by \(\text{body}^+(r)\) (respectively \(\text{body}^-(r)\)) all positive (respectively negative) literals in \(\text{body}(r)\). As is standard, we assume that the free variables in a rule's head are a subset of the free variables of the positive literals in the rule's body, i.e., \(\text{vars}(\text{head}(r)) \subseteq \bigcup_{l \in \text{body}^+(r)} \text{vars}(l) \cup \bigcup_{l = \text{strat}(r) \land x \in \text{dom}} \{x\}\). Finally, observe that a ground atom can be seen as a rule with an empty body.

We are now ready to define logic programs. Formally, a \((\Sigma, \text{dom})\)-logic program is a set of \((\Sigma, \text{dom})\)-ground atoms and \((\Sigma, \text{dom})\)-rules. Given a program \(p\), we denote by \(\text{facts}(p)\) the set of ground atoms in \(p\) and by \(\text{rules}(p)\) all the rules that are not ground atoms, i.e., \(\text{rules}(p) = p \setminus \text{facts}(p)\). Note that we ignore the distinction between extensional relation schemas (also called edbs), which are used only in ground atoms, and intensional relation schemas (also called ibds), which are used in the head of the rules.

We consider only programs \(p\) that do not contain negative cycles in the rules as is standard for stratified Datalog with negation [10]. Namely, we consider only those programs \(p\) that admit a stratification. Following [10], we now formalize stratifications.

**Definition 2.5.** Let \(D = (\Sigma, \text{dom})\) be a database schema and \(p\) be a \((\Sigma, \text{dom})\)-logic program. A stratification \(n\) of \(p\) is a sequence of \((\Sigma, \text{dom})\)-logic programs \(p_1, \ldots, p_n\) such that for some mapping \(\mu\) from the relation schemas in \(\Sigma\) to \(\{0, \ldots, n\}\) the following conditions hold:
1. The programs \(p_1, \ldots, p_n\) are a partition of \(\text{rules}(p)\) (i.e., \(\bigcup_{i \leq n} p_i = p\) and \(p_i \cap p_j = \emptyset\) for all distinct \(i, j \in \{1, \ldots, n\}\)).
2. For all rules \(r \in \text{rules}(p)\) and all positive literals \(l \in \text{body}^+(r)\), \(\mu(\text{pred}(l)) \leq \mu(\text{pred}((\text{head}(r))))\).
3. For all rules \(r \in \text{rules}(p)\) and all negative literals \(l \in \text{body}^-(r)\), \(\mu(\text{pred}(l)) < \mu(\text{pred}((\text{head}(r))))\).
4. For all rules \(r \in \text{rules}(p)\), \(r \in p_{\mu(\text{pred}((\text{head}(r))))}\) and \(1 \leq \mu(\text{pred}((\text{head}(r))))\) \(\leq n\).

A logic program \(p\) is stratified iff there is a stratification \(p_1, \ldots, p_n\) for it, and we call each \(p_i\) a stratum.

**Semantics.** Let \(D = (\Sigma, \text{dom})\) be a database schema and \(r\) be a \((\Sigma, \text{dom})\)-rule. We denote by \(\text{ASGN}(r)\) the set of all assignments from the free variables in \(r\) to values in \(\text{dom}\) that satisfy the equality and inequality constraints in \(r\). Furthermore, given a mapping \(\theta\) from variables in \(\text{Vars}\) to values in \(\text{dom}\) and a literal \(l\), we denote by \(l\theta\) the literal obtained by replacing all free variables in \(l\) with the corresponding values in \(\theta\).

One of the key concepts of the Datalog’s semantics is the immediate consequence operator.

**Definition 2.6.** Let \(D = (\Sigma, \text{dom})\) be a database schema. The immediate consequence operator \(\text{T}\) takes as input a \((\Sigma, \text{dom})\)-logic program and returns another \((\Sigma, \text{dom})\)-logic program. The operator is defined as follows: \(\text{T}(p) = p \cup \{\text{head}(r)\theta \mid r \in p \land \theta \in \text{ASGN}(r) \land \forall l \in \text{body}^+(r). \theta \in p \land \forall l \in \text{body}^+(r)\}.\)

The semantics of a Datalog program is the database state, which can be seen as a set of ground atoms, produced by executing the program. We first define the semantics of semi-positive programs [10], i.e., programs whose stratification contains just one stratum. The semantics of a semi-positive program is the least fixpoint of the immediate consequence operator. Note that such a fixpoint always exists [10]. Hence, the semantics is well-defined.

**Definition 2.7.** Let \(D = (\Sigma, \text{dom})\) be a database schema and \(p\) be a semi-positive \((\Sigma, \text{dom})\)-logic program. The semantics of \(p\) is defined as follows: \(\text{Facts}(p) = \bigcup_{\text{gen}} \text{T}^i(p)\).

We are now ready to introduce the semantics of stratified Datalog with negation.

**Definition 2.8.** Let \(D = (\Sigma, \text{dom})\) be a database schema, \(p\) be a stratified \((\Sigma, \text{dom})\)-logic program with stratification \(p_1, \ldots, p_n\), and \(p_0 = \text{facts}(p)\). The semantics of \(p\) up to the \(i\)-th stratum is \(\text{Facts}(p)_i = \text{Facts}(p) \cup \cdots \cup \text{Facts}([p_i]. \cdots \cup [p_1])\). The semantics of the program \(p\) is \(\text{Facts}(p)\).

The stratified semantics simply executes the programs in the stratification sequentially, using the result of a stratum to derive additional facts for the next one. Note that the above semantics is well-defined since \(p_i \cup [p]_{i-1}\), for each \(1 \leq i \leq n\), is a semi-positive program.

### 2.4 Bayesian Networks

A Bayesian Network (BN) is a graphical way of compactly representing a probability distribution by specifying a set of random variables and their dependencies as a directed acyclic graph. We now present an overview of BNs, which we use in Chapter 4. We refer the interested reader to [77, 105] for

---

1 Without loss of generality, we assume that equality constraints involving a variable \(x \in \text{Vars}\) and a constant \(c \in \text{dom}\) are of the form \(x = c\).
Definition 2.9. Let \( V, V_1, \ldots, V_n \) be finite sets of possible outcomes. A probability distribution over \( V \) is a function \( P : V \to [0, 1] \) such that \( \sum_{v \in V} P(v) = 1 \). A conditional probability table over \( V \) indexed by \( V_1, \ldots, V_n \) is a function \( CPT : V_1 \times \ldots \times V_n \to \{ 0, 1 \} \) such that for all \( \forall \in V_1 \times \ldots \times V_n, CPT(\forall) \) is a probability distribution over \( V \). A random variable \( X \) is a pair \( (V, P) \), where \( V \) is a finite domain and \( P \) is a probability distribution over \( V \).

Given a random variable \( X = (V, P) \), the probability that \( v \in V \) is \( X \)'s outcome, denoted \( P(X = v) \), is \( P(v) \).

Joint probability distributions specify the probabilities of multiple random variables.

Definition 2.10. Let \( X_1 = (V_1, P_1), \ldots, X_n = (V_n, P_n) \) be random variables. A joint probability distribution over \( X_1, \ldots, X_n \) is a probability distribution \( P \) over \( V_1 \times \ldots \times V_n \) such that for each \( 1 \leq i \leq n \) and each \( v_i \in V_i, \sum_{v_{i-1} \in V_{i-1}} \ldots \sum_{v_{i+1} \in V_{i+1}} \sum_{v_{n} \in V_{n}} P((v_1, \ldots, v_i-1, v_i, v_{i+1}, \ldots, v_n)) = P_i(v_i) \).

Let \( P \) be a joint probability distribution over \( X_1 = (V_1, P_1), \ldots, X_n = (V_n, P_n) \). The probability that \( (v_1, \ldots, v_n) \in V_1 \times \ldots \times V_n \) is the outcome of \( X_1, \ldots, X_n \), written \( P(X_1 = v_1, \ldots, X_n = v_n) \), is \( P((v_1, \ldots, v_n)) \). One can derive joint distributions over subsets of \( X_1, \ldots, X_n \) by just summing out some of the variables. Let \( \mathcal{X} = \{ X_1, \ldots, X_n \} \) and \( \mathcal{Z}_1 = \{ X_1, \ldots, X_k \}, \mathcal{Z}_2 = \{ X_1, \ldots, X_j \} \) be two sets defining a partition of \( \mathcal{X} \), where each \( X_i \) is of the form \( (V_i, P_i) \) and each \( X_h \) is of the form \( (V^h, P^h) \), for \( 1 \leq i \leq k \) and \( 1 \leq h \leq j \). The probability distribution \( \lambda X_1 = v_1, \ldots, v_i \in V_i \), \( v_j \in V_j \). \( \sum_{i \in V_i} \sum_{j \in V_j} P(X_1 = v_1, \ldots, X_i = v_i, X_j = v_j, \ldots, X_j = v_j) \) is a joint probability distribution over \( X_1, \ldots, X_{i+k} \). We denote the above probability distribution as \( P X_{i+1}^{i+k} \).

A conditional probability distribution specifies the probability distribution of a random variable given a fixed outcome for other variables.

Definition 2.11. Let \( P \) be a joint probability distribution over \( X_1 = (V_1, P_1), \ldots, X_n = (V_n, P_n) \). The conditional probability of \( X_i \) given outcomes \( v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \) for \( X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n \), written \( P(X_i | X_1 = v_1, \ldots, X_{i-1} = v_{i-1}, X_{i+1} = v_{i+1}, \ldots, X_n = v_n) \), is the function \( \lambda v_i \in V_1, P(X_i = v_i, \ldots, X_n = v_n) \cdot \left( \sum_{v_i \in V_i} P(X_i = v_i, \ldots, X_n = v_n) \right)^{-1} \).

With a slight abuse of notation, we write \( P(X_i | X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) \) to denote the CPT assignning to each outcome of \( X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n \) the conditional probability of \( X_i \), i.e., the function \( \lambda v_1 \in V_1, \ldots, v_{i-1} \in V_{i-1}, v_{i+1} \in V_{i+1}, \ldots, v_n \in V_n, P(X_i | X_1 = v_1, \ldots, X_{i-1} = v_{i-1}, X_{i+1} = v_{i+1}, \ldots, X_n = v_n) \). We call \( P(X_i | X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) \) the CPT of \( X_i \) indexed by \( X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n \).

Conditional probability distributions (and tables) can be simplified by exploiting the independence and conditional independence between random variables.

Definition 2.12. Let \( P \) be a joint probability distribution over \( X_1 = (V_1, P_1), \ldots, X_n = (V_n, P_n) \). Furthermore, let \( X_i = (V_i, P_i), X_j = (V_j, P_j) \), and \( X_k = (V_k, P_k) \) be random variables in \( \{ X_1, \ldots, X_n \} \). Two random variables \( X_i \) and \( X_j \) are independent iff for all \( v_i \in V_i \) and \( v_j \in V_j \), \( P_{X_i, X_j}(X_i = v_i, X_j = v_j) = P_{X_i}(X_i = v_i) P_{X_j}(X_j = v_j) \). Two random variables \( X_i \) and \( X_j \) are conditionally independent given \( X_k \) iff for all \( v_i \in V_i, v_j \in V_j, v_k \in V_k \), \( P_{X_i, X_j}(X_i = v_i, X_j = v_j | X_k = v_k) = P_{X_i}(X_i = v_i) P_{X_j}(X_j = v_j) \).

We can, therefore, represent a joint probability distribution over \( X_1, \ldots, X_n \) as a sequence of conditional probability tables. Furthermore, we can exploit independence and conditional independence among random variables to simplify these tables.

We are now finally ready to define Bayesian Networks.

Definition 2.13. A Bayesian Network \( BN \) is a tuple \( \langle N, E, D, CPT, \leq \rangle \), where \( N \) is the set of nodes, \( E \) is the set of directed edges, \( D \) is a function associating to each node \( n \in N \) its domain, \( CPT \) is a function associating to each node \( n \in N \) its conditional probability distribution, and \( \leq \) is a total ordering of the nodes in \( N \). We denote by \( \text{parents}(n) \) the set of \( n \)'s parents, i.e., \( \text{parents}(n) = \{ n' | n' \to n \in E \} \). Furthermore, we denote by \( \text{ancestors}(n) \) the set of \( n \)'s ancestors, i.e., \( \text{ancestors}(n) = \text{parents}(n) \cup \bigcup_{n' \in \text{parents}(n)} \text{ancestors}(n') \).

A Bayesian Network \( \langle N, E, D, CPT, \leq \rangle \) must satisfy the following constraints:

- \( \langle N, E \rangle \) is a directed acyclic graph.
Let $\mathcal{N}$ be a $\mathcal{N}$-tuple assignment.

For each node $n \in \mathcal{N}$ such that $\text{parents}(n) = \emptyset$, $\text{CPT}(n)$ is a probability distribution over $D(n)$.

Furthermore, let $\text{toTpl}$ be a function that takes as input a subset $\mathcal{S}$ of $\mathcal{N}$ and returns the tuple obtained by sorting the nodes in $\mathcal{S}$ according to $\preceq$. Finally, let $\text{CPT}^*$ be the function defined as follows: $\text{CPT}^*(n) = \text{CPT}(n)$ if $\text{parents}(n) = \emptyset$ and otherwise $\text{CPT}^*(n)$ is

$$\lambda v \in D(n). \sum_{v_{p_1} \in D(p_1)} \ldots \sum_{v_{p_k} \in D(p_k)} \text{CPT}(n)((v_{p_1}, \ldots, v_{p_k}))(v) \cdot \prod_{p_i \in \text{parents}(n)} \text{CPT}^*(p_i)(v_{p_i}),$$

where $(p_1, \ldots, p_k) = \text{toTpl}(\text{parents}(n))$. Note that $\text{CPT}^*(n)$ is a probability distribution over $D(n)$.

The BN $\mathcal{B}$ defines a joint probability distribution $[\mathcal{B}]$ over the random variables $X_1 = \langle D(n_1), CPT^*(n_1) \rangle, \ldots, X_k = \langle D(n_k), CPT^*(n_k) \rangle$, where

$$[\mathcal{B}] = \lambda v_1 \in D(n_1), \ldots, v_k \in D(n_k). \prod_{n_i \in \mathcal{N}'} \text{CPT}(n_i)(v_i). \prod_{n_i \in \mathcal{N}''} \text{CPT}(n_i)(\text{toTpl}(\text{parents}(n_i)))(\langle n_1, \ldots, n_k \rangle \mapsto \langle v_1, \ldots, v_k \rangle)(v_i),$$

where $\mathcal{N}' = \{ n \in \mathcal{N} \mid \text{parents}(n) = \emptyset \}$, $\mathcal{N}'' = \{ n \in \mathcal{N} \mid \text{parents}(n) \neq \emptyset \}$. □

When $\mathcal{D}$ and $\preceq$ are clear from the context, we refer to a Bayesian Network as a triple $(\mathcal{N}, E, \text{CPT})$ instead of a 5-tuple $(\mathcal{N}, E, \text{CPT}, D, \preceq)$.

Before concluding, we introduce some notation to query Bayesian Networks. Let $\mathcal{B} = (\mathcal{N}, E, D, \text{CPT}, \preceq)$ be a Bayesian Network. A $\mathcal{B}$-total assignment is a total function $\nu$ that associates to each $n \in \mathcal{N}$ a value in $v \in D(n)$, whereas a $\mathcal{B}$-partial assignment is a partial function $\nu$ that associates to each $n \in \mathcal{N}'$, where $\mathcal{N}' \subseteq \mathcal{N}$, a value in $v \in D(n)$. We now lift the semantics of a BN to assignments. The probability defined by $\mathcal{B}$ given a partial assignment $\nu$, written $[\mathcal{B}]\!(\nu)$ is

$$\sum_{w_1 \in D(m_1)} \cdots \sum_{w_l \in D(m_l)} [\mathcal{B}]\!(\text{toTpl}(\mathcal{N})(\nu \uplus \{(m_1, \ldots, m_l) \mapsto \langle w_1, \ldots, w_l \rangle\})),$$

where $\text{dom}(\nu) = \{ n_1, \ldots, n_k \}$ and $\mathcal{N} \setminus \text{dom}(\nu) = \{ m_1, \ldots, m_l \}$. Namely, $[\mathcal{B}]\!(\nu)$ returns the probability that the random variables associated with the nodes for which $\nu$ is defined have the outcome indicated by $\nu$. □
Part II

Protecting databases against SELECT-only attackers
Chapter 3

Database Access Control for SELECT-only attackers

Retrieving data from a database is usually done via SELECT queries, which allow one to retrieve all tuples matching a given criterion. SELECT queries are one of the most common ways of interacting with a database. For this reason, many DBAC approaches target the so-called SELECT-only attacker model, where attackers can interact with the database only through SELECT queries. In this setting, therefore, both the database’s content and the security policy are fixed.

In this chapter, we formalize and study Security-Aware Query Processing, the problem of computing answers to SELECT queries in the presence of security policies. Observe that Security-Aware Query Processing models DBAC in the SELECT-only setting. In the following, we present general impossibility results for the existence of optimal algorithms for Security-Aware Query Processing and classify query languages for which such algorithms exist. In particular, we show that for the relational calculus there are no optimal algorithms, whereas optimal algorithms exist for some of its fragments, such as the existential fragment.

We also establish relationships between two different models of Fine-Grained Access Control, called Truman and Non-Truman models, which have been previously presented in the literature as distinct. For optimal Security-Aware Query Processing, we show that the Non-Truman model is a special case of the Truman model for boolean queries in the relational calculus, moreover the two models coincide for more powerful query languages, such as the relational calculus with aggregation operators. In contrast, these two models are distinct for non-boolean queries. Note that this chapter is largely based on [84].

Organization. In Section 3.1, we introduce the Truman and Non-Truman models for database access control and the concept of optimal algorithms. In Section 3.2, we present the fragments of the relational calculus used in the rest of the chapter. In Section 3.3 we introduce our security policies, and in Section 3.4 we introduce Security-Aware Query Processing. We present our results in Section 3.5 for boolean queries and in Section 3.6 for non-boolean queries. In Section 3.7, we review related work and we draw conclusions in Section 3.8. Most of the proofs of our results are directly given in this chapter. For the sake of readability, some proofs are given in Appendix A.

3.1 Introduction

Computing answers to SELECT queries given a security policy is the main task of DBAC mechanisms in the SELECT-only attacker model. There is, however, no common terminology to refer to this problem. For example, “secure querying” [58], “enforcing data confidentiality” [125], and “Fine-Grained Access Control” [13,131,165] have all been used in this context. In the following, we refer to this problem as Security-Aware Query Processing (SAQP). This differs from Secure Query Processing since the latter usually refers to querying encrypted data. Security-Aware Query Processing algorithms are implemented in commercial databases [8,41,150], and solutions have been proposed for other settings like XML files [58] or RDF graphs [125].

Rizvi et al. [131] identified two distinct models, called Truman and Non-Truman models, that capture different approaches to the SAQP problem. The term Truman model comes from the protagonist of the movie The Truman Show who is unaware that he lives in an artificial environment where
Theorems 3.3 and 3.4 provide sufficient conditions for the decidability of the existential relational calculus (ERC). Finally, Theorem 3.6 gives an undecidable fragment of the relational calculus.

### Table 3.1: Summary of Results

<table>
<thead>
<tr>
<th>Queries</th>
<th>SAQP</th>
<th>Truman Model</th>
<th>Non-Truman Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>¬∃optimal</td>
<td>undecidable fragments (Theorem 3.2)</td>
<td>undecidable fragments (Theorem 3.4)</td>
</tr>
<tr>
<td></td>
<td>∃optimal</td>
<td>sufficient conditions (Lemma 3.1) ERC (Theorem 3.3)</td>
<td>sufficient conditions (Lemma 3.2) ERC (Theorem 3.5)</td>
</tr>
<tr>
<td>Non-Boolean</td>
<td>¬∃optimal</td>
<td>undecidable fragments (Theorem 3.6)</td>
<td>undecidable fragments (Theorem 3.8)</td>
</tr>
<tr>
<td></td>
<td>∃optimal</td>
<td>sufficient conditions (Lemma 3.3) ERC (Theorem 3.7)</td>
<td>sufficient conditions (Lemma 3.4) ERC (Theorem 3.9)</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of results. Additionally, Theorem 3.4 proves that the Non-Truman model is a special case of the Truman model for boolean queries, whereas Corollary 3.1 proves that the two models are distinct for non-boolean queries.

We now introduce the fragments of the relational calculus that we study in this chapter. A fragment \( F \) of the relational calculus is a subset \( F \subseteq RC \) of the relational calculus formulae. To stress that a formula \( \phi \) belongs to a fragment \( F \), we call it an \( F \)-formula. We say that a boolean query \( \phi \) (respectively a non-boolean query \( \{\tau \mid \phi\} \)) is in a fragment \( F \) iff \( \phi \in F \).
We now introduce the existential fragment ERC of the relational calculus. Note that the ERC fragment strictly contains conjunctive queries (CRC).

**Definition 3.1.** Let $D$ be a database schema. The formula $\phi(\overline{y}) = \exists \overline{x}. \psi(\overline{x}, \overline{y})$ is a ERC-formula over $D$ iff $\psi$ is a quantifier-free RC-formula over $D$.

For technical reasons, we also consider the Bernays-Schönfinkel-Ramsey fragment BSRRC.

**Definition 3.2.** Let $D$ be a database schema. The formula $\phi(\overline{z}) = \exists \overline{x}. \forall \overline{y}. \psi(\overline{x}, \overline{y}, \overline{z})$ is a BSRRC-formula over $D$ iff $\psi$ is a quantifier-free RC-formula over $D$.

In the following, we use the term *query language* to refer to the relational calculus and its fragments, such as ERC. We assume that there is a unique encoding of formulae as natural numbers, and in our proofs we switch freely between formulae and their encodings.

### 3.2.1 Decision Problems

Here we define the finite satisfiability and finite validity decision problems for (fragments of) the relational calculus.

**Definition 3.3.** Given a query language $F$, $\text{FINSAT}^F$ denotes the following decision problem:

**Input:** A database schema $D$ and a sentence $\phi \in F$.

**Question:** Is there a state $db \in \Omega_D$ such that $[\phi]^db = \top$?

**Definition 3.4.** Given a query language $F$, $\text{FINVAL}^F$ denotes the following decision problem:

**Input:** A database schema $D$ and a sentence $\phi \in F$.

**Question:** For all states $db \in \Omega_D$, is $[\phi]^db = \top$?

### 3.3 Security Policies

Various Fine-Grained Access Control models for databases have been proposed in the literature [8, 41, 110, 150]. Although each has its own features, the models share common characteristics: (1) they support access control constraints at the row or column level, and (2) access control constraints are given by formulae, e.g., expressed in SQL [110], referring to the database’s current state or to the user’s credentials. We now describe a security policy model that captures the main features of existing fine-grained access control models.

Let $F$ be a query language. An $F$-constraint is a pair $\langle \{\overline{x} \mid \psi\}, \phi \rangle$, where $\{\overline{x} \mid \psi\}$ is a non-boolean $F$-query and $\phi$ is an $F$-formula such that $\text{free}(\phi) \subseteq \text{free}(\psi)$. A row-level $F$-constraint is just an $F$-constraint $\langle \{\overline{x} \mid \psi\}, \phi \rangle$ that specifies the conditions $\phi$ under which we are authorized to read a tuple in the result of the query $\{\overline{x} \mid \psi\}$. A column-level $F$-constraint is a triple $\langle \{\overline{x} \mid \psi\}, \phi, i \rangle$ that specifies the conditions $\phi$ under which we are authorized to read the $i$-th value of a tuple in the result of the query $\{\overline{x} \mid \psi\}$, where $\{\overline{x} \mid \psi\}$ is an $F$-constraint and $i \in \{1, \ldots, |\overline{x}|\}$. Note that given an $F$-constraint $\langle \{\overline{x} \mid \psi\}, \phi \rangle$, a free variable $z$ in $\phi$ refers to the $i$-th value of the tuple on which the constraint $\phi$ is evaluated, where $i = \text{pos}(\overline{x}, z)$. In the following, we consider only domain independent $F$-constraints, i.e., constraints $\langle \{\overline{x} \mid \psi\}, \phi \rangle$ such that $\{\overline{x} \mid \psi \land \phi\}$ is domain-independent. As a result, the access control decision does not depend on the underlying domain.

The constraints are expressed using formulae that can refer to the current state and to the values of the tuple being accessed. Note that the constraints can be defined over arbitrary non-boolean queries. We can therefore restrict access not only to the data in the relation schemas but also to derived information, such as tuples in views.

A security policy in our model is defined as follows.

**Definition 3.5.** Let $F$ be a query language and $D$ be a database schema. An $F$-security policy $S$ over $D$ is a pair $\langle \text{ROW}, \text{COL} \rangle$, where $\text{ROW}$ is a set of row-level $F$-constraints and $\text{COL}$ is a set of column-level $F$-constraints, such that for any non-boolean query $q$: (1) there is at most one constraint in $\text{ROW}$ associated with $q$, and (2) there is at most one constraint in $\text{COL}$ associated with the $i$-th value of the tuples in the result of $q$. 

<table>
<thead>
<tr>
<th>Language</th>
<th>$\text{FINSAT}^F$</th>
<th>$\text{FINVAL}^F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>BSRRC</td>
<td>Decidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>ERC</td>
<td>Decidable</td>
<td>Decidable</td>
</tr>
</tbody>
</table>

Table 3.2: Decidability of the $\text{FINSAT}^F$ and $\text{FINVAL}^F$ decision problems (taken from [37]).
Let $D$ be a database schema, $S = \langle \text{ROW}, \text{COL} \rangle$ be a security policy over $D$, $\{\tau \mid \psi\}$ be a non-boolean query, and $db \in \Omega_D$ be a state. We say that $S$ discloses the tuple $\tau \in [\{\tau \mid \psi\}]^{db}$, denoted by $\text{Disc}_S(\tau, \{\tau \mid \psi\}, db)$, iff there is a constraint $\langle \{\tau \mid \psi\}, \phi \rangle \in \text{ROW}$ such that $[\phi[\tau \rightarrow \tau]]^{db} = T$. Similarly, we say that $S$ discloses the $i$-th value of the tuple $\tau \in [\{\tau \mid \psi\}]^{db}$, denoted by $\text{Disc}_S(\tau, \{\tau \mid \psi\}, db, i)$, iff $\text{Disc}_S(\tau, \{\tau \mid \psi\}, db)$ and there is a constraint $\langle \{\tau \mid \psi\}, \phi, i \rangle \in \text{COL}$ such that $[\phi[\tau \rightarrow \tau]]^{db} = T$, where $i \in \{1, \ldots, |\tau|\}$. Note that the authorization to read a tuple does not imply the authorization to read any of its values. Given a constraint $\langle q, \phi \rangle \in \text{ROW}$, we denote by $\text{Auth}_{S,q}(db)$ the set of tuples in $[q]^{db}$ disclosed by $S$ in the state $db$, i.e., $\text{Auth}_{S,q}(db) := \{t \in [q]^{db} \mid \text{Disc}_S(t, q, db)\}$.

**Example 3.1.** Let $D$ be a database schema containing only one relation schema $\text{Employee}$ with arity 3. We assume that the first column in the schema stores employees’ names, the second column stores their ages, and the last one stores their jobs. We want to prevent the access to tuples representing secret agents. Although name is not protected, we want to disclose only the age of employees over 18, and only the job of the Clerks. We also want to disclose the set of the jobs in the database. Let $q$ be the query $\{x, y, z \mid \text{Employee}(x, y, z)\}$, and $r$ be the query $\{z \mid \exists x, y, \text{Employee}(x, y, z)\}$. The security policy $S = \langle \text{ROW}, \text{COL} \rangle$ is as follows: $\text{ROW} = \{\langle q, z \neq \text{SecretAgent}, (r, \top) \rangle\}$, and $\text{COL} = \{\langle q, (\top, 1)\rangle, \langle q \cup_{i \in \{1, \ldots, 17\}} y \neq i, 2\rangle, \langle q, z = \text{Clerk}, 3\rangle, (r, \top, \top)\}$. 

Note that a policy specifies which tuples and values we are authorized to read; it does not directly specify the result of a given query. The semantics of Security-Aware Query Processing will be introduced in the following sections.

We now introduce the concept of masked tuple from [165]. A masked tuple is a tuple where some values are replaced by the distinguished value $\dagger$. This value prevents the disclosure of data that may be sensitive. A tuple $\tau$ is subsumed by another tuple $\tau'$, written $\tau \subseteq \tau'$, iff $\forall i \in [\tau], \tau'(i) = \dagger$.

**Example 3.2.** Let $q$ be the query $\{x, y, z \mid \text{Employee}(x, y, z)\}$ from Example 3.1. The result of $q$ in the state $db_1$ in Figure 3.1 contains three tuples: $\mathcal{T} = \langle \text{John, 25, Clerk} \rangle$, $\mathcal{V} = \langle \text{Frank, 17, Admin} \rangle$, and $\mathcal{W} = \langle \text{Jack, 36, SecretAgent} \rangle$. The policy $S$ discloses $\mathcal{T}$ and $\mathcal{V}$ but not $\mathcal{W}$ because Jack is a secret agent. Moreover, the tuple $\mathcal{W}$ is only partially disclosed. Indeed, we are authorized to read Frank’s name, but not his age or his job. The policy $S$ also fully discloses the result of the query $r$, i.e., the set $\{(\text{Clerk}), (\text{Admin}), (\text{SecretAgent})\}$. Note that the tuple $\mathcal{T} = \langle \text{Frank}, \dagger, \dagger \rangle$ is subsumed by the tuple $\mathcal{W}$, i.e., $\mathcal{T} \subseteq \mathcal{W}$.

Note that our security policy model uses a set semantics derived from the relational calculus, whereas related approaches use a multi-set semantics derived from SQL. The models in [8,150] support only row-level constraints, those in [41,110] support only column-level constraints, and the model in [13] supports both. Our security policy model subsumes all these approaches as well as approaches where the security policy is expressed using views. Note that our security policies can represent, by combining column-level and row-level constraints, cell-level disclosure policies [13,110,165].

For simplicity, we ignore users and their credentials. This neither restricts nor limits our theoretical results. Users’ credentials can be modeled as a mapping $\text{cred}$ from the set $\mathcal{U}$ of users to the set $\mathcal{S}$ of security policies that assigns to each user $u \in \mathcal{U}$ a security policy $S \in \mathcal{S}$ where the users’ credentials are hard-coded in $S$ using constant values.
### 3.4. Security-Aware Query Processing

In this section, we introduce security-aware query processors, indistinguishable states, and correctness criteria for boolean queries in the Truman model.

**Definition 3.6.** Let $D$ be a database schema, $F$ be a query language, $U$ be the set of all possible results, and $S$ be the set of $F$-policies. An $F$-Security-Aware Query Processor ($F$-SAQP) is a function $\mathcal{M} : F \times S \times \Omega_D \to U$.

Note that the set $U$ depends on the type of query, i.e., boolean or non-boolean. Although boolean queries can be only true ($\top$) or false ($\bot$) in a given state, a security-aware query processor may also return the third value $\dagger$, i.e., $U = \{ \top, \bot, \dagger \}$. This value is used to prevent the leakage of sensitive information, which is in contrast to [165] where $\dagger$ is used only to mask tuples in the query’s result.

We now define indistinguishability, adapted from [165] to our setting.

**Definition 3.7.** Let $D$ be a database schema, $F$ be a query language, and $S = \langle ROW, COL \rangle$ be an $F$-policy over $D$. We say that two database states $db_1$ and $db_2$ in $\Omega_D$ are indistinguishable according to $S$, written $db_1 \cong_S db_2$, iff for all $(q, \phi) \in ROW$, there is a bijection $f$ from $\text{Auth}_{S,q}(db_1)$ to $\text{Auth}_{S,q}(db_2)$ such that for all $\bar{t} \in \text{Auth}_{S,q}(db_1)$ and all $i \in \{1, \ldots, |\bar{t}|\}$, (1) $\text{Disc}_{S}(\bar{t}, q, db_1, i)$ iff $\text{Disc}_{S}(\bar{t}, q, db_2, i)$, and (2) if $\text{Disc}_{S}(\bar{t}, q, db_1, i)$, then $\bar{t}(i) = f(\bar{t})(i)$. \hfill $\square$

**Example 3.3.** The database states $db_1$ and $db_2$ in Figure 3.1 are indistinguishable according to the policy $S$ given in Example 3.1. In contrast, $db_1 \not\cong_S db_2$ because the value of the attribute age for Frank is protected in state $db_1$, but not in state $db_3$, and $db_1 \not\cong_S db_4$ because the result of the query $r$ is different in the two states.

Note that all the states are indistinguishable according to the empty policy $S = \emptyset$. As stated in Proposition 3.1, given a database schema $D$ and a policy $S$, the indistinguishability relation $\cong_S$ is an equivalence relation over $\Omega_D$. Given a state $s \in \Omega_D$, the equivalence class of $s$ defined by $\cong_S$ is denoted by $[s]_{\cong_S}$.

**Proposition 3.1.** Let $D$ be a database schema and $S$ be a security policy over $D$. The indistinguishability relation $\cong_S$ is an equivalence relation over $\Omega_D$.

**Proof.** Let $D$ be a database schema and $S = \langle ROW, COL \rangle$ be a security policy over $D$. We now show that $\cong_S$ is reflexive, symmetric, and transitive.

**Reflexivity:** Let $db \in \Omega_D$. For each $((\tau, \psi), \phi) \in ROW$, the identity function $i : \text{Auth}_{S,(\tau,\psi)}(db) \to \text{Auth}_{S,(\tau,\psi)}(db)$ is a bijection on the set $\text{Auth}_{S,(\tau,\psi)}(db)$ to itself such that for all $\bar{t} \in \text{Auth}_{S,(\tau,\psi)}(db)$ and all $j \in \{1, \ldots, |\bar{t}|\}$, (1) $\text{Disc}_{S}(\bar{t}, (\tau, \psi), j)$ iff $\text{Disc}_{S}(i(\bar{t}), (\tau, \psi), j)$, and (2) if $\text{Disc}_{S}(\bar{t}, (\tau, \psi), j)$, then $\bar{t}(j) = i(\bar{t})(j)$. Therefore $db \cong_S db$.

**Symmetry:** Let $db_1, db_2 \in \Omega_D$ be two database states such that $db_1 \cong_S db_2$. From the indistinguishability definition, it follows that for each $((\tau, \psi), \phi) \in ROW$, there is a bijection $f_{(\tau,\psi)}$ from $\text{Auth}_{S,(\tau,\psi)}(db_1)$ to $\text{Auth}_{S,(\tau,\psi)}(db_2)$ such that for all $\bar{t} \in \text{Auth}_{S,(\tau,\psi)}(db_1)$ and all $i \in \{1, \ldots, |\bar{t}|\}$, (1) $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_1, i)$ iff $\text{Disc}_{S}(f_{(\tau,\psi)}(\bar{t}), (\tau, \psi), db_2, i)$, and (2) if $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_1, i)$, then $\bar{t}(i) = f_{(\tau,\psi)}(\bar{t})(i)$. Therefore, for each $((\tau, \psi), \phi) \in ROW$, there is a bijection $f_{(\tau,\psi)}$ from $\text{Auth}_{S,(\tau,\psi)}(db_1)$ to $\text{Auth}_{S,(\tau,\psi)}(db_2)$ such that for all $\bar{t} \in \text{Auth}_{S,(\tau,\psi)}(db_2)$ and all $i \in \{1, \ldots, |\bar{t}|\}$, (1) $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_2, i)$ iff $\text{Disc}_{S}(f_{(\tau,\psi)}(\bar{t}), (\tau, \psi), db_1, i)$, and (2) if $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_2, i)$, then $\bar{t}(i) = f_{(\tau,\psi)}(\bar{t})(i)$. Hence, $db_2 \cong_S db_1$.

**Transitivity:** Let $db_1, db_2, db_3 \in \Omega_D$ be three states such that $db_1 \cong_S db_2$ and $db_2 \cong_S db_3$. From $db_1 \cong_S db_2$, it follows that for each $((\tau, \psi), \phi) \in ROW$, there is a bijection $f_{(\tau,\psi)}$ from $\text{Auth}_{S,(\tau,\psi)}(db_1)$ to $\text{Auth}_{S,(\tau,\psi)}(db_2)$ such that for all $\bar{t} \in \text{Auth}_{S,(\tau,\psi)}(db_1)$ and all $i \in \{1, \ldots, |\bar{t}|\}$, (1) $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_1, i)$ iff $\text{Disc}_{S}(f_{(\tau,\psi)}(\bar{t}), (\tau, \psi), db_2, i)$, and (2) if $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_1, i)$, holds, then $\bar{t}(i) = f_{(\tau,\psi)}(\bar{t})(i)$. From $db_2 \cong_S db_3$, it follows that for each $((\tau, \psi), \phi) \in ROW$, there is a bijection $f_{(\tau,\psi)}$ from $\text{Auth}_{S,(\tau,\psi)}(db_2)$ to $\text{Auth}_{S,(\tau,\psi)}(db_3)$ such that for all $\bar{t} \in \text{Auth}_{S,(\tau,\psi)}(db_2)$ and all $i \in \{1, \ldots, |\bar{t}|\}$, (1) $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_2, i)$ iff $\text{Disc}_{S}(f_{(\tau,\psi)}(\bar{t}), (\tau, \psi), db_3, i)$, and (2) if $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_2, i)$, holds, then $\bar{t}(i) = f_{(\tau,\psi)}(\bar{t})(i)$. Therefore, for each $((\tau, \psi), \phi) \in ROW$, there is a bijection $f_{(\tau,\psi)} := f_{(\tau,\psi)} \circ f_{(\tau,\psi)}$ from $\text{Auth}_{S,(\tau,\psi)}(db_1)$ to $\text{Auth}_{S,(\tau,\psi)}(db_3)$ such that for all $\bar{t} \in \text{Auth}_{S,(\tau,\psi)}(db_1)$ and all $i \in \{1, \ldots, |\bar{t}|\}$, (1) $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_1, i)$ iff $\text{Disc}_{S}(f_{(\tau,\psi)} \circ f_{(\tau,\psi)}(\bar{t}), (\tau, \psi), db_3, i)$, and (2) if $\text{Disc}_{S}(\bar{t}, (\tau, \psi), db_1, i)$, holds, then $\bar{t}(i) = f_{(\tau,\psi)} \circ f_{(\tau,\psi)}(\bar{t})(i)$. Hence, $db_3 \cong_S db_1$. \hfill $\square$
**Definition 3.8.** Let $D$ be a database schema and $F$ be a query language. An $F$-security-aware query processor $M$ is **secure** iff for all $F$-policies $S$ over $D$, all $F$-queries $q$, and all $db, db' \in \Omega_D$, if $db \succeq_S db'$, then $M(q, S, db) = M(q, S, db')$.

A sound SAQP $M$ returns only correct information, i.e., information already returned by the original query. Note that $M(q, S, db)$ may return less information than $[q]^{db}$, but never more information. For boolean queries, $M(q, S, db)$ can return either $[q]^{db}$ or $\top$.

**Definition 3.9.** Let $D$ be a database schema and $F$ be a query language. An $F$-security-aware query processor $M$ is **sound** iff for all queries $q \in F$, all $F$-policies $S$ over $D$, and all $db \in \Omega_D$, $M(q, S, db) = [q]^{db}$ or $M(q, S, db) = \top$.

Finally, maximality formalizes that $M$ returns as much information as possible. For boolean queries, a maximal SAQP, given a query $q$, returns $[q]^{db}$ for all states $db$ where it is secure to do so.

**Definition 3.10.** Let $D$ be a database schema and $F$ be a query language. An $F$-security-aware query processor $M$ is **maximal** iff for all $F$-security policies $S$ over $D$, all $F$-queries $q$, and all $db \in \Omega_D$, if $[q]^{db} = [q]^{db'}$ for all $db' \in [db]_{\succeq_S}$, then $M(q, S, db) = [q]^{db}$.

We are now ready to define optimal algorithms for boolean queries in the Truman model.

**Definition 3.11.** Let $F$ be a query language. An $F$-SAQP for boolean queries is **optimal in the Truman model** iff it satisfies the Definitions 3.8–3.10.

Optimal algorithms are those algorithms that return as much information as possible without returning incorrect information or violating the security policy. Figure 3.2 describes an optimal SAQP for boolean queries. Observe that, depending on the query language, $M_{opt}$ may not be computable.

Studying optimal algorithms is important as they are the best one can do in terms of SAQP. In particular, considering only two out of three correctness criteria is usually not enough. Indeed, a function $M_1(q, S, db) = \top$ for all $q, S$, and $db$ is secure and sound but is completely useless. Similarly, a function $M_2(q, S, db) = [q]^{db}$ for all $q, S$, and $db$ is sound and maximal but leaks sensitive information. Finally, a function $M_3$ that satisfies the security and maximality criteria can systematically return arbitrary results without violating security and maximality. Indeed, let $E$ be one of the partitions of $\Omega_D$ defined by $\succeq_S$ such that $[q]^{db} \neq [q]^{db'}$ for some $db, db' \in E$, then $M_3$ can return an arbitrary value for all states in $E$.

### 3.5 Boolean Queries

In this section, we study the existence of optimal Security-Aware Query Processing algorithms for boolean queries.

#### 3.5.1 Preliminaries

We first define the query agreement decision problem. Afterwards, we show how this problem is related to optimal Security-Aware Query Processing.

**Definition 3.12.** Given a language $F$, $\mathit{AGREE}^F$ denotes the following decision problem:

**Input:** A database schema $D$, an $F$-security policy $S$ over $D$, a boolean query $q \in F$, and a state $db \in \Omega_D$.

**Question:** For all states $db' \in [db]_{\succeq_S}$, is $[q]^{db'} = [q]^{db}$? \hfill \□

Theorem 3.1 establishes that the decidability of $\mathit{AGREE}^F$ is related to the existence of optimal SAQP algorithms for the query language $F$.

**Theorem 3.1.** Let $F$ be a query language. There is a computable optimal $F$-SAQP algorithm $M$ for boolean queries iff $\mathit{AGREE}^F$ is decidable.

**Proof.** ($\Rightarrow$) Let $M$ be a computable optimal $F$-SAQP algorithm for boolean queries. We use $M$ as a subroutine in a decision procedure for $\mathit{AGREE}^F$. $\mathit{AGREE}^F$ takes as input a database schema $D$,
Lemma 3.1. Let \( \mathcal{M} \) be a database schema. \( \mathcal{M} \) takes as input an \( F \)-policy \( S \) over \( D \), a boolean \( F \)-query \( q \), and a state \( db \). If \( \mathcal{M}(q, S, db) = \top \), then \( \text{AGREE}^F(D, S, q, db) = \top \). Otherwise, \( \text{AGREE}^F(D, S, q, db) = \bot \).

(\( \Rightarrow \)) We use \( \text{AGREE}^F \) to build an optimal \( F \)-SAQP \( \mathcal{M} \). Let \( D \) be a database schema. \( \mathcal{M} \) takes as input an \( F \)-policy \( S \) over \( D \), a boolean \( F \)-query \( q \), and a state \( db \in \Omega_D \). If \( \text{AGREE}^F(D, S, q, db) = \top \), then \( \mathcal{M}(q, S, db) = [q]^{db} \), otherwise \( \mathcal{M}(q, S, db) = \top \). It easy to see that \( \mathcal{M} \) is optimal and computable.

We next study the decidability of \( \text{AGREE}^F \) for \( RC \) and \( ERC \), and we extend the results to the existence of optimal algorithms for these query languages. In our theorem statements and proofs, we will ignore sub-recursive complexity bounds and all reductions will be Turing reductions.

3.5.2 Impossibility Results

We now show that there are no optimal SAQP algorithms for boolean \( RC \) queries. Since all query languages used in practice, such as SQL, are Codd-complete \([10]\), i.e., they are at least as expressive as \( RC \), it follows from our impossibility result that there are no optimal SAQP algorithms for the boolean query languages currently in use.

Theorem 3.2. \( \text{AGREE}^{RC} \) is undecidable.

Proof. Let \( S \) be the empty policy \((\emptyset, \emptyset)\) and \( D \) be a database schema. Then, \( \equiv_S \) defines just one equivalence class containing every state. An \( RC \)-sentence \( \phi \) is finitely satisfiable iff \( \text{AGREE}^{RC}(D, S, \phi, db) = \top \) or \( [\phi]^{db} = \top \), where \( db \) is the empty state. The reduction can be implemented by a total Turing machine. Since \( \text{FINSAT}^{RC} \) is undecidable, so is \( \text{AGREE}^{RC} \).

We now show the reduction’s correctness. Assume that \( \phi \) is finitely satisfiable. If \( [\phi]^{db} = \top \), then \( \text{AGREE}^{RC}(D, S, \phi, db) = \top \) or \( [\phi]^{db} = \top \) trivially holds. Assume, then, that \( [\phi]^{db} = \bot \). From this and \( \phi \) is satisfiable, it follows that there exists a database state \( db' \in \Omega_D \) such that \( [\phi]^{db'} \neq [\phi]^{db} \).

Hence, \( \text{AGREE}^{RC}(D, S, \phi, db) = \bot \). For the other direction, assume that \( \text{AGREE}^{RC}(D, S, \phi, db) = \bot \) or \( [\phi]^{db} = \bot \). If \( [\phi]^{db} = \bot \), \( \phi \) is trivially finitely satisfiable. In contrast, if \( \text{AGREE}^{RC}(D, S, \phi, db) = \bot \), there is a database state \( db' \in \Omega_D \) such that \( [\phi]^{db'} \neq [\phi]^{db} \). Therefore, \( \phi \) is satisfiable in \( db \) or \( db' \).

From Theorem 3.2, it follows that there are no optimal SAQP algorithms for boolean \( RC \)-queries. Similarly to Theorem 3.2, we can prove an even stronger result: for any fragment \( F \) of \( RC \) such that \( \text{FINSAT}^F \) (or \( \text{FINVAL}^F \)) is undecidable, then \( \text{AGREE}^F \) is also undecidable. Therefore, from well-known undecidability results for fragments of \( RC \), one can identify fragments for which there are no optimal SAQP algorithms. Note also that considering functional dependencies and other integrity constraints in a particular fragment of \( RC \) might cause the undecidability of the \( \text{AGREE} \) problem, and therefore the impossibility of optimal SAQP.

3.5.3 Possibility Results

Although \( \text{AGREE}^{RC} \) is undecidable, there are fragments of \( RC \) for which the problem is decidable. In this section, we present sufficient conditions for the decidability of \( \text{AGREE}^F \), and we use these conditions to prove the decidability of \( \text{AGREE}^{ERC} \). Note that we do not provide a complete classification of the fragments of \( RC \) for which \( \text{AGREE}^F \) is decidable and there are fragments that meet neither the conditions for undecidability stated above nor the conditions of Lemma 3.1 below. Moreover, although we prove decidability, we do not derive complexity bounds for optimal SAQP algorithms.

We first introduce the notion of encoding the indistinguishability relation in a formula. Let \( D \) be a database schema, \( S \) be a security policy over \( D \), and \( db \) be a state in \( \Omega_D \). We say that a sentence \( \phi_{\text{INDIST}(S, db)} \) encodes the indistinguishability relation defined by the policy \( S \) on the state \( db \) if \( \phi_{\text{INDIST}(S, db)} \) is domain independent and for all \( db' \in \Omega_D \), \( [\phi_{\text{INDIST}(S, db)}]^{db'} = \top \) if \( db \equiv_S db' \).

We now present sufficient conditions for the decidability of the \( \text{AGREE}^F \) decision problem.

Lemma 3.1. Let \( F \) be a query language. \( \text{AGREE}^F \) is decidable if there is a query language \( F' \) such that:

1. \( \text{FINSAT}^F \) is decidable.
2. For any \( db \in \Omega_D \) and any \( F \)-policy \( S \), we can compute a sentence \( \phi_{\text{INDIST}(S, db)} \) that encodes the indistinguishability relation.
3. We can compute sentences \( \gamma, \gamma' \in F' \) equivalent to \( \phi_{\text{INDIST}(S, db)} \land \psi \) and \( \phi_{\text{INDIST}(S, db)} \land \lnot \psi \) respectively, for any \( db \in \Omega_D \), any \( F \)-policy \( S \), and any sentence \( \psi \in F \).
Proof. Let $F$ be a query language and $F'$ be another query language satisfying the conditions (1)–(3). We prove the theorem by reducing $\text{AGREE}^F$ to $\text{FINSAT}^{F'}$. The inputs to the $\text{AGREE}^F$ problem are: a database schema $D$, an $F$-security policy $S$, an $F$-sentence $\phi$, and a state $db \in \Omega_D$. From (2) and (3), we can compute two $F'$-formulae $\gamma_1(S, db, \psi)$ and $\gamma_2(S, db, \psi)$ that are respectively equivalent to $\phi_{\text{INDIST}(S, \psi)} \land \neg \psi$ and $\phi_{\text{INDIST}(S, \psi)} \land \psi$ for any state $db \in \Omega_D$, any $\psi \in F$, and any $F$-policy $S$ over $D$. The algorithm for $\text{AGREE}^F$ first computes $k = |\psi|_{db}$. Then, $\text{AGREE}^F(D, S, \psi, db) = \top$ iff $\text{FINSAT}^{F'}(\gamma_k(S, db, \phi)) = \bot$. This procedure can be implemented by a total Turing machine. Moreover, from (1) and (3), it follows that $\gamma_k(S, db, \phi) \in F'$ and $\text{FINSAT}^{F'}$ is decidable. Hence, $\text{AGREE}^F$ is decidable.

To show this reduction’s correctness, we prove both directions.

($\Rightarrow$). Assume that $\text{FINSAT}^{F'}(\gamma_k(S, db, \phi)) = \bot$. Without loss of generality, we assume that $|\psi|_{db} = \bot$. Therefore, $\gamma_k(S, db, \phi) = \phi_{\text{INDIST}(S, \psi)} \land \psi$. From this and $\text{FINSAT}^{F'}(\gamma_k(S, db, \phi)) = \bot$, it follows that there is no database state $db'$ such that $[\phi_{\text{INDIST}(S, \psi)} \land \psi]_{db'} = \top$. From this, it follows that for all database states $db'$ such that $[\phi_{\text{INDIST}(S, \psi)}]_{db'} = \top$, $[\psi]_{db'} = \bot$. From this and (2), it follows that for all database states $db'$ such that $[\phi_{\text{INDIST}(S, \psi)}]_{db'} = \top$, $[\psi]_{db'} = \bot$. Hence, $\text{AGREE}^F(D, S, \psi, db) = \top$.

($\Leftarrow$). Assume that $\text{AGREE}^F(D, S, \psi, db) = \top$. Without loss of generality, we assume $|\psi|_{db} = \top$. The proof for the other case is analogous. From $\text{AGREE}^F(D, S, \psi, db) = \top$ and $|\psi|_{db} = \top$, it follows that $[\psi]_{db'} = \top$ for all $db' \in [db]_{\text{INDIST}(S, \psi)}$. From this and (2), $[\psi]_{db'} = \top$ for all $db'$ such that $[\phi_{\text{INDIST}(S, \psi)}]_{db'} = \top$. From this, the formula $\gamma_\top(S, db, \psi) = \phi_{\text{INDIST}(S, \psi)} \land \neg \psi$ is unsatisfiable and, therefore, $\text{FINSAT}^{F'}(\gamma_k(S, db, \phi)) = \bot$.

There are fragments of $RC$ that are not expressive enough to encode the indistinguishability relation. For this reason, in Lemma 3.1, we introduced another fragment $F'$ that is more expressive and allows such an encoding. For instance, in the $ERC$ fragment there is no encoding of the indistinguishability relation, but it can be encoded in the more expressive $BSRRC$ fragment.

There is a similar set of preconditions for solving $\text{AGREE}^F$ using $\text{FINSAT}^{F'}$; we just need to consider the formula $\phi_{\text{INDIST}(S, \psi)} \rightarrow \psi$ and $\phi_{\text{INDIST}(S, \psi)} \rightarrow \neg \psi$ instead of the formula $\phi_{\text{INDIST}(S, \psi)} \land \neg \psi$. Before proving the decidability of $\text{AGREE}^{\text{ERC}}$, we introduce some notation. Given a policy $S = (\text{ROW}, \text{COL})$, a state $db$, a constraint $(q, \phi) \in \text{ROW}$, and a tuple $t \in \text{Auth}_{S,q}(db)$, we denote by $\text{mask}_{S,q,t}(\overline{t})$ the masked tuple $\overline{t}$ from $\overline{t}$ by replacing all the undistinguished values with $\ddagger$, i.e., $[t] = [\ddagger]$ and for all $i \in \{1, \ldots, |\overline{t}|\}$, if $\text{Discs}(t, q, db, i)$, then $\tau(i) = \ddagger$; otherwise $\tau(i) = t(i)$.

Let $D$ be a database schema, $S = (\text{ROW}, \text{COL})$ be a security policy over $D$, $(q, \phi)$ be a constraint in $\text{ROW}$, and $db$ be a database state in $\Omega_D$. The set $\text{Ind}_{S,q}(db)$ contains all the masked tuples $\overline{t}$ that can be obtained from tuples in $\text{Auth}_{S,q}(db)$ using the function $\text{mask}_{S,q,t}(\overline{t})$, i.e., $\text{Ind}_{S,q}(db) := \{\text{mask}_{S,q,t}(\overline{t}) \mid \overline{t} \in \text{Auth}_{S,q}(db)\}$. Given a tuple $\overline{t} \in \text{Ind}_{S,q}(db)$, we denote by $\text{card}_{S,q,t}(\overline{t})$ the number of tuples $\overline{t} \in \text{Auth}_{S,q}(db)$ that cannot be distinguished from $\overline{t}$ according to the security policy $S$, i.e., $\text{card}_{S,q,t}(\overline{t}) := |\{\overline{t}' \in \text{Auth}_{S,q}(db) \mid \text{mask}_{S,q,t}(\overline{t}') = \overline{t}\}|$.

Let $S = (\text{ROW}, \text{COL})$ be a security policy, $i \in \mathbb{N}$, and $q = \{q\} \phi$ be a non-boolea query. We denote by $\psi_i^q$ the condition in the constraint associated with $q$, i.e., $(q, \psi_i^q) \in \text{ROW}$. Similarly, we denote by $\psi_i^q$, the condition associated with the i-th value of $q$, i.e., $(q, \psi_i^q) \in \text{COL}$. If there is no such a constraint, then $\psi_i^q = \bot$ (respectively $\psi_i^q = \bot$).

We now use Lemma 3.1 to prove that $\text{AGREE}^{\text{ERC}}$ is decidable. Due to the limited expressiveness of the $\text{ERC}$ fragment, we cannot encode the indistinguishability relation in it. However, we can prove the decidability of $\text{AGREE}^{\text{ERC}}$ using the more expressive $BSRRC$ fragment. The full proof of this result is given in Appendix A.

Theorem 3.3. $\text{AGREE}^{\text{ERC}}$ is decidable.

Proof Sketch. Let $S = (\text{ROW}, \text{COL})$ be an $\text{ERC}$-policy, $db$ be a state, and $q = \{q\} \phi$ be a non-boolea query such that there is a constraint for $q$ in $S$. The formula $\theta_{\{q\}, S}(\overline{y}, \overline{t})$, where $\overline{y}$ is a tuple of variables and $\overline{t}$ is a possibly masked tuple of values in $\text{dom}$ such that $[\overline{y}] = [\overline{t}]$, is as follows:

$$\theta_{\{q\}, S}(\overline{y}, \overline{t}) := \bigwedge_{i \in \{1, \ldots, |\overline{t}|\} \land \tau(i) \neq \ddagger} (\psi_i^q_{\overline{t}} \rightarrow [\overline{y}] \land \overline{y}(i) = \tau(i)) \land \bigwedge_{i \in \{1, \ldots, |\overline{t}|\} \land \tau(i) = \ddagger} \neg \psi_i^q_{\overline{t}} \rightarrow [\overline{y}].$$

The formula $\theta_{\{q\}, S}(\overline{y}, \overline{t})$ forces the masked version of the tuple associated with the values of the variables $\overline{y}$ to be $\overline{t}$.

In our encoding $\phi_{\text{INDIST}(S, \psi)}$, we use counting quantifiers. A counting quantifier $\exists^{\text{opm}}_{\overline{y}} \phi(\overline{x})$, where $\overline{x} \in \{=, \leq, \geq, <, >\}$ and $m \in \mathbb{N}$, is a quantifier that specifies the number of tuples that can be
mapped to $\pi$ and satisfy $\phi(\pi)$. For instance, the formula $\exists x_1 \phi(x_1)$ is true iff there are exactly two distinct elements $x_1$ satisfying $\phi(x_1)$. Note that counting quantifiers do not add expressive power to $RC$.

Given a tuple of variables $\pi$, we denote by $\nu_\pi$ the tuple of variables $y_1, \ldots, y_{|\pi|}$. The encoding $\phi^{INDIST(S, db)}_t$ is:

$$\psi_S,\pi, db := \exists^{\exists \text{Auth}_S, \pi, (db)} \nu_\pi \left( \psi[\pi \mapsto \nu_\pi] \land \psi^S_{\pi, db} \right)$$

$$\gamma_S,\pi, db, t := \exists^{\text{card}_S db, t} \nu_\pi \left( \psi[\pi \mapsto \nu_\pi] \land \phi^{INDIST(S, db)}_t \right)$$

$$\phi^{INDIST(S, db)}_t := \bigwedge_{\langle q, \phi \rangle \in \text{ROW}} \left( \psi_{S, q, db} \land \bigwedge_{\gamma \in \text{Ind}_{S, db, t}} \gamma_{S, q, db, t} \right).$$

The sentence $\psi_S,\pi, db$ states that, for any state $db'$ such that $[\psi_S,\pi, db][db']$ is $\top$, the sets $\text{Auth}_S,\pi, db'$ and $\text{Auth}_S,\pi, db$ have the same cardinality. The sentence $\gamma_S,\pi, db, t$ states that, for any state $db'$ such that $[\gamma_S,\pi, db, t][db']$ is $\top$, there are at least $\text{card}_S db, t$ tuples $\pi$ in $\text{Auth}_S,\pi, db'$ such that $\text{mask}_{S, q, \pi}[\pi]$ is $\top$. The sentence $\phi^{INDIST(S, db)}_t$ encodes the indistinguishability relation for the ERC-policy $S$ and the state $db$.

Although $\phi^{INDIST(S, db)}_t$ is not in $BSRRC$, it can be rewritten as a $BSRRC$-sentence. Furthermore, there are $BSRRC$-sentences equivalent to $\phi^{INDIST(S, db)}_t \land \psi$ and $\phi^{INDIST(S, db)}_t \land \lnot \psi$ for any ERC-sentence $\psi$ and any ERC-policy $S$. Therefore, $\text{AGREE}^{CRC}$ is decidable by Lemma 3.1.

From Theorem 3.3 and the fact that conjunctive queries ($CRC$) are a strict subset of ERC, it also follows that $\text{AGREE}^{CRC}$ is decidable. Therefore, there are optimal SAQP algorithms for boolean conjunctive queries.

### 3.5.4 Truman and Non-Truman models

In this section, we study the connections between Truman and Non-Truman models as defined in [131]. In the Non-Truman model, the security policy is expressed using a set of authorization views. Authorization views are standard database views extended with parameters referring to users’ credentials. They can be used to specify the data a user is authorized to read. Authorization views in [131] are expressed in SQL, whereas here we study authorization views in the relational calculus. In the following, we ignore users’ credentials, which does not limit our results.

**Definition 3.13.** Let $D$ be a database schema and $F$ be a query language. An $F$-authorization view is defined by assigning a relation identifier $V$ not in $D$ to a non-boolean $F$-query $\{ \pi \mid \phi(\pi) \}$, i.e., $V = \{ \pi \mid \phi(\pi) \}$.

Let $s \in \Omega_D$ be a state and $V = \{ \pi \mid \phi(\pi) \}$ be a view. The materialization of $V$ in $db$, denoted by $db(V)$, is $[\{ \pi \mid \phi(\pi) \}[db]]^{db}$. Views can be used in $RC$-formulae in the same way as relation schemas, but in this case we consider the materialized views instead of the relation instances.

We now introduce the notion of equivalence with respect to a set of authorization views $AV$ ($AV$-equivalence) defined in [131]. Two states are $AV$-equivalent iff their views’ materializations are the same.

**Definition 3.14.** Let $D$ be a database schema, $AV$ be a set of authorization views over $D$, and $db, db' \in \Omega_D$ be two states. Then, $db$ and $db'$ are $AV$-equivalent, written $db \equiv_{AV} db'$, iff for all $V \in AV$, $db(V) = db'(V)$.

For any database schema $D$ and any set of views $AV$, the relation $\equiv_{AV}$ is an equivalence relation over $\Omega_D$.

We now introduce row-level policies. Intuitively, a row-level policy does not disclose partial tuples.

**Definition 3.15.** Let $D$ be a database schema and $S = \{ROW, COL\}$ be a security policy over $D$. $S$ is a row-level policy iff for all $(q, \phi) \in ROW$, all $db \in \Omega_D$, and all $\tau \in [q]^{db}$, if $\text{Disc}_S(\tau, q, db)$, then $\text{Disc}_S(\tau, q, db)$ for all $i \in \{1, \ldots, |\tau|\}$.

We say that a security policy $S$ and a set of views $AV$ are equivalent iff $\equiv_{AV} \equiv S$. We prove below that $RC$-authorization views are as expressive as row-level $RC$-policies. It is easy to see that row-level $RC$-policies are strictly less expressive than $RC$-policies, and therefore $RC$-authorization views are strictly less expressive than $RC$-policies.

**Proposition 3.2.** Let $D$ be a database schema. For each set of $RC$-authorization views over $D$, there is an equivalent row-level $RC$-security policy over $D$ and vice versa.
Proof. ($\Rightarrow$). Let $AV$ be a set of $RC$-authorization views. We now show how to construct the row-level security policy $S$. The security policy $S = (ROW, COL)$ is as follows:

- For each view $V = \{\tau | \phi\} \in AV$, we define a constraint $\{|\tau | \phi\}, \top \in ROW$.
- For each view $V = \{\tau | \phi\} \in AV$ and each $i \in \{1, \ldots, |\tau|\}$, we define a constraint $\{|\tau | \phi\}, \top, i \in ROW$.

It is easy to see that $S$ is equivalent to $AV$, i.e., any two states $AV$-equivalent are also indistinguishable according to $S$ and vice versa. Indeed, the data disclosed by $S$ are exactly the materialization of the views in $AV$. It is also easy to check that $S$ is row-level by construction. Finally, observe that if $AV$ is a set of $F$-authorization views for some query language $F$, then $S$ is an $F$-security policy.

($\Leftarrow$). Let $S = (ROW, COL)$ be a row-level $RC$-security policy. We can define a set of authorization views $AV$ as follows. For each $\{|\tau | \psi\}, \phi \in ROW$, we define the authorization view $V_1(\tau, \psi)$ as follows:

$$V_1(\tau, \psi) = \{\tau | \psi \land \phi\}.$$ 

Given a state $db$ and a constraint $\langle q, \phi \rangle \in ROW$, the set $Auth_{S,q}(db)$ is exactly the materialization of the view $V_1$ in the state $db$. Note also that given a row-level policy $S$, then $Auth_{S,q}(db) = Ind_{S,q}(db)$ for all queries $q$ and all states $db$. We now prove that $S$ is equivalent to $AV$. Let $db_1$ and $db_2$ be two indistinguishable states according to $S$. Since the views in $AV$ disclose only values that we are authorized to read according to $S$, then $db_1$ and $db_2$ are $AV$-equivalent. Similarly, let $db_1$ and $db_2$ be two equivalent states. Given a constraint $\{|\tau | \psi\}, \phi \in ROW$ and a state $db$, then the set $Auth_{S,\tau,\psi}(q)$ is exactly the materialization of the view $V_1(\tau, \psi)$ in the state $db$. Therefore, since $db_1$ and $db_2$ are $AV$-equivalent, i.e., the materialization of the views in $AV$ are the same, then they are also indistinguishable. Note that if $F$ is closed under conjunction, then given any $F$-policy, the resulting equivalent set of authorization views is in $F$.

Example 3.4. Let $AV$ be the set of views $\{q_1, q_2\}$, where $q_1$ is the query $\{z | \exists x, y. \text{Employee}(x, y, z)\}$ and $q_2$ is the query $\{y | \exists x, z. \text{Employee}(x, y, z)\}$. The equivalent row-level policy is $S = \{\{q_1, \top\}, \{q_2, \top\}\}$. Let $S$ be the row-level policy $\{\{\{q_1, \phi\}, \{q_2, \psi\}\}, \{\{q_1, \top, 1\}, \{q_2, \top, 1\}\}\}$. The equivalent view of views $\{q_1, q_2\}$ is $\{q_1, q_2\}$, where $q_1 = \{z | \exists x, y. \text{Employee}(x, y, z) \land \phi\}$ and $q_2 = \{y | \exists x, z. \text{Employee}(x, y, z) \land \psi\}$. From Proposition 3.2, it follows that, given a set of authorization views $AV$, two states are $AV$-equivalent iff they are indistinguishable according to the equivalent security policy and vice versa.

In the relational calculus, authorization views are strictly less expressive than security policies. This is no longer the case for sufficiently powerful query languages. For instance, in the relational calculus extended with the count aggregation operator, authorization views are as expressive as security policies, as shown in Example 3.5. This also implies that, in contrast to the case of $RC$, for sufficiently powerful languages, row-level policies are as expressive as policies that combine both row-level and column-level constraints.

Example 3.5. In this example, we use the aggregation operator count [10]. Informally, $\text{count}(\tau | \psi(\tau))$ returns the number of tuples in the result of the query $\{\tau | \psi(\tau)\}$.

Let $q$ be the query $\{x, z | \alpha\}$, where $\alpha$ is the $RC$-formula $\exists y. \text{Employee}(x, y, z)$, and let $S$ be the policy $\{\{q, \phi\}, \{q, \psi, 1\}, \{q, \gamma, 2\}\}$, where $\phi$, $\psi$, and $\gamma$ are $RC$-formulas. The set of authorization views $AV$ that is equivalent to $S$ is:

$$\{x, z, s | \alpha \land \phi \land \psi \land \gamma \land s = 1\},$$
$$\{x, s | \exists z. (\alpha \land \phi \land \psi \land \neg \gamma) \land s = \text{count}(z | \alpha \land \phi \land \psi \land \neg \gamma)\},$$
$$\{z, s | \exists x. (\alpha \land \phi \land \gamma \land \neg \psi) \land s = \text{count}(x | \alpha \land \phi \land \gamma \land \neg \psi)\},$$
$$\{s | \exists x, z. (\alpha \land \phi \land \neg \psi \land \neg \gamma) \land s = \text{count}(x, z | \alpha \land \phi \land \neg \psi \land \neg \gamma)\}.$$ 

Let $db$ be a state. The materialization of the views in $AV$ in the state $db$ discloses, for each $\tau \in Ind_{S,q}(db)$, the values in $\tau$ different from $\top$ and the value of $\text{count}_{S,a}(\tau)$, which is stored in the variable $s$. Therefore, $\Xi_{AV} = \Xi_S$.

The intuition behind Example 3.5 is formalized in Proposition 3.3. Note that we denote by $RC^+$ the relational calculus extended with the count aggregation operator [10].

Proposition 3.3. Let $D$ be a database schema. For each set of $RC^+$-authorization views over $D$, there is an equivalent $RC^+$-security policy over $D$, and vice versa.

Proof. ($\Rightarrow$). The proof in this direction is exactly the same as the proof (in the same direction) of Proposition 3.2.
3.5. Boolean Queries

(⇐). Let $S = (ROW, COL)$ be a $RC^*$-security policy. We define a set of $RC^*$-authorization views $AV$ as follows. We denote by $Q_S$ the set containing all the non-boolean queries on which there are constraints in $ROW$. Given a tuple of variables $\pi$ and a finite set of variables $A$ such that all $y \in A$ occur in $\pi$, we denote by $\text{var}_\pi(A)$ the tuple of variables $\pi$ that contains exactly the variables in $A$ ordered according to their position in $\pi$. We also denote by $\text{rem}_\pi(A)$ the result of $\text{var}_\pi(\pi \setminus A)$.

For each $q = \{\pi | \psi\}$ in $Q_S$, and each set $A \in 2^{\text{free}(\psi)}$, we define the authorization view $V_{\pi,\psi}(A)$:

$$V_{\pi,\psi}(A) := \{\text{rem}_\pi(A), s \mid \exists \text{rem}_\pi(A), (\psi \land \phi^S_s \land \bigwedge_{y \in A} \psi^S_{q,pos(\pi,y)} \land \bigwedge_{y \in \text{free}(\psi) \setminus A} \neg \psi^S_{q,pos(\pi,y)} \land \neg \psi^S_{q,pos(\pi,y)} \land s = \text{count}[\text{rem}_\pi(A) | \psi \land \phi^S_s \land \bigwedge_{y \in A} \psi^S_{q,pos(\pi,y)} \land \bigwedge_{y \in \text{free}(\psi) \setminus A} \neg \psi^S_{q,pos(\pi,y)}]\}.$$ 

Note that if $|\text{rem}_\pi(A)| = 0$, then we replace the count operator with 1, i.e., $s = 1$ in this case. Note that we assume, without loss of generality, that $s \not\in \text{free}(\psi)$.

Given a constraint $(\langle \pi | \psi \rangle, \phi) \in ROW$, a state $db$, and a set of variables $A \subseteq \text{free}(\psi)$, the materialization of the view $V_{\pi,\psi}(A)$ in the state $db$ contains a tuple $\tau \circ \langle m \rangle$, where $\circ$ indicates the concatenation between two tuples and $m \in \mathbb{N}$, if there is a tuple $\tau \in \text{Ind}_{S,q}(db)$ such that $\text{card}_{\text{Ind}_{S,q}(db)} = m$ and for all $y \in \text{free}(\psi)$, if $y \in A$, then $\text{Ind}(\text{var}(\phi(A), y)) = \text{Ind}(\text{var}(\phi(y)))$ and otherwise $\text{Ind}(\text{var}(\phi(y))) = \tau^\dagger$. From this, it follows that $S$ is equivalent to $AV$. Indeed, let $db_1$ and $db_2$ be two indistinguishable states according to $S$. For all $(q, \phi) \in ROW$, $\text{Ind}_{S,q}(db_1) = \text{Ind}_{S,q}(db_2)$ and $\text{card}_{\text{Ind}_{S,q}(db)} = \text{card}_{\text{Ind}_{S,q}(db)}$ for all $\tau \in \text{Ind}_{S,q}(db_1)$. Therefore, the materializations of the views in $AV$ are the same in both states, and thus $db_1 \equiv_{AV} db_2$. Similarly, let $db_1$ and $db_2$ be two $AV$-equivalent states. Then, for all $(q, \phi) \in ROW$, we have that $\text{Ind}_{S,q}(db_1) = \text{Ind}_{S,q}(db_2)$ and also that $\text{card}_{\text{Ind}_{S,q}(db)} = \text{card}_{\text{Ind}_{S,q}(db)}$ for all $\tau \in \text{Ind}_{S,q}(db_1)$. Therefore, $db_1 \equiv_{AV} db_2$. 

We now introduce the concept of conditional validity [131], which is different from validity in FOL and $RC$. Afterwards, we define optimal algorithms in the Non-Truman model.

**Definition 3.16.** Let $D$ be a database schema, $db \in \Omega_D$ be a state, and $AV$ be a set of authorization views over $D$. An $RC$-query $q$ is conditionally valid in $db$ if there is another $RC$-query $q'$ written only in terms of the views in $AV$ that is equivalent to $q$ on all states $db'$ such that $db \equiv_{AV} db'$.

**Definition 3.17.** Let $F$ be a query language. An $F$-SAQP $M$ is optimal in the Non-Truman model iff it executes exactly the conditionally valid queries.

Before reconciling Truman and Non-Truman models, we state the following lemma.

**Lemma 3.2.** Let $D$ be a database schema, $F$ be a query language, $AV$ be a set of $F$-authorization views, $db \in \Omega_D$ be a state, and $\phi$ be a boolean $F$-query. Moreover, let $S$ be the $F$-security policy equivalent to $AV$. The query $\phi$ is conditionally valid in the state $db$ with respect to $AV$ iff $AGREE^F(D, S, \phi, db) = \top$.

**Proof.** ($\Rightarrow$). Let $\phi$ be a query that is conditionally valid in the state $db$ with respect to $AV$. Since $\phi$ is conditionally valid, then there is another query $\phi'$ expressed only in terms of $AV$ such that $[\phi']^{db} = [\phi]^{db}$ for all states $db'$ $AV$-equivalent to $db$. Note that two states are $AV$-equivalent if they have the same views’ materializations. Since $\phi'$ is expressed only in terms of $AV$, it is domain independent, the query language is deterministic, and the views’ materializations are fixed, then $\phi'$ returns always the same result $k \in \{\top, \bot\}$ on all states $db'$ $AV$-equivalent to $db$. Therefore, $[\phi]^{db'} = [\phi']^{db'} = k$ for all states $db'$ such that $db \equiv_{AV} db'$. Since $AV$ and $S$ are equivalent, it follows that $db \equiv_{AV} db'$ if $db \equiv_{S} db'$. It follows that $[\phi]^{db} = k$ for all states $db'$ such that $db \equiv_{AV} db'$, and therefore $AGREE^F(D, S, \phi, db) = \top$.

($\Leftarrow$). Let $\phi$ be a query such that $AGREE^F(D, S, \phi, db) = \top$. Moreover, let $k \in \{\top, \bot\}$ be the value $[\phi]^{db}$. Let $S$ be the $F$-security policy equivalent to $AV$. Since $AGREE^F(D, S, \phi, db) = \top$, we know that the result of $\phi$ is $k$ for all states indistinguishable from $db$ according to $S$. Since $S$ and $AV$ are equivalent, it follows that the result of $\phi$ is $k$ for all states $AV$-equivalent to $db$. Therefore, the query $\phi' = k$ is equivalent to $\phi$ for all states $AV$-equivalent to $db$. Thus, $\phi$ is conditionally valid in $db$ according to $AV$.

Finally, we can connect optimal SAQP in the Truman and Non-Truman model.

**Theorem 3.4.** Let $F$ be a query language. An optimal $F$-SAQP for boolean queries in the $\hat{\Delta}$-model is an optimal $F$-SAQP for boolean queries in the Non-Truman model when $\hat{\Delta}$ is interpreted as rejecting the query.

**Proof.** The theorem follows from Theorem 3.1, Proposition 3.2, Lemma 3.2, and the definition of optimal SAQP in the Non-Truman model.
In the past, Truman and Non-Truman models have been presented as two distinct and alternative approaches: the former concerned with returning as much information as possible and the latter concerned with avoiding inconsistencies. For boolean RC-queries, Theorem 3.4 shows that an optimal SAQP in the Truman model can be used as an optimal SAQP in the Non-Truman model. The reason is that, for boolean queries, the only way to protect sensitive information is to return \( \dagger \) and there is no way to return partial results. Therefore, for boolean RC-queries, the Non-Truman model is a special case of the Truman model. Indeed, RC-authorization views are strictly less expressive than RC-policies. For sufficiently powerful query languages, such as the relational calculus extended with the \texttt{count} operator, the two models coincide for boolean queries because authorization views are as expressive as security policies.

The practical consequence of Theorem 3.4 is that we can straightforwardly use an optimal algorithm in the Truman model to compute the answer to a query in the Non-Truman model. It also follows that the results in Sections 3.5.2 and 3.5.3 apply to the Non-Truman model and to the conditional validity problem.

### 3.6 Non-boolean queries

In this section, we study the existence of optimal SAQP algorithms for non-boolean queries and the connections between optimal SAQP algorithms in the Truman and Non-Truman models.

#### 3.6.1 Correctness Criteria

The answer to a non-boolean query under a given security policy is a multi-set of masked tuples and not a set of tuples as is standard in database theory. Indeed, some values may be set to \( \dagger \) because we are not authorized to read them. However, not all multi-sets of tuples are valid results for SAQP algorithms. We are interested only in those multi-sets that can be obtained from a set of unmasked tuples by replacing values with \( \dagger \). The set \( M \) is the set of all multi-sets of tuples \( V \) such that there is a set of unmasked tuples \( T \) and a bijection \( f \) from \( V \) to \( T \) such that \( T \subseteq f(T) \) for all \( T \in V \). In the following, we always refer just to multi-sets in \( M \).

Following [165], we define a subsumption relation \( \preceq \) on multi-sets, which is a partial order on \( M \). A multi-set \( K \in M \) is subsumed by another multi-set \( K' \in M \), written \( K \preceq K' \), iff there is an injective mapping \( f : K \rightarrow K' \) such that for all \( T \in K \), \( f(T) \subseteq T \).

**Example 3.6.** Let \( \bar{t} \) and \( \bar{z} \) be the tuples \( (John, 25, Clerk) \) and \( (Frank, \dagger, \dagger) \) respectively. The multi-set \( \{\bar{t}, \bar{z}, \bar{z}\} \) is not in \( M \) because there are two occurrences of the unmasked tuple \( \bar{t} \). In contrast, the multi-set \( J = \{\bar{z}, \bar{z}, \bar{t}\} \) is in \( M \). Let \( T \) be the set \( \{\langle Frank, 46, Clerk \rangle, \langle Frank, 27, SecretAgent \rangle, \langle John, 25, Clerk \rangle\} \). Then, \( J \preceq T \).

We now introduce the correctness criteria for non-boolean queries. Security is the same as in Definition 3.8 and we state here only soundness and maximality. Wang et al. [165] studied only secure algorithms and considered only algorithms that return a unique multi-set of masked tuples as a query’s result. However, there are cases in which the optimal answer is not unique, i.e., there are finitely many different multi-sets of masked tuples that are all optimal and incomparable. In the following, we consider algorithms that might return as a query’s result a set of multi-sets. Therefore, for non-boolean queries, the set \( U \), which contains all possible results of optimal SAQPs, is \( 2^M \). For this reason, we must modify the criteria given in [165].

For non-boolean queries, a sound algorithm must return a result subsumed by the original query’s result.

**Definition 3.18.** Let \( D \) be a database schema and \( F \) be a query language. An \( F \)-security-aware query processor \( \mathcal{M} \) is sound iff for all non-boolean queries \( q \in F \), all \( F \)-policies \( S \) over \( D \), all \( db \in \Omega_{F} \), and all \( V \in \mathcal{M}(q, S, db) \), \( V \preceq [q]^{db} \).

A maximal algorithm must return a result that subsumes any multi-set \( T \in M \) that is subsumed by the original query’s result in all indistinguishable states.

**Definition 3.19.** Let \( D \) be a database schema and \( F \) be a query language. An \( F \)-security-aware query processor \( \mathcal{M} \) is maximal iff for all \( F \)-policies \( S \) over \( D \), all non-boolean queries \( q \in F \), all \( db \in \Omega_{F} \), and all \( T \in M \), if \( T \preceq [q]^{db} \) for all \( db \in [db]_{\sim S} \), then there is a \( V \in \mathcal{M}(q, S, db) \) such that \( T \preceq V \).

We now define optimal algorithms for non-boolean queries in the Truman model.

**Definition 3.20.** Let \( F \) be a query language. An \( F \)-SAQP for non-boolean queries is optimal in the Truman model iff it satisfies the Definitions 3.8, 3.18, and 3.19.
Theorem 3.5. Let $D$ be a database schema with only one relation schema $R$ of arity 2. For simplicity, we call $a$ and $b$ the two attributes in $R$, and we assume that $\text{dom}$ is $\mathbb{N}$. Moreover, let $\phi(n,m)$ be the formula $x = n \land y = m \land \forall y. (R(n,y) \rightarrow y = m) \land \forall x. (R(x,m) \rightarrow x = n)$. Let $q$ be the query \{ $x,y \mid R(x,y)$ \}$ \land S$ be the security policy \{(q,T),(q,\psi,1),(q,\psi,2)\}, \}$ where $\psi$ is the formula $\neg \phi(1,3) \land \neg \phi(1,4) \land \neg \phi(2,3) \land \neg \phi(2,4)$. One of the equivalence classes defined by $S$ contains exactly the two states $db$ and $db'$ such that $db(R) = \{(1,3), (2,4)\}$ and $db'(R) = \{(1,4), (2,3)\}$. In this case, $M_{opt}(q,S,db) = M_{opt}(q,S,db') = K$ and $K = \{V \in M \mid V \not\subseteq \{(1,1),(2,1)\} \lor V \not\subseteq \{(1,3),(1,4)\}\}$. Therefore, in $db$ and $db'$, we are authorized to read the values of the attributes $a$ and $b$ separately, but not together. \hfill $\blacksquare$

In the Truman model, boolean queries are not a special case of non-boolean queries. For fragments that are not closed under negation, we cannot use an optimal SAQP $M$ for non-boolean queries to distinguish whether a sentence returns $\bot$ or $\top$ because $M$ returns $\{\emptyset\}$ in both cases.

We now define a decision problem, denoted as $\text{SUBSUME}^F$, that will be used in a way similar to $\text{AGREE}^F$.

Definition 3.21. Given a query language $F$, $\text{SUBSUME}^F$ denotes the problem:

**Input:** A database schema $D$, an $F$-security policy $S$, a non-boolean $F$-query $q = \{\pi \mid \phi(\pi)\}$, a multi-set $T \in M$ such that $[T] = [\pi]$ for all $\pi \in T$, and a state $db \in \Omega_D$.

**Question:** For all states $db' \in [db]_{\pi,S}$, is $T \subseteq [q]^{db'}$?

The $\text{SUBSUME}^F$ decision problem is the existence of optimal SAQP algorithms.

Theorem 3.5. Let $F$ be a query language. There is a computable optimal $F$-SAQP algorithm $M$ for non-boolean queries iff $\text{SUBSUME}^F$ is decidable.

**Proof.** ($\Rightarrow$). Let $M$ be a computable optimal $F$-SAQP algorithm for non-boolean queries. We use $M$ as a subroutine in a decision procedure for $\text{SUBSUME}^F$. Let $D$, $S$, $q$, $T$, and $db$ be the inputs of the $\text{SUBSUME}^F$ problem. Then, $\text{SUBSUME}^F(D,S,q,T,db) = \top$ iff there is a $V \in M(q,S,db)$ such that $T \subseteq V$.

($\Leftarrow$). We use $\text{SUBSUME}^F$ to build an optimal $F$-SAQP $M$. Let $D$ be a database schema, $S$ be an $F$-security policy over $D$, $db \in \Omega_D$ be a state, and $q$ be a non-boolean $F$-query. $M(q,S,db)$ is $\{T \in M \mid T \subseteq [q]^{db} \land \text{SUBSUME}^F(D,S,q,T,db) = \top\}$. It is easy to see that $M$ is optimal and computable. \hfill $\blacksquare$

The $\text{SUBSUME}^F$ decision problem plays the same role for optimal algorithms for non-boolean queries as $\text{AGREE}^F$ does for boolean queries. We now analyze the decidability of $\text{SUBSUME}^F$.

### 3.6.2 Impossibility Results

In this section, we extend the impossibility results in Section 3.5.2 from boolean $RC$-queries to non-boolean $RC$-queries.

**Theorem 3.6.** $\text{SUBSUME}^{RC}$ is undecidable.

**Proof.** Let $D$ be a database schema. We define a new schema $D'$ obtained from $D$ by adding a new relation schema $T$ with arity 1. Let $v$ be a value in $\text{dom}$, $db$ be the state in $\Omega_D$, such that $db(T) = \{v\}$ and $db(R) = \emptyset$ for all relation schemas $R$ in $D$, and $S$ be the $RC$-policy $\{\{\{x \mid T(x)\}, T\} \cup \{\{x \mid T(x)\}, T\}, 1\}$. Then, an $RC$-sentence $\phi$ over $D$ is finitely valid iff $\text{SUBSUME}^{RC}(D',S',\{x \mid T(x)\} \land \phi), \{v\}, db = \top$. To show the correctness of this reduction, consider that the equivalence class $[db]_{\pi,S}$ defined by the policy $S$ contains a database state $db''$ for each database state $db'' \in \Omega_D$ such that $db''(T) = \{v\}$. Then, $\text{SUBSUME}^{RC}(D',S',\{x \mid T(x)\} \land \phi), \{v\}, db = \top$ iff $\phi$ holds in all database states in $[db]_{\pi,S}$, i.e., iff $\phi$ is finitely valid. This reduction can be implemented by a total Turing machine. Therefore, since $\text{FINVAL}^{RC}$ is undecidable, so is $\text{SUBSUME}^{RC}$. \hfill $\blacksquare$
From Theorem 3.6, it follows that, for non-boolean queries, there are no computable optimal SAQP algorithms in the Truman model for the relational calculus. Therefore, it is impossible to securely process queries without either violating the security policy, losing some information, or returning incorrect results.

Note that, following the proof of Theorem 3.6, we can prove an even stronger result: for any fragment $F$ of RC such that (1) $F$ is closed under conjunction, and (2) FINVAL$^F$ is undecidable, then SUBSUME$^F$ is also undecidable. Therefore, from well-known undecidability results for fragments of first-order logic, we can identify fragments of RC for which there are no optimal algorithms for non-boolean queries, such as the BSRRC fragment.

### 3.6.3 Possibility Results

We now present a general criterion for identifying fragments of RC where SUBSUME$^F$ is decidable. We first introduce the notion of encoding the subsumption relation $\leq$ in a formula. Let $D$ be a database schema, $\psi(\overline{x})$ be a formula with free variables $\overline{x}$, and $T \in M$ be a multi-set such that $|\overline{t}| = |\overline{x}|$ for all $\overline{t} \in T$. We say that a sentence $\phi_{T,\psi(\overline{x})}$ encodes the subsumption relation between $T$ and $\psi(\overline{x})$ if $\phi_{T,\psi(\overline{x})}$ is domain independent and for all $\overline{d} \in \Omega_D$, $[\phi_{T,\psi(\overline{x})}]^\overline{d} = \top$ if $T \preceq ([\overline{x}]\phi(\overline{x}))^\overline{d}$.

Lemma 3.3 presents sufficient conditions for the decidability of the SUBSUME$^F$ decision problem.

**Lemma 3.3.** Let $F$ be a query language. SUBSUME$^F$ is decidable if there is a query language $F'$ such that:

1. $F \subseteq F'$.
2. AGREE$^{F'}$ is decidable for all $F$-policies, and
3. for any multi-set $T \in M$ and any $F$-formula $\psi(\overline{x})$ such that $|\overline{t}| = |\overline{x}|$ for all $\overline{t} \in T$, we can compute a sentence $\phi_{T,\psi(\overline{x})} \in F'$ that encodes the subsumption relation between $T$ and $\psi(\overline{x})$.

**Proof.** Let $D$ be a database schema, $\overline{d} \in \Omega_D$ be a state, $\{\overline{x} | \psi\}$ be an $F$-query, $S$ be an $F$-policy, and $T \in M$ be a multi-set of tuples such that $|\overline{t}| = |\overline{x}|$ for all $\overline{t} \in T$. Let $\phi_{T,\psi}$ be the $F'$-sentence encoding the subsumption relation. Then, SUBSUME$^F(D,S,\{\overline{x} | \psi\},T,\overline{d}) = \top$ iff $[\phi_{T,\psi}]^\overline{d} = \top$ and AGREE$^{F'}(D,S,\phi_{T,\psi},\overline{d}) = \top$. This reduction can be implemented by a Turing machine. The decidability of SUBSUME$^F$ directly follows from (1)–(3).

We now prove the correctness of the above reduction. If $[\phi_{T,\psi(\overline{x})}]^\overline{d} = \top$ and AGREE$^{F'}(D,S,\phi_{T,\psi(\overline{x})},\overline{d}) = \top$, then for all states $\overline{d}'$ indistinguishable from $\overline{d}$, we have that $T \preceq ([\overline{x}]\phi(\overline{x}))^{\overline{d}'}$. Hence, SUBSUME$^F(D,S,\{\overline{x} | \psi(\overline{x})\},T,\overline{d}) = \top$. If $[\phi_{T,\psi(\overline{x})}] = \bot$, then $T \not\preceq ([\overline{x}]\phi(\overline{x}))^{\overline{d}}$ and thus SUBSUME$^F(D,S,\{\overline{x} | \psi(\overline{x})\},T,\overline{d}) = \bot$. Similarly, if AGREE$^{F'}(D,S,\phi_{T,\psi(\overline{x})},\overline{d}) = \bot$, then there is at least a state $\overline{d}'$ indistinguishable from $\overline{d}$ such that $T \not\preceq ([\overline{x}]\phi(\overline{x}))^{\overline{d}'}$. Therefore, also in this case SUBSUME$^F(D,S,\{\overline{x} | \psi(\overline{x})\},T,\overline{d}) = \bot$.

We now use Theorem 3.3 and Lemma 3.3 to prove the decidability of SUBSUME$^{ERC}$. From Theorem 3.7, whose full proof is given in Appendix A, it follows that there are optimal SAQP algorithms for the existential fragment of the relational calculus.

**Theorem 3.7.** SUBSUME$^{ERC}$ is decidable.

**Proof Sketch.** Given a multi-set of tuples $T$ in $M$ and a tuple $\overline{t} \in T$, let $K_{T,T'}$ be the multi-set $\{\overline{t} | \overline{t} \in T \land \overline{t} \subseteq \overline{t}'\}$. The encoding is given by:

$$\phi_{T,\psi(\overline{x})} := \bigwedge_{\overline{t} \in T} \exists^{|K_{T,T'}|} \overline{t}, (\psi(\overline{x}) \land \bigwedge_{i \in \{1,\ldots,|\overline{x}|\} \land \overline{t}(i) \neq 1} \overline{x}(\overline{t}))$$

$\phi_{T,\psi(\overline{x})}$ can be equivalently rewritten as an ERC-sentence.

Since ERC strictly contains conjunctive queries, also SUBSUME$^{ERC}$ is decidable.

### 3.6.4 Truman and Non-Truman models

We now study the connections between Truman and Non-Truman models for non-boolean queries. We first introduce the notion of a strongly-optimal security-aware query processor. Using strongly-optimal SAQPs, we generalize the Non-Truman model approach for non-boolean queries [131] to our security policies, which are more expressive than authorization views. Afterwards, we study the relationships between strongly-optimal SAQPs and optimal SAQPs in the Non-Truman model and extend our decidability results.

A strongly-optimal SAQP for non-boolean queries returns the original query’s result whenever it is secure to do so, and otherwise returns $\bot$. Let $L$ be the set of all finite sets of unmasked tuples. A security-aware query processor for non-boolean queries is called strongly-optimal if it satisfies the
3.6. Non-boolean queries

\[ M_{s,a,q}(q, S, db) = \begin{cases} \{q\}^{db} & \text{if } \forall db' \in [db]_{\sim S} \cdot [q]^{db'} = [q]^{db} \\ \top & \text{otherwise} \end{cases} \]

Figure 3.4: A strongly-optimal SAQP for non-boolean queries

Definitions 3.8–3.10 and the set \( U \), containing all the possible results, is \( L \cup \{\top\} \). Figure 3.4 describes a strongly optimal SAQP for non-boolean queries. Depending on the query language, this function may not be computable.

Optimal SAQPs and strongly-optimal SAQPs are distinct. For instance, optimal SAQPs can return partial results while strongly-optimal SAQPs cannot. This holds even if we consider just row-level policies. Note that too, in the Non-Truman model, boolean queries are a special case of non-boolean queries.

We first define a new decision problem corresponding to \( \text{SUBSUME}^F \).

**Definition 3.22.** Given a query language \( F \), \( \text{EQUAL}^F \) denotes the problem:

**Input:** A database schema \( D \), an \( F \)-security policy \( S \), a non-boolean \( F \)-query \( q = \{\tau | \phi(\tau)\} \), a set of tuples \( T \in L \) such that \( [\tau] = [\tau] \) for all \( \tau \in T \), and a state \( db \in \Omega_D \).

**Question:** For all states \( db' \in [db]_{\sim S} \), is \( T = [q]^{db'} \)?

Similarly to Theorems 3.1 and 3.5, we can prove:

**Theorem 3.8.** Let \( F \) be a query language. There is a computable strongly-optimal \( F \)-SAQP algorithm \( M \) for non-boolean queries iff \( \text{EQUAL}^F \) is decidable.

*Proof.* \((\Rightarrow)\). Let \( M \) be a computable strongly-optimal \( F \)-SAQP \( M \) for non-boolean queries. We use \( M \) as a subroutine in a decision procedure for \( \text{EQUAL}^F \), which takes as input a database schema \( D \), a policy \( S \), a set of tuples \( T \), a non-boolean \( F \)-query \( q \), and a database state \( db \). Then, \( \text{EQUAL}^F(D, S, q, T, db) = \top \) if \( T = M(q, S, db) \).

\((\Leftarrow)\). We use \( \text{EQUAL}^F \) to construct a strongly-optimal \( F \)-SAQP \( M \). Let \( D \) be a database schema. \( M \) takes as input a security policy \( S \) over \( D \), a non-boolean query \( q \in F \), and a state \( db \in \Omega_D \). We first compute the value \( T = [q]^{db} \). If \( \text{EQUAL}^F(D, S, q, T, db) = \top \), then \( M(q, S, db) = \top \), otherwise \( M(q, S, db) = \bot \). It is easy to see that \( M \) is strongly-optimal and computable.

We can easily adapt the proof in Section 3.6.2 to obtain the following undecidability result:

**Theorem 3.9.** \( \text{EQUAL}^{RC} \) is undecidable.

*Proof.* To prove the theorem, we reduce \( \text{AGREE}^{RC} \) to \( \text{EQUAL}^{RC} \). The undecidability of \( \text{EQUAL}^{RC} \) then directly follows from Theorem 3.2.

Let \( D \) be a database schema, \( \phi \) be a boolean formula, \( S \) be a security policy over \( D \), and \( db \in \Omega_D \) be a state. We define a database schema \( D' \) from \( D \) by adding a new relation schema \( T \) with arity 1. Let \( v \) be a value in \( \text{dom} \) and \( db' \in \Omega_D^{v} \) be the database state such that \( [db'](T) = \{v\} \) and \( [db'](R) = [db](R) \) for all relation schemas \( R \) in \( D \). Finally, we extend the security policy \( S = \{\text{ROW}, \text{COL}\} \) on \( D \) to a new security policy \( S' = \{\text{ROW}', \text{COL}'\} \) on \( D' \), where \( \text{ROW}' = \text{ROW} \cup \{\{x | T(x)\}, \top\} \) and \( \text{COL}' = \text{COL} \cup \{\{x | T(x)\}, \top, 1\} \). The equivalence class \( [db']_{\sim S} \) contains a state \( db \) for each state \( db' \in [db]_{\sim S} \) such that the relation instances in \( db_1 \) and \( db_2 \) are identical for all relation schema in \( D \). Note that there is a bijection from the set \( [db]_{\sim S} \) to the set \( [db']_{\sim S} \). We define the query \( q' = \{x | T(x) \land \phi\} \). The set \( K' = \{v\} \), whereas \( K'' = \emptyset \). Then, \( \text{AGREE}^{R}(D, S, \phi, db) = \top \) iff \( \text{EQUAL}^{R}(D', S', q', \{v\}, db') = \top \) or \( \text{EQUAL}^{R}(D', S', q', \emptyset, db') = \top \). The reduction described above can be implemented by a total Turing machine. We now prove its correctness. If \( \text{EQUAL}^{R}(D', S', q', \{v\}, db') = \top \), then \( [\phi]^{db''} = \top \) for all \( db'' \in [db]_{\sim S} \) and thus \( \text{AGREE}^{R}(D, S, \phi, db) = \top \). Similarly, if \( \text{EQUAL}^{R}(D', S', q', \emptyset, db') = \top \), then \( [\phi]^{db''} = \bot \) for all \( db'' \in [db]_{\sim S} \) and therefore \( \text{AGREE}^{R}(D, S, \phi, db) = \bot \). Otherwise, if \( \text{EQUAL}^{R}(D', S', q', \{v\}, db') = \bot \) and \( \text{EQUAL}^{R}(D', S', q', \emptyset, db') = \top \), then there are two states \( db_1, db_2 \in [db]_{\sim S} \) such that \( [\phi]^{db_1} \neq [\phi]^{db_2} \) and therefore \( \text{AGREE}^{R}(D, S, \phi, db) = \bot \).}

Therefore, even strongly-optimal Security-Aware Query Processing is impossible for the relational calculus. Furthermore, \( \text{EQUAL}^F \) is undecidable for any fragment \( F \) (closed under conjunction) where \( \text{FINVAL}^F \), \( \text{FINSAT}^F \), or \( \text{AGREE}^F \) are undecidable.

We can also provide some decidability results. We first define a new encoding. Let \( D \) be a database schema, \( \psi(T) \) be a formula with free variables \( T \), and \( T \in L \) be a finite set of tuples such that \( [\tau] = [\tau] \) for all \( \tau \in T \). We say that a sentence \( \phi_{T, \psi(T)} \) encodes the property \( \psi(T) \) satisfied by the set \( T \) iff \( \phi_{T, \psi(T)} \) is domain independent and for all \( db \in \Omega_D \), \( [\phi_{T, \psi(T)}]^{db} = \top \) iff \( T = [\{\tau | \psi(T)\}]^{db} \).

We can now give sufficient conditions for the decidability of the \( \text{EQUAL}^F \) problem.
Lemma 3.4. Let \( F \) be a query language. \( \text{EQUAL}^F \) is decidable if there is a query language \( F' \) such that:

1. \( F \subseteq F' \),
2. \( \text{AGREE}^{F'} \) is decidable for all \( F' \)-policies, and
3. for any finite set \( T \subseteq \Omega \) and any formula \( \psi(\bar{t}) \in F' \) such that \( \{\bar{t}\} = \{\bar{t}\} \) for all \( \bar{t} \in T \), we can compute a sentence \( \phi_{T,\psi(\bar{t})} \in F' \) that encodes the property \( \psi(\bar{t}) \) satisfied by the set \( T \).

Proof. Let \( D \) be a database schema, \( S \) be an \( F \)-security policy over \( D \), \( db \in \Omega_D \) be a database state, \( q = \{\bar{t} \mid \psi(\bar{t})\} \) be an \( F \)-query, and \( T \subseteq \Omega \) be a finite set of tuples such that \( \{\bar{t}\} = \{\bar{t}\} \) for all \( \bar{t} \in T \). From (3), it follows that we can compute the encoding \( \phi_{T,\psi(\bar{t})} \). Then, \( \text{EQUAL}^F(D,S,q,T,db) = \top \) iff \( \phi_{T,\psi(\bar{t})} \) is true and \( \text{AGREE}^{F'}(D,S,\phi_{T,\psi(\bar{t})},db) = \top \). The decidability of \( \text{EQUAL}^F \) directly follows from (1) and (2).

We now show the correctness of our reduction. In case \( \phi_{T,\psi(\bar{t})} \) is true and \( \text{AGREE}^{F'}(D,S,\phi_{T,\psi(\bar{t})},db) = \top \), then for all states \( db' \) indistinguishable from \( db \), we have that \( T = \{\{\bar{t} \mid \psi(\bar{t})\}\}^{db'} \), and therefore \( \text{EQUAL}^F(D,S,\{\bar{t} \mid \psi(\bar{t})\},T,db) = \top \). If \( \phi_{T,\psi(\bar{t})} = \bot \), then \( T \neq \{\{\bar{t} \mid \psi(\bar{t})\}\}^{db} \) and thus \( \text{EQUAL}^F(D,S,\{\bar{t} \mid \psi(\bar{t})\},T,db) = \bot \). Similarly, if \( \text{AGREE}^{F'}(D,S,\phi_{T,\psi(\bar{t})},db) = \bot \), then there is at least a state \( db' \) indistinguishable from \( db \) such that \( T \neq \{\{\bar{t} \mid \psi(\bar{t})\}\}^{db'} \) and thus \( \text{EQUAL}^F(D,S,\{\bar{t} \mid \psi(\bar{t})\},T,db) = \bot \).

Using Lemma 3.4, we derive the following result, whose proof is given in Appendix A.

Theorem 3.10. \( \text{EQUAL}^{\text{F-RC}} \) is decidable.

Similarly to Section 3.5.4, we have:

Lemma 3.5. Let \( D \) be a database schema, \( F \) be a query language (that supports equality and is closed under conjunction and disjunction), \( AV \) be a set of \( F \)-authorization views, \( db \in \Omega_D \) be a state, and \( q \) be a non-boolean \( F \)-query. Moreover, let \( S \) be the \( F \)-security policy equivalent to \( AV \). The query \( q \) is conditionally valid in the state \( db \) with respect to \( AV \) iff \( \text{EQUAL}^F(D,S,q,\{\bar{t}\}^{db'},db) = \top \).

Proof. \((\Rightarrow)\). Let \( \{\bar{t}\} \) be a query conditionally valid in the state \( db \) with respect to \( AV \). From this, it follows that there exists a query \( q' \) expressed only in terms of \( AV \) such that \( \{\bar{t}\}^{db'} = \{\bar{t}\}^{db'} \) for all states \( db' \) \( AV \)-equivalent to \( db \). Note that two states are \( AV \)-equivalent if they have the same views' materializations. Since \( q' \) is expressed only in terms of \( AV \), it is domain independent, the query language is deterministic, and the views' materializations are fixed, then \( q' \) returns always the same result \( K \) on all states \( db' \) \( AV \)-equivalent to \( db \). Therefore, \( \{\bar{t}\}^{db'} = \{\bar{t}\}^{db'} = K \) for all states \( db' \) such that \( db \in \text{AV}_{db'} \). Since \( AV \) and \( S \) are equivalent, it follows that \( db \in \text{AV}_{db'} \) iff \( db \in \text{AV}_{db'} \). It follows that \( \{\bar{t}\}^{db'} = K \) for all states \( db' \) such that \( db \in \text{AV}_{db'} \). Note that \( \{\bar{t}\}^{db} = K \) and, therefore, \( \text{EQUAL}^F(D,S,q,\{\bar{t}\}^{db},db) = \top \).

\((\Leftarrow)\). Let \( q = \{\bar{t} \mid \phi(\bar{t})\} \) be a query such that \( \text{EQUAL}^F(D,S,q,\{\bar{t}\}^{db},db) = \top \) holds. Moreover, let \( K = \{\bar{t}\}^{db} \). Let \( S \) be the \( F \)-security policy equivalent to \( AV \). Since \( \text{EQUAL}^F(D,S,q,\{\bar{t}\}^{db},db) = \top \), the result of \( q \) is \( K \) for all states indistinguishable from \( db \) according to \( S \). Since \( AV \) and \( S \) are equivalent, it follows that the result of \( q \) is \( K \) for all states \( AV \)-equivalent to \( db \). We can now define the query \( q' \) as

\[
\{y_1, \ldots, y_{|T|} \mid \bigvee_{i \in \{1, \ldots, |T|\}} y_i = T(i)\}
\]

Observe that (1) the result of \( q' \) for all states \( db' \) \( AV \)-equivalent to \( db \) is exactly \( K \), and (2) \( q' \) does not depend on database tables. This means that \( q' \) is equivalent to \( q \) for all states \( AV \)-equivalent to \( db \), and therefore \( q \) is conditionally valid in \( db \) according to \( AV \).

Theorem 3.11. Let \( F \) be a query language. A strongly-optimal \( F \)-SAQP for non-boolean queries is an optimal \( F \)-SAQP for non-boolean queries in the Non-Truman model when \( \dagger \) is interpreted as rejecting the query.

Proof. The theorem follows from Proposition 3.2, Theorem 3.8, Lemma 3.5, and the definition of optimal SAQP in the Non-Truman model.

For non-boolean queries, strongly-optimal algorithms and optimal algorithms are distinct. Therefore, from Theorem 3.11, we have:

Corollary 3.1. There is a non-boolean \( RC \)-query \( q \), a state \( db \), a set \( AV \) of \( RC \)-views, and the equivalent row-level \( RC \)-policy \( S \), such that the result of an optimal SAQP for non-boolean queries in the Non-Truman model is different from the result of an optimal SAQP for non-boolean queries in the Truman model.
3.7. Related Work

Security-Aware Query Processing. Security-Aware Query Processing algorithms are implemented in commercial databases [8, 41, 150]. Despite that, only limited work has been done on the theoretical aspects of this problem. In [131], Rizvi et al. proposed the notions of Truman and Non-Truman models. They provided inference rules, which are sound but not complete, for determining whether a query is conditionally valid. The undecidability of the unconditional validity problem follows from well-known results on query rewriting using views [124]. Zhang et al. [170] studied the conditional validity problem for conjunctive queries and showed that it is decidable. We improve these results in that we provide sufficient conditions for the decidability of conditional validity. We also show that this problem is decidable for the existential fragment of the relational calculus, which contains conjunctive queries, and for our security policies, which are more expressive than authorization views.

Wang et al. [165] were the first to propose correctness criteria for algorithms in the Truman model. They proposed a secure and sound SAQP algorithm. Other secure and sound algorithms have been proposed since then, such as [90, 145]. Our results prove the claim of Wang et al. that optimal Security-Aware Query Processing is difficult. We also prove that optimal SAQP in the Truman model is possible for the existential fragment of the relational calculus.

Instance-based Determinacy. The instance-based determinacy problem [107] consists of checking whether, given a database state \( db \), a set of views \( V \), and a query \( q \), the materialization of the views in \( V \) in the state \( db \) fully determines the result of \( q \) in \( db \). Koutris et al. [107] proved that the problem of instance-based determinacy for unions of conjunctive queries under the set semantics is decidable and is \( \text{coNP-complete} \) in terms of data complexity.

When restricted to row-level policies, which are equivalent to \( RC \)-views, the \( AGREE \) and \( EQUAL \) problems are equivalent to the instance-based determinacy problem for boolean and non-boolean queries respectively. In the unrestricted case, however, these equivalences do not hold because security policies are more expressive than \( RC \)-views and, therefore, \( AGREE \) and \( EQUAL \) are more general than instance-based determinacy under the set semantics. Indeed, masked tuples introduce a kind of bag semantics that cannot be captured using \( RC \)-views under the set semantics. We are not aware of any work exploring instance-based determinacy under the bag semantics.

Certain Answers. The problem of computing certain answers using views under the closed world assumption [9] shares similarities with optimal SAQP. But it also has important differences. For instance, the result of a boolean query \( \phi \) according to optimal SAQP is one of \( \{\top, \bot, \upsilon\} \) whereas the certain answer is one of \( \{\top, \bot\} \). Indeed, one must compute both the certain answer and the possible answer to compute the result of boolean optimal SAQP.

For non-boolean queries, optimal algorithms in the Truman model return a set of results, whereas the certain answer is unique. Moreover, while optimal SAQP in the Truman model considers masked tuples, the certain answer problem considers only unmasked tuples. This seriously limits Fine-Grained Access Control. For example, suppose we have a policy with a constraint on the \( i \)-th value of a query \( q = \{\bar{V} \mid v\} \), for some \( i \in \{1, \ldots, |\bar{V}|\} \). The certain answer of \( q \) will be \( \emptyset \), whereas the result of an optimal SAQP will be a multi-set of masked tuples where the \( i \)-th value is replaced with \( \upsilon \). In the Non-Truman model, an algorithm either returns the query’s result or rejects the query. In contrast, the certain answer to a query is generally different from the query’s result.
The main difference between optimal SAQP and the certain answer problem is that security policies are more expressive than views in the relational calculus. As we previously noted, the presence of masked tuples introduces a kind of bag semantics. However, while the problem of querying views has been studied extensively under the set semantics [9, 124], only limited work has been done under the bag-set and bag semantics [12]. Note that our work can be viewed in the general setting of querying views under the bag-set semantics. Despite that, we decided to keep the terminology of Security-Aware Query Processing for consistency with previous works on Fine-Grained Access Control in databases, e.g., [131, 165].

Another related problem is computing the certain answer to a query in an incomplete database [112]. The same considerations for certain answers using views apply to this case. Observe also that the notion of indistinguishability appears related to the notion of semantics of incomplete databases [112].

3.8 Conclusions

We presented the first analysis of optimal Security-Aware Query Processing. Our results show that (1) it is impossible to build optimal SAQP algorithms for Codd-complete query languages, such as SQL, and (2) this impossibility is not due to specific characteristics of these query languages but rather to the undecidability of the relational calculus and several of its fragments. Note that our results also do not depend on the particular characteristics of our security model. We showed that there are interesting fragments of $RC$ for which optimal Security-Aware Query Processing is possible, such as the existential fragment of $RC$. Our results may be used to prove the decidability of optimal Security-Aware Query Processing for other fragments of $RC$, such as the monadic fragment and the guarded fragment, and other query languages.

For boolean queries, we showed that, for the relational calculus, optimal SAQP in the Non-Truman model is a special case of optimal SAQP in the Truman model, and that optimal algorithms in the two models coincide for the relational calculus extended with aggregation operators. In contrast, for non-boolean queries, optimal algorithms in the two models are distinct. This has direct consequences for developing algorithms for those models.

Optimal Security-Aware Query Processing is a difficult problem: it is intractable even for conjunctive queries. Indeed, it is \textit{coNP-complete} in terms of data complexity for boolean conjunctive queries and row-level security policies [107]. Despite that, optimal SAQP can still have practical applications. For instance, there are fragments of conjunctive queries for which optimal SAQP in the Non-Truman model is in \textit{PTIME} in terms of data complexity for row-level policies [107]. This suggests that there might be other fragments of conjunctive queries for which optimal SAQP is tractable.

Non-optimal SAQP algorithms can benefit from efficient optimal algorithms for some special cases. Namely, we can use tractable optimal algorithms when it is possible, and fall back to efficient non-optimal algorithms for the cases where optimal SAQP is undecidable or intractable. Furthermore, the study of optimal SAQP may shed some light on the trade-offs between efficiency and optimality, and can therefore lead to improvements for non-optimal algorithms.
Chapter 4

Securing Databases from Probabilistic Inference

Probability is expectation founded upon partial knowledge. A perfect acquaintance with all the circumstances affecting the occurrence of an event would change expectation into certainty, and leave nether room nor demand for a theory of probabilities.

George Boole

Databases can leak confidential information when users combine query results with probabilistic data dependencies, such as those found in genomics [97,105,109], social networks [92], and location tracking [117]. Attackers can exploit these dependencies to infer sensitive information with high confidence. To effectively prevent probabilistic inferences, DBIC mechanisms should (1) support a large class of probabilistic dependencies, and (2) have tractable runtime performance. The former is needed to express different attacker models. The latter is necessary for mechanisms to scale to real-world databases. Most existing DBIC mechanisms, however, support only precise data dependencies [36,40,157,158,168] or just limited classes of probabilistic dependencies [44,45,91,102,120,121,166]. As a result, they cannot reason about the complex probabilistic dependencies that exist in many realistic settings. Mardziel et al.’s mechanism [116] instead supports arbitrary probabilistic dependencies, but no complexity bounds have been established and their algorithm appears to be intractable.

In this chapter, we propose foundations for DBIC based on ProbLog, a probabilistic logic programming language. We leverage these foundations to develop Angerona\(^1\), a provably secure enforcement mechanism that prevents information leakage in the presence of probabilistic dependencies. We then provide a tractable inference algorithm for a practically relevant fragment of ProbLog. Finally, we empirically evaluate Angerona’s performance showing that it scales to relevant security-critical problems. Note that this chapter is largely based on [88].

Structure. In Section 4.1, we illustrate the security risks associated with probabilistic data dependencies. In Section 4.2, we present our system model, which we formalize in Section 4.3. In Section 4.4, we introduce AtkLog, a language, based on ProbLog, for formalizing attacker models in DBIC, whereas in Section 4.5 we present our dedicated inference engine for acyclic ProbLog programs. In Section 4.6, we present Angerona. We discuss related work in Section 4.7 and draw conclusions in Section 4.8. Finally, in Section 4.9 we present some extensions, technical details, and additional examples. The proof of all our results are given in Appendix B, and a prototype of our enforcement mechanism is available at [85].

4.1 Motivating Example

Hospitals and medical research centres store large quantities of health-related information for purposes ranging from diagnosis to research. As this information is extremely sensitive, the databases used must be carefully secured [7,130]. This task is, however, challenging due to the dependencies between health-related data items. For instance, information about someone’s hereditary diseases or genome can be inferred from information about her relatives. Even seemingly non-sensitive information, such as someone’s job or habits, may leak sensitive health-related information such as her predisposition to diseases. Most of these dependencies can be formalized using probabilistic models developed by medical researchers.

Consider a database storing information about the smoking habits of patients and whether they have been diagnosed with lung cancer. The database contains the tables patient, smokes, cancer,

\(^1\)Angerona is the Roman goddess of silence and secrecy. She is the keeper of the city’s sacred, and secret, name.
The database system consists of a set of Security policy, and is executed if and only if the command is authorized by the security policy. Each command is checked by the inference control system, which consists of a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). We assume that all communication between users and the components whose relatives also opted-in to the user study, independently of the amount of leaked information, researchers, even honest ones, would be able to access the health-related status only of those patients whose relatives have cancer, (b) for each parent with cancer, the likelihood that a child develops cancer increases by 15%, and (c) if a patient smokes regularly, his probability of developing cancer increases by 25%. We intentionally work with a simple model since, despite its simplicity, it illustrates the challenges of securing data with probabilistic dependencies. We refer the reader to medical research for more realistic probabilistic models [14,161].

The database is shared between different medical researchers, each conducting a research study on a subset of the patients. All researchers have access to the patient, smoker, father, and mother tables. Each researcher, however, has access only to the subset of the cancer table associated with the patients that opted-in to his research study. We want to protect our database against a malicious researcher whose goal is to infer the health status of patients not participating in the study. This is challenging since restricting direct access to the cancer table is insufficient. Sensitive information may be leaked even by queries involving only authorized data. For instance, the attacker may know that the patient Carl, which has not disclosed his health status, smokes regularly. From this, he can infer that Carl’s probability of developing lung cancer is, at least, 30%. If, additionally, Carl’s parents opted-in to the research study and both have cancer, the attacker can directly infer that the probability of Carl developing lung cancer is 60% by accessing his parents’ information.

Security mechanisms that ignore such probabilistic dependencies allow attackers to infer sensitive information. An alternative is to use standard DBIC mechanisms and encode all dependencies as precise, non-probabilistic, dependencies. This, however, would result in an unusable system. Medical researchers, even honest ones, would be able to access the health-related status of those patients whose relatives also opted-in to the user study, independently of the amount of leaked information, which may be negligible. Hence, to secure the database and retain usability, it is essential to reason about the probabilistic dependencies.

### 4.2 System Model

Figure 4.1 depicts our system model. Users interact with two components: a database system and an inference control system, which consists of a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). We assume that all communication between users and the components and between the components themselves is over secure channels.

**Database System.** The database system manages the system’s data. Its state is a mapping from tables to sets of tuples.

**Users.** Each user has a unique account used to retrieve information from the database system by issuing SELECT commands. Note that these commands do not change the database state. This reflects settings where users have only read-access to a database. Each command is checked by the inference control system and is executed if and only if the command is authorized by the security policy.

**Security policy.** The system’s security policy consists of a set of negative permissions specifying information to be kept secret. These permissions express bounds on users’ beliefs, formalized as probability distributions, about the actual database content. Negative permissions are formalized using commands of the form \( \text{SECRET } q \text{ FOR } u \text{ THRESHOLD } l \), where \( q \) is a query, \( u \) is a user identifier, and \( l \) is a rational number, \( 0 \leq l \leq 1 \). This represents the requirement that “A user \( u \)'s belief in the result of \( q \) must be less than \( l \).” Namely, the probability assigned by \( u \)'s belief to \( q \)'s result must be less than \( l \). Requirements like “A user \( u \) is not authorized to know the result of \( q \)” can be formalized as \( \text{SECRET } q \text{ FOR } u \text{ THRESHOLD } 1 \). The system also supports commands of the form \( \text{SECRET } q \text{ FOR USERS NOT IN } \{ u_1, \ldots, u_n \} \text{ THRESHOLD } l \), which represents the requirement that “For all users \( u \not \in \{ u_1, \ldots, u_n \}, u \)'s belief in the result of \( q \) must be less than \( l \).”
4.3. Formal Model

**Attacker.** An attacker is a system user with an assigned user account, and each user is a potential attacker. An attacker’s goal is to violate the security policy, that is, to read or infer information about one of the *SECRET* with a probability of at least the given threshold.

An attacker can interact with the system and observe its behavior in response to his commands. Furthermore, he can reason about this information and infer information by exploiting domain-specific relationships between data items. We assume that attackers know the database schema as well as any integrity constraints on it.

**Attacker Model.** An attacker model represents each user’s initial beliefs about the actual database state and how he updates his beliefs by interacting with the system and observing its behavior in response to his commands. These beliefs may reflect the attacker’s knowledge of domain-specific relationships between the data items or prior knowledge.

**Inference Control System.** The inference control system protects the confidentiality of database data. It consists of a PEP and a PDP, configured with a security policy $P$ and an attacker model $ATK$. For each user, the inference control system keeps track of the user’s beliefs according to $ATK$.

The system intercepts all commands issued by the users. When a user $u$ issues a command $c$, the inference control system decides whether $u$ is authorized to execute $c$. If $c$ complies with the policy, i.e., the users’ beliefs still satisfy $P$ even after executing $c$, then the system forwards the command to the database, which executes $c$ and returns its result to $u$. Otherwise, it raises a *security exception* and rejects $c$.

### 4.3 Formal Model

Here we formalize the various components of our system model.

#### 4.3.1 Database Model

We use the database model presented in Section 2.2, and we use the relational calculus as a query language (see Section 2.2.2). Observe that we support arbitrary integrity constraints. Finally, we assume that the domain $\text{dom}$ is finite, as is standard for many application areas combining databases and probabilistic reasoning [60, 77, 105, 153]. In this case, the set of all states $\Omega_D$ is finite.

**Example 4.1.** The database associated with the example in Section 4.1 consists of five relational schemas $\text{patient}$, $\text{smokes}$, $\text{cancer}$, $\text{father}$, and $\text{mother}$, where the first three schemas have arity 1 and the last two have arity 2. We assume that there are only three patients Alice, Bob, and Carl, so the domain $\text{dom}$ is $\{\text{Alice}, \text{Bob}, \text{Carl}\}$. The integrity constraints are as follows:

- Alice, Bob, and Carl are patients.
  \[
  \text{patient}(\text{Alice}) \land \text{patient}(\text{Bob}) \land \text{patient}(\text{Carl})
  \]

- Alice and Bob are Carl’s parents.
  \[
  \forall x, y. \ (\text{father}(x,y) \leftrightarrow (x = \text{Bob} \land y = \text{Carl})) \land
  \forall x, y. \ (\text{mother}(x,y) \leftrightarrow (x = \text{Alice} \land y = \text{Carl}))
  \]

- Alice does not smoke, whereas Bob and Carl do.
  \[
  \neg \text{smokes}(\text{Alice}) \land \text{smokes}(\text{Bob}) \land \text{smokes}(\text{Carl})
  \]

Given these constraints, there are just 8 possible database states in $\Omega_D$, which differ only in their *cancer* relation. The content of the *cancer* relation is a subset of $\{\text{Alice}, \text{Bob}, \text{Carl}\}$, whereas the content of the other tables is shown in Figure 4.2. We denote each possible world as $s_C$, where the set $C \subseteq \{\text{Alice}, \text{Bob}, \text{Carl}\}$ denotes the users having cancer.
4.3.2 Security Policies

Existing access control models for databases are inadequate to formalize security requirements capturing probabilistic dependencies. For example, SQL cannot express statements like “A user u’s belief that φ holds must be less than l.” We present a simple framework, inspired by knowledge-based policies [116], for expressing such requirements.

A D-secret is a tuple ⟨U, φ, l⟩, where U is either a finite set of users in U or a co-finite set of users, i.e., U = U \ U′ for some finite U′ ⊆ U, φ is a relational calculus sentence over D, and l is rational number 0 ≤ l ≤ 1 specifying the uncertainty threshold. Abusing notation, when U consists of a single user u, we write u instead of {u}. Informally, ⟨U, φ, l⟩ represents that for each user u ∈ U, u’s belief that φ holds in the actual database state must be less than l. Therefore, a command of the form SECRET q FOR USERS NOT IN {u1, ..., un} THRESHOLD l can be represented as ⟨U \ {u1, ..., un}, q⟩, l.

Finally, a D-security policy is a finite set of D-secrets. Given a D-security policy P, we denote by secrets(P, u) the set of D-secrets associated with the user u, i.e., secrets(P, u) = {⟨u, φ, l⟩ | ⟨U, φ, l⟩ ∈ P ∧ u ∈ U}. Note that the function secrets is computable since the set U is always either finite or co-finite.

Our framework also allows the specification of lower bounds. Requirements of the form “A user u’s belief that φ holds must be greater than l” can be formalized as {u, ¬φ, 1 − l} (since the probability of ¬φ is 1 − P(φ), where P(φ) is φ’s probability). Security policies can be extended to support secrets over non-boolean queries. A secret {u, ⟨T | φ(T)⟩}, l can be seen as a shorthand for the set {⟨u, φ[−⟩ → T⟩, l | T ∈ ∪t∈D ⟨T | φ(T)⟩∗}, i.e., u’s belief in any tuple T being in the query’s result must be less than l.

Example 4.2. Let Mallory denote the malicious researcher from Section 4.1 and D be the database schema in Example 4.1. Consider the requirement from Section 4.1: Mallory’s belief in a patient having cancer must be less than 50%. This can be formalized as ⟨Mallory, cancer(⟨Alice⟩), 1/2⟩, ⟨Mallory, cancer(⟨Bob⟩), 1/2⟩, and ⟨Mallory, cancer(⟨Carl⟩), 1/2⟩, or equivalently as ⟨Mallory, [p, cancer(p)], 1/2⟩.

In contrast, the requirement “For all users u that are not Carl, u’s belief in Carl having cancer must be less than 50%” can be formalized as ⟨{u} \ {Carl}, cancer(⟨Carl⟩), 1/2⟩, where Carl denotes the user identifier associated with Carl.

4.3.3 Formalized System Model

We now formalize our system model. We first define a system configuration, which describes the database schema and the integrity constraints. Afterwards, we define the system’s state. Finally, we define a system run, which represents a possible interaction of users with the system.

A system configuration is a tuple ⟨D, Γ⟩, where D is a database schema and Γ is a set of integrity constraints over D. Let C = ⟨D, Γ⟩ be a system configuration. A C-system state is a tuple ⟨db, U, P⟩, where db ∈ ΩD is a database state, U ⊆ U is a finite set of users (representing all active users), and P is a D-security policy. A C-query is a pair ⟨u, φ⟩ where u ∈ U is a user and φ is a relational calculus sentence over D.2 We denote by ΩC the set of all system states and by QC the set of all queries.

A C-event is a triple ⟨q, a, res⟩, where q is a C-query in QC, a ∈ {⊥, ⊤} is a security decision, where ⊤ stands for “authorized query” and ⊥ stands for “unauthorized query”, and res ∈ {⊥, ⊤} is the query’s result, where ⊤ and ⊥ represent the usual boolean values and stands that the query was not executed as access was denied. Given a C-event e = ⟨q, a, res⟩, we denote by q(e) (respectively a(e) and res(e)) the query q (respectively the decision a and the result res). A C-history is a finite sequence of C-events. We denote by HC the set of all possible C-histories.

We now formalize Policy Decision Points. A C-PDP is a function f : ΩC × QC × HC → {⊤, ⊥} taking as input a system state, a query, and a history and returning the security decision, accept (⊤) or deny (⊥).

Let C be a system configuration, s = ⟨db, U, P⟩ a C-state, and f be a C-PDP. A C-history is compatible with s and f iff for each 1 ≤ i ≤ |h|, (1) f(s, q(h(i))), h[i−1] = a(h(i)), (2) if a(h(i)) = ⊥, then res(h(i)) = ⊥, and (3) if a(h(i)) = ⊤, then res(h(i)) = q(φh(i)), where q(φh(i)) = ⟨u, φ⟩. In other words, h is compatible with s and f iff it was generated by the PDP f starting in state s.

A (C, f)-run is a pair ⟨s, h⟩, where s is a system state in ΩC and h is a history in HC compatible with s and f. Since all queries are SELECT queries, the system state does not change along the run. Hence, our runs consist of a state and a history instead of e.g., an alternating sequence of states and actions (as is standard for runs). We denote by runs(C, f) the set of all (C, f)-runs. Furthermore,

---

2Without loss of generality, we focus only on boolean queries. We can support non-boolean queries as follows. Given a query q := ⟨T | φ⟩ and a database state db, we first compute the set R of all possible results of q, i.e., R = {qφ| db ∈ ΩD}. Then, we authorize the query q and return its result qφdb iff for all R ∈ R, the inference control mechanism authorizes the boolean query \( \bigwedge_{φ| db} φ[−] → T \) ∧ (∀T. φ → ∨_{φ| db} T).
given a run \( r = (\langle db, U, P \rangle, h) \), we denote by \( r^i \) the run \( (\langle db, U, P \rangle, h^i) \), and we use a dot notation to access to \( r^i \)'s components. For instance, \( r.db \) denotes the database state \( db \) and \( r.h \) denotes the history.

**Example 4.3.** Consider the run \( r = (\langle db, U, P \rangle, h) \), where the database state \( db \) is the state \( s_{(a,b,c)} \), where \( Alice, Bob, \) and \( Carl \) have cancer, the policy \( P \) is defined in Example 4.2, the set of users \( U \) contains only \( Mallory \), and the history \( h \) is as follows (here we assume that all queries are authorized):
1. Mallory checks whether Carl smokes. Thus, \( h(1) = (\langle Mallory, smokes(Carl) \rangle, T, T) \).
2. Mallory checks whether Carl is Alice's and Bob's son. Therefore, \( h(2) = (\langle Mallory, father(Bob, Carl) \rangle, T, T) \).
3. Mallory checks whether Alice has cancer. Thus, \( h(3) = (\langle Mallory, cancer(Alice) \rangle, T, T) \).
4. Mallory checks whether Bob has cancer. Thus, \( h(4) = (\langle Mallory, cancer(Bob) \rangle, T, T) \). □

### 4.3.4 Attacker Model

To reason about DBIC, it is essential to precisely define (1) how users interact with the system, (2) how they reason about the system's behavior, (3) their initial beliefs about the database state, and (4) how these beliefs change by observing the system's behavior. We formalize this in an attacker model.

Each user has an initial belief about the database state. Following [52, 53, 71, 116], we represent a user's beliefs as a probability distribution over all database states. Furthermore, users observe the system's behavior and derive information about the database content. We formalize a user's observations as an equivalence relation over runs, where two runs are equivalent iff the user's observations are the same in both runs, as is standard in information-flow control [18, 20]. A user's knowledge is the set of all database states that he considers possible given his observations. Finally, we use Bayesian conditioning to update a user's beliefs given his observations.

Let \( C = (\{D, Γ\}) \) be a system configuration and \( f \) be a C-PDP. A C-*probability distribution* is a discrete probability distribution given by a function \( P : \Omega_D → [0, 1] \) such that \( \sum_{db ∈ Ω_D} P(db) = 1 \).

Given a set \( E ⊆ Ω_D \), \( P(E) \) denotes \( \sum_{s ∈ E} P(s) \). Furthermore, given two sets \( E', E'' ⊆ Ω_D \) such that \( P(E') ≠ 0 \), \( P(E'' \mid E') \) denotes \( P(E'' \cap E') / P(E') \) as is standard. We denote by \( PC \) the set of all possible C-probability distributions. Abusing notation, we extend probability distributions to formulae: \( P(ψ) = P(ψ|v) \), where \( [v] = \{ db ∈ Ω_D \mid P(db) = 1 \} \).

We now introduce *indistinguishability*, an equivalence relation used in information-flow control [93]. Let \( C \) be a system configuration and \( f \) be a C-PDP. Given a history \( h \) and a user \( u ∈ U \), \( h_u \) denotes the history obtained from \( h \) by removing all C-events from users other than \( u \), namely \( ε_u = ε \), and if \( h = ⟨⟨ u', q, a, res⟩⟩ \) then \( h_u = ⟨⟨ u, q, a, res⟩⟩ \) if \( u = u' \). Given two runs \( r = (db, U, P, h) \) and \( r' = (db', U', P', h') \) in \( runs(C, f) \) and a user \( u ∈ U \), we say that \( r \) and \( r' \) are indistinguishable for \( u \), written \( r ∼_u r' \), iff \( h_u = h'_u \). This means that \( r \) and \( r' \) are indistinguishable for a user \( u \) iff the system's behavior in response to \( u \)'s commands is the same in both runs. Note that \( ∼_u \) depends on both \( C \) and \( f \), which we generally leave implicit. Given a run \( r \), \( [r]_={} \) is the equivalence class of \( r \) with respect to \( ∼_u \), i.e., \( [r]_={} = \{ r' ∈ runs(C, f) \mid r ∼_u r' \} \), whereas \( [r]_u \) is set of all databases associated with the runs in \( [r]_={} \), i.e., \( [r]_u = \{ db \mid ∃ U, P, h, (db, U, P, h) ∈ [r]_={} \} \).

**Definition 4.1.** Let \( C = (\{D, Γ\}) \) be a configuration and \( f \) be a C-PDP. A \((C, f)-attacker model\) is a function \( ATK : U → PC \) associating to each user \( u ∈ U \) a C-probability distribution representing \( u \)'s initial beliefs. Additionally, for all users \( u ∈ U \) and all states \( db ∈ Ω_D \), we require that \( ATK(u)(db) > 0 \). The semantics of \( ATK \) is \( [ATK][u, r] = λdb ∈ Ω_D^P, ATK(u)(db) > 0 \), where \( u ∈ U \) and \( r ∈ runs(C, f) \). □

The semantics of an attacker model \( ATK \) associates to each user \( u \) and each run \( r \) the probability distribution obtained by updating \( u \)'s initial beliefs given his knowledge with respect to the run \( r \). We informally refer to \( [ATK][u, r](φ) \) as \( u \)'s beliefs in a sentence \( φ \) (given a run \( r \)).

**Example 4.4.** The attacker model for our motivating example from Section 4.1 is as follows. Let \( X_{Alice}, X_{Bob}, \) and \( X_{Carl} \) be three boolean random variables, representing the probability that the corresponding patient has cancer. They define the following joint probability distribution, which represents a user's initial beliefs about the actual database state: \( P(X_{Alice}, X_{Bob}, X_{Carl}) = P(X_{Alice}) \cdot P(X_{Bob}) \cdot P(X_{Carl} | X_{Alice}, X_{Bob}) \). The probability distributions of these variables are given in Figure 4.3 and they are derived from the probabilistic model in Section 4.1. We associate each outcome \( (x, y, z) \) of \( X_{Alice}, X_{Bob}, X_{Carl} \) with the corresponding database state \( s_C \), where \( C \) is the set of patients such that the outcome of the corresponding variable is \( T \). For each user \( u ∈ U \), the distribution \( P_u \) is defined as \( P_u (s_C) = P(X_{Alice} = x, X_{Bob} = y, X_{Carl} = z) \), where \( x \) (respectively \( y \) and \( z \)) is \( T \) if \( Alice \) (respectively \( Bob \) and \( Carl \)) is in \( C \) and \( ⊥ \) otherwise. Figure 4.4 shows the probabilities associated with each state in \( Ω_D^P \), i.e., a user's initial beliefs. Finally, the attacker model is \( ATK = u ∈ U, P_u \). □
Let \( C \) be a configuration, \( f \) be a \( C \)-PDP, and \( ATK \) be a \((C, f)\)-attacker model. A run \( r \) is secrecy-preserving for a secret \((u, \phi, l)\) and \( ATK \) iff for all \( 0 \leq i < |r| \), \( \lceil ATK \rceil(u, r^i)(\phi) < l \) implies \( \lceil ATK \rceil(u, r^{i+1})(\phi) < l \).

We now formalize our confidentiality notion. A PDP provides data confidentiality for an attacker model \( ATK \) iff all runs are secrecy-preserving for \( ATK \). Note that our security notion can be seen as a probabilistic generalization of opacity [142] for the database setting. Our notion is also inspired by the semantics of knowledge-based policies [116].

**Definition 4.3.** Let \( C = (D, \Gamma) \) be a system configuration, \( f \) be a \( C \)-PDP, and \( ATK \) be a \((C, f)\)-attacker model. We say that the PDP \( f \) provides data confidentiality with respect to \( C \) and \( ATK \) iff for all runs \( r = (\langle db, U, P \rangle, h) \) in \( runs(C, f) \), for all users \( u \in U \), for all secrets \( s \in secrets(P, u) \), \( r \) is secrecy-preserving for \( s \) and \( ATK \).

A PDP providing confidentiality ensures that if an attacker’s initial belief in a secret \( \phi \) is below the corresponding threshold, then there is no way for the attacker to increase his belief in \( \phi \) above the threshold by interacting with the system. However, this guarantee does not apply to trivial non-secrets, i.e., those secrets an attacker knows with a probability at least the threshold even before interacting with the system. No PDP can prevent their disclosure since the disclosure does not depend on the attacker’s interaction with the database.

**Example 4.5.** Let \( r \) be the run given in Example 4.3, \( ATK \) be the attacker model in Example 4.4, and \( u \) be the user \( Mallory \). In the following, \( \phi_1, \phi_2, \) and \( \phi_3 \) denote \( cancer(Carl) \), \( cancer(Bob) \), and \( cancer(Alice) \) respectively, i.e., the three secrets from Example 4.2. Furthermore, we assume that the policy contains an additional secret \( \langle Mallory, \phi_4, \frac{1}{2} \rangle \), where \( \phi_4 := \neg cancer(Alice) \).

Figure 4.5 illustrates \( Mallory \)'s beliefs about \( \phi_1, \ldots, \phi_4 \) and whether the run is secrecy-preserving for the secrets \( \phi_1, \ldots, \phi_4 \). The probabilities in the tables can be obtained by combining the states in \( [r^i]_{\text{tau}} \), for \( 0 \leq i \leq 4 \), and \( [\phi_j] \), for \( 1 \leq j \leq 4 \), with the probabilities from Figure 4.4. As shown in Figure 4.5, the run is not secrecy-preserving for the secrets \( \phi_1 \) and \( \phi_2 \) as it completely discloses that \( Alice \) and \( Bob \) have cancer, in the third and fourth steps respectively. Secrecy-preservation is also violated for the secret \( \phi_3 \), even though \( r \) does not directly disclose any information about \( Carl \)'s health status. Indeed, in the last step of the run, \( Mallory \)'s belief in \( \phi_3 \) is 0.6, which is higher than the threshold \( \frac{1}{2} \), even though his belief in \( \phi_1 \) before learning that \( Bob \) had cancer was below the threshold. Note that \( \phi_4 \) is a trivial non-secret: even before interacting with the system, \( Mallory \)'s belief in \( \phi_4 \) is 0.95.

**4.3.6 Discussion**

Our approach assumes that the attacker’s capabilities are well-defined. While this, in general, is a strong assumption, there are many domains where such information is known. There are, however,
domains where this information is lacking. In these cases, security engineers must (1) determine the appropriate beliefs capturing the desired attacker models, and (2) formalize them. The latter can be done, for instance, using AtkLog (see Section 4.4). Note however that precisely eliciting the attackers’ capabilities is still an open problem in DBIC.

4.4 AtkLog

We now describe AtkLog, a language for formalizing a large class of attacker models. AtkLog is built on top of ProbLog [60,61,73], a state-of-the-art probabilistic logic programming language. We first briefly review ProbLog’s syntax and semantics. Afterwards, we present AtkLog and use it to formalize the attacker model associated with our motivating example from Section 4.1.

4.4.1 Probabilistic Logic Programming

ProbLog [60,61,73] is a probabilistic logic programming language with associated tool support. An exact inference engine for ProbLog is available at [6].

Here we introduce ProbLog by building on top of our logic programming’s foundations from Section 2.3. Thus, following our treatment of logic programs in Section 2.3, we restrict ourselves to function-free stratified ProbLog programs with negation. As a result, in our setting ProbLog is a probabilistic extension of stratified DATALOG with negation. In the following, let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite set.

**Syntax.** To reason about probabilities, ProbLog extends logic programs with probabilistic atoms. A $(\Sigma, \text{dom})$-probabilistic atom is a $(\Sigma, \text{dom})$-atom $a$ annotated with a rational value $0 \leq v \leq 1$, denoted $v:a$. If $v = 1$, then we write $a(\top)$ instead of $1:a(\top)$. A $(\Sigma, \text{dom})$-ProbLog program is a finite set of ground $(\Sigma, \text{dom})$-probabilistic atoms and $(\Sigma, \text{dom})$-rules. Note that ground atoms $a \in \Sigma \cup \text{dom}$ are represented as $1:a$. Given a ProbLog program $p$, we denote by $\text{prob}(p)$ the set of all probabilistic ground atoms $v:a$ in $p$, i.e., $\text{prob}(p) := \{v:a \in p \mid 0 \leq v \leq 1 \wedge a \in \Sigma \cup \text{dom}\}$, and by $\text{rules}(p)$ the (non-probabilistic) rules in $p$, i.e., $\text{rules}(p) := p \setminus \text{prob}(p)$. Observe that ProbLog programs are subject to all restrictions we introduced in Section 2.3 for logic programs. For instance, we consider only programs $p$ that admit a stratification. Finally, we say that a ProbLog program $p$ is a logic program iff $v = 1$ for all $v:a \in \text{prob}(p)$, i.e., $p$ does not contain probabilistic atoms.

**Semantics.** Given a $(\Sigma, \text{dom})$-ProbLog program $p$, a $p$-grounded instance is a logic program $A \cup R$, where the set of ground atoms $A$ is a subset of $\{a \mid \exists v, v:a \in \text{prob}(p)\}$ and $R = \text{rules}(p)$. Informally, a grounded instance of $p$ is one of the logic programs that can be obtained by selecting some of the probabilistic atoms in $p$ and keeping all rules in $p$. A p-probabilistic assignment is a total function associating to each probabilistic atom $v:a$ in $\text{prob}(p)$ a value in $(\top, \bot)$. We denote by $A(p)$ the set of all $p$-probabilistic assignments. The probability of a p-probabilistic assignment $f$ is $\text{prob}(f) = \prod_{v:a \in \text{prob}(p)} (v:a = \top) \cdot \prod_{v:a \in \text{prob}(p)} (v:a = \bot) (1 - v)$. Given a p-probabilistic assignment $f$, instance$(p,f)$ denotes the grounded instance $\{a \mid \exists v, f(v:a) = \top\} \cup \text{rules}(p)$.

The semantics of a $(\Sigma, \text{dom})$-ProbLog program $p$ is defined as a probability distribution over all possible $p$-grounded instances. Note that ProbLog’s semantics relies on the closed world assumption, namely every fact that is not in a given model is considered false. The semantics of $p$, denoted by $[p]$, is as follows: $[p] = \sum_{f \in F_{p,s}} \text{prob}(f)$, where $F_{p,s} = \{f \in A(p) \mid f = \text{instance}(p,f)\}$. We remark that a $(\Sigma, \text{dom})$-ProbLog program $p$ implicitly defines a probability distribution over $(\Sigma, \text{dom})$-structures. Indeed, the probability of a given $(\Sigma, \text{dom})$-structure $s$ is the sum of the probabilities of all $p$-grounded instances $f$ such that $[p'] = s$.

\[\text{With a slight abuse of notation, we treat a set of ground } (\Sigma, \text{dom})\text{-atoms as a } (\Sigma, \text{dom})\text{-structure. This is without loss of generality. Under the closed world assumption, a set of ground atoms directly defines a structure (and vice versa).}\]
semantics of $p$ to $(\Sigma, \text{dom})$-structures and sentences as follows: $[p](s) = \sum_{f \in \mathcal{M}(p,s)} \text{prob}(f)$, where $s$ is a $(\Sigma, \text{dom})$-structure and $\mathcal{M}(p,s)$ is the set of all assignments $f$ such that $\text{instance}(p,f) = s$. Finally, $p$’s semantics can be lifted to sentences as follows: $[p](\phi) = \sum_{s \in \text{dom}[p]} [p](s)$, where $[\phi]$ is the set of all $(\Sigma, \text{dom})$-structures satisfying $\phi$.

**Evidence.** ProbLog supports expressing evidence inside programs [60]. To express evidence, i.e., to condition a probability distribution on some event, we use statements of the form $\text{evidence}(a,v)$, where $a$ is a ground atom and $v \in \{\text{true}, \text{false}\}$. Let $p$ be a $(\Sigma, \text{dom})$-ProbLog program $p$ with evidence $\text{evidence}(a_1,v_1), \ldots, \text{evidence}(a_n,v_n)$, and $p'$ be the program without the evidence statements. Furthermore, let $\text{POX}(p’)$ be the set of all $(\Sigma, \text{dom})$-structures $s$ complying with the evidence, i.e., the set of all states $s$ such that $a_i$ holds in $s$ iff $v_i = \text{true}$. Then, $[p](s)$, for a $(\Sigma, \text{dom})$-structure $s \in \text{POX}(p)$, is $[p’](s) \cdot \left(\sum_{s’ \in \text{POX}(p)} [p’](s’)^{-1}\right)$.

**Syntactic Sugar.** Following [60, 61, 73], we extend ProbLog programs with two additional constructs: probabilistic rules and annotated disjunctions. As shown in [60], these constructs are just syntactic sugar. A probabilistic rule is a ProbLog rule whose head is a probabilistic atom. The probabilistic rule $v_i : h \leftarrow l_1, \ldots, l_n$ can be encoded using the additional probabilistic atoms $v_i : \text{sw}(\_)$ and the rule $h \leftarrow l_1, \ldots, l_n, \text{sw}(\pi)$, where $\text{sw}$ is a fresh predicate symbol, $\pi$ is the tuple containing the variables in $\text{vars}(h) \cup \bigcup_{1 \leq i \leq n} \text{vars}(l_i)$, and $v_i : \text{sw}(\_)$ is a shorthand representing the fact that there is a probabilistic atom $v_i : \text{sw}(\pi)$ for each tuple $t \in \text{dom}[\pi]$.

An annotated disjunction $v_1 : a_1(\bar{t}_1); \ldots; v_n : a_n(\bar{t}_n)$, where $a_1(\bar{t}_1), \ldots, a_n(\bar{t}_n)$ are ground atoms and $\left(\sum_{1 \leq i \leq n} v_i\right) \leq 1$, denotes that $a_1, \ldots, a_n$ are mutually exclusive probabilistic events happening with probabilities $v_1, \ldots, v_n$. This annotated disjunction can be encoded as:

\[
p_1 : \text{sw}_1(\_)
\]
\[
\vdots
\]
\[
p_n : \text{sw}_n(\_)
\]
\[
a_1(\bar{t}_1) \leftarrow \text{sw}_1(\bar{t}_1)
\]
\[
a_2(\bar{t}_2) \leftarrow \neg \text{sw}_1(\bar{t}_1), \text{sw}_2(\bar{t}_2)
\]
\[
\vdots
\]
\[
a_n(\bar{t}_n) \leftarrow \neg \text{sw}_1(\bar{t}_1), \ldots, \neg \text{sw}_{n-1}(\bar{t}_{n-1}), \text{sw}_n(\bar{t}_n),
\]

where each $p_i$, for $1 \leq i \leq n$, is $v_i \cdot \left(1 - \sum_{1 \leq j \leq i} v_j\right)^{-1}$. Probabilistic rules can be easily extended to support annotated disjunctions in their heads.

**Example 4.6.** Let $\Sigma$ be a first-order signature with two predicate symbols $V$ and $W$, both with arity 1, $\text{dom}$ be the domain $\{a, b\}$, and $p$ be the program consisting of the facts $1/2: T(a)$ and $1/2: T(b)$, the annotate disjunction $1/2: W(a); 1/2: W(b)$, and the rule $1/2: T(x) \leftarrow W(x)$. The probability associated to each $(\Sigma, \text{dom})$-structure by the program $p$ is shown in the following table.

<table>
<thead>
<tr>
<th>$W$</th>
<th>$\emptyset$</th>
<th>${a}$</th>
<th>${b}$</th>
<th>${a, b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\frac{3}{32}$</td>
<td>$\frac{9}{64}$</td>
<td>$\frac{3}{32}$</td>
<td>0</td>
</tr>
<tr>
<td>$T$</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{9}{64}$</td>
<td>$\frac{1}{32}$</td>
<td>0</td>
</tr>
<tr>
<td>${a}$</td>
<td>$\frac{3}{32}$</td>
<td>$\frac{9}{64}$</td>
<td>$\frac{9}{32}$</td>
<td>0</td>
</tr>
<tr>
<td>${b}$</td>
<td>$\frac{3}{32}$</td>
<td>$\frac{9}{64}$</td>
<td>$\frac{3}{32}$</td>
<td>0</td>
</tr>
<tr>
<td>${a, b}$</td>
<td>$\frac{1}{32}$</td>
<td>$\frac{9}{64}$</td>
<td>$\frac{3}{32}$</td>
<td>0</td>
</tr>
</tbody>
</table>

The empty structure has probability $\frac{3}{32}$. The only grounded instance producing the empty database is the instance $i_1$ that does not contain ground atoms. Its probability is $\frac{3}{32}$ because the probability that $T(a)$ is not in $i_1$ is $\frac{3}{4}$, the probability that $T(b)$ is not in $i_1$ is $\frac{1}{2}$, and the probability that neither $W(a)$ nor $W(b)$ are in $i_1$ is $\frac{1}{4}$ and all these events are independent.

The probability of some structures is determined by more than one grounded instance. For example, the probability of the structure $s$ such that $s(T) = \{a, b\}$ and $s(W) = \{a\}$ is $\frac{9}{64}$. There are two grounded instances $i_2$ and $i_3$ that produce $s$. The instance $i_2$ has probability $\frac{1}{16}$ and it consists of the atoms $T(b), W(a), \text{sw}(a)$ and the rule $T(x) \leftarrow W(x), \text{sw}(x)$, whereas the instance $i_3$ has probability $\frac{1}{4}$ and it consists of the atoms $T(a), T(b), W(a)$ and the rule $T(x) \leftarrow W(x), \text{sw}(x)$. Note that before computing the ground instances, we translated probabilistic rules and annotated disjunctions into standard ProbLog rules.

**Medical Data.** We formalize the probability distribution from Example 4.4 as a ProbLog program. We use the database schema from Example 4.1 as the database schema for the ProbLog program.
We encode the template shown in Figure 4.2 using ground atoms: patient(Alice), patient(Bob), patient(Carl), smokes(Bob), smokes(Carl), father(Bob, Carl), and mother(Alice, Carl). Finally, we encode the probability distribution associated with the possible values of the cancer table using the following probabilistic rules:

\[
\begin{align*}
1/20: & \text{cancer}(x) \leftarrow \text{patient}(x) \\
5/19: & \text{cancer}(x) \leftarrow \text{smokes}(x) \\
3/14: & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{cancer}(x), \text{mother}(z, y), \neg \text{cancer}(z) \\
3/14: & \text{cancer}(y) \leftarrow \text{father}(x, y), \neg \text{cancer}(x), \text{mother}(z, y), \text{cancer}(z) \\
3/14: & \text{cancer}(y) \leftarrow \text{father}(x, y), \neg \text{cancer}(x), \text{mother}(z, y), \neg \text{cancer}(z)
\end{align*}
\]

The coefficients in the above example are derived from Section 4.1. For instance, the probability that a smoking patient whose parents are not in the cancer relation has cancer is 30%. The coefficient in the first rule is $1/20$ since each patient has a 5% probability of having cancer. The coefficient in the second rule is $5/19$, which is $(9/20 - 1/20) \cdot (1 - 1/20)^{-1}$, i.e., the probability that cancer(x) is derived from the second rule given that it has not been derived from the first rule. This ensures that the overall probability of deriving cancer(x) is $9/20$, i.e., 30%. The coefficients for the last two rules are derived analogously.

Informally, a probabilistic ground atom $1/2::\text{cancer}(\text{Bob})$ expresses that cancer(Bob) holds with a probability $1/2$. Similarly, the rule $1/20::\text{cancer}(x) \leftarrow \text{patient}(x)$ states that, for any x such that patient(x) holds, then cancer(x) can be derived with probability $1/20$. This program yields the probability distribution shown in Figure 4.4.

### 4.4.2 AtkLog’s Foundations

We first introduce belief programs, which formalize an attacker’s initial beliefs. Afterwards, we formalize AtkLog.

**Belief Programs.** A belief program formalizes an attacker’s beliefs as a probability distribution over the database states.

A database schema $D' = (\Sigma', \text{dom})$ extends a schema $D = (\Sigma, \text{dom})$ iff $\Sigma'$ contains all relation schemata in $\Sigma$. The extension $D'$ may extend $\Sigma$ with additional predicate symbols necessary to encode probabilistic dependencies. Given an extension $D'$, a $D'$-state $s'$ agrees with a $D$-state $s$ iff $s'(R) = s(R)$ for all $R$ in $D$. Given a $D$-state $s$, we denote by $\text{EXT}(s, D, D')$ the set of all $D'$-states that agree with $s$.

A $(\Sigma', \text{dom})$-ProbLog program $p$, where $D' = (\Sigma', \text{dom})$ extends $D$, is a belief program over $D$. The $D$-semantics of $p$ is $[p]_D = \lambda s \in \Omega_D. \sum_{s' \in \text{EXT}(s, D, D')} [p]_{D'}(s')$. Given a system configuration $C = (D, \Gamma)$, a belief program $p$ over $D$ complies with $C$ iff $[p]_{D'}$ is a $C$-probability distribution. With a slight abuse of notation, we lift the semantics of belief programs to sentences: $[p]_{D'} = \lambda \phi \in \text{RC}_{\text{base}}. \sum_{s' \in \Omega_{D'}(s'[\phi] = 1)} [p]_{D'}(s')$.

**AtkLog.** An AtkLog model specifies the initial beliefs of all users in $\mathcal{U}$ using belief programs.

Let $D$ be a database schema and $C = (D, \Gamma)$ be a system configuration. A C-AtkLog model $\text{ATK}$ is a function associating to each user $u \in U$, where $U \subseteq \mathcal{U}$ is a finite set of users, a belief program $p_u$ and to all users $u \in U \setminus U$ a belief program $p_0$, such that for all users $u \in U$, $[\text{ATK}(u)]_D$ complies with $C$ and for all database states $db \in \Omega_D$, $[\text{ATK}(u)]_D(db) > 0$, i.e., all database states satisfying the integrity constraints are possible. Informally, a C-AtkLog model associates a distinct belief program to each user in $U$, and it associates to each user in $U \setminus U$ the same belief program $p_0$.

Given a C-PDP $f$, a C-AtkLog model $\text{ATK}$ defines the $(C,f)$-attacker model $\lambda u \in U. [\text{ATK}(u)]_D$ that associates to each user $u \in U$ the probability distribution defined by the belief program $\text{ATK}(u)$. The semantics of this $(C,f)$-attacker model is: $\lambda u \in U. \lambda r \in \text{runs}(C,f). \lambda db \in \Omega_D. [\text{ATK}(u)]_D(db \mid [r]_u)$. Informally, given a C-AtkLog model $\text{ATK}$, a C-PDP $f$, and a user $u$, $u$’s belief in a database state $db$, given a run $r$, is obtained by conditioning the probability distribution defined by the belief program $\text{ATK}(u)$ given the set of database states corresponding to all runs $r' \sim_u r$.

### 4.5 Tractable Inference for ProbLog programs

Probabilistic inference in ProbLog is intractable in general. Its data complexity, i.e., the complexity of inference when only the programs’ probabilistic ground atoms are part of the input and the rules are considered fixed and not part of the input, is $\#P$-hard; see Appendix B. This limits the practical applicability of ProbLog (and AtkLog) for DBIC. To address this, we define acyclic ProbLog programs, a class of programs where the data complexity of inference is PTime.

Given a ProbLog program $p$, our inference algorithm consists of three steps: (1) we compute all of $p$’s derivations, (2) we compile these derivations into a Bayesian Network (BN) $bn$, and (3) we
perform the inference over bn. To ensure tractability, we leverage two key insights. First, we exploit
guarded negation [25] to develop a sound over-approximation, called the relaxed grounding, of all
derivations of a program that is independent of the presence (or absence) of the probabilistic atoms.
This ensures that whenever a ground atom can be derived from a program (for a possible assignment
to the probabilistic atoms), the atom is also part of this program’s relaxed grounding. This avoids
grounding p for each possible assignment to the probabilistic atoms. Second, we introduce syntactic
constraints (acyclicity) that ensure that bn is a forest of poly-trees. This ensures tractability since
inference for poly-tree BNSs can be performed in polynomial time in the network’s size [105].

We also precisely characterize the expressiveness of acyclic PROLOG programs. In this respect,
we prove that acyclic programs are as expressive as forests of poly-tree BNSs, one of the few classes
of BNSs with tractable inference.

As mentioned in Section 4.4, probabilistic rules and annotated disjunctions are just syntactic
sugar. Hence we consider PRoLOG programs consisting just of probabilistic ground
atoms and non-probabilistic rules. Note also that we treat ground atoms as rules with an empty body.

In the following, let (Σ, dom) be a database schema such that dom is a finite domain and p be a
(Σ, dom)-program. Without loss of generality, we assume that there are no distinct v₁ and v₂ such
that v₁:a(Σ) ∈ p and v₂:a(Σ) ∈ p.

4.5.1 Preliminaries

Negation-guarded Programs. A rule r is negation-guarded [25] iff all the variables occurring in
negative literals also occur in positive literals, namely for all negative literals l in body⁻(r), vars(l) ⊆
∪v∈body⁺(r) vars(l'). To illustrate, the rule C(x) ← A(x), ¬B(x) is negation-guarded, whereas the
rule C(x) ← A(x), ¬B(x, y) is not since the variable y does not occur in any positive literal. We
say that a program p is negation-guarded if all rules r ∈ p are. Observe that negation-guarded rules
ensure domain independence, i.e., the result of a negation-guarded program does not depend on the
domain dom.

Relaxed Grounding. The relaxed grounding of a program p is obtained by considering all prob-
abilistic atoms as certain and by grounding all positive literals. For all negation-guarded programs,
the relaxed grounding of p is a sound over-approximation of all possible derivations in p. Given a
program p and a rule r ∈ p, ground(p) denotes r’s relaxed grounding and ground(p, r) denotes the
set of r’s ground instances.

Formally, ground(p) and ground(p, r) are defined as follows. Let (Σ, dom) be a database schema
such that dom is a finite domain, p be a (Σ, dom)-negation-guarded PRoLOG program, and i ∈ N
be an natural number. The function ground(p, i) is inductively defined as follows: ground(p, 0) =
{a(Σ) | a(Σ) ∈ p ∪ 3v, v∈a(Σ) ∈ p}, and ground(p, i) = ground(p, i - 1) ∪ {kΘ | ∃v ∈ p. (Θ ∈
ASGN(r) ∧ h = head(r) ∧ vl ∈ body⁺(r). Θ ∈ ground(p, i - 1) ∧ ∃i, j, consistent(body(r, i),Θ, body(r,
j)Θ) = T)}, where consistent(a(Σ), ¬a(Σ)) = consistent(¬a(Σ), a(Σ)) = ⊤ and consistent(l, l') = ⊤
otherwise. The function ground(p, r, i), where p ∈ r, is inductively defined as follows: ground(p,
a(Σ), i) = {a(Σ)}, ground(p, v:a(Σ), i) = {a(Σ)}, ground(p, r, 0) = ∅, and ground(p, r, i) is the set
{hΘ ← l₁Θ, ..., lᵢΘ | Θ ∈ ASGN(r) ∧ ∃vl ∈ body⁺(r), l₁Θ ∈ ground(p, i - 1) ∧ ∃i, j, consistent(l₁Θ,
lᵢΘ) = T}, where r = h ← l₁, ..., lᵢ. Finally, the relaxed grounding of p, denoted by ground(p), is
the set ground(p) = ∪i∈N ground(p, i), whereas ground(p, r) = ∪i∈N ground(p, r, i).

Example 4.7. Let p be the program consisting of the facts 1/2::A(1), A(2), A(3), D(1), E(2),
F(1), O(1,2), and z/2::O(2,3), and the rules r₁ = B(x) ← A(x), D(x), r₂ = B(x) ← A(x), E(x),
and r₃ = B(y) ← B(x), ¬F(x), O(x, y). The relaxed grounding of p consists of the initial facts
together with B(1), B(2), and B(3), whereas ground(p, rᵢ) consists of B(2) ← B(1), ¬F(1), O(1,2)
and B(3) ← B(2), ¬F(2), O(2,3).

Dependancy and Ground Graphs. The dependency graph of a program p, denoted graph(p), is
the labelled directed graph having as nodes all the predicate symbols in p and having an edge a −→ b
iff there is a rule r such that a occurs in r’s literal in b’s body and b occurs in r’s head. Formally,
the p-dependency graph is the labelled directed graph (N, E) where N = Σ and E = {pred(body(r,
i)) −→ pred(head(r)) | r ∈ p ∧ 1 ≤ i ≤ |body(r)|}. Figure 4.6 depicts the dependency graph from
Example 4.7.

The ground graph of a program p is the graph obtained from its relaxed grounding. Hence, there
is an edge a −→ b from the ground atom a to the ground atom b iff there is a rule r and a ground
rule gr ∈ ground(p, r) such that body(gr, i) ∈ {a, ¬a} and head(gr) = b. Formally, the ground graph
of p, denoted ggr(p), is the labelled directed graph (N, E) where N = ground(p) ∪ {l ∈ body⁺(r') | 3r ∈ p.
ᵢ ∈ ground(p, r')} ∪ {a(Σ) | 3r ∈ p, rᵢ ∈ ground(p, r), l ∈ body⁺(r'), a = pred(l) ∧ T = args(l)} and
E = {a −→ b | r ∈ p ∧ gr ∈ ground(p, r) ∧ 1 ≤ i ≤ |body(r)| ∧ body(gr, i) ∈ {a, ¬a} ∧ head(gr) = b}. 
4.5. Tractable Inference for PROBLOG programs

We now introduce some terminology and notation for paths and cycles in graphs. Let $G = (N, E)$ be a directed graph. A directed path in $G$ is a sequence of edges $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$, such that for all $1 \leq i \leq m$, $n_i \rightarrow n_i' \in E$ and if $i \neq m$, $n_i' = n_{i+1}$. We observe that a path $n_1 \rightarrow n_1' \rightarrow n_2 \rightarrow \ldots \rightarrow n_{m-1} \rightarrow n_m$ is simply as $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$ simply. Furthermore, the empty path is just the empty sequence $\epsilon$. A directed cycle in $G$ is a directed path $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m \rightarrow n_1$ such that $n_1 = n_m$. We say that a directed cycle $C$ is simple if each edge occurs at most once in $C$.

Given a directed path $P = n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$, we denote by $\text{start}(P)$ the node $n_1$ and by $\text{end}(P)$ the node $n_m$. Two directed paths $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$ and $n'_1 \rightarrow n'_2 \rightarrow \ldots \rightarrow n'_m$ are head-connected (respectively tail-connected) if $n_1 = n'_1$ (respectively $n_m = n'_m$). Furthermore, a directed path $P_1$ is connected with a directed path $P_2$ if $\text{end}(P_1) = \text{end}(P_2)$, $\text{start}(P_1) = \text{start}(P_2)$, or $\text{end}(P_1) = \text{start}(P_2)$. Finally, given a node $n$, we denote by $\text{reach}(n)$ the set of all nodes $n'$ such that there is a direct path from $n$ to $n'$.

A reversed path is a sequence of reversed edges $n_1 \leftarrow n_1' \rightarrow n_2 \leftarrow n_2' \rightarrow \ldots \rightarrow n_{m-1} \rightarrow n_m$, such that $n_m \rightarrow n_m' \rightarrow \ldots \rightarrow n_2 \rightarrow n_2' \rightarrow n_1$ is a directed path. Given a directed path $P$, we denote by $P^{-1}$ the corresponding reversed path. Similarly, given a reversed path $P$, we denote by $P^{+1}$ the corresponding directed path.

An undirected path in $G$ is a sequence of undirected edges $n_1 \rightarrow n_1' \rightarrow n_2 \rightarrow n_2' \rightarrow \ldots \rightarrow n_m$, such that for all $1 \leq i \leq m$, $n_i \rightarrow n_i' \in E \lor n_i' \rightarrow n_i \in E$ and if $i \neq m$, $n_i' = n_{i+1}$. Again, we denote a path $n_1 \rightarrow n_1' \rightarrow n_2 \rightarrow n_2' \rightarrow \ldots \rightarrow n_m$, such that $n_1 = n_m$. An undirected cycle in $G$ is an undirected path $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$, such that $n_1 = n_m$. Observe that any undirected cycle $C$ can be seen as a sequence of directed and reversed paths $D_1, \ldots, D_m$. Observe also that a directed path (respectively cycle) is also an undirected path (respectively cycle). We say that two (directed or undirected) cycles are equivalent if they are the same cycle.

Propagation Maps. We use propagation maps to track how information flows inside rules. Given a rule $r$ and a literal $l \in \text{body}(r)$, the $(r,l)$-vertical map is the mapping $\mu$ from $[1, \ldots, |l|]$ to $[1, \ldots, |\text{head}(r)|]$ such that $\mu(i) = j$ if $\text{args}(l)(i) = \text{args}(\text{head}(r))(j)$ and $\text{args}(l)(i) \in \text{Var}$. Given a rule
r and literals l and l’ in r’s body, the (r, l, l’)-horizontal map is the mapping µ from \{1, \ldots, |l|\} to \{1, \ldots, |l’|\} such that µ(i) = j iff args(l)(i) = args(l’)(j) and args(l)(i) ∈ Var.

We say that a path links to a literal l if information flows along the rules to l. This can be formalized by posing constraints on the mapping obtained by combining horizontal and vertical maps along the path. Formally, given a literal l and a mapping ν : N → N, a directed path \(pr_1 \rightarrow \cdots \rightarrow pr_n\) \(\nu\)-downward links to l iff there is a \(0 \leq j < n - 1\) such that the function \(\mu := \mu_j \circ \mu_{j+1} \circ \cdots \circ \mu_0\) satisfies \(\mu(k) = \nu(k)\) for all k for which \(\nu(k)\) is defined, where for \(1 \leq h \leq j\), \(\mu_h\) is the \((r_h, \text{body}(r_h, i_h))\)-vertical map, and \(\mu^\prime\) is the horizontal map connecting \(\text{body}(r_{j+1}, i_{j+1})\) with \(l\).

Similarly, a directed path \(pr_1 \rightarrow \cdots \rightarrow pr_n\) \(\nu\)-upward links to l iff there is an \(1 \leq j \leq n - 1\) such that the function \(\mu := \mu^{-1}_j \circ \mu^{-1}_{j+1} \circ \cdots \circ \mu^{-1}_0\) satisfies \(\mu(k) = \nu^{-1}(k)\) for all k for which \(\nu(k)\) is defined, where \(\mu_h\) is the \((r_h, \text{body}(r_h, i_h))\)-vertical map, for \(j < h \leq n - 1\), and \(\mu^\prime\) is the \((r_j, l)\)-vertical map. Given a predicate symbol \(a\), a path \(P\) \(\nu\)-upward links to a (respectively \(\nu\)-downward links to a) if there is an atom \(a(\overline{x})\) such that \(P\) \(\nu\)-upward links (respectively \(\nu\)-downward links) to \(a(\overline{x})\).

Example 4.8. The horizontal map connecting \(A(x)\) and \(D(x)\) in \(r_n\), i.e., the \((r_n, A(x), D(x))\)-horizontal map, is \(\{1 \rightarrow 1\}\). The horizontal map connecting \(A(x)\) and \(E(x)\) in \(r_3\) is \(\{1 \rightarrow 1\}\) as well. Hence, the path \(A \rightarrow r_{n-1} B\rightarrow B\) downward links to D and the path \(A \rightarrow r_{k-1} B\rightarrow B\) downward links to E for the mapping \(\{1 \rightarrow 1\}\). Furthermore, the path \(B \rightarrow r_{n-1} B\rightarrow B\) downward links to O for \(\{1 \rightarrow 1\}\) since \((r_c, B(x), O(x, y))\)-horizontal map is \(\{1 \rightarrow 1\}\). Finally, the path \(B \rightarrow r_{n-1} B\rightarrow B\) upward links to O for \(\{2 \rightarrow 1\}\) since the \((r_c, O(x, y))\)-vertical map is \(\{2 \rightarrow 1\}\).

4.5.2 Annotations

Annotations represent properties of the relations induced by the program \(p\), and they are syntactically derived by analyzing \(p\)’s ground atoms and rules. We now illustrate how to derive three kinds of annotations: disjointness, ordering, and uniqueness annotations. In this section, whenever we use the notation reach(·) (defined in Section 4.5.1), we always refer to \(p\)’s dependency graph.

Disjointness annotations. Let \(a, a’ \in \Sigma\) be two predicate symbols such that \(|a| = |a’|\). A disjointness annotation \(\text{DIS}(a, a’)\) represents that the relations induced by \(a\) and \(a’\) (given \(p\)’s relaxed grounding) are disjoint. Informally, \(\text{DIS}(a, a’)\) can be derived from \(p\) iff (1) no rules in \(p\) contain \(a\) or \(a’\) in their heads, and there is no \(\overline{x} \in \text{dom}^\overline{a}\) where both \(a(\overline{x})\) and \(a’(\overline{x})\) appear as (possibly probabilistic) ground atoms in \(p\), or (2) \(a\) and \(a’\) can be derived only from other disjoint predicate symbols. Hence, the relations induced by \(a\) and \(a’\) are disjoint. Formally, the annotation \(\text{DIS}(a, a’)\) can be derived from \(p\) iff one of the following conditions hold:

1. There are no rules \(r \in p\) such that \(\text{body}(r) \neq \emptyset\) and \(\text{pred}(\text{head}(r)) = a\) or \(\text{pred}(\text{head}(r)) = a’\), and the sets \(\{\overline{x} \mid a(\overline{x}) \in p\} \cup \{\overline{x} \mid a’(\overline{x}) \in p\}\) are disjoint.
2. There are no rules \(r \in p\) such that \(\text{body}(r) \neq \emptyset\) and \(\text{pred}(\text{head}(r)) = a\), the annotation \(\text{DIS}(a, b)\) can be derived from \(p\), \(b \notin \text{reach}(a)\), \(b \neq a’\), and for all rules \(r \in p\) such that \(\text{head}(r) = a(\overline{x})\), \(b(\overline{x}) \in \text{body}(r)\).
3. There are no rules \(r \in p\) such that \(\text{body}(r) \neq \emptyset\) and \(\text{pred}(\text{head}(r)) = a’\), the annotation \(\text{DIS}(b, b’)\) can be derived from \(p\), \(b \notin \text{reach}(a)\), \(b \neq a’\), and for all rules \(r \in p\) such that \(\text{head}(r) = a(\overline{x})\), \(b(\overline{x}) \in \text{body}(r)\).
4. The annotation \(\text{DIS}(b, b’)\) can be derived from \(p\), \(\{b, b’\} \cap \text{reach}(a) = \emptyset\), \(b \neq a\), \(b’ \neq a’\), and for all rules \(r \in p\) such that \(\text{head}(r) = a(\overline{x})\), \(b(\overline{x}) \in \text{body}(r)\) and if \(\text{head}(r) = a(\overline{x})\), \(b’(\overline{x}) \in \text{body}(r)\).

Ordering annotations. Let \(n \in N\) and \(A \subseteq \Sigma\) be a set of predicate symbols such that \(|a| = 2n\) for all \(a \in A\). An ordering annotation \(\text{ORD}(A)\) represents that the transitive closure of the union of the relations induced by predicates in \(A\) given \(p\)’s relaxed grounding is a strict partial order over \(\text{dom}^\overline{a}\). The annotation \(\text{ORD}(A)\) can be derived from the program \(p\) iff (1) there is no rule \(r \in p\) that contains any of the predicates in \(A\) in its head and the transitive closure of \(\bigcup_{a \in A} \{\langle v_1, \ldots, v_n \rangle \mid |k| = |a(v_1, \ldots, v_n) \in p\}\) is a strict partial order over \(\text{dom}^\overline{a}\), or (2) the predicates in \(A\) are derived from other predicates for which we can derive an ordering annotation. Hence, the closure of the relation \(\bigcup_{a \in A} \{\langle v_1, \ldots, v_n \rangle \mid a(v_1, \ldots, v_n) \in \text{ground}(p)\} \cup \text{ground}(p)\) induced by the relaxed grounding is a strict partial order. Formally, the annotation \(\text{ORD}(A)\) can be derived from the program \(p\) iff one of the following conditions hold:

1. For all \(pr \in A\), there is no rule \(r \in p\) such that \(\text{pred}(\text{head}(r)) = pr\) and \(\text{body}(r) \neq \emptyset\), and the transitive closure of the relation \(R = \bigcup_{pr \in A} \{\langle \overline{r}, \overline{v} \rangle \mid |\overline{r}| \in p. (\langle \text{head}(r) = pr \rangle \lor \exists \overline{v’}. \text{head}(r) = \overline{v’}. \langle \overline{r}, \overline{v} \rangle \land \text{body}(r) = \emptyset \} \land |\overline{r}| = |pr|/2\}\) is strict partial order.
2. The annotation \(\text{ORD}(A)\) can be derived from \(p\) and there are two distinct predicate symbols \(pr, pr’ \in A\) such that (a) \(|pr| = |pr’|\), (b) \(A = (A' \setminus \{pr’\}) \cup \{pr\}\), (c) \(pr’ \notin \text{dom}^\overline{a}\).
4.5. Tractable Inference for ProbLog programs

\[ \bigcup_{\sigma \in \mathcal{A}} \text{reach}(\sigma), \text{ and (d) for all rules } r \text{ in } p \text{ such that } \text{pred}(\text{head}(r)) = pr, \text{ there are sequences of variables } \mathcal{T} \text{ and } \mathcal{Y} \text{ such that } \text{head}(r) = pr(\mathcal{T}, \mathcal{Y}), pr'((\mathcal{T}, \mathcal{Y})) \in \text{body}(r), \text{ and } |\mathcal{T}| = |\mathcal{Y}|. \]

Uniqueness annotations. Let \( a \in \Sigma \) be a predicate symbol and \( K \subseteq \{1, \ldots, |a|\} \). A uniqueness annotation \( \text{UNQ}(a, K) \) represents that the attributes in \( K \) are a primary key for the relation induced by \( a \) given the relaxed grounding. We say that \( \text{UNQ}(a, K) \) can be derived from a program \( p \) iff no rule contains \( a \) in its head and for all \( \mathcal{T}, \mathcal{Y} \in \text{dom}^{\mathcal{A}} \), if (1) \( \mathcal{Y}(i) = \mathcal{T}(i) \) for all \( i \in K \), and (2) there are \( k \) and \( k' \) such that \( a(\mathcal{T}) \in p \) and \( k':=a(\mathcal{Y}) \in p \), then \( \mathcal{T} = \mathcal{Y} \). This ensures that whenever \( a(\mathcal{T}) \), \( a(\mathcal{Y}) \in \text{ground}(p) \) and \( \mathcal{Y}(i) = \mathcal{T}(i) \) for all \( i \in K \), then \( \mathcal{T} = \mathcal{Y} \). Alternatively, uniqueness annotations can be derived from other uniqueness annotations under appropriate conditions. Formally, the annotation \( \text{UNQ}(pr, K) \) can be derived from \( p \) iff one of the following conditions hold:

1. There are no rules \( r \in p \) such that \( \text{body}(r) \neq \emptyset \) and \( \text{pred}(\text{head}(r)) = pr \), and for all \( \mathcal{T}, \mathcal{Y} \in \{\mathcal{T} \mid pr(\mathcal{T}) \in p \lor \exists \mathcal{V}: pr(\mathcal{V}) \in p\} \), if \( \mathcal{T}(i) = \mathcal{Y}(i) \) for all \( i \in K \), then \( \mathcal{T} = \mathcal{Y} \).
2. Both the following conditions hold:
   (a) For each rule \( r \in p \) such that \( \text{head}(r) = pr(\mathcal{T}, \mathcal{Y}) \) and \( \text{body}(r) \neq \emptyset \), there is a mapping \( \mu' \) associating to each predicate \( pr' \) not in \( \text{reach}(pr) \) a set \( K \) such that (a) the annotation \( \text{UNQ}(pr', \mu(pr')) \) can be derived from \( p \), and (b) \( V \subseteq \bigcup_{\mathcal{L} \in \text{bound}(\text{head}(r)), K, \text{body}(r), \mu'} \text{vars}(l) \) where \( V = \{\mathcal{T}(i) \mid i \notin K \land \mathcal{T}(i) \in \text{Var}\} \), \( u(\mathcal{Y}, K) = \{\mathcal{Y}(i) \mid i \in K\} \), and \( \text{bound}(h, K, L, \mu') = \bigcup_{\mathcal{L} \in \text{bound}(\text{head}(r)), \mu, \mu'} \{\{\mathcal{L}\}\} \).
   (b) For all rules \( r_1, r_2 \in p \), if \( r_1 \neq r_2, \text{pred}(\text{head}(r_1)) = \text{pred}(\text{head}(r_2)) = pr \), and \( K \neq \{1, \ldots, |pr|\} \), then there is a value \( i \in K \) such that \( \text{args}(\text{head}(r_1))(i) \in \text{dom}, \text{args}(\text{head}(r_2))(i) \in \text{dom}, \text{and args}(\text{head}(r_1))(i) \neq \text{args}(\text{head}(r_2))(i) \).

Templates. A \( \Sigma \)-template \( T \) is a set of annotations.

Example 4.9. We can derive \( \text{DIS}(D, E) \) from the program \( P \) since no rule generates facts for \( D \) and \( E \) and the relations defined by the ground atoms are \( \{1\} \) and \( \{2\} \). We can also derive \( \text{ORD}(\{O\}) \) since the relation defined by \( O \)'s ground atoms is \( \{(1, 2), (2, 3)\} \), whose transitive closure is a strict partial order. Finally, we can derive \( \text{UNQ}(O, \{1\}), \text{UNQ}(O, \{2\}), \) and \( \text{UNQ}(O, \{1, 2\}) \) since both arguments of \( O \) uniquely identify the tuples in the relation induced by \( O \).

4.5.3 Acyclic ProbLog programs

A sufficient condition for tractable inference is that \( p \)'s ground graph is a forest of poly-trees. This requires that \( p \)'s ground graph neither contains directed nor undirected cycles, or, equivalently, the undirected version of \( p \)'s ground graph is acyclic. To illustrate, the ground graph in Figure 4.7 is a poly-tree. The key insight here is that a cycle among ground atoms is caused by a (directed or undirected) cycle among \( p \)'s predicate symbols. In a nutshell, acyclicity requires that all possible cycles in \( \text{graph}(p) \) are guarded. This ensures that cycles in \( \text{graph}(p) \) do not lead to cycles in the ground graph. Additionally, acyclicity requires that programs are negation-guarded. This ensures that the relaxed grounding and the ground graph are well-defined.

Unsafe structures. An unsafe structure models a part of the dependency graph that may introduce cycles in the ground graph. We define directed and undirected unsafe structures which may respectively introduce directed and undirected cycles.

A directed unsafe structure \( \text{graph}(p) \) in \( \text{graph}(p) \) is a directed cycle \( C \) in \( \text{graph}(p) \). We say that a directed unsafe structure \( C \) covers \( \text{graph}(p) \) if \( C \) is equivalent to \( C' \).

An undirected unsafe structure \( \text{graph}(p) \) is quadruple \( \langle D_1, D_2, D_3, U \rangle \) such that (1) \( D_1, D_2, \) and \( D_3 \) are directed paths whereas \( U \) is an undirected path, (2) \( D_1 \) and \( D_2 \) start from the same node, (3) \( D_2 \) and \( D_3 \) end in the same node, and (4) \( D_1 \cdot U \cdot D_3 \cdot D_2 \) is an undirected cycle in \( \text{graph}(p) \). We say that an unsafe structure \( \langle D_1, D_2, D_3, U \rangle \) covers an undirected cycle \( U' \) in \( \text{graph}(p) \) if \( D_1 \cdot U \cdot D_3 \cdot D_2 \) is equivalent to \( U' \).

Example 4.10. The cycle introduced by the rule \( r_a \) is captured by the directed unsafe structure \( B \xrightarrow{r_a} B \), while the cycle introduced by \( r_a \) and \( r_b \) is captured by the structure \( \langle A \xrightarrow{r_a} \epsilon, B, A \xrightarrow{r_a} \epsilon, B \rangle \), where \( \epsilon \) denotes the empty path.

Connected Rules. A connected rule \( r \) ensures that a grounding of \( r \) is fully determined either by the assignment to the head’s variables or to the variables of any literal in \( r \)'s body. Formally, a strongly connected rule \( r \) guarantees that for any two groundings \( gr', gr'' \) of \( r \), if \( \text{head}(gr') = \text{head}(gr'') \), then \( gr' = gr'' \). In contrast, a weakly connected rule \( r \) guarantees that for any two groundings \( gr', gr'' \) of \( r \), if \( \text{body}(gr', i) = \text{body}(gr'', i) \) for some \( i \), then \( gr' = gr'' \). This is done by exploiting uniqueness annotations and the rule’s structure.

Before formalizing connected rules, we introduce join trees. A join tree represents how multiple predicate symbols in a rule share variables. In the following, let \( r \) be a rule and \( T \) be a template. A
join tree for a rule \( r \) is a rooted labelled tree \( \langle N, E, root, \lambda \rangle \), where \( N \subseteq body(r) \), \( E \) is a set of edges (i.e., unordered pairs over \( N^2 \)), \( root \in N \) is the tree’s root, and \( \lambda \) is the labelling function. Moreover, we require that for all \( n, n' \in N \), if \( n \neq n' \) and \( (n, n') \in E \), then \( \lambda(n, n') = \vars(n) \cap \vars(n') \) and \( \lambda(n, n') \neq \emptyset \). A join tree \( \langle N, E, root, \lambda \rangle \) covers a literal \( l \) if \( l \in N \). Given a join tree \( J = \langle N, E, root, \lambda \rangle \) and a node \( n \in N \), the support of \( n \), denoted \( support(n) \), is the set \( \vars(\text{head}(r)) \cap \{ x \mid (x = c) \in \text{csb}(r) \land c \in \text{dom} \} \cup \{ \vars(n') \mid n' \in \text{anc}(J, n) \} \), where \( \text{anc}(J, n) \) is the set of \( n \)'s ancestors in \( J \), i.e., the set of all nodes (different from \( n \)) on the path from \( root \) to \( n \). A join tree \( J = \langle N, E, root, \lambda \rangle \) is \( T \)-strongly connected iff for all positive literals \( l \in N \), there is a set \( K \subseteq \{ i \mid x \in arg(l) \land x \in support(l) \} \) such that \( UNQ(\text{pred}(l), K) \in T \) and for all negative literals \( l \in N \), \( vars(l) \subseteq support(l) \). In contrast, a join tree \( \langle N, E, root, \lambda \rangle \) is \( T \)-weakly connected iff for all \((\alpha (\pi), \alpha (\pi')) \in E\), there are \( K \subseteq \{ i \mid \pi(i) \in L \} \) and \( K' \subseteq \{ i \mid \pi'(i) \in L \} \) such that \( UNQ(\alpha (K), UNQ(\alpha (K'), W)) \in T \), where \( W = \lambda(\pi(\alpha (\pi)), \alpha (\pi')) \).

We now formalize strongly and weakly connected rules. A rule \( r \) is \( T \)-strongly connected iff there exist \( T \)-strongly connected join trees \( J_1, \ldots, J_k \) that cover all literals in \( r \)'s body. This guarantees that for any two groundings \( gr', gr'' \) of \( r \), if \( head(gr') = head(gr'') \), then \( gr' = gr'' \).

Given a rule \( r \), a set of literals \( L \), and a template \( \mathcal{T} \), a literal \( l \in body(r) \) is \( (r, T, L) \)-strictly guarded iff \( 1 \) \( \vars(l) \subseteq \bigcup_{l' \in L \cap body^+(r)} \vars(l') \cup \{ x \mid (x = c) \in \text{csbr}(r) \land c \in \text{dom} \} \), and \( 2 \) there is a positive literal \( a(\pi) \in L \) and an annotation \( UNQ(a, K) \in T \) such that \( \{ \pi(i) \mid i \in K \} \subseteq vars(l) \).

A rule \( r \) is \( T \)-weakly connected for \( \mathcal{T} \) iff there exists a \( T \)-weakly connected join tree \( J = \langle N, E, root, \lambda \rangle \) such that \( N \subseteq body^+(r) \), and all literals in \( body(r) \setminus N \) are \( (r, T, N) \)-strictly guarded. This guarantees that for any two groundings \( gr', gr'' \) of \( r \), if \( body(gr', body(gr'') \) for some \( i \), then \( gr' = gr'' \).

**Example 4.11.** Let \( \mathcal{T} \) be the template from Example 4.9. The rule \( r_c := B(y) \leftarrow B(x), \neg F(x), O(x, y) \) is \( T \)-strongly connected. Indeed, the join tree having \( O(x, y) \) as root and \( B(x) \) and \( \neg F(x) \) as leaves is such that \( 1 \) \( \vars(O(x, y)) \subseteq \bigcup_{l' \in L \cap body^+(r)} \vars(l') \cup \{ x \mid (x = c) \in \text{csbr}(r) \land c \in \text{dom} \} \), and \( 2 \) there is a positive literal \( a(\pi) \in L \) and an annotation \( UNQ(O, \{ 2 \}) \in T \) such that \( \{ \pi(i) \mid i \in K \} \subseteq vars(O(x, y)) \).

**Guarded undirected structures.** Guarded undirected structures ensure that undirected cycles in the dependency graph do not correspond to undirected cycles in the ground graph by exploiting disjointness and unification annotations. Formally, an undirected structure \( \langle D_1, D_2, D_3, D_4 \rangle \) is guarded by a template \( \mathcal{T} \) if and only if \( D_1 \) and \( D_2 \) are \( \mathcal{T} \)-head-guarded and \( D_3 \) and \( D_4 \) are \( \mathcal{T} \)-tail-guarded.

A pair of non-empty paths \( P_1, P_2 \) sharing the same initial node \( a \) is \( \mathcal{T} \)-head guarded if \( 1 \) \( P_1 = P_2 \), all rules in \( P_1 \) are weakly connected for \( \mathcal{T} \), and \( 2 \) if \( P_1 \neq P_2 \), then there is an annotation \( DIS(pr, pr') \in T \), \( K \subseteq \{ 1, \ldots, |a| \} \), and a bijection \( \nu : K \rightarrow \{ 1, \ldots, |pr| \} \) such that \( P_1 \) is a downward link to \( pr \) and \( P_2 \) is a downward link to \( pr' \). Given two ground paths \( P_1' \) and \( P_2' \) corresponding to \( P_1 \) and \( P_2 \), the first condition ensures that \( P_1' = P_2' \) whereas the second ensures that \( P_1' \neq P_2' \) are not in the ground graph.

Similarly, a pair of non-empty paths \( P_1, P_2 \) sharing the same final node \( a \) is \( \mathcal{T} \)-tail guarded if \( 1 \) \( P_1 = P_2 \), all rules in \( P_1 \) are strongly connected for \( \mathcal{T} \), and \( 2 \) if \( P_1 \neq P_2 \), there is an annotation \( DIS(pr, pr') \in T \), \( K \subseteq \{ 1, \ldots, |a| \} \), and a bijection \( \nu : K \rightarrow \{ 1, \ldots, |pr| \} \), such that \( P_1 \) \( \nu \)-upward links to \( pr \) and \( P_2 \) \( \nu \)-upward links to \( pr' \).

**Example 4.12.** The only non-trivially guarded undirected cycle in the graph from Figure 4.6 is the one represented by the undirected unsafe structure \( (A \overset{r_1}{\longrightarrow} B, A \overset{r_2}{\longrightarrow} B, \varepsilon, \varepsilon) \). The structure is guarded since the paths \( A \overset{r_1}{\longrightarrow} B \) and \( A \overset{r_2}{\longrightarrow} B \) are head guarded by \( DIS(D, E) \). Indeed, for the same ground atom \( A(v) \), for some \( v \in \{ 1, 2, 3 \} \), only one of \( r_1 \) and \( r_2 \) can be applied since \( D \) and \( E \) are disjoint.

**Guarded directed structures.** Guarded directed structures exploit ordering annotations to ensure that directed cycles in the dependency graph do not correspond to directed cycles among ground atoms. A directed unsafe structure \( pr_1 \overset{r_1}{\longrightarrow} \ldots \overset{r_m}{\longrightarrow} pr_j \) is guarded by a template \( \mathcal{T} \) if there is an annotation \( ORD(O) \in T \), integers \( 1 \leq y_1 < y_2 < \ldots < y_k = n \), atoms \( a_1(\pi_1), \ldots, a_n(\pi_n) \) (where \( a_1, \ldots, a_n \in O \)), an empty set \( \{ 1, \ldots, |pr_1| \} \), and a bijection \( \nu : K \rightarrow \{ 1, \ldots, |pr| \} \) such that for each \( 0 \leq k < e \), \( 1 \) \( pr_{y_k} \overset{r_{y_k+1}}{\longrightarrow} \ldots \overset{r_{y_{k+1}-1}}{\longrightarrow} pr_{y_{k+1}} \) \( \nu \)-downward connects to \( a_{k+1}(\pi_{k+1}) \), and \( 2 \) \( pr_{y_{k+1}} \overset{r_{y_{k+1}+1}}{\longrightarrow} \ldots \overset{r_{y_{k+1}-1}}{\longrightarrow} pr_{y_{k+1}} \) \( \nu' \)-upward connects to \( a_{k+1}(\pi_{k+1}) \), where \( \nu'(i) = \nu(x) + \lfloor |\pi|/2 \rfloor \) for all \( 1 \leq i \leq |\pi|/2 \), and \( y_0 = 1 \).

**Example 4.13.** The directed unsafe structure \( B \overset{r_1}{\longrightarrow} B \) is guarded by \( ORD(\{ O \}) \) in the template from Example 4.9. Indeed, the strict partial order induced by \( O \) breaks the cycle among ground atoms belonging to \( B \). In particular, the path \( B \overset{r_1}{\longrightarrow} B \) both downward links and upward links to \( O(x, y) \); see Example 4.8.
4.5. Tractable Inference for ProbLog programs

A (1) D (1) E (1) B (1) A (2) E (2) F (1) O (1, 2) B (2) A (3) E (3) F (2) O (2, 3) B (3) ra,r′ a,1 rb,r′ b,1 ra,r′ a,2 rb,r′ b,2 ra,r′ a,3 rb,r′ b,3 rc,r′ c,1 rc,r′ c,2 rc,r′ c,3

Figure 4.8: Ground graph for the program in Example 4.7 extended with the atom E (1). The additional edges and nodes are represented using dashed lines. The ground rules r′ a, r′ b, r′ c are as in Figure 4.7, and r′ = B (1) ← A (1), E (1).

X[B(2)] X[F(2)] X[O(2,3)]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>⊥</td>
<td>⊥</td>
<td>0 1</td>
</tr>
<tr>
<td>T</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>0 1</td>
</tr>
<tr>
<td>⊥</td>
<td>T</td>
<td>⊥</td>
<td>⊥</td>
<td>1 0</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>T</td>
<td>⊥</td>
<td>0 1</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>T</td>
<td>0 1</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>0 1</td>
</tr>
</tbody>
</table>

Selected CPTs.

X[rc,r′ c,B(3)]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>⊤</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>⊤</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>⊤</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

Figure 4.9: Portion of the resulting BN for the atoms B (2), F (2), and O (2,3), the rule r c = B (y) ← B (x), ¬F (x), O (x, y), and the ground rule r′ c = B (3) ← B (2), ¬F (2), O (2,3), together with the CPT encoding r′ c’s semantics.

Acyclic Programs. Let p be a negation-guarded program and T be the template containing all annotations that can be derived from p. We say that p is acyclic if (a) for all undirected cycles U in graph(p) that are not directed cycles, there is a T-guarded undirected unsafe structure that covers U, and (b) for all directed cycles C in graph(p), there is a T-guarded directed unsafe structure that covers C. This ensures the absence of cycles in the ground graph.

Proposition 4.1. Let p be a ProbLog program. If p is acyclic, then the ground graph of p is a forest of poly-trees.

Example 4.14. The program p from Example 4.7 is acyclic. This is reflected in the ground graph in Figure 4.7. The program q = p ∪ {E (1)}}, however, is not acyclic: we cannot derive DIS(D, E) from q and the undirected unsafe structure ⟨A ra,1 B, A rb,1 B, e, c⟩ is not guarded. As expected, q’s ground graph contains an undirected cycle between A (1) and B (1), as shown in Figure 4.8. ■

Expressiveness. Acyclicity trades some of ProbLog’s expressiveness for a tractable inference procedure. Acyclic programs, nevertheless, can encode many relevant probabilistic models.

Proposition 4.2. Any forest of poly-tree BNs can be represented as an acyclic ProbLog program.

To clarify this proposition’s scope, observe that poly-tree BNs are one of the few classes of BNs with tractable inference procedures. From Proposition 4.2, it follows that a large class of probabilistic models with tractable inference can be represented as acyclic programs. This supports our thesis that our syntactic constraints are not overly restrictive. In Section 4.9, we relax acyclicity to support a limited form of annotated disjunctions and rules sharing a part of their bodies, which are needed to encode the example from Section 4.4 and for Proposition 4.2. The proofs of all our results are in Appendix B.
4.5.4 Inference Engine

Our inference algorithm for acyclic PROBLOG programs consists of three steps: (1) we compute the relaxed grounding of \( p \) (see Section 4.5.1), (2) we compile the relaxed grounding into a Bayesian Network (BN), and (3) we perform the inference using standard algorithms for poly-tree Bayesian Networks [105].

Encoding as BNs. We compile the relaxed grounding \( ground(p) \) into the BN \( bn(p) \). The boolean random variables in \( bn(p) \) are encoded as follows: (a) for each atom \( a \) in \( ground(p) \) and ground literal \( \neg a \) occurring in any \( gr \in \bigcup_{r \in \text{poly-tree}} ground(p, r) \), there is a random variable \( X[a] \), (b) for each rule \( r \in p \) and each ground atom \( a \) such that there is a ground rule \( gr \in ground(p, r) \) satisfying \( a = \text{head}(gr) \), there is a random variable \( X[r, a] \), and (c) for each rule \( r \in p \), each ground atom \( a \), and each ground rule \( gr \in ground(p, r) \) such that \( a = \text{head}(gr) \), there is a random variable \( X[r, gr, a] \).

The edges in \( bn(p) \) are as follows: (a) for each ground atom \( a \), rule \( r \), and ground rule \( gr \), there is an edge from \( X[r, gr, a] \) to \( X[r, a] \) and an edge from \( X[r, a] \) to \( X[a] \), and (b) for each ground atoms \( a \) and \( b \), rule \( r \), and ground rule \( gr \), there is an edge from \( X[b] \) to \( X[r, gr, a] \) if \( b \) occurs in \( gr \)'s body as a positive or negative literal.

Finally, the Conditional Probability Tables (CPTs) of the variables in \( bn(p) \) are as follows. The CPT of variables of the form \( X[a] \) and \( X[r, a] \) is just the OR of the values of their parents, i.e., the value is \( \top \) with probability 1 iff at least one of the parents has value \( \top \). For variables of the form \( X[r, gr, a] \) such that \( \text{body}(r) \neq \emptyset \), the variable's CPT encode the semantics of the rule \( r \), i.e., the value of \( X[r, gr, a] \) is \( \top \) with probability 1 iff all positive literals have value \( \top \) and all negative literals have value \( \bot \). In contrast, for variables of the form \( X[r, gr, a] \) such that \( \text{body}(r) = \emptyset \), the variable has value \( \top \) with probability \( v \) and \( \bot \) with probability \( 1 - v \), where \( r \) is of the form \( v :: a \) (if \( r = a \), then \( v = 1 \)).

To ensure that the size of the CPT of variables of the form \( X[r, a] \) is independent of the size of the relaxed grounding, instead of directly connecting variables of the form \( X[r, gr, a] \) with \( X[r, a] \), we construct a binary tree of auxiliary variables where the leaves are all variables of the form \( X[r, gr, a] \) and the root is the variable \( X[r, a] \). Figure 4.9 depicts a portion of the BN for the program in Example 4.7.

Complexity. We now introduce the main result of this section.

Theorem 4.1. The data complexity of inference for acyclic PROBLOG programs is \( \text{Ptime} \).

This follows from (1) the relaxed grounding and the encoding can be computed in \( \text{Ptime} \) in terms of data complexity, (2) the encoding ensures that, for acyclic programs, the resulting Bayesian Network is a forest of poly-trees, and (3) inference algorithms for poly-tree BNs [105] run in polynomial time in the BN's size. In Section 4.9, we extend our encoding to handle additional features such as (a) a list of (b) annotated disjunctions, whereas in Appendix B we prove its correctness and complexity.

4.6 Angerona

ANGERONA is a DBIC mechanism that provably secures databases against probabilistic inferences. ANGERONA is parametrized by an ATKLOG model representing the attacker’s capabilities and it leverages PROBLOG’s inference capabilities.

4.6.1 Checking Query Security

Algorithm 1 presents ANGERONA. It takes as input a system state \( s = (db, U, P) \), a history \( h \), the current query \( q \) issued by the user \( u \), a system configuration \( C \), and an ATKLOG model \( ATK \) formalizing the users’ beliefs. ANGERONA checks whether disclosing the result of the current query \( q \) may violate any secrets in \( \text{secrets}(P, u) \). If this is the case, the algorithm concludes that \( q \)'s execution would be insecure and returns \( \bot \). Otherwise, it returns \( \top \) and authorizes \( q \)'s execution. Note that once we fix a configuration \( C \) and an ATKLOG model \( ATK \), ANGERONA is a C-PDP as defined in Section 4.3.3.

To check whether a query \( q \) may violate a secret \( (u, \psi, l) \in \text{secrets}(P, u) \), ANGERONA first checks whether the secret has already been violated. If this is not the case, ANGERONA checks whether disclosing \( q \) violates any secret. This requires checking that \( u \)'s belief about the secret \( \psi \) stays below the threshold independently of the result of the query \( q \); hence, we must ensure that \( u \)'s belief is below the threshold both in case the query \( q \) holds in the actual database and in case \( q \) does not hold (this ensures that the security decision itself does not leak information). ANGERONA, therefore, first checks whether there exists at least one possible database state where \( q \) is satisfied given \( h \), using the procedure \( \text{pax} \). If this is the case, the algorithm extends the current history \( h \) with the new event recording that the query \( q \) is authorized and its result is \( \top \) and it checks whether \( u \)'s belief about \( \psi \) is still below the corresponding threshold once \( q \)'s result is disclosed, using the secure
Algorithm 1: ANGERONA Enforcement Algorithm.

Input: A system state $s = (db, l, P)$, a history $h$, an action $(u, q)$, a system configuration $C$, and a $C$-AtkLog model ATK.
Output: The security decision in $\{T, L\}$.

begin
for $(u, \psi, l) \in \text{secrets}(P, u)$ do
  if secure($C, ATK, h, (u, \psi, l)$)
    if pos($C, ATK, h, (u, q))$
      $h' := h \cdot ((u, q), T, T)$
    if $\neg$secure($C, ATK, h', (u, \psi, l)$)
      return $\perp$
  if pos($C, ATK, h, (u, \neg q))
    $h' := h \cdot ((u, q), T, \perp)$
  if $\neg$secure($C, ATK, h', (u, \psi, l)$)
    return $\perp$
return $\top$

function secure($D, \Gamma, ATK, h, (u, \psi, l)$)
  $p := ATK(u)$
  for $\phi \in \text{observations}(h, u)$ do
    $p := p \cup PL(\phi) \cup \{\text{evidence}(\text{head}(\phi), \text{true})\}$
  $p := p \cup PL(\psi)$
  return $[p]_{\mathcal{D}}(\text{head}(\psi)) < l$

function pos($D, \Gamma, ATK, h, (u, \psi))$
  $p := ATK(u)$
  for $\phi \in \text{observations}(h, u)$ do
    $p := p \cup PL(\phi) \cup \{\text{evidence}(\text{head}(\phi), \text{true})\}$
  $p := p \cup PL(\psi)$
  return $[p]_{\mathcal{D}}(\text{head}(\psi)) > 0$

procedure. Afterwards, ANGERONA checks whether there exists at least a possible database state where $q$ is not satisfied given $h$, it extends the current history $h$ with another event representing that the query $q$ does not hold, and it checks again whether disclosing that $q$ does not hold in the current database state violates the secret. Note that checking whether there is a database state where $q$ is (or is not) satisfied is essential to ensure that the conditioning that happens in the secure procedure is well-defined, i.e., the set of states we condition on has non-zero probability.

ANGERONA uses the secure subroutine to determine whether a secret’s confidentiality is violated. This subroutine takes as input a system configuration, an AtkLog model $ATK$, a history $h$, and a secret $(u, \psi, l)$. It first computes the set $\text{observations}(h, u)$ containing all the authorized queries in the $\mathcal{W}$-projection of $h$, i.e., $\text{observations}(h, u) = \{\phi \upharpoonright \exists i. h_u(i) = ((u, \phi), T, T)\} \cup \{\neg \phi \upharpoonright \exists i. h_u(i) = ((u, \phi), T, \perp)\}$. Afterwards, it generates a ProbLog program $p$ by extending $ATK(u)$ with additional rules. In more detail, it translates each relational calculus sentence $\phi \in \text{observations}(h, u)$ to an equivalent set of ProbLog rules $PL(\phi)$. The translation $PL(\phi)$ is standard [10]. For example, given a query $\phi = (A(1) \land B(2)) \lor \neg C(3)$, the translation $PL(\phi)$ consists of the rules $(h_1 \leftarrow A(1)), (h_2 \leftarrow B(2)), (h_3 \leftarrow \neg C(3)), (h_4 \leftarrow h_1, h_2), (h_5 \leftarrow h_3, h_4)$, where $h_1, \ldots, h_4$ are fresh predicate symbols. We denote by $\text{head}(\phi)$ the unique predicate symbol associated with the sentence $\phi$ by the translation $PL(\phi)$. In our example, $\text{head}(\phi)$ is the fresh predicate symbol $h_5$. The algorithm then conditions the initial probability distribution $ATK(u)$ based on the sentences in $\text{observations}(h, u)$. This is done using evidence statements, which are special ProbLog statements of the form $\text{evidence}(a, v)$, where $a$ is a ground atom and $v$ is either true or false (see Section 4.4.1). For each sentence $\phi \in \text{observations}(h, u)$, the program $p$ contains a statement $\text{evidence}(\text{head}(\phi), \text{true})$. Finally, the algorithm translates $\psi$ to a set of logic programming rules and checks whether $\psi$’s probability is below the threshold $l$.

The pos subroutine takes as input a system configuration, an AtkLog model $ATK$, a history $h$, and a query $(u, \psi)$. It determines whether there is a database $db'$ that satisfies $\psi$ and complies with the history $h_{\upharpoonright u}$. Internally, the routine again constructs a ProbLog program starting from $ATK$, $\text{observations}(h, u)$, and $\psi$. Afterwards, it uses the program to check whether the probability of $\psi$ given $h_{\upharpoonright u}$ is greater than $0$.

Given a run $(s, h)$ and a user $u$, the secure and pos subroutines condition $u$’s initial beliefs based on the sentences in $\text{observations}(h, u)$, instead of using the set $[r]_{\sim}^{\leftarrow s}$ as in the AtkLog semantics. The key insight is that, as we prove in Appendix B, the set of possible database states defined by the sentences in $\text{observations}(h, u)$ is equivalent to $[r]_{\sim}^{\leftarrow s}$, which contains all database states derivable from the runs $r' \sim u, r$. This allows us to use ProbLog to implement AtkLog’s semantics without explicitly computing $[r]_{\sim}^{\leftarrow s}$.

Example 4.15. Let $ATK$ be the attacker model in Example 4.4, $u$ be the user Mallory, the database
state be $s_{(u,q)}$, where Alice, Bob, and Carl have cancer, and the policy $P$ be the one from Example 4.2. Furthermore, let $q_1, \ldots, q_4$ be the queries issued by Mallory in Example 4.3. ANGERONA permits the execution of the first two queries since they do not violate the policy. In contrast, it denies the execution of the last two queries as they leak sensitive information.

**Confidentiality.** As we prove in Appendix B, ANGERONA provides the desired security guarantees for any ATkLOG-attacker. Namely, it authorizes only those queries whose disclosure does not increase an attacker's beliefs in the secrets above the corresponding thresholds. ANGERONA also provides precise completeness guarantees: it authorizes all secrecy-preserving queries. Informally, a query $(u,q)$ is secrecy-preserving given a run $r$ and a secret $(u,\psi,l)$ iff disclosing the result of $(u,q)$ in any run $r' \sim_u r$ does not violate the secret.

**Theorem 4.2.** Let a system configuration $C$ and a C-ATkLOG model ATK be given, and let ANGERONA be the C-PDP $f$. ANGERONA provides data confidentiality with respect to $C$ and $\lambda u \in U. \ [ATK(u)]_D$, and it authorizes all secrecy-preserving queries.

**Complexity.** ANGERONA’s complexity is dominated by the complexity of inference. We focus our analysis only on data complexity, i.e., the complexity when only the ground atoms in the ProbLog programs are part of the input while everything else is fixed. A literal query is a query consisting either of a ground atom $a(\tau)$ or its negation $\neg a(\tau)$. We call an ATkLOG model acyclic if all belief programs in it are acyclic. Furthermore, a literal secret is a secret $(U,\phi,l)$ such that $\phi$ is a literal query. We prove in Appendix B that for acyclic ATkLOG models, literal queries, and literal secrets, the ProbLog programs produced by the secure and pax subroutines are acyclic. We can therefore use our dedicated inference engine from Section 4.5 to reason about them. Hence, ANGERONA can be used to protect databases in PTIME in terms of data complexity.

**Theorem 4.3.** For all acyclic ATkLOG attackers, for all literal queries $q$, for all runs $r$ whose histories contain only literal queries and contain only secrets expressed using literal queries, ANGERONA’s data complexity is PTIME.

**Discussion.** Our tractability guarantees apply only to acyclic ATkLOG models, literal queries, and literal secrets. Nevertheless, ANGERONA can still handle relevant problems of interest. As stated in Section 4.5, acyclic models are as expressive as poly-tree Bayesian Networks, one of the few classes of Bayesian Networks with tractable inference. Hence, for many probabilistic models that cannot be represented as acyclic ATkLOG models, exact probabilistic inference is intractable.

Literal queries are expressive enough to state simple facts about the database content. For instance, they can be used to formulate queries such as “does Alice have cancer?”. More complex (non-literal) queries can be simulated using (possibly large) sequences of literal queries. Similarly, policies with non-literal secrets can be implemented as sets of literal secrets, and the Boole–Fréchet inequalities [89] can be used to derive the desired thresholds. In both cases, however, our completeness guarantees hold only for the resulting literal queries, not for the original ones.

Finally, whenever our tractability constraints are violated, ANGERONA can still be used by directly using ProbLog’s inference capabilities. In this case, one would retain the security and completeness guarantees (Theorem 4.2) but lose the tractability guarantees (Theorem 4.3).

**4.6.2 Implementation and Empirical Evaluation**

To evaluate the feasibility of our approach in practice, we implemented a prototype of ANGERONA, available at [85]. The prototype implements our dedicated inference algorithm for acyclic ProbLog programs (Section 4.5), which computes the relaxed grounding of the input program $p$, constructs the Bayesian Networks $BN$, and performs the inference over $BN$ using belief propagation [105]. For inference over $BN$, we rely on the GRM library [154]. Observe that evidence statements in ProbLog are encoded by fixing the values of the corresponding random variables in the BN. Note also that computing the relaxed grounding of $p$ takes polynomial time in terms of data complexity, where the exponent is determined by $p$’s rules. A key optimization is to pre-compute the relaxed grounding and construct $BN$ off-line. This avoids grounding $p$ and constructing the (same) Bayesian Network for each query. In our experiments we measure this time separately.

We use our prototype to study ANGERONA’s efficiency and scalability. We run our experiments on a PC with an Intel i7 processor and 32GB of RAM. For our experiments, we consider the database schema from Section 4.3. For the belief programs, we use the ProbLog program given in Section 4.4, which can be encoded as an acyclic program whenever each person has at most one son. We evaluate ANGERONA’s efficiency and scalability in terms of the number of ground atoms in the belief programs. We generate synthetic belief programs containing 1,000 to 100,000 patients and for each of these instances, we generate 100 random queries of the form $R(\overline{t})$, where $R$ is a predicate symbol and $\overline{t}$ is a tuple. For each instance and sequence of queries, we check the security of each query with our
4.7 Related Work

**Database Inference Control.** Existing DBIC approaches protect databases (either at design time \[120,121\] or at runtime \[44,45,91\]) only against restricted classes of probabilistic dependencies, e.g., those arising from functional and multi-valued dependencies. AtkLog, instead, supports arbitrary probabilistic dependencies, and even our acyclic fragment can express probabilistic dependencies that are not supported by \[44,45,91,120,121\]. Wiese \[166\] proposes a DBIC framework, based on possibilistic logic, that formalizes secrets as sentences and expresses policies by associating bounds to secrets. Possibility theory differs from probability theory, which results in subtle differences. For instance, there is no widely accepted definition of conditioning for possibility distributions, cf. \[38\]. Thus, the probabilistic model from Section 4.1 cannot be encoded in Wiese’s framework \[166\].

Statistical databases store information associated to different individuals and support queries that return statistical information about an entire population \[47\]. DBIC solutions for statistical databases \[11,47,62,65,67\] prevent leakages of information about single individuals while allowing the execution of statistical queries. These approaches rely on various techniques, such as perturbing the original data, synthetically generating data, or restricting the data on which the queries are executed. Instead, we protect specific secrets in the presence of probabilistic data dependencies and we return the original query result, without modifications, if it is secure.

**Controlled Query Evaluation.** Controlled Query Evaluation (CQE) is an inference control framework initially proposed by Biskup and Bonatti \[28,29,36\]. The framework has been specialized to several settings, such as deductive databases \[36,146\], incomplete databases \[33–35\], databases in the...
presence of updates [31, 32], possibilistic databases [166], and ontologies [82]. In a CQE system, each query \( q \) is executed on the database and its result is inspected by a censor, which decides whether the query’s result violates the security policy. In case the result is considered sensitive, the censor may reject the result [28, 29] or modify it [36] (or a combination of both strategies [30]). Observe that the first strategy corresponds to the Non-Truman model, whereas the last one corresponds to the Truman model.

Works on CQE formalize security policies using secrecies and secrets. A secret is a sentence \( \phi \) defined on the database, whereas a secrecy is a set \( \{ \phi, \neg \phi \} \) where \( \phi \) is a secret. A policy then is either a set of secrets or a set of secrecies. Note that secrecies can be reduced to secrets. Our security policies from Section 4.3 can be seen as an extension of secrets to the probabilistic setting.

In a nutshell, the two security conditions studied in CQE are defined as follows:

- **Secrecies**: a censor is secure iff for all sequences of queries \( Q \), for all database states \( db \), for all secrecies \( \{ \phi, \neg \phi \} \) in \( S \), there exists a state \( db' \) such that the results of all queries in \( Q \) on \( db \) and \( db' \) are the same, \( \phi \) holds in one state, and \( \neg \phi \) holds in the other one.

- **Secrets**: a censor is secure iff for all sequences of queries \( Q \), for all database states \( db \), for all secrets \( \phi \) in \( S \), there exists a state \( db' \) such that the results of all queries in \( Q \) on \( db \) and \( db' \) are the same, and in one of the two states \( \neg \phi \) holds.

Intuitively, the security condition for secrecies guarantees that an attacker is not able to infer \( \phi \)’s truth value, whereas the one for secrets only guarantees that an attacker is not able to infer that \( \phi \) holds in current database states. Observe that our data confidentiality property from Section 4.3 generalizes the latter condition to the probabilistic setting.

**Differential Privacy.** Differential Privacy [68, 69] is widely used for privacy-preserving data analysis. Systems such as Airavat [132], ProPer [70], and PINQ [118] provide users with automated ways to perform differentially private computations. A differentially private computation guarantees that the presence (or absence) of an individual’s data in the input data set affects the probability of the computation’s result only in limited way, i.e., by at most a factor \( e^\epsilon \) where \( \epsilon \) is a parameter controlling the privacy-utility trade-off. While differential privacy does not make any assumption about the attacker’s beliefs, we assume that the attacker’s belief is known and we guarantee that for all secrets in the policy, no user can increase his beliefs, as specified in the attacker model, over the corresponding thresholds by interacting with the system.

**Database Access Control.** Rastogi et al. [129] propose an access control language for uncertain data and a perturbation algorithm to securely answer queries. Their security notion differs from ours, as it is based on differential privacy. To prevent leaks, their algorithm adds noise to the original query answer. Instead, ANGERONA returns the original result to the user if this does not violate the security policy.

**Query Auditing.** Query auditing [71, 103, 123] is the task of determining whether the answer to a user’s query may lead to a privacy breach. Evfimievski et al.’s security notion [71] guarantees that for a secure query, a user cannot increase his confidence in the secret by observing the query’s output. Instead, our security notion guarantees that users cannot use queries to increase their confidence in a secret above a given threshold. Our approach works for arbitrary relational calculus queries. In contrast, existing online query auditing approaches [103, 123] are limited to restricted classes of queries, such as COUNT and MAX queries.

**Information-flow Control.** Quantified Information Flow [16, 51, 106, 115] aims at quantifying the amount of information leaked by a program. Instead of measuring the amount of leaked information, we focus on restricting the information that an attacker may obtain about a set of given secrets.

Non-interference has been extended to consider probabilities [15, 136, 164] for concurrent programs. Our security notion, instead, allows those leaks that do not increase an attacker’s beliefs in a secret above the threshold, and it can be seen as a probabilistic extension of opacity [142], which allows any leak except leaking whether the secret holds.

Mardziel et al. [116] present a general DBIC architecture, where users’ beliefs are expressed as probabilistic programs, security requirements as threshold on these beliefs, and the beliefs are updated in response to the system’s behaviour. Our work directly builds on top of this architecture. However, instead of using an imperative probabilistic language, we formalize beliefs using probabilistic logic programming, which provides a natural and expressive language for formalizing dependencies arising in the database setting, e.g., functional and multi-valued dependencies, as well as common probabilistic models, like Bayesian Networks.

Mardziel et al. [116] also propose a DBIC mechanism based on abstract interpretation. They do not provide any precise complexity bound for their mechanism. Their algorithm’s complexity class, however, appears to be intractable, since they use a probabilistic extension of the polyhedra abstract domain, whose asymptotic complexity is exponential in the number of program variables [148]. In contrast, ANGERONA exploits our inference engine for acyclic programs to secure databases against
a practically relevant class of probabilistic inferences, and it provides precise tractability and completeness guarantees.

We now compare (unrestricted) ATKLOG with the imperative probabilistic language used in [116]. ATKLOG allows one to concisely encode probabilistic properties specifying relations between tuples in the database. For instance, a property like “the probability of $A(x)$ is $1/n$” where $n$ is the number of tuples $(x,y)$ in $B$ can be encoded as $\frac{1}{n}:A(x) \leftarrow B(x,y)$. Encoding this property as an imperative program is more complex; it requires a for statement to iterate over all variables representing tuples in $B$ and an if statement to filter the tuples. In contrast to [116], ATKLOG provides limited support for numerical constraints (as we support only finite domains). Mardziel et al. [116] formalize queries as imperative probabilistic programs. They can, therefore, also model probabilistic queries or the use of randomization to limit disclosure. While all these features are supported by ATKLOG, our goal is to protect databases from attackers that use standard query languages like SQL. Hence, we formalize queries using relational calculus and ignore probabilistic queries. Similarly to [116], our approach can be extended to handle some uncertainty on the attackers’ capabilities. In particular, we can associate to each user a finite number of possible beliefs, instead of a single one. However, like [116], we cannot handle infinitely many alternative beliefs.

**Probabilistic Programming.** Probabilistic programming is an active area of research [81]. Here, we position PROBLOG with respect to expressiveness and inference. Similarly to [77,116], PROBLOG can express only discrete probability distributions, and it is less expressive than languages supporting continuous distributions [76,80,144]. Current exact inference algorithms for probabilistic programs are based on program analysis techniques, such as symbolic execution [76,144] or abstract interpretation [116]. In this respect, we present syntactic criteria that ensure tractable inference for PROBLOG. Sampson et al. [139] symbolically execute probabilistic programs and translate them to BNs to verify probabilistic assertions. In contrast, we translate PROBLOG programs to BNs to perform exact inference and our translation is tailored to work together with our acyclicity constraints to allow tractable inference.

### 4.8 Conclusions

Effectively securing databases that store data with probabilistic dependencies requires an expressive language to capture the dependencies and a tractable enforcement mechanism. To address these requirements, we developed ATKLOG, a formal language providing an expressive and concise way to represent attackers’ beliefs while interacting with the system. We leveraged this to design ANGERONA, a provably secure DBIC mechanism that prevents the leakage of sensitive information in the presence of probabilistic dependencies. ANGERONA is based on a dedicated inference engine for a fragment of PROBLOG where exact inference is tractable. We see these results as providing a foundation for building practical protection mechanisms, which include probabilistic dependencies, as part of real-world database systems.

### 4.9 Technical Details, Extensions, and Additional Examples

In Section 4.9.1, we introduce an extension of acyclic programs. We present a simple encoding from relational calculus queries to logic programs in Section 4.9.2. Afterwards, we show that the program associated with our medical example from Section 4.1 is a relaxed acyclic program (Section 4.9.3). Finally, in Section 4.9.4 we present an additional example from the genomic data domain.

#### 4.9.1 Relaxed acyclic programs

Here we present an extension of acyclic programs, called relaxed acyclic programs. Relaxed acyclic programs support a restricted class of annotated disjunctions. They also support rules with overlapping bodies. Relaxed acyclic programs are needed to encode the example from Section 4.4 and for Proposition 4.2. We also generalize our BN encoding to relaxed acyclic programs. Observe that (1) acyclic programs are also relaxed acyclic programs, and (2) all our results extend to relaxed acyclic programs.

The key components of this new fragment are two syntactic transformations over PROBLOG programs that preserve certain key aspects of the program’s structure. Intuitively, a PROBLOG program $p$ is a relaxed acyclic program iff the program obtained from $p$ by applying the transformations is an acyclic PROBLOG program. Finally, we develop a procedure for compiling any relaxed acyclic PROBLOG program into a poly-tree Bayesian Network. Since any acyclic program is a relaxed acyclic program as well, this procedure can be applied also to acyclic programs.
4.9.1.1 Rule Domination

Let r₁ and r₂ be two rules. Rule r₁ is dominated by rule r₂, written r₁ ⊆ r₂, iff: (a) head(r₁) = head(r₂), (b) cstr(r₁) = cstr(r₂), (c) for all 1 ≤ i ≤ |body(r₁)|, then pred(body(r₁, i)) = pred(body(r₂, i)) and args(body(r₁, i)) = pred(body(r₂, i)). We extend the domination relation also to atoms and probabilistic atoms as follows: a₁(σ₁) ⊆ a₂(σ₂) iff a₁ = a₂ and σ₁ = σ₂, and v₁::a₁(σ₁) ⊆ v₂::a₂(σ₂) iff v₁ = v₂ = a₂ and σ₁ = σ₂.

Given a program p and a rule r ∈ P, [r]ₚ denotes the set {r′ ∈ P | r′ ⊆ r}. We say that a rule r is maximal in a program p, written maximal(r, p), iff there is no rule r′ ∈ p such that r ⊆ r′. The kernel of [r]ₚ, denoted k([r]ₚ), is the rule r′ defined as follows: head(r′) = head(r), body(r′) = |body(r)|, cstr(r′) = cstr(r), and for all 1 ≤ i ≤ |body(r′)|, then body(r′, i) = aᵢ(σᵢ). If for all rules r′ ∈ [r]ₚ (1) |body(r′)| ≥ i and (2) body(r′, i) is a positive literal and body(r′, i) = ¬aᵢ(σᵢ) otherwise, where aᵢ = pred(body(r, i)) and σᵢ = args(body(r, i)).

The maximal projection of r, denoted pₚ ⊆ₚ, is {k([r]ₚ) | r ∈ p ∧ maximal(r, p)}. The syntactic transformation α, which removes all non-maximal rules, takes as input a PROBLOG program p and returns as output its maximal projection pₚ ⊆ₚ.

4.9.1.2 Safe annotated disjunctions

We now present safe annotated disjunctions predicates, a special kind of probabilistic structures that can be formalized using annotated disjunctions. While our fragment does not support the unrestricted use of annotated disjunctions, as they may introduce cycles in the ground graph, safe annotated disjunctions can be still encoded as poly-trees (using non-random boolean variables).

Notation. Let (Σ, dom) be a database schema such that dom is a finite domain, pr be a predicate symbol in Σ of arity |pr|, p be a (Σ, dom)-PROBLOG program, and K be a set of distinct integer values such that for all i ∈ K, 1 ≤ i ≤ |pr|. Given a tuple τ, we denote by τ₁(i₁,...,iₙ) the tuple (t_1(i_1),...,t_n(i_n)), where i_1 ≤ i_2 ≤ ... ≤ i_n. The syntactic transformation ψpr.K is defined as follows:

- (pr(τ))ₚ,ψpr.K = pr(τ₁|pr.K),
- (pr(τ))ₚ,ψpr,K = pr(τ), where |pr| ≠ |pr′|
- (v::(τ))ₚ,ψpr.K = v::(τₚ,ψpr,K),
- (¬a(τ))ₚ,ψpr.K = ¬(a(τ)),
- (h ← t₁,...,tₙ)ₚ,ψpr,K = hₚ,ψpr,K ← t₁ₚ,ψpr,K,...,tₙₚ,ψpr,K
- pₚₚ,ψpr.K, where p is a program, is {τₚ,ψpr.K | r ∈ p}.

Given (pr₁, K₁),..., (prₙ, Kₙ), the transformation ψ₁,...,ₙₚ,ψpr₁,...,ₙ,ψprₙ,K₁,...,ₙ is ψ₁,ψpr₁.K₁ ∘ ..., ∘ ψₙ,ψprₙ,Kₙ. Finally, given a partial function μ assigning to predicate symbols pr sets K ⊆ {1,...,|pr|}, the transformation ψμ is ψ₁,μ(pr₁),...,ₙ,μ(prₙ), where pr₁,...,ₙ,μ(prₙ) are all predicate symbols for which μ is defined.

Partitionings and CPT-schemas. Let p be a program.

An horizontal partitioning πḥ is a function taking as input a program p and a predicate symbol pr and returning as output an element of 2^{2^{|args(pr)|}} such that the following conditions hold:

1. For all RR ∈ πḥ(p, pr) and for all R ∈ RR, RR ≠ ∅ and R ≠ ∅.
2. For all RR ∈ πḥ(p, pr), for all distinct R₁ and R₂ in RR, R₁ ∩ R₂ = ∅.
3. For all distinct RR₁ and RR₂ in πḥ(p, pr), ∪ₚ∈πḥ,p∈RR₁RR₂ ∈ πḥ(p, pr).
4. ∪ₚ∈πḥ,p∈RR₁RR₂ ∈ πḥ(p, pr) if r ∈ rules(p) ∧ pred(head(r)) = pr.

A vertical partitioning πᵥ is a function that takes as input rules in p and returns as output a triple ⟨row, sel, sw⟩, where row, sel, and sw are sequences of literals, such that for each rule r ∈ rules(p), πᵥ(r) = ⟨row, sel, sw⟩ and row ∪ row ∪ row ∪ row ∪ row ∪ row ∪ row = body(r). A CPT-schema is a triple (πḥ, πᵥ, μ) such that πḥ is a horizontal partitioning, πᵥ is a vertical partitioning, and μ is a total function associating to each predicate symbol pr in p a set in 2^{πḥ} such that μ(pr) ⊆ {1,...,|pr|}.

K-fixed domain predicates. Let p be a program, pr be a predicate symbol, and K ⊆ {1,...,|pr|} be a non-empty set. We say that pr is a K-fixed domain with respect to p iff for all rules r ∈ p, the following conditions hold: (1) if pred(head(r)) = pr, then arg(s}(head(r))ₚ,K is a tuple in dom[K], and (2) if pred(body(r, j)) = pr, for some 1 ≤ j ≤ |body(r)|, then arg(s}(body(r, j))ₚ,K is a tuple in dom[K]. We denote by p-dom(pr, p, K) the set of [K]-tuples representing all constant values associated with the positions in K. Namely, p-dom(pr, p, K) = {arg(s}(head(r))ₚ,K | r ∈ p ∧ pred(head(r)) = pr} ∪ {arg(s}(body(r, j))ₚ,K | r ∈ p ∧ 1 ≤ j ≤ |body(r)| ∧ pred(body(r, j)) = pr}.

Switch predicates. Let p be a program. We say that a predicate symbol pr is a switch predicate for p iff (1) there is no rule in rules(p) having pr in the head, (2) for any v::pr(τ₁) ∈ p, v ≠ 0, and (3) for any two ground atoms v₁::pr(τ₁) ∈ p and v₂::pr(τ₂) ∈ p, then v₁ ≠ v₂.

Safe schemas. Let p be a program, (πḥ, πᵥ, μ) be a CPT-schema for p, and τ be the template with all annotations that can be derived from p. We say that (πḥ, πᵥ, μ) is a safe schema iff for all pr in Σ such that μ(pr) ≠ ∅ the following conditions hold:
1. For all rules \( r \in p \) such that \( \text{pred}(\text{head}(r)) = pr \), \( r \) is strongly connected for \( T \) and \( \text{body}(r) \neq \emptyset \). Note that this implies that there are no ground atoms \( pr(T) \) in \( p \).

2. The predicate symbol \( pr \) is \( \mu(pr) \)-fixed domain with respect to \( p \).

3. For all \( RR \in \pi_{\mu}(p, pr) \), the following conditions hold:
   (a) For all \( R \in RR \), the following conditions hold:
      i. For any two distinct \( r_1, r_2 \in R \), the following conditions hold:
         A. \( row_1 = row_2 \) and \( sel_1 = sel_2 \), where \( \pi_V(r_1) = \langle row_1, sel_1, sw_1 \rangle \) and \( \pi_V(r_2) = \langle row_2, sel_2, sw_2 \rangle \).
         B. \( \tau_{r_1} \neq \tau_{r_2} \) and \( \tau_{\pi_V(r_1)} \neq \tau_{\pi_V(r_2)} \), where \( \pi_V(r_1) = \langle row_1, sel_1, sw_1 \rangle \) and \( \pi_V(r_2) = \langle row_2, sel_2, sw_2 \rangle \).
   ii. One of the following conditions hold:
      A. There is an ordering \( r_1, \ldots, r_n \) of the rules in \( R \) and switch predicates \( sw_1, \ldots, sw_{n-1} \) such that (1) for all \( 1 \leq i < n \), \( psw_i = \neg sw_{i+1} \) and \( \tau_{\pi_V(r_i)} = \langle row_i, sel_i, sw_i \rangle \), and \( psw_n = \neg sw_1 \), \( \neg sw_2 \), ... , \( \neg sw_{n-1} \), where \( \pi_V(r_i) = \langle row_i, sel_i, sw_i \rangle \) and \( \pi_{\tau(r_i)} = \langle row_i, sel_i, psw_i \rangle \) and \( \pi_{\tau(r_2)} = \text{args}(\text{head}(r_2)) \) for \( 1 \leq j \leq n \) and (2) \( \sum_{1 \leq i \leq n-1} p(i) = 1 \), where \( p(i) = \frac{1}{\mu} \cdot \left( 1 - \sum_{1 \leq j < i} p(j) \right) \) and \( \mu \) is the probability associated to the predicate symbol \( sw_i \) in the program \( p \).

   iii. For all \( r \in R \), \( \text{pred}(\text{head}(r)) = pr \), \( \text{args}(\text{head}(r)) \cup \mu(pr) \in \text{dom}(pr, p, \mu(pr)) \), and \( \text{vars}(\text{head}(r)) \subseteq \bigcup_{\pi \in \pi_{\mu}(p, pr)} \text{vars}(l) \), where \( \pi_V(r) = \langle row, sel, sw \rangle \).
   (b) For any two distinct \( R_1, R_2 \in RR \), for all \( r_1 \in R_1 \) and \( r_2 \in R_2 \), where \( \pi_V(r_1) = \langle row_1, sel_1, sw_1 \rangle \) and \( \pi_V(r_2) = \langle row_2, sel_2, sw_2 \rangle \), the following conditions hold:
      i. \( \text{args}(\text{head}(r_1)) \cup \mu(pr) \neq \emptyset \), \( \text{args}(\text{head}(r_2)) \cup \mu(pr) \neq \emptyset \), and \( \text{args}(\text{head}(r_1)) \cup \mu(pr) \neq \text{args}(\text{head}(r_2)) \cup \mu(pr) \).
      ii. For all \( 1 \leq i \leq \min(|\pi_{\text{row}}(\text{row}_1)|, |\pi_{\text{row}}(\text{row}_2)|) \), the following conditions hold:
         A. \( \text{pred}(\text{row}_1(i)) = \text{pred}(\text{row}_2(i)) \).
         B. \( \text{args}(\text{row}_1(i)) \cup K_{1,i} \neq \text{args}(\text{row}_2(i)) \cup K_{2,i} \), where \( K_{1,i} = \langle 1, \ldots, |\text{pred}(\text{row}_1(i))| \rangle \setminus \mu(\text{pred}(\text{row}_1(i))) \) and \( K_{2,i} = \langle 1, \ldots, |\text{pred}(\text{row}_2(i))| \rangle \setminus \mu(\text{pred}(\text{row}_2(i))) \).
      iii. There exists a \( 1 \leq i \leq \min(|\pi_{\text{row}}(\text{row}_1)|, |\pi_{\text{row}}(\text{row}_2)|) \) such that one of the following conditions hold:
         A. \( \mu(\text{pred}(\text{row}_1(i))) \neq \emptyset \), \( \text{row}_1(i) \) is a positive literal, \( \text{row}_2(i) \) is a positive literal, and \( \text{args}(\text{row}_1(i)) \cup K_{1,i} \neq \emptyset \), where \( K_{1,i} = \mu(\text{pred}(\text{row}_1(i))) \) and \( K_{2,i} = \mu(\text{pred}(\text{row}_2(i))) \).
         B. \( \mu(\text{pred}(\text{row}_1(i))) = \emptyset \), \( \text{row}_1(i) \) is a positive literal, and \( \text{row}_2(i) \) is a negative one or vice versa.
   (c) The set \{ \( \{ \pi_T(T) \} \mid r \in \bigcup_{R \in RR} R \land \text{maximal}(r, \bigcup_{R \in RR} R) \} \) has cardinality 1.

3. For any two distinct \( RR_1 \) and \( RR_2 \) in \( \pi_{\mu}(p, pr) \), the following conditions hold:
   (a) For all \( R_1 \in RR_1 \), \( R_2 \in RR_2 \), \( r_1 \in R_1 \), and \( r_2 \in R_2 \), \( \text{args}(\text{head}(r_1)) \cup \mu(pr) = \text{args}(\text{head}(r_2)) \cup \mu(pr) \), where \( \text{args}(\text{head}(r_1)) \cup \mu(pr) = \text{body}^{-}(r_1, i) \) and \( \text{args}(\text{head}(r_2)) \cup \mu(pr) = \text{body}^{-}(r_2, j) \).
   (b) There is a disjointness annotation \( \text{DIS}(a, b) \in T \) and indexes \( i, j \) such that for all \( R_1 \in RR_1 \) and \( R_2 \in RR_2 \), all \( r_1 \in R_1 \) and all \( r_2 \in R_2 \), \( a(\text{args}(\text{head}(r_1)) \cup \mu(pr)) = \text{body}^{-}(r_1, i) \) and \( b(\text{args}(\text{head}(r_2)) \cup \mu(pr)) = \text{body}^{-}(r_2, j) \).
   (c) The set \{ \( \{ \pi_T(T) \} \mid r \in \bigcup_{R_1 \in RR_1} R_1 \cup \bigcup_{R_2 \in RR_2} R_2 \cup \text{maximal}(r, \bigcup_{R_1 \in RR_1} R_1 \cup \bigcup_{R_2 \in RR_2} R_2) \} \) has cardinality 2.

4. Syntactic transformation. Let \( p \) be a program and \( \langle \pi_H, \pi_V, \mu \rangle \) be a CPT-schema for \( p \). The syntactic transformation \( \beta_{\mu}^{-} \) takes as input a ProbLog program \( p \) and it returns as output the program \( p' = p_{\mu(\pi_H(\pi_V(pr_1)), \ldots, \pi_H(\pi_V(pr_n)))} \) where \( pr_1, \ldots, pr_n \) are the predicate symbols in \( \mu \)'s domain. Note that \( p' \) is defined on a different database schema than \( p \). The program \( p' \) is defined over the reduced database schema obtained by applying \( \beta \) to the atoms in \( p \).

4.9.1.3 Relaxed Acyclic ProbLog programs

Let \( p \) be a \( (\Sigma, \text{dom}) \)-ProbLog program. An acyclicity witness for a program \( p \) is safe CPT-schema \( W = (\pi_H, \pi_V, \mu) \) such that \( \alpha(\beta_{\mu}(p)) \) is an acyclic ProbLog program. We say that \( p \) is a relaxed acyclic \( (\Sigma, \text{dom}) \)-ProbLog program iff there is an acyclicity witness \( W = (\pi_H, \pi_V, \mu) \) for \( p \). We remark that an acyclic program is also a relaxed acyclic program (for any assignment \( \langle \pi_H, \pi_V, \mu \rangle \) such that \( \mu(pr) = \emptyset \) for any \( pr \)). Given a relaxed acyclic \( (\Sigma, \text{dom}) \)-ProbLog program \( p \) and a
Furthermore, we do not explicitly refer to each variable’s domain, as it can be immediately derived.

\[ \{ r'' \in p \mid \beta_W(r'') = r' \} \]

The set \([r]_{p,W} \) contains all rules in \( p \) that are represented by the rule \( r \) in \( \alpha(\beta_W(p)) \).

### 4.9.1.4 Notation

Let \( p \) be a relaxed acyclic \textsc{Problog} program and \( W = \langle \pi_H, \pi_V, \mu \rangle \) be a witness for \( p \). For each atom \( a(\tau) \), we denote by \( a(\tau)\|_p \) the atom obtained by applying the transformations \( \alpha \) and \( \beta_W \), whereas we denote by \( a(\tau)\|_p \) the constants not used in the transformed program, namely \( a(\tau)\|_p \) is the set \( \{ r \in [r]_{p,W} \mid \beta_W(r) \in p \} \). Furthermore, given a rule \( r \in p \) (respectively a ground rule \( s \in ground(p,r) \)), we denote by \( r\|_p \) (respectively \( s\|_p \)) the rule obtained by applying the transformations \( \alpha \) and \( \beta_W \). If \( \mu(a) = \emptyset \), then \( a(\tau)\|_p = \top \) and \( \neg a(\tau)\|_p = \bot \).

### 4.9.1.5 Compilation to Bayesian Networks

Given a relaxed acyclic \textsc{Problog} program \( p \), its encoding as a Bayesian Network \( BN = \langle N, E, D, CPT, \preceq \rangle \), denoted \( bn(p) \), is defined by Algorithm 2. We remark that this encoding extends the one we presented in Section 4.5 in two main aspects:

1. It supports safe annotated disjunctions by using non-boolean random variables (instead of just using boolean random variables), and
2. It supports rules with overlapping bodies by encoding the semantics of multiple rules in the conditional probability table of a single random variable (see the \( CPT \) auxiliary function in Algorithm 4).

For simplicity, in the following we assume given the total ordering \( \preceq \) over random variables in \( N \). Furthermore, we do not explicitly refer to each variable’s domain, as it can be immediately derived from \( CPT \). Hence, we refer to a Bayesian Network as a triple \( \langle N, E, CPT, \preceq \rangle \) instead of a 5-tuple \( \langle N, E, D, CPT, \preceq \rangle \).

In Algorithm 2, given a ground rule \( s = h \leftarrow l_1, \ldots, l_m, e_1, \ldots, e_m \), we denote by \( pos(s) \) the rule \( h \leftarrow atom(l_1), \ldots, atom(l_m), e_1, \ldots, e_m \), where each negative literal is replaced with the corresponding positive literal.

### 4.9.2 From relational calculus to logic programs

In the following, let \( D = \langle \Sigma, \text{dom} \rangle \) be a database schema. Without loss of generality, we focus on relational calculus formulae \( \phi \) where no distinct pair of quantifiers binds the same variable.

**Normal Form.** We say that a formula \( \psi \land \neg \gamma \) is guarded iff \( free(\gamma) \subseteq free(\psi) \). We say that a relational calculus formula \( \phi \) is in Normal Form (NF) iff (1) \( \phi \) uses only existential quantifiers, (2) negation is used only in sub-formulae of the form \( \psi \land \neg \gamma \) and it is always guarded, (3) for any sub-formula \( \psi \lor \gamma \), \( free(\psi) = free(\gamma) \), (4) no distinct pair of quantifiers binds the same variable, and (5) there are no equality and inequality constraints.

Most of the time, domain-independent relational calculus formulae can be easily written in NF by just re-arranging sub-formulae. We remark that any domain-independent relational calculus formula can be written in NF by (1) extending the database schema with two relations \( eq \) and \( neq \) encoding = and \( \neq \) among constants in \( \text{dom} \) (this is always possible because \( \text{dom} \) is finite), (2) renaming the quantified variables in a unique way, (3) replacing all universally quantified sub-formula \( \forall x. \phi_x \rightarrow \psi \) with the equivalent existentially quantified version \( \exists x. \phi_x \land \neg \psi \), (4) replacing each negated sub-formula \( \neg \psi \) with the equivalent sub-formula \( (\forall x \in free(\psi) \adom(x)) \land \neg \psi \), where \( \adom(x) = \bigvee_{R \in D} \bigwedge_{1 \leq i \leq |R|} x \in R, x_{i-1}, x_{i+1}, \ldots, x_{|R|}, R(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{|R|}) \), and (5) replacing sub-formulae of the form \( \psi \lor \gamma \) with the equivalent formula \( (\exists x \in free(\gamma) \adom(x)) \lor \forall x \in free(\psi) \adom(x) \lor \gamma \). Note that the resulting NF formula is equivalent to the original one (because the rewriting does not modify the formula’s semantics). Therefore, in the following we consider only NF relational calculus formulae.

**From Relational Calculus to Logic Programming.** Let \( \phi \) be a NF relational calculus sentence. We denote by \( sub(\phi) \) the sequence \( \phi_1, \ldots, \phi_n \) of all \( \phi \)'s sub-formulae ordered from the smallest to the largest, i.e., \( \phi_n = \phi \), and by \( D\text{Is}\text{ub}(\phi) \) the sequence obtained by removing from the sequence \( sub(\phi) \) all formulae of the form \( \neg \psi \). Since \( \phi \) is in NF, all negated sub-formulae in \( \phi \) appear only in NF in the sequence \( D\text{Is}\text{ub}(\phi) \).

The function \( PL(\phi) \) encodes any NF relational calculus sentence as a set of equivalent logic programming rules. We associate a unique predicate symbol \( H_i \) and a set of rules \( r(H_i) \) to each \( \psi_i \) in \( D\text{Is}\text{ub}(\phi) \) as follows:

- If \( \psi_i := \phi_i \rightarrow \phi_j \), then \( r(H_i) \) is \( \{ H_i(\tau) \leftarrow neq(x_i, x_j) \} \).

**Chapter 4. Securing Databases from Probabilistic Inference**
4.9. Technical Details, Extensions, and Additional Examples

Algorithm 2: Constructing the Bayesian Network. The sub-routines $CPT$ is shown in Algorithm 3, whereas the sub-routines $tree$ and $CPT_\oplus$ are shown in Algorithm 4.

**Input:** A relaxed acyclic $(\Sigma, \text{dom})$-ProbLog program $p_0$ and a witness $W = \langle \pi_H, \pi_V, \mu' \rangle$ for $p_0$.

**Output:** A Bayesian Network $(N, E, CPT)$.

1. Transform the original program.
   - $p = \alpha(b_H(p_0))$
   - Initialize the domain of each predicate symbol.
   - $D = \emptyset$
   - for $pr \in \Sigma$ do
     - if $|\mu'(pr)| > 0$ then
       - $D(pr) = \text{dom}(pr, p_0, \mu(pr)) \cup \{\bot\}$
     - else
       - $D(pr) = \{\top, \bot\}$
   - Initialize the BN.
   - $N = \emptyset$
   - $E = \emptyset$
   - $CPT = \emptyset$
   - Creates the nodes.
     - for $r \in p$ do
       - for $r' \in \text{ground}(p, r)$ do
         - $N = N \cup \{X[\text{head}(r')]\}$
         - for $a(\overline{r}) \in \text{body}^+(r')$ do
           - $N = N \cup \{X[a(\overline{r})]\}$
         - for $\neg a(\overline{r}) \in \text{body}^-(r')$ do
           - $N = N \cup \{X[a(\overline{r})]\}$
       - for $r \in p$ do
         - $\mu = \emptyset$
         - for $s \in \text{ground}(p, r)$ do
           - if $\text{head}(s) \notin \text{dom}(\mu)$ then
             - $\mu = \mu[\text{head}(s) \mapsto \emptyset]$
             - $\mu[\text{head}(s)] = \mu[\text{head}(s)] \cup \{\text{pos}(s)\}$
           - for $a(\overline{r}) \in \text{dom}(\mu)$ do
             - $N = N \cup \{X[r, a(\overline{r})]\}$
             - $E = E \cup \{X[r, a(\overline{r})] \rightharpoonup X[a(\overline{r})]\}$
             - $K = \emptyset$
             - for $I \in \mu(a(\overline{r}))$ do
               - if $I = \emptyset$ then
                 - $\text{Here, } [r]_{p_0,w} = \{r\}$ and $D(a) = \{\top, \bot\}$.
                 - if $\exists v: \text{head}(r) = v.a(\overline{r})$ then
                   - $\text{prob} = v$
                 - else
                   - $\text{prob} = 1$
                 - $CPT(X[r, \emptyset, a(\overline{r})]) = (\top \mapsto \text{prob}, \bot \mapsto (1 - \text{prob}))$
               - else
                 - for $b(\overline{r}) \in I$ do
                   - $E = E \cup \{X[b(\overline{r})] \rightharpoonup X[r, I, a(\overline{r})]\}$
                 - for $\neg b(\overline{r}) \in I$ do
                   - $E = E \cup \{X[b(\overline{r})] \rightharpoonup X[r, i, a(\overline{r})]\}$
                 - $\text{CPT}(X[r, I, a(\overline{r})]) = \text{CPT}(I, r, D, W, p_0)$
                 - $(N', E', \text{CPT}') = \text{tree}(K, X[r, a(\overline{r})], D(a))$
                 - $N = N' \cup N$
                 - $E = E \cup E'$
                 - $\text{CPT} = \text{CPT} \cup \text{CPT}'$
               - end
             - end
           - end
         - end
       - end
     - end
   - end
   - end
   - end

2. Set the CPT for the variables associated to the atoms.
   - for $X[a(\overline{r})] \in N$ do
     - $\text{CPT}(X[a(\overline{r})]) = \text{CPT}_\oplus(\{X[r, a(\overline{r})] \in N\} + 1, D(a))$
   - return $(N, E, \text{CPT})$
Chapter 4. Securing Databases from Probabilistic Inference

Algorithm 3: Auxiliary functions used in Algorithm 2.

```plaintext
function CPT((a₁, ..., aₙ), r, D, W, p₀)
    A = D(a₁) × ... × D(aₙ) × D(pred(head(r)))
    CPT = ∅
    for a ∈ A do
        a' = removeDuplicates((a₁, ..., aₙ), a)
        if satisfiable(r, a', D, W, p₀) ∧ ∀ 1 ≤ i, j ≤ n. ((i ≠ j ∧ aᵢ = aⱼ) → aᵢ(i) = aⱼ(j)) then
            CPT = CPT ∪ {a' → 1}
        else
            CPT = CPT ∪ {a' → 0}
    return CPT

function satisfiable(r, (v₁, ..., vₙ), D, W, p₀)
    if vₙ ≠ ⊥ then
        return res = ⊤
        for r' ∈ [r]p₀, W do
            if filter(head(r'), D, µ) = vₙ then
                for 1 ≤ i ≤ |body(r')| do
                    if vᵢ ∉ filter(body(r', i), D, µ) then
                        sat = ⊥
                    res = res ∨ sat
                return res
    else
        return res = ⊥
        for r' ∈ [r]p₀, W do
            sat = ⊤
            for 1 ≤ i ≤ |body(r')| do
                if vᵢ ∉ filter(body(r', i), D, µ) then
                    sat = ⊥
                res = res ∨ sat
            return ¬res

function filter(l, D, (π_H, π_V, µ))
    if ∃ a ∈ Σ, π ∈ (Var ∪ dom)ₛ, l = a(π) then
        if µ(pred(l)) ≠ ∅ then
            return {args(l)|µ(pred(l))}
        else
            return {⊥}
    if ∃ a ∈ Σ, π ∈ (Var ∪ dom)ₛ, l = ¬a(π) then
        if µ(pred(l)) ≠ ∅ then
            return {D(pred(l)) \ {args(l)|µ(pred(l))}} ∪ {⊥}
        else
            return {⊥}
```

Function $CPT_{\oplus}(n, D)$

1. If $D \neq \{\top, \bot\}$, then $n = 2$ (since the program is relaxed-acyclic).
2. For $(v_1, v_2) \in D^2$ do:
   - if $v_1 = v_2$ then $CPT = CPT \cup \{(v_1, v_2) \mapsto 1\}$
   - else $CPT = CPT \cup \{(v_1, v_2) \mapsto 0\}$
3. Return $CPT$

Function $tree(X, root, D)$

1. $p = \text{nil}$
2. $N = X$
3. $E = \emptyset$
4. $CPT = \emptyset$
5. $X' = \emptyset$
6. if $X = \{x\}$ then:
   - $N' = N \cup \{\text{root}\}$
   - $E' = E \cup \{x \rightarrow \text{root}\}$
   - $CPT(root) = CPT_{\oplus}(2, D)$
7. else:
   - for $v \in X$ do:
     - if $p = \text{nil}$ then
       - $p = v$
     - else
       - $X' = X' \cup \{X[p, v]\}$
       - $N = N \cup \{X[p, v]\}$
       - $E = E \cup \{p \rightarrow X[p, v], v \rightarrow X[p, v]\}$
       - $CPT(X[p, v]) = CPT_{\oplus}(3, D)$
     - $p = \text{nil}$
   - if $p \neq \text{nil}$ then
     - $X' = X' \cup \{p\}$
   - $(N', E', CPT') = tree(X', root, D)$
8. Return $(N \cup N', E \cup E', CPT \cup CPT')$
In the above program, we placed the rule \( H_i(\mathbf{F}_i) \leftarrow \text{eq}(x_i, x_j) \).

If \( \psi_i := x_i = x_j \), then \( r(H_i) \) is \( \{ H_i(\mathbf{F}_i) \leftarrow \text{eq}(x_i, x_j) \} \).

If \( \psi_i := R(\mathbf{F}_i) \), for some \( R \in \Sigma \), then \( r(H_i) \) is \( \{ H_i(\mathbf{F}_i) \leftarrow R(\mathbf{F}_i) \} \), where \( \mathbf{F}_i \) are \( \psi_i \)’s free variables.

If \( \psi_i := \psi_j \land \neg \psi_k \), then \( r(H_i) \) is \( \{ H_i(\mathbf{F}_i) \leftarrow H_j(\mathbf{F}_j), \neg H_k(\mathbf{F}_k) \} \), where \( \mathbf{F}_j \) are \( \psi_j \)’s free variables, \( \mathbf{F}_k \) are \( \psi_k \)’s free variables, \( \mathbf{F}_j \) are \( \psi_j \)’s free variables, and \( \mathbf{F}_k \) are \( \psi_k \)’s free variables (note that \( \mathbf{F}_i \cap \mathbf{F}_j \subset \mathbf{F}_k \)).

If \( \psi_i = \psi_j \land \psi_k \), then \( r(H_i) \) is \( \{ H_i(\mathbf{F}_i) \leftarrow H_j(\mathbf{F}_j), H_k(\mathbf{F}_k) \} \), where \( \mathbf{F}_j \) are \( \psi_j \)’s free variables, \( \mathbf{F}_k \) are \( \psi_k \)’s free variables, and \( \mathbf{F}_k \) are \( \psi_k \)’s free variables.

If \( \psi_i = \psi_j \lor \psi_k \), then \( r(H_i) \) contains the rules \( H_i(\mathbf{F}_i) \leftarrow H_j(\mathbf{F}_j) \) and \( H_i(\mathbf{F}_i) \leftarrow H_k(\mathbf{F}_k) \) (note that \( \text{free}(\psi_i) = \text{free}(\psi_j) = \text{free}(\psi_k) \)).

If \( \psi_i = \exists x \psi_j \), then \( r(H_i) \) is \( \{ H_i(\mathbf{F}_i) \leftarrow H_j(\mathbf{F}_j) \} \), where \( \mathbf{F}_i \) are \( \psi_i \)’s free variables and \( \mathbf{F}_j \) are \( \psi_j \)’s free variables (i.e., those in \( \mathbf{F}_i \) and \( x \)).

Additionally, we add ground atoms encoding the equality and inequality relations \text{eq} and \text{neq}. Furthermore, we denote by \text{head}(\phi) the predicate symbol associated to \( \psi_m \).

### 4.9.3 Acyclicity of the medical data example

We now show, step by step, that the ProbLog program we presented in Section 4.4, which captures the motivating example from Section 4.1, is a relaxed acyclic program.

**Original program.** For simplicity, we report here the rules defining the original ProbLog program.

\[
\begin{align*}
    1/20: & \text{cancer}(x) \leftarrow \text{patient}(x) \\
    5/19: & \text{cancer}(x) \leftarrow \text{smokes}(x) \\
    3/14: & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{cancer}(x), \text{mother}(z, y), \neg \text{cancer}(z) \\
    3/14: & \text{cancer}(y) \leftarrow \text{father}(x, y), \neg \text{cancer}(x), \text{mother}(z, y), \text{cancer}(z) \\
    3/7: & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{cancer}(x), \text{mother}(z, y), \text{cancer}(z) \\
    \end{align*}
\]

We do not fix the set of ground atoms. Instead, below we present a set of requirements for acyclicity.

**Removing syntactic sugar.** The first step is removing the syntactic sugar, which, in this case, consists of the probabilistic rules. We do so exactly as specified in Section 4.4. We therefore obtain the following program:

\[
\begin{align*}
    1/20: & \text{sw}_1(,) \\
    5/19: & \text{sw}_2(,) \\
    3/14: & \text{sw}_3(,) \\
    3/7: & \text{sw}_4(,) \\
    & \text{cancer}(x) \leftarrow \text{patient}(x), \text{sw}_1(x) \\
    & \text{cancer}(x) \leftarrow \text{smokes}(x), \text{sw}_2(x) \\
    & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \text{cancer}(x), \neg \text{cancer}(z), \text{sw}_3(y) \\
    & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \neg \text{cancer}(x), \text{cancer}(z), \text{sw}_3(y) \\
    & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \text{cancer}(x), \text{cancer}(z), \text{sw}_4(y) \\
    \end{align*}
\]

**Some rewriting.** The last three rules in the above program use the same predicate symbols in their first four literals. However, they differ on the last literal, since the first two rules use \( \text{sw}_3(y) \) while the last one uses \( \text{sw}_4(y) \). We thus rewrite the program in an equivalent way:

\[
\begin{align*}
    1/20: & \text{sw}_1(,) \\
    5/19: & \text{sw}_2(,) \\
    3/14: & \text{sw}_3(,) \\
    3/7: & \text{sw}_4(,) \\
    & \text{cancer}(x) \leftarrow \text{patient}(x), \text{sw}_1(x) \\
    & \text{cancer}(x) \leftarrow \text{smokes}(x), \text{sw}_2(x) \\
    & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \text{cancer}(x), \neg \text{cancer}(z), \text{sw}_3(y) \\
    & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \neg \text{cancer}(x), \text{cancer}(z), \text{sw}_3(y) \\
    & \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \text{cancer}(x), \text{cancer}(z), \text{sw}_4(y) \\
    \end{align*}
\]

In the above program, we replaced the rule \( \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \text{cancer}(x), \text{cancer}(z), \text{sw}_4(y) \) with the two rules \( \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \text{cancer}(x), \text{cancer}(z), \text{sw}_3(y), \text{sw}_4(y) \)
4.9. Technical Details, Extensions, and Additional Examples

and \( \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \text{cancer}(x), \text{cancer}(z), \neg \text{sw}_3(y), \text{sw}_4(y) \). This does not modify the program’s semantics.

**Syntactic transformations.** By applying the syntactic transformation for relaxed acyclic programs (see Section 4.9.1), we obtain the following program:

1/20: \( \text{sw}_1(\_\_) \)

5/19: \( \text{sw}_2(\_\_) \)

3/14: \( \text{sw}_3(\_\_) \)

3/7: \( \text{sw}_4(\_\_) \)

\( \text{cancer}(x) \leftarrow \text{patient}(x), \text{sw}_1(x) \)

\( \text{cancer}(x) \leftarrow \text{smokes}(x), \text{sw}_2(x) \)

\( \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \neg \text{cancer}(x), \neg \text{cancer}(z), \neg \text{sw}_3(y), \text{sw}_4(y) \)

In the above transformation we considered a CPT-schema \( \langle \pi_V, \pi_H, \mu \rangle \) such that \( \mu(pr) = \emptyset \) for any predicate symbol. Note that such a CPT-schema is always safe.

The dependency graph of this program is depicted in Figure 4.11, where \( r_1 \) denotes \( \text{cancer}(x) \leftarrow \text{patient}(x), \text{sw}_1(x) \), \( r_2 \) denotes the rule \( \text{cancer}(x) \leftarrow \text{smokes}(x), \text{sw}_2(x) \), and \( r_3 \) denotes the rule \( \text{cancer}(y) \leftarrow \text{father}(x, y), \text{mother}(z, y), \neg \text{cancer}(x), \neg \text{cancer}(z), \neg \text{sw}_3(y), \text{sw}_4(y) \).

**Restrictions and annotations.** To ensure that the parent-child relationship forms a poly-tree, we require that (1) each patient has at most one son, and (2) the relations induced by \( \text{father} \) and \( \text{mother} \) form a (strict) partial order. Furthermore, we require that the \( \text{father} \) and \( \text{mother} \) relations are disjoint. Whenever the ground atoms in the program satisfy these conditions, we can derive the annotations \( \text{UNQ}(\text{father}, \{1\}), \text{UNQ}(\text{father}, \{2\}), \text{UNQ}(\text{mother}, \{1\}), \text{UNQ}(\text{mother}, \{2\}), \text{ORD}(\{\text{father}, \text{mother}\}) \), and \( \text{DIS}(\text{father}, \text{mother}) \) from the program.

**Relaxed Acyclicity.** We now show that the program is relaxed acyclic. First, we observe that all rules in the program are both weakly and strongly connected. Hence, all undirected structures of the form \( (P, P, \epsilon, \epsilon) \) and \( (\epsilon, P, P, \epsilon) \), where \( P \) is a directed path in the dependency graph, are guarded. Therefore, any undirected cycle involving predicates other than \( \text{cancer} \) is immediately guarded (see Proposition B.8 in Appendix B).

Second, consider an undirected cycle, which is not a directed cycle, that involves only the predicate symbol \( \text{cancer} \). For simplicity, we consider only cycles consisting of 2 distinct paths, denoted \( P_1 \) and \( P_2 \), of length 1. There are two possible cases:

1. \( P_1 = \text{cancer} \xrightarrow{r_{3,3}} \text{cancer} \) and \( P_2 = \text{cancer} \xrightarrow{r_{3,4}} \text{cancer} \). The cycle is guarded by the structure \( \langle \text{cancer} \xrightarrow{r_{3,3}} \text{cancer}, \text{cancer} \xrightarrow{r_{3,4}} \text{cancer}, \epsilon, \epsilon \rangle \) (since we can derive the annotation \( \text{DIS}(\text{father}, \text{mother}) \)).

2. \( P_1 = \text{cancer} \xrightarrow{r_{3,4}} \text{cancer} \) and \( P_2 = \text{cancer} \xrightarrow{r_{3,3}} \text{cancer} \). The cycle is guarded by the structure \( \langle \text{cancer} \xrightarrow{r_{3,4}} \text{cancer}, \text{cancer} \xrightarrow{r_{3,3}} \text{cancer}, \epsilon, \epsilon \rangle \) (since we can derive the annotation \( \text{DIS}(\text{father}, \text{mother}) \)).

Hence, all undirected cycles consisting of two paths of length 1 are guarded. We can extend this result to all undirected cycles (without restrictions) using Proposition B.8 from Appendix B.

Third, consider the two directed cycles \( C_1 = \text{cancer} \xrightarrow{r_{3,3}} \text{cancer} \) and \( C_2 = \text{cancer} \xrightarrow{r_{3,4}} \text{cancer} \). Both are guarded, since we can derive \( \text{ORD}(\{\text{father}, \text{mother}\}) \). Furthermore, any directed cycle can be obtained only by combining \( C_1 \) and \( C_2 \). From this and Proposition B.9 from Appendix B, it follows that all directed cycles are guarded.
Chapter 4. Securing Databases from Probabilistic Inference

<table>
<thead>
<tr>
<th>Mother</th>
<th>Father</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>CC</td>
<td>1 0 0</td>
</tr>
<tr>
<td>CC</td>
<td>CT</td>
<td>1/2 1/2 0</td>
</tr>
<tr>
<td>CC</td>
<td>TT</td>
<td>0 1 0</td>
</tr>
<tr>
<td>CT</td>
<td>CC</td>
<td>1/2 1/2 0</td>
</tr>
<tr>
<td>CT</td>
<td>CT</td>
<td>1/4 1/2 1/4</td>
</tr>
<tr>
<td>CT</td>
<td>TT</td>
<td>0 1/2 1/2</td>
</tr>
<tr>
<td>TT</td>
<td>CC</td>
<td>0 1 0</td>
</tr>
<tr>
<td>TT</td>
<td>CT</td>
<td>0 1/2 1/2</td>
</tr>
<tr>
<td>TT</td>
<td>TT</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

Figure 4.12: Probability distribution for a child’s genome given his parents’ genome, for a fixed position.

Since all directed and undirected cycles are guarded, the program is relaxed acyclic.

4.9.4 Genomic Data Example

We first present an overview of our example. Afterwards, we encode it in ProbLog, and we show how it can be represented as a relaxed acyclic program. Finally, we present some experimental results obtained with Angerona.

4.9.4.1 Overview

Genomic data is extremely sensitive. From an individual’s genome, one can derive his predisposition to different diseases and his relationship to other persons. At the same time, genomic data is becoming increasingly available in hospitals, research centres, and online [1,2,5]. It is thus essential to understand how to secure genomic databases.

We now briefly summarize relevant aspects of genomics, see [42,109] for an in-depth introduction to the topic. A chromosome is a string of DNA consisting of a sequence of complementary nucleotide pairs, where each pair consists either of Adenine (A) and Thymine (T), or Cytosine (C) and Guanine (G). Each cell’s genetic material is stored in 23 pairs of chromosomes, where in each pair one of the chromosomes derives from the corresponding maternal chromosome and the other derives from the paternal one. Figure 4.12 illustrates the probabilistic model defined by Mendel’s inheritance laws.

Humans have 99.9% of their DNA in common. Hence, what characterizes and differentiates us are the variations in our DNA. The most common variation is called Single Nucleotide Polymorphism (SNP), which occurs when a nucleotide (at a specific position) varies among the individuals in a population. SNPs are related to the susceptibility to various diseases, and therefore information about SNPs, precise or probabilistic, is sensitive. For instance, the SNP rs11200638 in the HTRA1 promoter is associated with a 10-fold increased risk of age-related macular degeneration [104].

A simple example of a database for storing and querying patients’ genomic data is as follows. The database contains: a table patient storing the patients’ information; a table doctor storing the doctors’ information; a table patientof, with two attributes doctor and patient, associating the doctors with the patients they treat; a table genome, with four attributes patient, position, value1, and value2, storing the genomic data in the form of SNPs; and tables father and mother, with two attributes parent and child, representing the parent-child relationships. This database is shared between different doctors and each doctor can access only the information associated with the patients he is treating. Furthermore, each doctor has access to the tables father, mother, and patient.

Controlling probabilistic inference is critical in this scenario. Restricting direct access to the genome of the patients not treated by a doctor, for instance by granting to each doctor d access to a view genome_d disclosing only the genome of d’s patients, is insufficient to prevent a malicious doctor from inferring sensitive information about patients that he is not treating. For instance, suppose that a malicious doctor d is treating both Alice and Bob but not Carl, which is Alice and Bob’s son. Suppose too that the nucleotide pair associated with the age-related macular disease is TT for Alice and CT for Bob. The doctor can use this information to infer that Carl has nucleotide TT with probability 50% for the position associated with age-related macular disease and is, therefore, 10 times more likely to suffer from the disease. Securing the database requires restricting direct and indirect access and reasoning about the probabilistic dependencies among the data.

4.9.4.2 Encoding

Original program. We encode the probability distribution associated with the Mendel’s inheritance laws in ProbLog. We assume given the tables patient, father, and mother (denoted respectively P, F, and M). Furthermore, we assume that there are two auxiliary tables withParents and noParents
(denoted WP and NP respectively), which represent whether a patient’s parents are in the database or not. Finally, we assume that the content of the above tables is the same in all possible states. The content of these tables can be encoded using ground non-probabilistic facts.

In contrast, the content of the genome table (denoted by \( G \)) follows the probability distribution shown in Figure 4.12. For simplicity, we assume that patients either have both parents in the database or neither of them, and in all possible database states, all patients have an entry in the genome table. For the patients without parents in the database, the three possible values are uniformly distributed. This is represented using the rule:

\[
\begin{align*}
1/3: & G(x, p, C, C) \leftarrow P(x), NP(x) \\
1/2: & G(x, p, C, T) \leftarrow P(x), NP(x) \\
\end{align*}
\]

In contrast, for patients with both parents in the database, the possible values are distributed according to Mendel’s laws:

\[
\begin{align*}
G(x, p, C, C) & \leftarrow WP(x), F(f, x), G(f, p, C, C), M(m, x), G(m, p, C, C) \\
1/2: & G(x, p, C, T) \leftarrow WP(x), F(f, x), G(f, p, C, C), M(m, x), G(m, p, C, T) \\
G(x, p, C, T) & \leftarrow WP(x), F(f, x), G(f, p, C, T), M(m, x), G(m, p, C, C) \\
1/2: & G(x, p, T, T) \leftarrow WP(x), F(f, x), G(f, p, T, T), M(m, x), G(m, p, C, T) \\
1/2: & G(x, p, C, T) \leftarrow WP(x), F(f, x), G(f, p, T, T), M(m, x), G(m, p, C, T) \\
1/2: & G(x, p, T, T) \leftarrow WP(x), F(f, x), G(f, p, T, T), M(m, x), G(m, p, T, T) \\
\end{align*}
\]

In the rules, we use annotated disjunctions to encode the probability distribution, once the parents’ genome is fixed.

**Remove syntactic sugar.** We now rewrite the program by removing all syntactic sugar, i.e., probabilistic rules and annotated disjunctions. We refer the reader to Section 4.4 for a discussion about how to encode these features in plain ProBLog. Note that we also perform some minor simplifications on the de-sugared program.

\[
\begin{align*}
1/3: & sw_1(\_ ) \\
1/2: & sw_2(\_ ) \\
1/2: & sw_3(\_ ) \\
1/2: & sw_4(\_ ) \\
G(x, C, C) & \leftarrow P(x), NP(x), sw_1(x, C, C) \\
G(x, C, T) & \leftarrow P(x), NP(x), \neg sw_1(x, C, C), sw_2(x, C, T) \\
G(x, T, T) & \leftarrow P(x), NP(x), \neg sw_1(x, C, C), \neg sw_2(x, C, T) \\
G(x, C, C) & \leftarrow WP(x), F(f, x), G(f, C, C), M(m, x), G(m, C, C) \\
G(x, C, T) & \leftarrow WP(x), F(f, x), G(f, C, C), M(m, x), G(m, C, T), sw_3(x, C, T) \\
G(x, T, T) & \leftarrow WP(x), F(f, x), G(f, C, T), M(m, x), G(m, C, T), \neg sw_3(x, C, T) \\
G(x, C, T) & \leftarrow WP(x), F(f, x), G(f, C, C), M(m, x), G(m, T, T), sw_3(x, C, T) \\
G(x, T, T) & \leftarrow WP(x), F(f, x), G(f, C, C), M(m, x), G(m, T, T), \neg sw_3(x, C, T) \\
G(x, C, C) & \leftarrow WP(x), F(f, x), G(f, C, C), M(m, x), G(m, C, T), sw_4(x, C, C) \\
G(x, C, T) & \leftarrow WP(x), F(f, x), G(f, C, C), M(m, x), G(m, C, T), \neg sw_4(x, C, C) \\
\end{align*}
\]

**Syntactic transformations.** By applying the syntactic transformation for relaxed acyclic programs (see Section 4.9.1), we obtain the following program (observe that the rules defining the predicate \( G \)
Whenever the ground atoms in the program satisfy these conditions, we can derive the annotations (Relaxed Acyclicity).

Similar to Section 4.9.3, for undirected cycles that involve only relations are disjoint. Furthermore, we require that these relations capture their intuitive semantics (e.g., a patient belongs to the database).

Restrictions and annotations. We make the following assumptions on the set of ground atoms:

1. To ensure that the parent-child relationship forms a poly-tree, we require that (1) each patient has at most one son, and (2) the relations induced by $F$ and $M$ form a (strict) partial order.
2. We require that the $F$ and $M$ relations are disjoint.
3. We require that the $WP$ and $NP$ relations are disjoint. Furthermore, we require that these relations are safe annotated disjunctions):

\[
\begin{align*}
1/2::sw_1(\_)& \\
1/2::sw_2(\_)& \\
1/2::sw_3(\_)& \\
1/2::sw_4(\_)& \\
G(x) & \leftarrow P(x), NP(x), \neg sw_1(x, c, c), \neg sw_2(x, c, T) \\
G(x) & \leftarrow WP(x), F(f, x), G(f), M(m, x), G(m), \neg sw_3(x, c, T), \neg sw_4(x, c, c)
\end{align*}
\]

The dependency graph of this program is shown in Figure 4.13, where $r_1$ denotes the rule $G(x) \leftarrow P(x), NP(x), \neg sw_1(x, c, c), \neg sw_2(x, c, T)$ and $r_2$ denotes the rule $G(x) \leftarrow WP(x), F(f, x), G(f), M(m, x), G(m), \neg sw_3(x, c, T), \neg sw_4(x, c, c)$. Whenever the ground atoms in the program satisfy these conditions, we can derive the annotations $UNQ(F, \{1\})$, $UNQ(F, \{2\})$, $UNQ(M, \{1\})$, $UNQ(M, \{2\})$, $ORD(\{F, M\})$, $DIS(F, M)$, and $DIS(WP, NP)$ from the program. Note that we can also derive other annotations, such as $UNQ(G, \{1\})$.

Relaxed Acyclicity. We can finally show that the program is relaxed acyclic. Observe that all rules in the program are both weakly and strongly connected. Hence, all undirected structures of the form $(P, P, e, e)$ (and $(e, P, P, e)$), where $P$ is a directed path in the dependency graph, are guarded. Therefore, any undirected cycle involving predicates other than $G$ is immediately guarded. Similar to Section 4.9.3, for undirected cycles that involve only $G$, the acyclicity directly follows from the annotations $DIS(F, M)$ and $DIS(WP, NP)$. Additionally, consider the two directed cycles $C_1 = G \rightarrowsw_3 G$ and $C_2 = G \rightarrowsw_5 G$. Both are guarded, since we can derive $ORD(\{F, M\})$. Furthermore, any directed cycle can be obtained only by combining $C_1$ and $C_2$. From this and Proposition B.9 from Appendix B, it follows that all directed cycles are guarded. Since all directed and undirected cycles are guarded, the program is relaxed acyclic.

4.9.4.3 Experiments

We run our experiments on a PC with an Intel i7 processor and 32GB of RAM. We consider the database schema and the probabilistic model we presented in the previous section. For our experiments, we generate synthetic belief programs containing 100 to 10,000 patients and for each of these instances, we generate 100 random literal queries extracting information from the genome table. For each instance and sequence of queries, we check the security of each query with our prototype, against a policy containing 100 randomly generated secrets specified as literal queries (note that the secrets are specified against the genome table). Note that in our experiments we repeated the above process 5 times, i.e., we generated 5 synthetic belief programs per data point and, for each of the programs, we generated random queries and policies. The average execution time for our experiments is shown in Figure 4.14. After the generation of the BN, ANGERONA takes under 100 milliseconds on average, even for our larger examples, to check a query’s security, whereas the grounding time is
less than 1.5 minutes on our largest examples. We remark that the grounding process is done off-line and just once.
Part III

Beyond SELECT-only attackers
Chapter 5

A Formal Model of Databases

Surely we want solid foundations. What kind of castle can we build on sand? What is the point of devoting effort to balconies and minarets, if the foundation may be so weak as to allow the structure to collapse of its own weight? We want our foundations set on bedrock, designed to last for generations. Who would want an architect who cannot certify the soundness of the foundations of his buildings?

Henry E. Kyburg – Why do we need foundations for modelling uncertainties?

Reasoning about the security of database access control mechanisms requires a formal model of database systems. Such a formal model specifies how the database behaves in response to user commands, and, ultimately, it will serve as a basis for the security proofs.

Existing DBAC works, however, either consider very simplistic formal models [131,165], which account only for \texttt{SELECT} queries, or do not provide any formal model at all [8,41,145]. To address this, we present a formal operational semantics of database systems. In contrast to existing models, our semantics supports advanced database features like triggers and views. As we show in Chapter 6, the advanced features supported by our model are security-critical, and attackers can exploit them to violate a database’s confidentiality and integrity. Note that this chapter is largely based on [87].

Structure. We informally present our system model together with the features supported by our formal model in Section 5.1. In Section 5.2, we formalize our database model as well as the other supported features, such as triggers and views. Finally, Section 5.3 presents our operational semantics of database systems.

5.1 Overview

Here, we overview our system model, and we present the supported features and commands.

5.1.1 System Model

In our system model, shown in Figure 5.1, users interact with two components: a database system and an access control system. The access control system contains both a policy enforcement point and a policy decision point. We assume that all the communication between the users and the components is over secure channels.

Database System. The database system (or database for short) manages the data. The database’s state is represented by a mapping from relation schemas to sets of tuples. We assume that all database operations are atomic.

Users. Users interact with the database where each command is checked by the access control system. Each user has a unique account through which he can issue commands. We overview the main features supported by our formal model in Section 5.1.2.
The system administrator is a distinguished user responsible for defining the database schema and the initial security policy. In addition to issuing queries and commands, he can create user accounts and assign them to users. The administrator interacts with the access control system through a special account admin.

Access Control System. The access control system protects the confidentiality and integrity of the data in the database. It is configured with a security policy $S$, it intercepts all commands issued by the users, and it prevents the execution of commands that are not authorized by $S$. When a user $u$ issues a command $c$, the access control system decides whether $u$ is authorized to execute $c$. If $c$ complies with the policy, then the access control system forwards the command to the database system, which executes $c$ and returns its result to $u$. Otherwise, it raises a security exception and rejects $c$. Note that this corresponds to the Non-Truman model [131]. The access control system also logs all issued commands. When evaluating a command, the access control system can access the database's current state and the log.

5.1.2 Supported features

In this chapter, we consider the following SQL features: SELECT, INSERT, DELETE, GRANT, REVOKE, CREATE TRIGGER, CREATE VIEW, and ADD USER commands. Figure 5.2 presents the syntax of the SQL fragment corresponding to the commands supported by our operational semantics. The fragment's syntax is loosely inspired by Transact-SQL, the SQL's dialect supported by Microsoft SQL Server. Observe that the supported fragment contains the most common SQL commands for data manipulation and access control as well as the core commands for creating triggers and views. The ideas and techniques presented in this chapter (and in the subsequent ones) are general and can be extended to the entire SQL standard.

Users can retrieve information from the database using SELECT commands. Rather than using SQL, however, we formalize queries using the relational calculus (RC), which has a simple and well-defined semantics (see Section 2.2).

We support changes to the database security policy through GRANT and REVOKE commands. In particular, we support GRANT commands to add permissions to the security policy. We also support delegation through GRANT commands with GRANT OPTION. Finally, privileges can be revoked using REVOKE commands with the CASCADE OPTION, i.e., when a user revokes a privilege, he also revokes all the privileges that depend on it [137,162]. Namely, when a user $u$ revokes a privilege $p$ (which has been granted before with GRANT OPTION) from a user $u'$, the system automatically revokes all the privileges that $u'$ has further delegated to other users. This check is recursively applied for each revoked privilege with GRANT OPTION, leading to the revocation of all dependent privileges.

Users can modify the database's content using INSERT and DELETE commands. Specifically, we support INSERT and DELETE commands that explicitly identify the tuple to be inserted or deleted, i.e., commands of the form INSERT INTO table$(x_1,\ldots,x_n)$ VALUES $(v_1,\ldots,v_n)$ and DELETE FROM table WHERE $x_1 = v_1 \land \ldots \land x_n = v_n$, where $x_1,\ldots,x_n$ are table's attributes and $v_1,\ldots,v_n$ are the tuple's values. More complex INSERT and DELETE commands, as well as UPDATES, can be simulated by combining SELECT, INSERT, and DELETE commands.

Our model also supports triggers, which are procedures automatically executed by the database system in response to user commands. In particular, we support only AFTER triggers on INSERT and DELETE events, i.e., triggers that are executed in response to INSERT and DELETE commands. Our operational semantics supports triggers that are executed either under the privileges of the trigger's owner or under the privileges of the user that executed the command that activated the trigger. The
triggers’ WHEN conditions are arbitrary boolean queries and their actions are GRANT, REVOKE, INSERT, or DELETE commands. Note that database systems usually impose severe restrictions on the WHEN clause, such as it must not contain sub-queries. However, most database systems can express arbitrary conditions on triggers by combining control flow statements with SELECT commands inside the trigger’s body. Thus, we support the class of triggers whose body is of the form BEGIN IF expr THEN act END, where act is either a GRANT, REVOKE, INSERT, or DELETE command.

Finally, we support views with both the owner’s privileges and the activator’s privileges. Additionally, we support CREATE commands for creating new triggers and views as well as ADD USER commands to create new user accounts. We also support two kinds of integrity constraints: functional dependencies and inclusion dependencies [10]. They model the most widely used families of SQL integrity constraints, namely the UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints.

5.2 Formal Database Model

We now formalize databases including features like views, security policies, and triggers. Our formalization of databases and queries follows Chapter 2, and our security policies formalize SQL policies. In the following, let $U$, $V$, and $T$ be mutually disjoint, countably infinite sets, respectively representing identifiers of users, views, and triggers. Furthermore, we assume that $R \cap U = \emptyset$, $R \cap V = \emptyset$, and $R \cap T = \emptyset$, where $R$ is the set of relation identifiers from Section 2.2.1.

5.2.1 Databases and Queries

We use the database model presented in Section 2.2. Furthermore, we use the Relational Calculus as query language (see Section 2.2.2). We also support integrity constraints. In this respect, we consider only functional and inclusion dependencies. Recall that functional dependencies are sentences of the form $x \rightarrow y$, where $x$ and $y$ are sets of attributes. Inclusion dependencies are sentences of the form $x \subseteq y$, where $x$ and $y$ are sets of attributes.

5.2.2 Views

Let $D$ be a schema. A view $V$ over $D$ is a tuple $⟨id, o, q, m⟩$, where $id \in V$ is the view identifier, $o \in U$ is the view’s owner, $q$ is the non-boolean query over $D$ defining the view, and $m \in \{A,O\}$ is the security mode, where $A$ stands for activator’s privileges and $O$ stands for owner’s privileges. Note that the query $q$ may refer to other views. We assume, however, that views have no cyclic dependencies between them. We denote by $\mathcal{VIEW}_D$ the set of all views over $D$.

The materialization of a view $(V,o,q,m)$ in a state $s$, denoted by $s(V)$, is $[q]^s$. We extend the relational calculus in the standard way to work with views [10]. Namely, we treat the view materialization in the current database state as the relation associated with the corresponding predicate symbol.

Given a view $V$, we denote $V$’s owner by $owner(V)$. Moreover, given a formula $\phi$, we denote by $\text{tables}(\phi)$ the set of tables occurring in the formula $\phi'$, where $\phi'$ is obtained from $\phi$ by (recursively) replacing all views with their definitions. Let $D$ be a database schema and $V$ be a set of views over $D$. A query $q$ is defined over $D$ and $V$, denoted by $\text{defined}(q,D,V)$, if $q$ refers only to relation schemas in $D$ and views in $V$. Moreover, a view $v$ is defined over $D$ and $V$ if $\text{query specifying } v \text{ is defined over } D$ and $V$.

5.2.3 Security Policies

We now formalize the SQL access control model [137,162]. We first formalize five privileges. Let $D$ be a database schema. A SELECT privilege over $D$ is a tuple $⟨\text{SELECT}, R⟩$, where $R$ is a relation schema in $D$ or a view over $D$. A CREATE VIEW privilege over $D$ is a tuple $⟨\text{CREATE VIEW}⟩$. An INSERT privilege over $D$ is a tuple $⟨\text{INSERT}, R⟩$, a DELETE privilege over $D$ is a tuple $⟨\text{DELETE}, R⟩$, and a CREATE TRIGGER privilege over $D$ is a tuple $⟨\text{CREATE TRIGGER}, R⟩$, where $R$ is a relation schema in $D$. We denote by $\mathcal{PRIV}_D$ the set of privileges over $D$. Given a database schema $D$ and a set of views $V$ over $D$, we say that a privilege $p$ is defined over $D$ and $V$, denoted by $\text{defined}(p,D,V)$, if $p$ refers only to tables and views in $D$ and $V$.

Following SQL, we use GRANT commands to assign privileges to users. Let $U \subseteq \mathfrak{U}$ be a set of users and $D$ be a database schema. We now define $(U,D)$-grants and $(U,D)$-revokes. There are two types of $(U,D)$-grants. A $(U,D)$-simple grant is a tuple $⟨\diamondsuit, u,p,u'⟩$, where $u \in U$ is the user receiving the privilege $p \in \mathcal{PRIV}_D$ and $u' \in U$ is the user granting this privilege. A $(U,D)$-grant with grant option is a tuple $⟨\neg\diamondsuit, u,p,u'⟩$, where $u$, $p$, and $u'$ are as before. $(U,D)$-revoke is a tuple $⟨\neg\diamondsuit, u,p,u'⟩$, where $u \in U$ is the user from which the privilege $p \in \mathcal{PRIV}_D$ will be revoked and $u' \in U$ is the user revoking this privilege. We denote by $\mathcal{GRANT}_D$ the set of all $(U,D)$-grants and $(U,D)$-revokes. A grant
(⊕, u, p, u′) models the command GRANT p TO u issued by u′, a grant with grant option (⊕∗, u, p, u′) models the command GRANT p TO u WITH GRANT OPTION issued by u′, and a revoke (⊖, u, p, u′) models the command REVOKE p FROM u CASCADE issued by u′.

Finally, we define a (U, D)-security policy S as a finite set of (U, D)-grants. We denote by $S_{U,D}$ the set of all (U, D)-policies.

In the following, we formalize the semantics of GRANT and REVOKE commands, i.e., how their execution modifies the security policy.

**Semantics of GRANT commands.** Executing a GRANT command $⟨⊕, u, p, u′⟩$ on a security policy S simply amounts to adding the GRANT to the policy, i.e., $S ∪ \{⟨⊕, u, p, u′⟩\}$. The same holds for GRANTS with grant option.

**Semantics of REVOKE commands.** We now define the function revoke that models the semantics of SQL’s REVOKE statements with cascade. In the following, let S be a security policy and p be a privilege. A *chain* is a sequence of grants $g_1, g_2, \ldots, g_n$ such that (1) there is a user identifier u and an op ∈ {⊕, ⊖} such that $g_1 = \langle op, u, p, start \rangle$, (2) if $p \neq \langle SELECT, V \rangle$, where V is a view with owner’s privileges, then $start = \langle admin \rangle$, and $start \in \{\langle admin \rangle, owner(V)\}$ otherwise, and (3) for each $1 \leq i \leq n - 1$, there are user identifiers $u, u′, u''$ and an op ∈ {⊕, ⊖} such that $g_i = \langle op, u', p, u \rangle$ and $g_{i+1} = \langle op, u'', p, u' \rangle$. The function $\text{chain} : S_{U,D} → 2^{\Omega_{U,D}}$, which takes as input a policy S and constructs all possible chains, is the smallest function satisfying the following recurrence relation:

\[
\text{chain}(S) = \{\langle op, u, p, u' \rangle \in S \mid u' = \langle admin \rangle\} ∪ \\
\{\langle op, u, \langle SELECT, V \rangle, u' \rangle \in S \mid V = \{v, o, q, O\} \land u' = o\} ∪ \\
\bigcup_{G \in \text{chain}(S)} \{G \cdot g \mid g \in S \land G = \langle op, u, p, u' \rangle \land G([G]) = \langle op^*, u', p, u'' \rangle \land \\
\forall i \in \{1, \ldots, |G|\}. G(i) \neq g\}.
\]

The function $\text{filter}$ takes as input a set of chains C and a grant g and outputs the set of all chains in C not containing g:

\[
\text{filter}(C, g) := \{g_1 \cdots g_n \in C \mid \forall i \in \{1, \ldots, n\}. g_i \neq g\}.
\]

The function $\text{policy}$ constructs a policy starting from a set of chains C:

\[
\text{policy}(C) := \bigcup_{g_1 \cdots g_n \in C} \bigcup_{1 \leq i \leq n} \{g_i\}.
\]

Finally, the function $\text{revoke}$, which models the semantics of the REVOKE command, is as follows:

\[
\text{revoke}(S, u, p, u') := \text{policy}(\text{filter}(\text{policy}(\text{filter}(\text{chain}(S), \langle ⊖, u, p, u' \rangle))), \langle ⊖^*, u, p, u' \rangle).
\]

Given a policy S, $\text{revoke}(S, u, p, u')$ denotes the policy obtained by applying $⟨⊖, u, p, u'⟩$ to S.

**Example 5.1.** Consider a database schema D with three relation schemas $N$, $P$, and $T$, all with arity 1. Moreover, consider the following policy: the user u has read and write access to the tables $P$ and $N$. Finally, assume that the only other user is the administrator $\langle admin \rangle$. This setting can be formalized as follows. The set $U = \{u, \langle admin \rangle\}$ and the policy $S$ contains the following grants: $⟨⊕, u, \langle SELECT, P \rangle, \langle admin \rangle⟩$, $⟨⊖, u, \langle SELECT, N⟩, \langle admin \rangle⟩$, $⟨⊖, u, \langle INSERT, N⟩, \langle admin \rangle⟩$, and $⟨⊖, u, \langle DELETE, N⟩⟩$.

**5.2.4 Triggers**

Let $D = (\Sigma, \text{dom})$ be a database schema. A trigger over D is a tuple $⟨id, u, e, R, φ, a, m⟩$, where id ∈ T is the trigger identifier, u ∈ U is the trigger’s owner, e ∈ {INS, DEL} is the trigger event (where INS stands for INSERT and DEL stands for DELETE), R is a relation schema in $\Sigma$, the trigger condition $φ$ is a relational calculus formula such that $\text{free}(φ) \subseteq \{x_1, \ldots, x_{|R|}\}$, and the trigger action a is one of: (1) $⟨\langle INSERT, R', T \rangle⟩$, where $R' \in D$ and T is a $|R'|$-tuple of values in dom and variables in $\{x_1, \ldots, x_{|R|}\}$, (2) $⟨\langle DELETE, R', T \rangle⟩$, where $R'$ and T are as before, or (3) $⟨\langle op, u, p, \rangle⟩$, where $op ∈ \{⊕, ⊖^*, \subset\}$, u ∈ U, and p is a privilege over D. Finally, $m ∈ \{A, O\}$ is the security mode, where A stands for activator’s privileges and O stands for owner’s privileges. We denote by $T\text{TRIGGER}_D$ the set of all triggers over D. Furthermore, given a trigger t, we denote t’s owner by $\text{owner}(t)$ and we denote its security mode as $\text{mode}(t)$. To illustrate, the trigger $(t, \text{admin}, \text{INS}, P, T(x_1)), \langle\langle INSERT, N, x_1⟩⟩, O)$ models a trigger with identifier t, created by $\text{admin}$, that is executed (under $\text{admin}$’s privileges) whenever a user inserts a tuple in the relation $P$, and if $T(v_1)$ holds in the current database state, it inserts $N(v_1)$ into the table $N$. Here, $x_1$ is bound, at run-time, to the value $v_1$ inserted in the table $P$ by the trigger’s invoker.
5.3 Operational Semantics

We formalize our system model as a labelled transition system (LTS). First, we define a system configuration, which describes the database schema and the integrity constraints, and the user actions. Afterwards, we define the system's state, which represents a snapshot of the system that contains the database's state, the identifiers of the users interacting with the system, the access control policy, and the current triggers and views in the system. Finally, we formalize the system's behavior as a small step operational semantics, including all features necessary to reason about security, even in the presence of attacks exploiting advanced database features.

System configurations. A system configuration is a tuple \( \langle D, \Gamma \rangle \) such that \( D \) is a database schema and \( \Gamma \) is a finite set of integrity constraints over \( D \). We assume that the constraints in \( \Gamma \) are functional or inclusion dependencies. Observe that functional and inclusion dependencies model primary key and foreign key constraints, some of the most common integrity constraints used in practice in database systems.

User actions. Let \( M = \langle D, \Gamma \rangle \) be a system configuration and \( u \in U \) be a user. A \( (D, u) \)-action is one of the following tuples:

- \( \langle u, \text{ADD}_u, u' \rangle \), where \( u = \text{admin} \) and \( u' \in U \setminus \{ \text{admin} \} \),
- \( \langle u, \text{SELECT}, q \rangle \), where \( q \) is a boolean query over \( D \),
- \( \langle u, \text{INSERT}, R, \bar{t} \rangle \), where \( R \in D \) and \( \bar{t} \in \text{dom}^{\bar{R}} \),
- \( \langle u, \text{DELETE}, R, \bar{t} \rangle \), where \( R \) and \( \bar{t} \) are as above,
- \( \langle \text{op}, u', p, u \rangle \), where \( \langle \text{op}, u', p, u \rangle \in \Omega^{\text{TRIGGER}} \), or
- \( \langle u, \text{CREATE}, o \rangle \), where \( o \in \text{TRIGGER} \). The set of all \( (D, u) \)-actions and by \( A_{D,u} \), for some \( U \subseteq U \), the set \( \bigcup_{u \in U} A_{D,u} \).

We denote by \( A_{D,U} \) the set of all \((D, u)\)-actions and by \( A_{D,U} \), for some \( U \subseteq U \), the set \( \bigcup_{u \in U} A_{D,u} \).

System states. Given a system configuration \( M = \langle D, \Gamma \rangle \), an \( M \)-system state is a tuple \( (db, U, \text{sec}, T, V) \) such that \( db \in \Omega^D \) is a database state, \( U \subseteq U \) is a finite set of users such that \( \text{admin} \in U \), \( \text{sec} \in \mathcal{S}_{U,D} \) is a security policy, \( T \) is a finite set of safe triggers over \( D \), and \( V \) is a finite set of views over \( D \). We denote by \( \Omega_M \) the set of all \( M \)-system states.

Contexts. An \( M \)-context, where \( M \) is a system configuration, describes the system's history, the scheduled triggers that must be executed, and how to modify the system's state in case a roll-back occurs. We denote by \( \mathcal{C}_M \) the set of all \( M \)-contexts. We assume that \( \mathcal{C}_M \) contains a distinguished element \( \varepsilon \) representing the empty context, which is the context in which the system starts.

We now formalize contexts and all their components. In the following, let \( M = \langle D, \Gamma \rangle \) be a system configuration and \( u \) be a user. An \( (M, u) \)-action effect is a tuple \( \langle \text{act}, \text{secDec}, \text{res}, E \rangle \), where \( \text{act} \in A_{D,u} \) is an action, \( \text{secDec} \in \{ \top, \bot \} \) is the security decision for that action (where \( \top \) stands for permit and \( \bot \) stands for deny), \( \text{res} \in \{ \top, \bot \} \) is the action's result, and \( E \subseteq \Gamma \) is the set of integrity constraints violated by the action. We denote by \( \Omega^{\text{actEff}} \) the set of all \((M, u)\)-action effects and by \( \Omega^{\text{actEff}} \), for some \( U \subseteq U \), the set \( \bigcup_{u \in U} \Omega^{\text{actEff}} \). An \( (M, u) \)-trigger effect is a tuple \( \langle t, \text{when}, \text{stmt} \rangle \) where \( t \in \text{TRIGGER} \) is a trigger, \( \text{when} \in \Omega^{\text{trigEff}} \), and \( \text{stmt} \in \Omega^{\text{stmtEff}} \) is the action effect associated with the trigger's \( \text{when} \) condition, and \( \text{stmt} \in \Omega^{\text{stmtEff}} \cup \{ \varepsilon \} \) is the action effect associated with the statement in the trigger's body. We denote by \( \Omega^{\text{trigEff}} \), the set of all \((M, u)\)-trigger effects and by \( \Omega^{\text{stmtEff}} \), for some \( U \subseteq U \), the set \( \bigcup_{u \in U} \Omega^{\text{stmtEff}} \).

\( ^7 \) Without loss of generality, we focus only on boolean queries [10]. We can support non-boolean queries as follows. Given a database state \( s \) and a query \( q := \{ \varphi | \psi \} \), if the access control mechanism authorizes the boolean query \( \bigwedge_{\varphi \in \varphi} \bigwedge_{\psi \in \psi} \varphi \rightarrow \psi \wedge (\forall \varphi, \psi \rightarrow \bigvee_{\varphi \in \varphi} \varphi \neq \top) \), then we return \( q \)’s result, and otherwise we reject \( q \) as unauthorized.
An $M$-trigger transaction is a 4-tuple $(s, t, u, tr)$, where $s \in \Pi_M \cup \{\epsilon\}$ is an $M$-system state representing the “roll-back state”, i.e., the state that should be restored in case a roll-back happens, $t \in \{\epsilon\} \cup \bigcup_{M \in M} \text{dom}$ is the tuple involved in the event that has fired the transaction, $u \in U \cup \{\epsilon\}$ is the user that has activated the triggers in the transactions, and $tr \in TRIGGER_D$ is a sequence of triggers. Note that we denote by $(e, e, e, e)$ the empty $M$-trigger transaction.

An $M$-history $h$ is a sequence of action effects and triggers, i.e., $h \in (\Omega_{M\text{actEff}} \cup \Omega_{M\text{trEff}})^*$. and we denote by $H_M$ the set of all $M$-histories.

We are now ready to formally define contexts. Let $M = (D, \Gamma)$ be a system configuration. An $M$-context is a tuple $(h, actEff, tr)$, where $h \in H_M$ models the system’s history, $actEff \in \Omega_{M\text{actEff}} \cup \Omega_{M\text{trEff}} \cup \{\epsilon\}$ describes the last action’s effect, i.e., whether the action has been accepted by the access control mechanism and the action’s result, and $tr$ is an $M$-trigger transaction. Furthermore, the empty context $\epsilon$ is the element $(\epsilon, \epsilon, (\epsilon, \epsilon, \epsilon))$.

**Runtime states.** An $M$-runtime state is a tuple $(db, U, sec, T, V, cxt)$ such that $(db, U, sec, T, V)$ is an $M$-system state and $cxt \in C_M$ is an $M$-context. We denote by $\Omega_M$ the set of all $M$-runtime states. Observe that we often write $(db, U, sec, T, V, h, actEff, tr)$ instead of $(db, U, sec, T, V, (h, actEff, tr))$. For simplicity, we sometimes write $(s, cxt)$, where $s$ is a system state $(db, U, sec, T, V)$ and $cxt$ is a context to represent the runtime state $(db, U, sec, T, V, c)$. Finally, whenever this is clear from the context, we often refer to system states and runtime states simply as states.

A runtime state $(db, U, sec, T, V, c)$ is initial iff (a) $sec$ contains only grants issued by $admin$, (b) $T$ (respectively $V$) contains only triggers (respectively viewed) owned by $admin$, and (c) $c = e$. We denote by $I_M$ the set of all initial states. Given a runtime state $s = (db, U, sec, T, V, c)$, we denote by $sysState(s)$ the $M$-system state $(db, U, sec, T, V)$ obtained from $s$ by dropping the context $c$ and by $cxt(s)$ its context $c$. Furthermore, given an $M$-state $s = (db, sec, U, T, V, c)$, we use a dot notation to refer to its components. For instance, we use $s.db$ to refer to the database’s state in $s$ and $s.sec$ to refer to the policy in $s$.

**Policy Decision Points.** In a system configuration $M$, an $M$-Policy Decision Point (M-PDP) is a total function $f: \Omega_M \times A_{D,M} \rightarrow \{\top, \bot\}$ that maps each runtime state $s$ and action $a$ to a security decision represented by a boolean value, where $\top$ stands for permit and $\bot$ stands for deny. An extended configuration $\langle M, f \rangle$ is a tuple $\langle M, f \rangle$, where $M$ is a system configuration and $f$ is an M-PDP.

**Operational Semantics.** We now define the LTS representing the system model.

**Definition 5.1.** Let $P = \langle M, f \rangle$ be an extended configuration, where $M = (D, \Gamma)$ and $f$ is an $M$-PDP. The $P$-LTS is the labelled transition system $\langle S, A, \rightarrow_f, I \rangle$, where $S = \Omega_M$ is the set of states, $A = A_{D,M} \cup TRIGGER_D$ is the set of actions, $\rightarrow_f \subseteq S \times A \times S$ is the transition relation, and $I = I_M$ is the set of initial states.

Let $P = \langle M, f \rangle$ be an extended configuration. A run $r$ of a $P$-LTS $L$ is a finite alternating sequence of runtime states and actions, which starts with an initial state $s$, ends in some state $s'$, and respects the transition relation $\rightarrow_f$. We denote by $\text{traces}(L)$ the set of all $L$’s runs. Given a run $r$, $|r|$ denotes the number of states in $r$, last($r$) denotes $r$’s last state, and $r^i$, where $1 \leq i \leq |r|$, denotes the run obtained by truncating $r$ at the $i$-th state. With a slight abuse of notation, we denote by $r^0$ the empty run, i.e., the one without states.

The relation $\rightarrow_f$ formalizes the system’s small step operational semantics, and it is defined in Figures 5.3-5.6. Observe that in the rules we assume fixed a database schema $D$. Note also that the rules rely on several auxiliary functions that we define in Section 5.3.1.

Figure 5.3 presents the rules for the ADD USER, SELECT, GRANT, and REVOKE commands. For each of these commands, there are two rules: one handling the command’s successful execution, which is applied in case the command is authorized by the PDP $f$, and the other handling commands whose execution has been deemed insecure by the PDP $f$. For instance, the rule SELECT SUCCESS models the system’s behavior when the user $u$ issues a SELECT query $q$ that is authorized by the PDP $f$, whereas the SELECT DENY rule models the system’s behavior whenever a SELECT query is unauthorized. Both rules check the security decision produced by the PDP $f$ and whether the query is well-defined given the database schema $D$ and the current set of views $V$. Additionally, the SELECT SUCCESS rule also computes $q$’s result on the current database state $db$. The only component of the $M$-state $s$ that changes is the context, which is updated by extending the history with the action event associated with the SELECT query. Observe that, for the rule SELECT SUCCESS, the action event also contains $q$’s result on the current database state. The rules for the other commands are similar to those for the SELECT command, the main difference being how they modify the database state.

Figure 5.4 formalizes the semantics for INSERT and DELETE commands. We illustrate the rules using INSERT commands as an example. The rules for DELETE commands are similar. The rules INSERT SUCCESS 1 and INSERT SUCCESS 2 formalize the successful execution of INSERT commands. They update the database state by adding the tuple $t$ to the relation $R$. They also check that (1) the PDP $f$ authorizes the INSERT command, and (2) the database state obtained by executing the
5.3. Operational Semantics

\[ \begin{align*}
\text{ADD USER Success} & \quad aE' = \langle \langle \text{admin}, \text{ADD USER}, u \rangle, \top, \top, \emptyset \rangle \quad \text{admin} \in U \quad f((db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle), \langle \text{admin}, \text{ADD USER}, u \rangle) = \top \\
\langle db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle \rangle \xrightarrow{\langle \text{admin}, \text{ADD USER}, u \rangle} \langle db, U \cup \{u\}, sec, T, V, h \cdot aE, aE', \langle e, e, e \rangle \rangle
\end{align*} \]

\[ \begin{align*}
\text{ADD USER Deny} & \quad aE' = \langle \langle \text{admin}, \text{ADD USER}, u \rangle, \bot, \bot, \emptyset \rangle \quad \text{admin} \in U \quad f((db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle), \langle \text{admin}, \text{ADD USER}, u \rangle) = \bot \\
\langle db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle \rangle \xrightarrow{\langle \text{admin}, \text{ADD USER}, u \rangle} \langle db, U \cup \{u\}, sec, T, V, h \cdot aE, aE', \langle e, e, e \rangle \rangle
\end{align*} \]

\[ \begin{align*}
\text{SELECT Success} & \quad u \in U \quad f(\langle db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle \rangle, \langle u, \text{SELECT}, q \rangle) = \top \\
\langle db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle \rangle \xrightarrow{\langle u, \text{SELECT}, q \rangle} \langle db, U, sec, T, V, h \cdot aE, aE', \langle e, e, e \rangle \rangle
\end{align*} \]

\[ \begin{align*}
\text{SELECT Deny} & \quad u \in U \quad f(\langle db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle \rangle, \langle u, \text{SELECT}, q \rangle) = \bot \\
\langle db, U, sec, T, V, h, aE, \langle rs, T', u', e \rangle \rangle \xrightarrow{\langle u, \text{SELECT}, q \rangle} \langle db, U, sec, T, V, h \cdot aE, aE', \langle e, e, e \rangle \rangle
\end{align*} \]

\[ \begin{align*}
\text{GRANT Success} & \quad u, u' \in U \quad f(\langle db, U, sec, T, V, h, aE, \langle rs, T, u'' \rangle \rangle, \langle op, u, p, u' \rangle) = \top \\
\langle db, U, sec, T, V, h, aE, \langle rs, T, u'' \rangle \rangle \xrightarrow{\langle op, u, p, u' \rangle} \langle db, U, sec', T, V, h \cdot aE, aE', \langle e, e, e \rangle \rangle
\end{align*} \]

\[ \begin{align*}
\text{REVOKE Success} & \quad u, u' \in U \quad f(\langle db, U, sec, T, V, h, aE, \langle rs, T, u'' \rangle \rangle, \langle \circ, u, p, u' \rangle) = \top \\
\langle db, U, sec, T, V, h, aE, \langle rs, T, u'' \rangle \rangle \xrightarrow{\langle \circ, u, p, u' \rangle} \langle db, U, sec', T, V, h \cdot aE, aE', \langle e, e, e \rangle \rangle
\end{align*} \]

\[ \begin{align*}
\text{GRANT-REVOKE Deny} & \quad u, u' \in U \quad f(\langle db, U, sec, T, V, h, aE, \langle rs, T, u'' \rangle \rangle, \langle op, u, p, u' \rangle) = \bot \\
\langle db, U, sec, T, V, h, aE, \langle rs, T, u'' \rangle \rangle \xrightarrow{\langle op, u, p, u' \rangle} \langle db, U, sec, T, V, h \cdot aE, aE', \langle e, e, e \rangle \rangle
\end{align*} \]

Figure 5.3: Rules for the ADD USER, SELECT, GRANT, and REVOKE commands.
Chapter 5. A Formal Model of Databases

**INSERT Success 1**

\[
\begin{align*}
&\text{act} = (u, \text{INSERT}, R, \overline{T}) \\
&\text{filter}(T, \text{INS}, R) = \epsilon \lor \overline{T} \in \text{db}(R) \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{INSERT}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

**INSERT Success 2**

\[
\begin{align*}
&\text{act} = (u, \text{INSERT}, R, \overline{T}) \\
&\text{filter}(T, \text{INS}, R) = \epsilon \lor \overline{T} \in \text{db}(R) \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{INSERT}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

**INSERT Exception**

\[
\begin{align*}
&\text{act} = (u, \text{INSERT}, R, \overline{T}) \\
&E' = \{ \phi \in \Gamma \mid [\phi]^{[R \ominus \overline{T}]} = \perp \} \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{INSERT}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

**INSERT Deny**

\[
\begin{align*}
&\text{act} = (u, \text{INSERT}, R, \overline{T}) \\
&\text{filter}(T, \text{INS}, R) = \epsilon \lor \overline{T} \in \text{db}(R) \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{INSERT}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

**DELETE Success 1**

\[
\begin{align*}
&\text{act} = (u, \text{DELETE}, R, \overline{T}) \\
&\text{filter}(T, \text{DEL}, R) = \epsilon \lor \overline{T} \in \text{db}(R) \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{DELETE}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

**DELETE Success 2**

\[
\begin{align*}
&\text{act} = (u, \text{DELETE}, R, \overline{T}) \\
&\text{filter}(T, \text{DEL}, R) = \epsilon \lor \overline{T} \in \text{db}(R) \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{DELETE}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

**DELETE Exception**

\[
\begin{align*}
&E' = \{ \phi \in \Gamma \mid [\phi]^{[R \ominus \overline{T}]} = \perp \} \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{DELETE}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

**DELETE Deny**

\[
\begin{align*}
&\text{act} = (u, \text{DELETE}, R, \overline{T}) \\
&\text{filter}(T, \text{DEL}, R) = \epsilon \lor \overline{T} \in \text{db}(R) \\
&\langle db, U, \text{sec}, T, V, h, aE, (rS, \overline{T}, u', \epsilon) \rangle \xrightarrow{(u, \text{DELETE}, R, \overline{T})} \langle db[R \ominus \overline{T}], U, \text{sec}, T, V, h \cdot aE, aE', (\epsilon, \epsilon, \epsilon, \epsilon) \rangle
\end{align*}
\]

Figure 5.4: Rules for the INSERT and DELETE commands.
Figure 5.5: Rules for triggers.
CREATE TRIGGER SUCCESS
\[ u \in U \quad \text{defined}(t, D, V) \quad \text{safe}(\{t\} \cup T) \quad \text{usersIn}(t, U) \]
\[
\tau = (id, u, ev, R, \phi, \text{stmt}, m) \\
\tau' = (id, ou', e', R', \phi', \text{stmt}', m')
\]
\[
\langle db, u, sec, T, V, h, aE, \langle s, \bar{t}, u', \epsilon \rangle \rangle \sim f \langle db, U, sec, T \cup \{t\}, V, h, aE, \langle e, \epsilon, \epsilon \rangle \rangle
\]

CREATE TRIGGER DENY
\[ u \in U \quad \text{defined}(t, D, V) \quad \text{safe}(\{t\} \cup T) \quad \text{usersIn}(t, U) \]
\[
\tau = (id, u, ev, R, \phi, \text{stmt}, m) \\
\tau' = (id, ou', e', R', \phi', \text{stmt}', m')
\]
\[
\langle db, u, sec, T, V, h, aE, \langle s, \bar{t}, u', \epsilon \rangle \rangle \sim f \langle db, U, sec, T \cup \{t\}, V, h, aE, \langle e, \epsilon, \epsilon \rangle \rangle
\]

CREATE VIEW SUCCESS
\[ u \in U \quad \text{defined}(v, D, V) \quad f(\langle db, U, sec, T, V, h, aE, \langle s, \bar{t}, u', \epsilon \rangle \rangle, \langle u, \text{CREATE}, v \rangle ) = \top \]
\[
v = (id, u, q, m) \\
aE' = \langle u, \text{CREATE}, v, T, T, \emptyset \rangle \\
\exists v' \in V. v' = (id, ou', q', m')
\]
\[
\langle db, U, sec, T, V, h, aE, \langle s, \bar{t}, u', \epsilon \rangle \rangle \sim f \langle db, U, sec, T \cup \{v\}, h, aE, \langle e, \epsilon, \epsilon \rangle \rangle
\]

CREATE VIEW DENY
\[ u \in U \quad \text{defined}(v, D, V) \quad f(\langle db, U, sec, T, V, h, aE, \langle s, \bar{t}, u', \epsilon \rangle \rangle, \langle u, \text{CREATE}, v \rangle ) = \top \]
\[
v = (id, u, q, m) \\
aE' = \langle u, \text{CREATE}, v, T, T, \emptyset \rangle \\
\exists v' \in V. v' = (id, ou', q', m')
\]
\[
\langle db, U, sec, T, V, h, aE, \langle s, \bar{t}, u', \epsilon \rangle \rangle \sim f \langle db, U, sec, T \cup \{v\}, h, aE, \langle e, \epsilon, \epsilon \rangle \rangle
\]

DELETE command still satisfies the integrity constraints. Observe that the INSERT SUCCESS 1 rule handles INSERT commands that do not activate triggers. This is done by ensuring that (1) there are no triggers in the system that could be executed in response to the INSERT command (using the filter function formalized in Section 5.3.1), or (2) the execution of the INSERT command does not modify the database state. The INSERT SUCCESS 2 rule, instead, handles INSERT commands that activate triggers. The rule updates the current context by appending the transaction associated with the triggers to be executed. Furthermore, the INSERT EXCEPTION rule handles authorized INSERT commands that violate the integrity constraints and therefore throw integrity exceptions. The rule updates the system’s context by recording the set of violated integrity constraints \( \{ \phi \in \Gamma | [\phi]^{\text{DB}\not\in\mathbb{R}_\mathbb{P}]} = \top \). Finally, the INSERT DENY rule handles INSERT commands that are deemed insecure by the PDP \( f \) and it updates the context accordingly.

The rules for triggers are presented in Figure 5.5. Note that these rules rely on the functions getActualUser, apply, and action, which are formalized in Section 5.3.1. The rules Trigger INSERT-DELETE SUCCESS, Trigger GRANT SUCCESS, and Trigger REVOKE SUCCESS formalize the semantics of triggers that successfully execute commands, without violating the security policy or causing integrity exceptions. These rules are inspired by the corresponding rules for the successful execution of INSERT, DELETE, GRANT, and REVOKE commands. Additionally, these rules modify the system’s context by consuming the first element in the trigger’s transaction. This ensures that triggers are executed in the order specified by the transaction. Furthermore, the Trigger DISABLED rule handles the execution of triggers whose WHEN condition is not satisfied in the current state. In this case, the database is not modified as a result of the trigger’s execution, and the system’s context is updated by consuming the first element in the current transaction. Finally, the rules Trigger INSERT-DELETE EXCEPTION, Trigger DENY CONDITION, and Trigger DENY ACTION formalize what happens whenever triggers throw exceptions. The first rule formalizes the semantics of triggers whose actions violate integrity constraints. The handling of the command execution and of the exceptions is similar to the INSERT EXCEPTION and DELETE EXCEPTION rules from Figure 5.4. The rules Trigger DENY CONDITION and Trigger DENY ACTION, instead, formalize the system’s behavior in case the condition or the action associated with the trigger violate the security policy. These three rules (1) terminate the execution of the triggers still in the transactions by setting the list of triggers to be processed to \( \epsilon \), and (2) roll-back the current database state to the one stored in the transaction (i.e., the one before executing the command that fired all the triggers). This guarantees that our semantics matches the behavior of existing database systems, where exceptions caused by triggers terminate the execution of all triggers and roll-back the database state.
Finally, Figure 5.6 illustrates the rules associated with the CREATE VIEW and CREATE TRIGGER commands. The rules CREATE TRIGGER SUCCESS and CREATE VIEW SUCCESS handle the successful execution of CREATE commands. The rules CREATE TRIGGER DENY and CREATE VIEW DENY formalize the unsuccessful execution of CREATE commands due to clashes between triggers and views identifiers. Finally, the rule CREATE DENY formalizes how the system handles CREATE commands that violate the security policy.

We developed an executable version of our operational semantics, available at [86], using the Maude term-rewriting framework [54]. The executable model acts as a reference implementation for our semantics. We remark that both our operational semantics and the corresponding executable model can be tailored to model the behavior of specific database systems. Given the complexity of databases and their features, having an executable version of our semantics provides a way to validate it against existing database systems.

5.3. Operational Semantics

Here we define several auxiliary functions to extract information from runtime states.

Function used in the operational semantics

We now define the functions mentioned in the rules in Figures 5.3–5.6. The getActualUser(m, invk, ow) function takes as input a security mode m ∈ {A, O} and two users invk and ow in U, and it outputs one of the two users depending on m:

\[
\text{getActualUser}(m, \text{invk}, \text{ow}) = \begin{cases} 
\text{invk} & \text{if } m = A \\
\text{ow} & \text{if } m = O
\end{cases}
\]

The ID function takes as input an action \( \text{act} \in \mathcal{A}_{D,M} \) and returns \( \top \) if \( \text{act} \) is either \( \langle u, \text{INSERT}, R, \mathcal{T} \rangle \) or \( \langle u, \text{DELETE}, R, \mathcal{T} \rangle \), for some \( u \in U, R \in D \), and \( \mathcal{T} \in \text{dom}(R) \). The function ID returns \( \bot \) otherwise.

The apply function takes as input an INSERT or DELETE action \( \text{act} \in \mathcal{A}_{D,M} \) and a database state \( db \in \Omega_D \) and it returns as output the updated database state. The function is defined as follows:

\[
\text{apply}(\text{act}, db) = \begin{cases} 
\text{db}[R \oplus \mathcal{T}] & \text{if } \text{act} = \langle u, \text{INSERT}, R, \mathcal{T} \rangle \\
\text{db}[R \ominus \mathcal{T}] & \text{if } \text{act} = \langle u, \text{DELETE}, R, \mathcal{T} \rangle
\end{cases}
\]

We are now ready to define the function action, which takes as input a trigger \( \langle id, ow, ev, R', \phi, \text{stmt}, m \rangle \), a user \( u \), and a tuple \( \mathcal{T} \in \text{dom}(R') \), and returns the concrete action executed by the system. Formally, action is defined as follows:

\[
\text{action}(\langle id, ow, ev, R', \phi, \text{stmt}, m \rangle, u, \mathcal{T}) = \begin{cases} 
\langle u, \text{INSERT}, R, \mathcal{T}[\mathcal{T}' \mapsto \mathcal{T}] \rangle & \text{if } \text{stmt} = \langle \text{INSERT}, R, \mathcal{T} \rangle \\
\langle u, \text{DELETE}, R, \mathcal{T}[\mathcal{T}' \mapsto \mathcal{T}] \rangle & \text{if } \text{stmt} = \langle \text{DELETE}, R, \mathcal{T} \rangle \\
\langle \text{op}, u', p, u \rangle & \text{if } \text{stmt} = \langle \text{op}, u', p \rangle \land \text{op} \in \{\ominus, \oplus, \ominus^*\}
\end{cases}
\]

where \( \mathcal{T}' \) denotes the tuple of variables \( \langle x_1, \ldots, x_{|R'|} \rangle \).

We assume given a total-order relation \( \leq_T \) over \( T \). We use this ordering to determine the order in which triggers are executed. Given a set of triggers \( T \) and a database schema \( D \), we denote by \( \text{filter}(T, ev, R) \), where \( ev \in \{\text{INS, DEL}\} \) and \( R \in D \), the sequence of triggers in \( T \) (ordered according to \( \leq_T \)) whose event is \( ev \) and whose relation schema is \( R \).

Auxiliary Functions over contexts

Given an \( M \)-context \( c = \langle h, aE, tr \rangle \), we denote by secEx the following function, which returns \( \top \) if the effect \( aE \) is associated with a security exception.

\[
\text{secEx}(\langle h, aE, tr \rangle) = \begin{cases} 
\top & \text{if } aE = \langle \text{act}, \bot, \text{res}, E \rangle \\
\top & \text{if } aE = \langle t, \langle \text{act}, \bot, \text{res}, E \rangle, e \rangle \\
\top & \text{if } aE = \langle t, \langle \text{when}, \langle \text{act}, \bot, \text{res}, E \rangle \rangle \rangle
\end{cases}
\]

\( \bot \) otherwise

Similarly, we denote by Ex(c) the function extracting the (violated) integrity constraints stored in the effect \( aE \).

\[
\text{Ex}(\langle h, aE, tr \rangle) = \begin{cases} 
E & \text{if } aE = \langle \text{act}, aC, \text{res}, E \rangle \\
E & \text{if } aE = \langle t, \langle \text{when}, \langle \text{act}, aC, \text{res}, E \rangle \rangle \rangle
\end{cases}
\]

\emptyset \) otherwise
Additionally, the functions $acA(c)$ and $acC(c)$ extract the access control decision associated with the trigger’s action and condition respectively:

\[
acA(\langle h, aE, tr \rangle) = \begin{cases} 
 aC' & \text{if } aE = \langle t, \langle act, aC, res, E \rangle, \langle act', aC', res', E' \rangle \rangle \wedge \langle act', aC', res', E' \rangle \neq \epsilon \\
 \bot & \text{otherwise}
\end{cases}
\]

\[
acC(\langle h, aE, tr \rangle) = \begin{cases} 
 aC & \text{if } aE = \langle t, \langle act, aC, res, E \rangle, \epsilon \rangle \\
 \bot & \text{otherwise}
\end{cases}
\]

The function $res(c)$, instead, extracts the action’s result from the effect $aE$:

\[
res(\langle h, aE, tr \rangle) = \begin{cases} 
 res & \text{if } aE = \langle act, aC, res, E \rangle \\
 aC & \text{if } aE = \langle t, \langle act, aC, res, E \rangle, \epsilon \rangle \\
 aC \wedge aC' \wedge res' & \text{if } aE = \langle t, \langle act, aC, res, E \rangle, \langle act', aC', res', E' \rangle \rangle \wedge \langle act', aC', res', E' \rangle \neq \epsilon 
\end{cases}
\]

We denote by $invoker(c)$ the function extracting the user in the transaction, i.e., $invoker(\langle h, aE, (s, \bar{I}, u, trL) \rangle) = u$. Similarly, we denote by $tpl(c)$ the function extracting the tuple that has fired the transaction, namely $tpl(\langle h, aE, (s, \bar{I}, u, trL) \rangle) = \bar{I}$, by $triggers(c)$ the function extracting the list of triggers, i.e., $triggers(\langle h, aE, (s, \bar{I}, u, trL) \rangle) = trL$, and by $trigger(c)$, or $tr(c)$ for short, the first trigger in the sequence $triggers(c)$. With a slight abuse of notation, we lift all the aforementioned functions from contexts to runtime states. For instance, $Ex(s)$, where $s$ is a runtime state, is simply $Ex(ctx(s))$. 
Chapter 6

Database Access Control in Modern Databases

If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.

Sun Tzu – The art of war

The SQL standard supports database access control, and almost all Database Management Systems (DBMSs) have accordingly developed access control mechanisms. The standard however fails to define a precise access control semantics, the attacker model, and the security properties that the mechanisms ought to satisfy. As a consequence, existing access control mechanisms are implemented in an ad hoc fashion, with neither precise security guarantees nor the means to verify them.

This deficit has dire and immediate consequences. In this chapter, we show that popular database systems are susceptible to two types of attacks. Integrity attacks allow an attacker to perform non-authorized changes to the database. Confidentiality attacks allow an attacker to learn sensitive data. These attacks exploit advanced SQL features, such as triggers, views, and integrity constraints, and they are easy to carry out.

Current research efforts in database security are neither adequate for evaluating the security of modern databases, nor do they account for their advanced features. In more detail, existing research [13, 40, 131, 165] implicitly considers attackers who use SELECT commands. But the capabilities offered by databases go far beyond SELECT. Users, in general, can modify the database’s state and security policy, as well as use features such as triggers, views, and integrity constraints. Consequently, all proposed research solutions fail to prevent attacks such as those we present in Section 6.1.

In summary, the database vendors have been left to develop access control mechanisms without guidance from either the SQL standard or existing research in database security. It is therefore not surprising that modern databases are open to abuse.

In this chapter, we develop a comprehensive formal framework for the design and analysis of database access control. Our framework, which relies on the operational semantics from Chapter 5, consists of a precise attacker model complemented with adequate security properties. We use our framework to design and verify an access control mechanism that prevents confidentiality and integrity attacks that defeat existing mechanisms. Note that this chapter is largely based on [87].

Organization. In Section 6.1, we present attacks that illustrate serious weaknesses in existing DBMSs. We introduce and formalize our attacker model in Section 6.2. We define our security properties in Sections 6.3 and 6.4. In Section 6.5, we present our provably secure DBAC mechanism, and in Section 6.6 we discuss related work. Finally, we draw conclusions in Section 6.7. For the sake of readability, some technical details are given in Section 6.8. Furthermore, the proof of all our results are given in Appendix C. A prototype of our enforcement mechanism is available at [86].

6.1 Illustrative Attacks

We demonstrate here how attackers can exploit existing DBMSs using standard SQL features. We classify these attacks as either Integrity Attacks or Confidentiality Attacks. In the former, an attacker makes unauthorized changes to the database, which stores the data, the policy, the triggers, and the views. In the latter, an attacker learns sensitive data by interacting with the system and observing the outcome. No existing access control mechanism prevents all the attacks we present. Moreover, many related attacks can be constructed using variants of the ideas presented here. We manually carried out the attacks against IBM DB2, Oracle Database, PostgreSQL, MySQL, SQL Server, and Firebird. We summarize our findings in Figures 6.1 and 6.2, and we discuss them at the end of this section. In the figures, ✓ indicates a successful attack, ◐ indicates a failed attack, and * indicates that the DBMS does not support the features necessary to launch the attack.
### 6.1 Integrity Attacks

Our three integrity attacks combine different database features: **INSERT**, **DELETE**, **GRANT**, and **REVOKE** commands together with views and triggers. In the first attack, an attacker creates a trigger, i.e., a procedure automatically executed by the DBMS in response to user commands, that will be activated by an unaware user with a higher security clearance and will perform unauthorized changes to the database. The attack requires triggers to be executed under the privileges of the users activating them. Such triggers are supported by PostgreSQL, SQL Server, and Firebird.

**Attack 6.1. Triggers with activator’s privileges.** Consider a database with two tables $P$ and $S$ and two users $u_1$ and $u_2$. The attacker is the user $u_1$, whose goal is to delete the content of $S$. The policy is that $u_1$ is not authorized to alter $S$, $u_1$ can create triggers on $P$, and $u_2$ can read and modify $S$ and $P$. The attack is as follows:

1. $u_1$ creates the trigger:

   ```sql
   CREATE TRIGGER t ON P AFTER INSERT
   DELETE FROM S;
   ```

2. $u_1$ waits until $u_2$ inserts a tuple into the table $P$. The trigger will then be invoked using $u_2$’s privileges and $S$’s content will be deleted.

   An attacker can use similar attacks to execute arbitrary commands with administrative privileges. Despite the threat posed by such simple attacks, the existing countermeasures [4] are unsatisfactory; they are either too restrictive, for instance completely disabling triggers in the database, or too time consuming and error prone, namely manually checking if “dangerous” triggers have been created.

   In our second attack, an attacker escalates his privileges by delegating the read permission for a table without being authorized to delegate this permission. The attacker first creates a view over the table and, afterwards, delegates the access to the view to another user. This attack exploits DBMSs, such as PostgreSQL, where a user can grant any read permission over his own views. Note that **GRANT** and **REVOKE** commands are **write operations**, which target the database’s internal configuration instead of the tables.

**Attack 6.2. Granting views.** Consider a database with a table $S$, two users $u_1$ and $u_2$, and the following policy: $u_1$ can create views and read $S$ (without being able to delegate this permissions), and $u_2$ cannot read $S$. The attack is as follows:

1. $u_1$ creates the view:

   ```sql
   CREATE VIEW v AS SELECT * FROM S.
   ```

2. $u_1$ issues the command **GRANT SELECT ON v TO u_2**. Now, $u_2$ can read $S$ through $v$. However, $u_1$ is not authorized to delegate the read permission on $S$.

   This attack exploits several subtleties in the commands’ semantics: (a) users can create views over all tables they can read, (b) the views are executed under the owner’s privileges, and (c) view’s owners...
can grant arbitrary permissions over their own views. These features give \( u_1 \) the implicit ability to delegate the read access over \( S \). As a result, the overall system’s behavior does not conform with the given policy. That is, \( u_1 \) should not be permitted to delegate the read access to \( S \) or to any view that depends on it. Note that the commands’ semantics may vary between different DBMSs.

In our third attack, an attacker exploits the failure of access control mechanisms to propagate REVOKE commands.

**Attack 6.3. Revoking views.** Consider a database with a table \( S \), three users \( u_1, u_2, \) and \( u_3 \), and the following policy: \( u_1 \) can read \( S \) and delegate this permission, \( u_2 \) can create views, and \( u_3 \) cannot read \( S \). The attack proceeds as follows:

1. \( u_1 \) issues the command `GRANT SELECT ON \( S \) TO \( u_2 \)` with GRANT OPTION.
2. \( u_2 \) creates the view: `CREATE VIEW \( v \) AS SELECT * FROM \( S \)`.
3. \( u_2 \) issues the command `GRANT SELECT ON \( v \)` TO \( u_3 \).
4. \( u_1 \) revokes the permission to read \( S \) (and to delegate the permission) from \( u_2 \): `REVOKE SELECT ON \( S \)` FROM \( u_2 \). Now, \( u_3 \) cannot read \( v \) because \( u_2 \), which is \( v \)'s owner, cannot read \( S \).
5. \( u_1 \) grants again the permission to read \( S \) to \( u_2 \): `GRANT SELECT ON \( S \)` TO \( u_2 \). Now, \( u_3 \) can again read \( v \) but \( u_2 \) can no longer delegate the read permission on \( v \). ■

This attack succeeds because, in the fourth step, the REVOKE statement does not remove the GRANT granted by \( u_2 \) to \( u_3 \) to read \( v \). This GRANT only becomes ineffective because \( u_2 \) is no longer authorized to read \( S \). However, after the fifth step, this GRANT becomes effective again, even though \( u_2 \) can no longer delegate the read permission on \( v \). Thus, the policy is left in an inconsistent state, i.e., it contains a privilege that should have been revoked according to the SQL access control model [137,162].

### 6.1.2 Confidentiality Attacks

We now present two attacks that use INSERT and SELECT commands together with triggers and integrity constraints. In our fourth attack, an attacker exploits integrity constraint violations to learn sensitive information. An integrity constraint is an invariant that must be satisfied for a database state to be considered valid. *Integrity constraint violations* arise when the execution of an SQL command leads the database from a valid state into an invalid one.

**Attack 6.4. Table updates and integrity constraints.** Consider a database with two tables \( P \) and \( S \). Suppose the primary key of both tables is the user’s identifier. Furthermore, the set of user identifiers in \( S \) is contained in the set of user identifiers in \( P \), i.e., there is a foreign key from \( S \) to \( P \). The attacker is the user \( u \) whose goal is to learn whether \( Bob \) is in \( S \). The security policy is that \( u \) can read \( P \) and insert tuples in \( S \). The attacker \( u \) can learn whether \( Bob \) is in \( S \) as follows:

1. \( u \) reads \( P \) and learns \( Bob \)'s identifier.
2. \( u \) issues an INSERT statement in \( S \) using \( Bob \)'s id.
3. If \( Bob \) is already in \( S \), then \( u \) gets an error message about the primary key’s violation. Alternatively, there is no violation and \( u \) learns that \( Bob \) is not in \( S \). ■

Although similar attacks have been identified before [99,143], existing DBMSs are still vulnerable. In our fifth attack, an attacker learns sensitive information by exploiting the system’s triggers. The trigger in this attack is executed under the privileges of the trigger’s owner. Such triggers are supported by IBM DB2, Oracle Database, PostgreSQL, MySQL, SQL Server, and Firebird.

**Attack 6.5. Triggers with owner’s privileges.** Consider a database with three tables \( N \), \( P \), and \( T \). The attacker is the user \( u \), who wishes to learn whether \( v \) is in \( T \). The policy is that \( u \) is not authorized to read the table \( T \), and he can read and modify the tables \( N \) and \( P \). Moreover, the following trigger has been defined by the administrator:

```
CREATE TRIGGER t ON P AFTER INSERT FOR EACH ROW
    IF exists(SELECT * FROM T WHERE id = NEW.id)
    INSERT INTO N VALUES (NEW.id);
```

The attack is as follows:

1. \( u \) deletes \( v \) from \( N \).
2. \( u \) issues the command `INSERT INTO P VALUES (v)`.
3. \( u \) checks the table \( N \). If it contains \( v \)'s id, then \( v \) is in \( T \). Otherwise, \( v \) is not in \( T \). ■

This attack exploits that the trigger \( t \) conditionally modifies the database. Furthermore, the attacker can activate \( t \), by inserting tuples in \( P \), and then observe \( t \)'s effects, by reading the table \( N \). He therefore can exploit \( t \)'s execution to learn whether \( t \)'s condition holds. We assume here that the attacker knows the triggers in the system. This is, in general, a weak assumption as triggers usually describe the domain-specific rules regulating a system’s behavior and users are usually aware of them.
6.1.3 Discussion

We manually carried out all five attacks against IBM DB2, Oracle Database, PostgreSQL, MySQL, SQL Server, and Firebird. Figures 6.1 and 6.2 summarize our findings. None of these systems prevent the confidentiality attacks. They are however more successful in preventing the integrity attacks. The most successful is Oracle Database, which prevents two of the three attacks, while Attack 6.1 cannot be carried out due to missing features. IBM DB2, MySQL, and Firebird prevent just one of the three attacks, namely Attack 6.2. However, they all fail to prevent Attack 6.3. Note that Firebird also fails to prevent Attack 6.1. In contrast, Attack 6.1 cannot be carried out against MySQL and IBM DB2 due to missing features. SQL Server also fails to prevent Attack 6.1; however the remaining two attacks cannot be carried out due to missing features. PostgreSQL fails to prevent all three attacks.

We argue that the direct state of database access control mechanisms, as illustrated by these attacks, comes from the lack of clearly defined security properties that such mechanisms ought to satisfy and the lack of a well-defined attacker model. We therefore develop a formal attacker model and precise security properties and we use them to design a provably secure access control mechanism that prevents all the above attacks.

6.2 Attacker Model

We next present our attacker model. In our setting, an attacker is a user, other than the administrator, with an assigned user account who attempts to violate the security policy. Namely, his goals are: (1) to read or infer data from the database for which he lacks the necessary SELECT privileges, and (2) to alter the state in unauthorized ways, e.g., changing data in relations for which he lacks the necessary INSERT and DELETE privileges. The attacker can issue any command available to users and he sees the results of his commands (see Chapter 5 for all supported commands). The attacker’s inference capabilities are specified using deduction rules.

6.2.1 Formal Attacker Model

We model attackers that interact with the system through SQL commands and infer information from the system’s behavior by exploiting triggers, views, and integrity constraints. We argue that database access control mechanisms should be secure with respect to such strong attackers, as this reflects how (malicious) users may interact with modern databases. Furthermore, any mechanism secure against such strong attackers is also secure against weaker attackers.

Any user other than the administrator can be an attacker, and we assume that users do not collude to subvert the system. Note that our attacker model, the security properties in Sections 6.3 and 6.4, and the mechanism we develop in Section 6.5 can all easily be extended to support colluding users. We also assume that an attacker can issue any command available to the system’s users, and he knows the system’s operational semantics, the database schema, and the integrity constraints.

We assume that an attacker has access to the system’s security policy, the set of users, and the definitions of the triggers and views in the system’s state. In more detail, given an M-runtime state (db, U, sec, T, V, ctx), an attacker can access U, sec, T, and V. Users interacting with existing DBMSs typically have access to some, although not all, of this information. For instance, in PostgreSQL a user can read all the information about the triggers defined on the tables for which he has some non-SELECT privileges. Note that the more information an attacker has, the more attacks he can launch. Finally, we assume that an attacker knows whether any two of his commands c and c’ have been executed consecutively by the system, i.e., if there are commands executed by other users occurring between c and c’. The attacker’s knowledge about the sequential execution of his commands is needed to soundly propagate his knowledge about the system’s state between his commands. Since the mechanism we develop in Section 6.5 is secure with respect to this attacker, it is also secure with respect to weaker attackers who have less information or cannot detect whether their commands have been executed consecutively.

An attacker model describes what information an attacker knows, how he interacts with the system, and what he learns about the system’s data by observing the system’s behavior. Since every user is a potential attacker, for each user u ∈ U we define an attacker model specifying u’s inference capabilities. To represent u’s knowledge, we introduce judgments. A judgment is a four-tuple (r, i, u, φ), written r, i ⊬ u φ, denoting that from the run r, which represents the system’s behavior, the user u can infer that φ holds in the i-th state of r. An attacker model for u is thus a set of judgments associating to each position of each run, the sentences that u can infer from the system’s behavior. The idea of representing the attacker’s knowledge using sentences φ is inspired by existing formalisms for Database Inference Control [40, 72] and Controlled Query Evaluation [36].

**Definition 6.1.** Let P be an extended configuration, L be the P-LTS, and u ∈ U be a user. A (P, u)-judgment is a tuple (r, i, u, φ), written r, i ⊬ u φ, where r ∈ traces(L), 1 ≤ i ≤ |r|, and...
The attacker is able to propagate the knowledge of a sentence \( \phi \) after a successful action leading to \( \text{SELECT} \), and after a successful \( \text{DELETE} \) command, then he knows that \( \phi \) also holds just before the \( \text{SELECT} \) command because \( \text{SELECT} \) commands do not modify the database state. The complete formalization of all rules is given in Section 6.8.1. In the following, when we say that a judgment \( r, i \vdash u \phi \) holds, we always mean with respect to the attacker model \( \text{ATK}_u \).

As we prove in Appendix C, \( \text{ATK}_u \) is sound with respect to the \( RC \) semantics, i.e., if \( r, i \vdash u \phi \) holds, then the formula \( \phi \) holds in the \( i \)-th state of \( r \). Intuitively, \( \text{ATK}_u \) models how \( u \) infers information from the system's behavior, namely (a) how \( u \) learns information from its commands and their results, (b) how \( u \) learns information from triggers, their execution, their interleavings, and their side effects, (c) how \( u \) propagates his knowledge along a run, and (d) how \( u \) learns information from exceptions caused by either integrity constraint violations or security violations. This model is substantially more powerful than the \( \text{SELECT} \)-only attacker studied in previous works [131, 165, 170].

The rules \( \text{DELETE SUCCESS} \) and \( \text{SELECT SUCCESS} \) describe how the user \( u \) infers information from his successful actions, i.e., those actions that generate neither security exceptions nor integrity violations. In the rules, \( \text{secEx} (s) \) denotes that there were no security exceptions caused by the action leading to \( s \), and \( \text{Ex} (s) = \emptyset \) denotes that the action leading to \( s \) has not violated the integrity constraints. After a successful \( \text{DELETE} \), \( u \) knows that the deleted tuple is no longer in the database, and after a successful \( \text{SELECT} \) he learns the query's result, denoted by \( \text{res} (s) \).

The rules \( \text{PROPAGATE BACKWARD SELECT} \) and \( \text{PROPAGATE FORWARD UPDATE SUCCESS} \) describe how the user \( u \) propagates information along the run. \( \text{PROPAGATE BACKWARD SELECT} \) states that if the user \( u \) knows that \( \phi \) holds after a \( \text{SELECT} \) command, then he knows that \( \phi \) also holds just before the \( \text{SELECT} \) command because \( \text{SELECT} \) commands do not modify the database state. \( \text{PROPAGATE FORWARD UPDATE SUCCESS} \) states that if \( u \) knows that \( \phi \) holds before a successful \( \text{INSERT} \) or \( \text{DELETE} \) command and he can determine that the command's execution does not influence \( \phi \)'s truth value, denoted by \( \text{revise} (r, \phi, \phi (r')) = T \), then he also knows that \( \phi \) holds after the command.

In our attacker model, we consider a very simple syntactic criterion for revising beliefs. Informally, the attacker is able to propagate the knowledge of a sentence \( \phi \) after (or before) an \( \text{INSERT} \) or a \( \text{DELETE} \) action on a table \( R \) iff the predicate \( R \) does not occur in \( \phi \). Formally, the function \( \text{revise} : \text{traces} (L) \times \text{RC}_{\text{bool}} \times \text{traces} (L) \rightarrow \{ T, \bot \} \) captures this belief revision approach and it is defined as follows:

\[
\text{revise} (r, \phi, \phi \text{-act-s}) = \begin{cases} 
T & \text{if } \text{act} = \{ \text{op}, \text{op}, \text{R}, \text{T} \} \land \text{R} \notin \text{tables} (\phi) \land \text{op} \in \{ \text{INSERT}, \text{DELETE} \} \\
T & \text{if } \text{act} = \{ \text{id}, \text{op}, \text{ev}, \text{R}', \phi, (\text{op}, \text{op}, \text{T}, \text{m}) \} \land \text{R} \notin \text{tables} (\phi) \land \\
& \text{op} \in \{ \text{INSERT}, \text{DELETE} \} \\
T & \text{if } \text{act} = \{ \text{id}, \text{op}, \text{ev}, \text{R}, \phi, (\text{op}, \text{u}, \text{p}) \} \land \text{op} \in \{ \oplus, \ominus , \ominus \} \\
\bot & \text{otherwise}
\end{cases}
\]
thereby, prevents integrity attacks, such as Attacks 6.1–6.3. mechanism providing database integrity prevents non-authorized changes to the system’s state and, that all actions modifying the system’s state are authorized by the system’s policy. An access control

\[ r, 2 \vdash u \neg N(v) \]

\[ r, 3 \vdash u \neg N(v) \]

\[ r, 3 \vdash u \top(v) \]

**DELETESUCCESS**

**UPDATESUCCESS**

**SELECTSUCCESS**

**PROPAGATEFORWARD**

**PROPAGATEBACKWARD**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ r, 5 \vdash u N(v) ]</td>
<td>SELECT SUCCESS</td>
</tr>
<tr>
<td>[ r, 4 \vdash u N(v) ]</td>
<td>PROPAGATE BACKWARD</td>
</tr>
<tr>
<td>[ r, 4 \vdash u N(v) ]</td>
<td>SELECT SUCCESS</td>
</tr>
<tr>
<td>[ r, 5 \vdash u N(v) ]</td>
<td>PROPAGATE FORWARD</td>
</tr>
<tr>
<td>[ r, 2 \vdash u \neg N(v) ]</td>
<td>DELETE SUCCESS</td>
</tr>
<tr>
<td>[ r, 3 \vdash u \neg N(v) ]</td>
<td>UPDATE SUCCESS</td>
</tr>
</tbody>
</table>

**Figure 6.4:** Template Derivation of Attack 6.5 (contains just selected subgoals)

Observe that above we define revise only for the inputs \( r, \phi, r' \) such that \( \phi \in RC_{\text{auth}} \) is a sentence and \( r' = r \cdot \text{act-s} \), where \( \text{act} \in A_{D, U} \cup \text{TRIGGER}_{D} \) and \( s \in \Omega_{M} \). If this is not the case, then revise\((r, \phi, r')\) = \( \bot \). Our attacker model, enforcement mechanism, and proofs can be easily extended to support other criteria for revising beliefs.

Finally, the rule LEARN INSERT BACKWARD models \( u \)’s reasoning when he activates a trigger that successfully inserts a tuple in the database. If \( u \) knows that immediately before the trigger the formula \( \psi \) does not hold and immediately after the trigger the formula \( \psi \) holds, then the trigger’s execution is the cause of the database state’s change. Therefore, \( u \) can infer that the trigger’s condition \( \phi \) holds just before the trigger’s execution. Note that \textit{invoker}(s) denotes the user who fired the trigger that is executed in the state \( s \), whereas \textit{tpl}(s) denotes the tuple associated with the action that fired the trigger that is executed in the state \( s \).

We developed an executable version of our attacker model, available at [86], using the Maude term rewriting framework [54]. Our executable models can be used for simulating the execution of our operational semantics, as well as computing the information that an attacker can infer from the system’s behavior. We have executed and validated all of the examples in this chapter using these models.

**Example 6.1.** Consider the policy described in Attack 6.5. The database \( D \) has three tables: \( N \), \( P \), and \( T \). The set \( U \) is \{\( u, \text{admin} \}\) and the policy \( S \) contains the following grants: \( \langle \oplus, u, \langle \text{SELECT}, P \rangle, \text{admin} \rangle \), \( \langle \oplus, u, \langle \text{INSERT}, P \rangle, \text{admin} \rangle \), \( \langle \oplus, u, \langle \text{DELETE}, P \rangle, \text{admin} \rangle \), \( \langle \oplus, u, \langle \text{INSERT}, N \rangle, \text{admin} \rangle \), and \( \langle \oplus, u, \langle \text{DELETE}, N \rangle, \text{admin} \rangle \). Furthermore, the database state \( db \) is \( db(N) = \{ v \} \), \( db(P) = \emptyset \), and \( db(T) = \{ v \} \), while the only trigger in the system is \( t = \langle \text{id, admin, INS}, P, T(x), \langle \text{INSERT}, N, x, \rangle, O \rangle \). The run \( r \) is as follows:

1. \( u \) deletes \( v \) from \( N \).
2. \( u \) inserts \( v \) in \( P \). This activates the trigger \( t \), which inserts \( v \) in \( N \).
3. \( u \) issues the \textit{SELECT} query \( N(v) \).

We used Maude to generate the following run, which illustrates how the system’s state changes. Note that there are no exceptions during the run.

\[ \langle db, U, S, \{ t \}, \emptyset, c_{1} \rangle \xrightarrow{ \langle u, \text{DELETE}, N, v \rangle } \langle db \oplus v, U, S, \{ t \}, \emptyset, c_{2} \rangle \]

\[ \langle db \oplus v, U, S, \{ t \}, \emptyset, c_{3} \rangle \]

\[ \langle u, \text{INSERT}, P, v \rangle \]

\[ \langle u, \text{SELECT}, N(v) \rangle \]

Figure 6.4 models \( u \)’s reasoning in Attack 6.5. The user \( u \) first applies the \textit{SELECT} SUCCESS rule to derive \( r, 5 \vdash u N(v) \), i.e., he learns the query’s result. By applying the rule PROPAGATE BACKWARD to \( r, 5 \vdash u N(v) \), he obtains \( r, 4 \vdash u N(v) \), i.e., he learns that \( N(v) \) holds before the \textit{SELECT} query. Similarly, he applies the rule DELETE SUCCESS to derive \( r, 2 \vdash u \neg N(v) \), and he obtains \( r, 3 \vdash u \neg N(v) \) by applying the PROPAGATE FORWARD UPDATE SUCCESS rule. Finally, by applying the rule LEARN INSERT BACKWARD to \( r, 3 \vdash u \neg N(v) \) and \( r, 4 \vdash u N(v) \), he learns the value of the trigger’s \textit{WHEN} condition \( r, 3 \vdash u T(v) \). Since the user \( u \) should not be able to learn information about \( T \), the attack violates the intended confidentiality guarantees. We used our executable attacker model [86] to derive the judgments.

### 6.3 Database Integrity

Here we define the first of our security properties: database integrity. Database integrity states that all actions modifying the system’s state are authorized by the system’s policy. An access control mechanism providing database integrity prevents non-authorized changes to the system’s state and, thereby, prevents integrity attacks, such as Attacks 6.1–6.3.

Database integrity requires a formalization of authorized actions. We therefore define the relation \( \sim_{\text{auth}} \) between runtime states and actions, modeling which actions are authorized in a given state.

We first introduce some notation. Let \( P = (M, f) \) be an extended configuration, where \( M = (D, \Gamma) \) and \( f \) is an \( M\text{-PDP} \). We denote by \( VIEW_{\text{owner}}^{D} \) the set of all \( D \)-views with owner’s privileges.
i.e., \( \mathcal{VIE}_D \) = \{ (V, q) \mid V \in \mathcal{VIE}_D \} and by \( \mathcal{PRIV}_D \) = \{ (SELECT, V) \mid V \in \mathcal{VIE}_D \} the set of all SELECT privileges associated with views in \( \mathcal{VIE}_D \), and by \( \mathcal{PRIV}^{\text{SELECT, VIE}_D}_D \) the set of all SELECT privileges associated with views in \( \mathcal{VIE}_D \). Furthermore, given an \( M \)-routine runtime state \( s = \langle db, U, sec, T, V, c \rangle \) and a \textit{REVOKE} command \( r = (u, p, u') \), we denote by \( \text{applyRev}(s, r) \) the state \( \langle db, U, \text{revok}(sec, u, p, u'), T, V, c \rangle \) obtained by executing the \textit{REVOKE} command. Finally, given a query \( q \), a set of views \( V \) with owner's privileges, and a set of tables \( T \), we write \( \text{determine}_{\text{sec}}(T, V, q) \) to denote that \( D, \Gamma \vdash Q \rightarrow q \), where \( Q \) is the set of queries containing the tables in \( T \) and the views in \( V \) (see Section 2.2.3 for a formalization of query determinacy).

We are now ready to formalize the notion of authorized actions. The relation \( \sim_{\text{auth}} \subseteq \Omega_M \times (\mathcal{A}_M \cup \mathcal{TRIGGER}_D) \), specifying which actions are authorized in a given state, is the smallest relation satisfying the inference rules given in Figure 6.5. In the following, let \( s = \langle db, T, sec, T, V, \text{ctx} \rangle \) be a runtime state. According to the \textit{ADD User} rule, the only authorized \textit{ADD USER} commands are those issued by the administrator. In contrast, \textit{SELECT} commands are always authorized, as they do not modify the system state. Furthermore, an \textit{INSERT}, \textit{DELETE}, \textit{CREATE VIEW}, or \textit{CREATE TRIGGER} command is authorized in two cases: (1) the user executing the command is the administrator (see the rules \textit{INSERT-DELETE ADMIN}, \textit{CREATE VIEW ADMIN}, and \textit{CREATE TRIGGER ADMIN}), or (2) the current policy \( sec \) contains a grant \( g \) that grants the corresponding privilege to the user executing the command and such that \( s \sim_{\text{auth}} g \) holds (see the rules \textit{INSERT-DELETE}, \textit{CREATE VIEW}, and \textit{CREATE TRIGGER}).

The \textit{REVOKE} rule says that a \textit{REVOKE} statement is authorized if the resulting state, obtained using the function \( \text{applyRev} \), has a consistent policy, namely one in which all the grants in the policy are authorized by \( \sim_{\text{auth}} \). The rules \textit{GRANT-1}, \textit{GRANT-2}, \textit{GRANT-3}, \textit{GRANT-4}, and \textit{GRANT-5}, instead, formalize in which cases a \textit{GRANT} command is authorized. Specifically, a \textit{GRANT} command is authorized if one of the following conditions hold:

- **Rule Grant-1**: the user executing the command is not the administrator and there is a grant \( g \) (with grant option) that authorizes the command and \( s \sim_{\text{auth}} g \).
- **Rule Grant-2**: the user executing the command is the administrator and the privilege is not associated with a view with the owner’s privileges.
- **Rule Grant-3**: the privilege used in the command is a \textit{SELECT} privilege over a view \( V \) in \( \mathcal{VIE}_D \), the user \( u \) executing the command is \( V \)'s owner, and \( u \) has the \textit{SELECT} privilege with grant option over a set of tables and views that determine \( V \).
- **Rule Grant-4**: the privilege used in the command is a \textit{SELECT} privilege over a view \( V \) in \( \mathcal{VIE}_D \), the user executing the command is the administrator, and \( V \)'s owner has the \textit{SELECT} privilege over a set of tables and views that determine \( V \).
- **Rule Grant-5**: the privilege used in the command is a \textit{SELECT} privilege over a view with the activator’s privileges and the user executing the command is the view’s owner.

The rules \textit{GRANT-3} and \textit{GRANT-4} use the function \textit{hasAccess} to decide whether a user has read access (with or without delegation) to a set of views and tables. Formally, the \textit{hasAccess} function is defined as follows:

\[
\text{hasAccess}(\langle db, U, sec, T, V, c \rangle, S, u, op) = \begin{cases} \top & \text{if } u \neq \text{admin} \land \forall v \in S, \exists u' \in U, g \in sec, op' \in \{ \text{op}, \oplus^* \}, \\
& \langle db, U, sec, T, V, c \rangle \sim_{\text{auth}} g \\
& \langle db, U, sec, T, V, c \rangle \sim_{\text{auth}} \langle \text{op'}, u \rangle \\
& \langle \text{op'}, u \rangle \in (\text{SELECT}, v) \\
\top & \text{otherwise} \\
\end{cases}
\]

The above function performs the following check. If the user \( u \) is not the administrator, \textit{hasAccess} checks that for all views and tables \( v \in S \), there is a \textit{GRANT} \( g \) in the current policy that is valid according to \( \sim_{\text{auth}} \), and that grants the privilege \( \langle \text{SELECT}, v \rangle \) (with or without \textit{GRANT OPTION} depending on \( op \)) to the user \( u \). In case the user \( u \) is the administrator, \textit{hasAccess} checks that for all views and tables in \( S \), the \textit{GRANT} \( \langle \text{op'}, \text{admin}, \langle \text{SELECT}, v \rangle, u' \rangle \) is valid according to \( \sim_{\text{auth}} \), where \( \text{op'} \) is \( \oplus \) or \( \oplus^* \) depending on \( op \) and \( u' \) is a user in \( U \).

Finally, the rules \textit{EXECUTE TRIGGER-1}, \textit{EXECUTE TRIGGER-2}, and \textit{EXECUTE TRIGGER-3} regulate the execution of triggers. According to the rule \textit{EXECUTE TRIGGER-1}, a trigger with the owner’s privileges whose \textit{WHEN} condition is satisfied is authorized if the trigger’s owner is authorized to execute the trigger’s action. The \textit{EXECUTE TRIGGER-2} rule says that the execution of a trigger (whose \textit{WHEN} condition is satisfied) with the activator’s privileges is authorized if both the invoker and the trigger’s owner are authorized to execute the trigger’s action according to \( \sim_{\text{auth}} \). Finally, the \textit{EXECUTE TRIGGER-3} rule says that the execution of triggers whose \textit{WHEN} condition is not satisfied is always authorized, as these triggers do not modify the system state. Observe that the \textit{action} function instantiates the action given in the trigger’s definition to a concrete action by identifying the user performing the action and replacing the free variables with values from \textit{dom}.

We now define database integrity. Intuitively, a PDP provides database integrity iff all the actions it authorizes are explicitly authorized by the policy, i.e., they are authorized by \( \sim_{\text{auth}} \). This notion comes directly from the SQL standard, and it is reflected in existing enforcement mechanisms. Recall
ADD User
\[ u \in U \quad u' = \text{admin} \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} (u', \text{ADD USER}, u) \]

SELECT
\[ u \in U \quad q \in RC \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} (u, SELECT, q) \]

Insert–Delete
\[ g = (op, u, (op', R), u') \]
\[ u, u' \in U \quad R \in D \]
\[ t \in \text{dom}^{R} \]
\[ g \in \text{sec} \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} g \]
\[ op' \in \{\text{INSERT, DELETE}\} \]

Create View
\[ u, u' \in U \quad v \in \text{VIEW} \]
\[ g = (op, u, (\text{CREATE VIEW}, u')) \]
\[ g \in \text{sec} \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} g \]

Create Trigger
\[ t \in \text{TRIGGER} \]
\[ g = (op, u, (\text{CREATE TRIGGER}, R), u') \]
\[ g \in \text{sec} \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} g \]

Insert–Delete admin
\[ R \in D \quad t \in \text{dom}^{R} \]
\[ op' \in \{\text{INSERT, DELETE}\} \]

Create View admin
\[ v \in \text{VIEW} \]
\[ v = (id, admin, q, m) \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} (admin, CREATE, v) \]

Revoke
\[ u, u' \in U \quad priv \in \text{PRIV} \]
\[ s = (db, U, sec, T, V, c) \]
\[ s' = (db, U, sec', T, V, c) \]
\[ s' = \text{applyRev}(s, (\text{id, admin, q, m})) \]
\[ \forall g \in \text{sec}. s' \rightarrow_{\text{auth}} g \]

Grant-1
\[ op \in \{\oplus, \ominus\} \]
\[ priv \in \text{PRIV} \]
\[ u, u', u'' \in U \]
\[ g = (\oplus, u', priv, u'') \]
\[ g \in \text{sec} \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} g \]

Grant-2
\[ u \in U \quad op \in \{\oplus, \ominus\} \]
\[ priv \in \text{PRIV} \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} (op, u, priv, u') \]

Grant-3
\[ op \in \{\oplus, \ominus\} \]
\[ priv = (\text{SELECT, v}) \]
\[ u, \text{owner} \in U \]
\[ v \in \text{V} \quad V' \subseteq V \cup \text{VIEW}^\text{owner} \]
\[ T' \subseteq D \]
\[ \text{determines}_{T'}(T', V', q) \]
\[ \text{hasAccess}((db, U, sec, T, V, c), V' \cup T', \text{owner}, \ominus) \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} (op, u, priv, admin) \]

Grant-4
\[ op \in \{\oplus, \ominus\} \]
\[ priv = (\text{SELECT, v}) \]
\[ u, \text{owner} \in U \]
\[ v \in \text{V} \]
\[ \text{hasAccess}((db, U, sec, T, V, c), V' \cup T', \text{owner}, \ominus) \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} (op, u, priv, admin) \]

Execute Trigger-1
\[ t \in T \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} t \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} \text{action}(\text{stmt}, \text{owner}, \text{tpl}(c)) \]
\[ [\phi[\text{R}^d] \rightarrow \text{tpl}(c)]^d = \top \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} t \]

Execute Trigger-2
\[ t \in T \quad (db, U, sec, T, V, c) \rightarrow_{\text{auth}} \text{action}(\text{stmt}, \text{invoker}(c), \text{tpl}(c)) \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} \text{action}(\text{stmt}, \text{owner}, \text{tpl}(c)) \]
\[ [\phi[\text{R}^d] \rightarrow \text{tpl}(c)]^d = \top \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} t \]

Execute Trigger-3
\[ t \in T \quad (db, U, sec, T, V, c) \rightarrow_{\text{auth}} t \]
\[ (db, U, sec, T, V, c) \rightarrow_{\text{auth}} t \]

Figure 6.5: Definition of the \( \rightarrow_{\text{auth}} \) relation.
that, given a state $s$, $\text{secEx}(s) = \bot$ denotes that there were no security exceptions caused by the action or leading to $s$.

**Definition 6.2.** Let $P = \langle M, f \rangle$ be an extended configuration, where $M = \langle D, \Gamma \rangle$ and $f$ is an $M$-PDP, and let $L$ be the $P$-LTS. We say that $f$ provides database integrity with respect to $P$ iff for all reachable states $s, s' \in \Omega_M$, if $s'$ is reachable in one step from $s$ by an action $a \in A_{D,U} \cup TRIGGER_D$ and $\text{secEx}(s) = \bot$, then $s \not\sim_{a\text{uth}} a$. □

**Example 6.2.** We consider a run corresponding to Attack 6.1, which illustrates a violation of database integrity. The database $db$ is such that $db(P) = \emptyset$ and $db(S) = \{ z \}$, the policy $sec$ is $\{ (\oplus, u_1, (\text{CREATE TRIGGER}, P), \text{admin}), (\oplus, u_2, (\text{INSERT}, P), \text{admin}), (\oplus, u_2, (\text{DELETE}, S), \text{admin}) \}$, and the set $U$ is $\{ u_1, u_2, \text{admin} \}$. The run $r$ is as follows:

1. The user $u_1$ creates the trigger $t = \langle id, u_1, \text{INS}, P, T, (\text{DELETE}, S, z), A \rangle$.
2. The user $u_2$ inserts the value $v$ in $P$. This activates the trigger $t$ and deletes the content of $S$, i.e., the value $z$.

We used Maude to generate the following run, which illustrates how the system’s state changes. Note that there are no exceptions during the run.

\[
\begin{align*}
\langle db, U, sec, \emptyset, c_1 \rangle & \xrightarrow{u_1, \text{CREATE}, t} \langle db, U, sec, \{ t \}, \emptyset, c_2 \rangle \\
\langle db[P \oplus v, S \oplus z], U, sec, \{ t \}, \emptyset, c_4 \rangle & \xrightarrow{u_2, \text{INSERT}, P, v} \langle db[P \oplus v], U, sec, \{ t \}, \emptyset, c_3 \rangle
\end{align*}
\]

Access control mechanisms that do not restrict the execution of triggers with activator’s privileges violate database integrity because they do not throw security exceptions when $\langle db[P \oplus v], U, sec, \{ t \}, \emptyset, c_3 \rangle \not\not\sim_{a\text{uth}} t$. □

### 6.4 Data Confidentiality

We now introduce data confidentiality, our second security property. Data confidentiality states that all information that an attacker can learn by observing the system’s behavior is authorized. Hence, by preventing the leakage of sensitive data, a mechanism providing data confidentiality prevents confidentiality attacks.

We first introduce indistinguishability and secure judgments, two key notions for data confidentiality, together with our security property. Afterwards, we formalize the notion of indistinguishability that we use in the rest of the chapter. We conclude the section with some examples.

#### 6.4.1 Definition

To model data confidentiality, we first introduce the concept of indistinguishability of runs, which formalizes the desired confidentiality guarantees by specifying whether users can distinguish between different runs based on their observations. Formally, a $P$-indistinguishability relation is an equivalence relation over traces($L$), where $P$ is an extended configuration and $L$ is the $P$-LTS. Indistinguishable runs, intuitively, should disclose the same information.

We now define the concept of a secure judgment, which is a judgment that does not leak sensitive information or, equivalently, one that cannot be used to differentiate between indistinguishable runs.

**Definition 6.3.** Let $P$ be an extended configuration, $L$ be the $P$-LTS, and $\equiv$ be a $P$-indistinguishability relation. A judgment $r, i \vdash_u \phi$ is secure with respect to $P$ and $\equiv$, written secure$_P,\equiv(r, i \vdash_u \phi)$, iff for all $r' \in$ traces$(L)$ such that $r' \equiv r'$, it holds that $[\phi]'^{db} = [\phi]'^{da'}$, where last$(r') = \langle db, U, sec, T, V, ctx \rangle$ and last$(r') = \langle db', U', sec', T', V', ctx' \rangle$. □

We are now ready to define data confidentiality. Intuitively, an access control mechanism provides data confidentiality iff all judgments that an attacker can derive are secure.

**Definition 6.4.** Let $P = \langle M, f \rangle$ be an extended configuration, $L$ be the $P$-LTS, $u \in U$ be a user, $A$ be a ($P, u$)-attacker model, and $\equiv$ be a $P$-indistinguishability relation. We say that $f$ provides data confidentiality with respect to $P$, $u$, $A$, and $\equiv$ iff secure$_P,\equiv(r, i \vdash_u \phi)$ for all judgments $r, i \vdash_u \phi$ that hold in $A$. □
6.4.2 Indistinguishability

We now define the indistinguishability relation that we use in the rest of the chapter. This relation captures what each user can observe (as specified in Section 6.2) and the effects of the system’s security policy. Let \( P = (D, \Gamma, f) \) be an extended configuration, \( L \) be the \( P \)-LTS, and \( u \) be a user in \( U \). Given a run \( r \in \text{traces}(L) \), the user \( u \) is aware only of his actions and not of the actions of the other users in \( r \). This is represented by the \( u \)-projection of \( r \), which is obtained by masking all sequences of actions that are not issued by \( u \) using a distinguished symbol \(*\). Specifically, the \( u \)-projection of \( r \) is an alternating sequence of states in \( \Omega_M \) and actions in \( A_{D,u} \cup \text{TRIGGER}_D \cup \{\ast\} \) that is obtained from \( r \) by (1) replacing each action not issued by \( u \) with \(*\), (2) replacing each trigger whose invoker is not \( u \) with \(*\), and (3) replacing all non-empty sequences of \(*\)-transitions with a single \(*\)-transition. For each user \( u \in U \), we define the \( P \)-indistinguishability relation \( \equiv_{P,u} \). Informally, two runs \( r \) and \( r' \) are \( \equiv_{P,u} \)-indistinguishable, denoted \( r \equiv_{P,u} r' \), iff (1) the labels of the \( u \)-projections of \( r \) and \( r' \) are the same, (2) \( u \) executes the same actions \( a_1, \ldots, a_n \) in \( r \) and \( r' \), in the same order, and with the same results, and (3) before each action \( a_i \), where \( 1 \leq i \leq n \), as well as in the last states of \( r \) and \( r' \), the policy, the views, the triggers, the users, and the data disclosed by the policy are the same in \( r \) and \( r' \).

We remark that there is a close relation between \( \equiv_{P,u} \) and existing indistinguishability relations over database states, such as those in [131,165] or the one from Section 3.4. For any two \( \equiv_{P,u} \)-indistinguishable runs \( r \) and \( r' \), the database states that precede all actions issued by \( u \) as well as the last states in \( r \) and \( r' \) are pairwise indistinguishable with respect to existing state-based notions [131,165].

Example 6.3 illustrates our indistinguishability notion.

**Example 6.3.** Let the schema, the set of users, the policy, and the triggers be as in Example 6.1. Consider the following run \( r(db) \), parametrized by the initial database state \( db \):

1. \( u \) deletes \( v \) from \( N \).
2. \( u \) inserts \( v \) in \( P \). If \( v \) is in \( T \), this activates the trigger \( t \), which, in turn, inserts \( v \) in \( N \).
3. \( u \) issues the \text{SELECT} query \( N(v) \).

Let \( db_1 \), \( db_2 \), and \( db_3 \) be three database states such that \( db_1(T) = \{v\} \), \( db_2(T) = \{j, v\} \), and \( db_3(T) = \emptyset \), whereas \( db_1(N) = \{v\} \) and \( db_2(P) = \emptyset \), for \( 1 \leq i \leq 3 \). Note that \( r(db_3) \) is the run used in Example 6.1. Figure 6.6 depicts how the database’s state changes during the runs \( r(db_i) \), for \( 1 \leq i \leq 3 \). Gray indicates those tables that the user \( u \) cannot read. The runs \( r(db_1) \) and \( r(db_2) \) are indistinguishable for the user \( u \). The only difference between them is the content of the table \( T \), which \( u \) cannot read. In contrast, \( u \) can distinguish between \( r(db_1) \) and \( r(db_3) \) because the trigger has been executed in the former and not in the latter.

Indistinguishability may also depend on the actions of the other users. Consider the runs \( r' \) and \( r'' \) obtained by extending \( r(db_1) \) respectively with one and two \text{SELECT} queries issued by the administrator just after \( u \)’s query. The user \( u \) can distinguish between \( r(db_3) \) and \( r'' \) because he knows that other users interacted with the system in \( r' \) but not in \( r(db_3) \), i.e., the \( u \)-projections have different labels. In contrast, the runs \( r' \) and \( r'' \) are indistinguishable for \( u \) because he only knows that, after his own \text{SELECT} query, other users interacted with the system, i.e., the \( u \)-projections have the same labels. However, he does not know the number of commands, the commands themselves, or their results.

We are now ready to formalize the indistinguishability relation \( \equiv_{P,u} \). We first formalize the notion of \( u \)-projection. Afterwards, we define the notion of consistency between \( u \)-projections. Finally, we formalize the indistinguishability relation \( \equiv_{P,u} \).
Projections

Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ and $f$ is an $M$-PDP, $L$ be the $P$-LTS, and $u$ be a user in $U$. As already mentioned, the $u$-projection of a run $r$ is obtained by (1) replacing each action not issued by $u$ with $*$, (2) replacing each trigger whose invoker is not $u$ with $*$, and (3) replacing all non-empty sequences of $*$-transitions with a single $*$-transition. Note that the $u$-projections in the $u$-projections represent whether $u$'s actions are executed consecutively or not. With a slight abuse of notation, we extend all the notation we use for runs also to $u$-projections. For instance, $r_u^i$ denotes the prefix obtained by truncating $r_u$ at its $i$-th state. Formally, the $u$-projection $r_u^i$ is defined as $c(v(r, u))$. The function $v$ takes as input a run $r$ and a user $u$ and returns another run in which all actions issued by users other than $u$ are replaced with $*$.

$$v(r, u) = \begin{cases} v(r', u) \cdot a \cdot s & \text{if } r = r' \cdot a \cdot s \in \text{traces}(L) \land s \in \Omega_M \land a \in A_{D, u} \land |r| > 1 \\
v(r', u) \cdot \ast \cdot s & \text{if } r = r' \cdot a \cdot s \in \text{traces}(L) \land s \in \Omega_M \land a \in A_{D, u} \land u' \neq u \land |r| > 1 \\
v(r', u) \cdot t \cdot s & \text{if } r = r' \cdot t \cdot s \in \text{traces}(L) \land s \in \Omega_M \land t \in TRIGGER_D \land invoker(last(r')) = u \land |r| > 1 \\
v(r', u) \cdot \ast \cdot s & \text{if } r = r' \cdot t \cdot s \in \text{traces}(L) \land s \in \Omega_M \land t \in TRIGGER_D \land invoker(last(r')) \neq u \land |r| > 1 \\
s & \text{if } r = s \land s \in \Omega_M \end{cases}$$

The function $c$ takes as input a run $r$ containing $*$-transitions and replaces each sequence of $*$-transitions with a single $*$-transition. Note that the function $c$ is obtained by repeatedly applying the function $c'$ until the computation reaches a fixed point. The function $c'$ is as follows:

$$c'(r) = \begin{cases} c'(r') \cdot a \cdot s & \text{if } r' = r' \cdot a \cdot s \land a \neq * \land r' \in \text{traces}(L) \land s \in \Omega_M \land |r| > 1 \\
c'(r') \cdot \ast \cdot s & \text{if } r' = r' \cdot \ast \cdot s \cdot s' \cdot s' \in \text{traces}(L) \land s, s', s' \in \Omega_M \land |r| > 2 \\
r & \text{if } r = s \land s \in \Omega_M \\
r & \text{if } r = s \cdot s' \land s, s' \in \Omega_M \\
\end{cases}$$

Consistency

We now define the notion of consistency between two $u$-projections. Observe that our consistency definition uses the function $labels$ that takes as input a run $r$ and returns as output the sequence of labels in the run, namely $labels(r)$ is obtained from $r$ by dropping all the states.

**Definition 6.5.** Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ and $f$ is an $M$-PDP, $L$ be the $P$-LTS, and $u$ be a user in $U$. Furthermore, let $r_u^i$ and $r_u'^j$ be the $u$-projections of the runs $r$ and $r'$ in traces($L$). We say that $r_u^i$ and $r_u'^j$ are consistent iff the following conditions hold:

1. $|r_u^i| = |r_u'^j|.$
2. $labels(r_u^i) = labels(r_u'^j).$
3. $\text{triggers} last(r_u^i) = \epsilon$ iff $\text{triggers} last(r_u'^j) = \epsilon.$
4. for all $i$ such that $1 \leq i \leq |r_u|,$ if $r_u^i = r_u'^i \cdot a \cdot s$ and $a \neq *,$ the following conditions hold:
   - $\text{res} last(r_u^i) = \text{res} last(r_u'^i).$
   - $\text{secEx} last(r_u^i) = \text{secEx} last(r_u'^i).$
   - if $a$ is a trigger, then $\text{accC} last(r_u^i) = \text{accC} last(r_u'^i).$
   - $\text{invoker} last(r_u^i) = \text{invoker} last(r_u'^i).$
   - $\text{triggers} last(r_u^i) = \text{triggers} last(r_u'^i).$
   - $\text{tpl} last(r_u^i) = \text{tpl} last(r_u'^i).$
   - $\text{sysState} last(r_u^i) = \text{sysState} last(r_u'^i).$

Indistinguishability

Let $M = (D, \Gamma)$ be an extended configuration, $s = (db, U, sec, T, V)$ be an $M$-system state, and $u \in U$ be a user. The set $\text{permissions}(s, u)$ is $\text{permissions}(s, u) := \{ (\oplus, \text{SELECT}, O) \mid \exists u' \in U, \text{op} \in \{ \oplus, \oplus' \}, \{ op, u' \}, (\text{SELECT}, O), u' \in sec \}.$ We also fix $\text{permissions}(s, admin)$ to be $D \cup V$ since the administrator has read access to the whole database. We extend permissions to $M$-states as follows. Given an $M$-state $s' = (db, U, sec, T, V, c),$ $\text{permissions}(s', u) = \text{permissions}(db, U, sec, T, V, u).$

We are now ready to introduce the notion of indistinguishability between two runs. Intuitively, two runs $r$ and $r'$ are indistinguishable for a user $u$ iff (1) their $u$-projections are consistent, and (2) for each action of the user $u$ as well as for the last states in the two runs, the policy, the triggers, the users, the views, and the data disclosed by the policy are the same in $r$ and $r'$.

**Definition 6.6.** Let $P = (M, f)$ be an extended configuration, $L$ be the $P$-LTS, and $u$ be a user.

We say that two runs $r$ and $r'$ in traces($L$) are $(P, u)$-indistinguishable, written $r \cong_{P, u} r'$, iff

1. $r_u$ and $r'_u$ are consistent,
2. $\text{sysState} last(r)$ and $\text{sysState} last(r')$ are $(M, u)$-data indistinguishable,
3. for all \( i \) such that \( 1 \leq i \leq |r|^u - 1 \), if \( r_i^u = r_i^u \cdot a \cdot s \), \( a \neq s \), and \( s \in \Omega_M \), then \( \text{sysState}(\text{last}(r_i^u)) \) and \( \text{sysState}(\text{last}(r_i^u)) \) are \((M, u)\)-data indistinguishable.

We say that two M-system states \( s = (db, U, sec, T, V) \) and \( s' = (db', U', sec', T', V') \) are \((M, u)\)-data indistinguishable, written \( s \equiv_{\text{data}} s' \), if (a) \( U = U' \), (b) \( sec = sec' \), (c) \( T = T' \), (d) \( V = V' \), (e) for all relation schema \( R \in D \) for which \((\oplus, \text{SELECT}, R) \in \text{permissions}(s, u)\), \( db(R) = db'(R) \), and (f) for all views \( v \in \text{VIEW}_D^{\text{owner}} \) for which \((\oplus, \text{SELECT}, v) \in \text{permissions}(s, u)\), \( db(v) = db'(v) \).

Given a run \( r \), we denote by \([r]_{F, u} \), the equivalence class of \( r \) defined by \( \equiv_{F, u} \) over \( \text{traces}(L) \). Similarly, we denote by \([s]_{M, u} \) the equivalence class of \( s \) defined by \( \equiv_{M, u} \) over \( \Pi_M \).

6.4.3 Examples

Example 6.4. In Example 6.1, we showed how the user \( u \) derives \( r, 3 \vdash_u T(v) \). The judgment is not secure because there is a run indistinguishable from \( r^3 \), i.e., the run \( r^3(d_b) \) in Example 6.3, in which \( T(v) \) does not hold.

Example 6.5 shows how views may leak information about the underlying tables. Even though this leakage might be considered legitimate, there is no way in our setting to distinguish between intended and unintended leakages. If this is desired, data confidentiality can be extended with the concept of declassification [20, 21].

Example 6.5. Consider a database with two tables \( T \) and \( Z \) and a view \( V = \{v \mid T(x) \land Z(x)\} \), \( O \). The set \( U \) is \( \{u, \text{admin}\} \) and the policy \( S \) is \( \{\oplus, u, (\text{SELECT } T), \text{admin}\}, \{\oplus, u, (\text{SELECT } V), \text{admin}\} \). Consider the following run \( r \), parametrized by the initial database state \( db \), where \( u \) first inserts 27 into \( T \) and afterwards issues the \text{SELECT} query \( V(27) \). We assume there are no exceptions in \( r \).

\[
\langle u, \text{INSERT}, T(27) \rangle \xrightarrow{(db[U, S, \emptyset, \{V\}, c_1] \in u, \text{INSERT}, T(27))} \langle db[T + 27], U, S, \emptyset, \{V\}, c_2 \rangle
\]

We used Maude to generate the runs \( r(d) \) and \( r(d') \) with the initial database states \( d \) and \( d' \) such that \( d(T) = d(Z) = d'(T) = \emptyset \) and \( d'(Z) = \{27\} \). The runs \( r^1(d) \) and \( r^1(d') \) are indistinguishable for \( u \) because they differ only in the content of \( Z \), which \( u \) cannot read. After the \text{INSERT}, \( u \) can distinguish between \( r^2(d) \) and \( r^2(d') \) by reading \( V \). Indeed, \( d[T + 27](V) = \emptyset \), because \( d(Z) = \emptyset \), whereas \( d'[T + 27](V) = \{27\} \). The user \( u \) derives \( r(d') \), \( 1 \vdash_u Z(27) \), which is not secure because \( r^1(d) \) and \( r^1(d') \) are indistinguishable for \( u \), but \( Z(27) \) holds just in the latter.

In contrast to existing security notions [131, 165], we have defined data confidentiality over runs. This is essential to model and detect attacks, such as those in Examples 6.4 and 6.5, where an attacker infers sensitive information from the transitions between states. For instance, the leakage in Example 6.5 is due to the execution of the \text{INSERT} command. Although the \text{SELECT} command is authorized by the policy, \( u \) can use it to infer sensitive information about the system’s state before the \text{INSERT} execution.

6.5 A Provably Secure PDP

We now present a PDP that provides both database integrity and data confidentiality. We first explain the ideas behind it using examples. Afterwards, we show that it satisfies the desired security properties and has acceptable overhead. Finally, we argue that it is more permissive than existing access control solutions for commands that do not violate our security properties.

Figure 6.7 depicts our PDP \( f \) together with the functions \( f_{\text{int}} \) and \( f_{\text{conf}} \). Additional details about the PDP are given in Section 6.8. The PDP takes as input a state \( s \) and an action \( a \) and outputs \( \top \) iff both \( f_{\text{int}} \) and \( f_{\text{conf}} \) authorize \( a \) in \( s \), i.e., iff \( a \)'s execution neither violates database integrity nor data confidentiality. Note that our algorithm is not \text{complete} in that it may reject some secure commands. However, from the impossibility results in Chapter 3, it follows that no algorithm can be complete and provide database integrity and data confidentiality for the relational calculus.

Our PDP is invoked by the database system each time a user \( u \) issues an action \( a \) to check whether \( u \) is authorized to execute \( a \). The PDP is also invoked whenever the database system executes a scheduled trigger \( t \): once to check if the \text{SELECT} statement associated with \( t \)'s \text{WHEN} condition is authorized and once, in case \( t \) is enabled, to check if \( t \)'s action is authorized.
6.5. A Provably Secure PDP

\[ f(s,a) \]

\[ \text{function } f(s,a) \]

1. return \( f_{\text{sec}}(s,a) \land f_{\text{conf}}(s,a, user(s,a)) \)

\[ \triangleright s \text{ is a state and } a \text{ is an action} \]

\[ \triangleright a \text{ is an action} \]

\[ \text{function } f_{\text{int}}(s,a) \]

1. if \( \text{try}(s) = \epsilon \) return \( \text{auth}(s,a) \)
2. else if \( a = \text{cond}(\text{try}(s), s) \) return \( \top \)
3. else if \( a = \text{act}(\text{try}(s), s) \)
4. return \( \text{auth}(s, \text{try}(s)) \)
5. else return \( \bot \)

\[ \triangleright s \text{ is a state and } a \text{ is an action} \]

\[ \text{function } f_{\text{conf}}(s,a,u) \]

1. switch \( a \)
2. case \( \langle u', \text{SELECT}, q \rangle : \text{return } \text{secure}(u,q,s) \)
3. case \( \langle u', \text{INSERT}, R, \Gamma \rangle : \text{case } \langle u', \text{DELETE}, R, \Gamma \rangle : \)
4. if \( \text{leak}(a,s,u) \lor \neg \text{secure}(u, \text{getInfo}(a), s) \)
5. return \( \bot \)
6. for \( \gamma \in \text{Dep}(a, \Gamma) \)
7. if \( \neg \text{secure}(u, \text{getInfo}(\gamma,a), s) \lor \neg \text{secure}(u, \text{getInfo}(a), s) \)
8. return \( \bot \)
9. case \( \langle \oplus, u''', pr, u' \rangle, \langle \ominus, u'', pr, u' \rangle : \)
10. return \( \neg \text{leak}(a,s,u) \)
11. return \( \top \)

\[ \begin{array}{c}
\text{Figure 6.7: The PDP } f \text{ uses the two subroutines } f_{\text{int}} \text{ and } f_{\text{conf}}. \text{ The former provides database integrity and the latter provides data confidentiality with respect to the user user}(s,a), \text{ which denotes either the user issuing the action, when the system is not executing a trigger, or the trigger's invoker.}
\end{array} \]

6.5.1 Enforcing Database Integrity

The function \( f_{\text{int}} \) takes as input a state \( s \) and an action \( a \). If the system is not executing a trigger, denoted by \( \text{try}(s) = \epsilon \), \( f_{\text{int}} \) checks (line 1) whether \( a \) is authorized with respect to \( s \). In line 2, \( f_{\text{int}} \) checks whether \( a \) is the current trigger’s condition. If this is the case, it returns \( \top \) because the triggers’ conditions do not violate database integrity. Finally, the algorithm checks (line 3) whether \( a \) is the current trigger’s action, and if this is the case, it checks whether the current trigger \( \text{try}(s) \) is authorized with respect to \( s \) (line 4). The function \( \text{auth} \) uses a sound and computable under-approximation of \( \text{\textit{auth}} \) to check if \( a \) is authorized with respect to \( s \). Thus, any action authorized by \( f_{\text{int}} \) is authorized according to \( \text{\textit{auth}} \). This ensures database integrity. Note that \( \text{\textit{auth}} \) relies on query determinacy (see Chapter 2) to decide whether a query is determined by a set of views. Since determinacy is undecidable [124], in \( \text{auth} \) we implement a sound under-approximation of it, given in Section 6.8.2. Our approximation checks syntactically if a query is determined by a set of views.

Example 6.6. Consider a database with three tables: \( R, T, \) and \( Z \). The set \( U \) is \( \{ u, u', \text{admin} \} \) and the policy \( S \) is \( \{ \langle \oplus, u, \text{SELECT}, R, \text{admin} \rangle, \langle \ominus, u, \text{SELECT}, T, \text{admin} \rangle, \langle \oplus, u, \text{SELECT}, Z, \text{admin} \rangle \} \).

There are two views \( W = \{ w, \text{admin}, \{ x \mid T(x) \land Z(x) \}, O \} \) and \( W = \{ w, u, \{ x \mid R(x) \lor V(x) \}, O \} \). The user \( u \) tries to grant to \( u' \) read access to \( W \), i.e., he issues \( \langle \oplus, u', \text{SELECT}, W, u \rangle \). The PDP \( f_{\text{int}} \) rejects the command and raises a security exception because \( u \) is not authorized to delegate the read access only for \( T \) and \( Z \) but \( W \)'s result depends also on \( R \), for which \( u \) cannot delegate read access. Assume now that the policy is \( \{ \langle \ominus, u, \text{SELECT}, R, \text{admin} \rangle, \langle \ominus, u, \text{SELECT}, T, \text{admin} \rangle, \langle \ominus, u, \text{SELECT}, Z, \text{admin} \rangle \} \).

In this case, \( f_{\text{int}} \) authorizes the \text{GRANT}. The reason is that \( W \)'s definition can be equivalently rewritten as \( \{ x \mid R(x) \lor (T(x) \land Z(x)) \} \) and \( u \) is authorized to delegate the read access for \( R, T, \) and \( Z \).

\[ \square \]

6.5.2 Enforcing Data Confidentiality

The function \( f_{\text{conf}} \), shown in Figure 6.7, takes as input an action \( a \), a state \( s \), and a user \( u \). Note that any user other than the administrator is a potential attacker. The requirement for \( f_{\text{conf}} \) is that it authorizes only those commands that result in secure judgments for \( a \) as required by Definition 6.4.

To achieve this, \( f_{\text{conf}} \) over-approximates the set of judgments that \( u \) can derive from \( a \)'s execution. For instance, the algorithm assumes that \( u \) can always derive the trigger’s condition from the run, even though this is not always the case. Then, \( f_{\text{conf}} \) authorizes \( a \) if it can determine that all \( u \)'s judgements are secure. This can be done by analysing just a finite subset of the over-approximated set of \( u \)'s judgments.

In more detail, \( f_{\text{conf}} \) performs a case distinction on the action \( a \) (line 1). If \( a \) is a \text{SELECT} command (line 2), \( f_{\text{conf}} \) checks whether the query is secure with respect to the current state \( s \) and the user \( u \) using the \text{secure} procedure. If \( a \) is an \text{INSERT} or \text{DELETE} command (lines 3–8), \( f_{\text{conf}} \) checks (lines 4–5), using the \text{leak} procedure, whether \( a \)'s execution may leak sensitive information through the views that \( u \) can read, as in Example 6.5. Afterwards, \( f_{\text{conf}} \) also checks (lines 4–5) whether the information \( u \) can learn from \( a \)'s execution, modelled by the sentence computed by the procedure \text{getInfo}(a), is secure. In line 6–8, \( f_{\text{conf}} \) computes the set of all integrity constraints that \( a \)'s execution may violate, denoted by \( \text{Dep}(a, \Gamma) \), and for all constraints \( \gamma \), it checks whether the information that \( u \) may learn from \( \gamma \) is secure. The procedure \text{getInfo}(S) (respectively \text{getInfo}(V)) computes the sentence
modelling the information learned by $\text{u}$ from $\gamma$ if $a$ is executed successfully (respectively violates $\gamma$).
If $a$ is a GRANT command (lines 9–10), $f_{\text{conf}}$ checks whether $a$’s successful execution discloses sensitive information to $\text{u}$. In the remaining cases (line 11), $f_{\text{conf}}$ authorizes $a$.

**Secure judgments.** Determining if a given judgment is secure is undecidable for RC (this directly follows from our results in Chapter 3). Hence, the secure procedure implements a sound and computable under-approximation of this notion. We now present our solution. Other sound under-approximations can alternatively be used without affecting $f_{\text{conf}}$’s data confidentiality guarantees.

Let $M = (D, \Gamma)$ be a system configuration, $r, i \vdash_{\text{u}} \phi$ be a judgment, and $s = \langle db, U, \text{sec}, T, V, c \rangle$ be the $i$-th state in $r$. As a first under-approximation, instead of the set of all runs indistinguishable from $r'$, we consider the largest set of all runs $r'$ whose last state $s' = \langle db', U, \text{sec}, T, V, c' \rangle$ is such that the disclosed data in $db$ and $db'$ are the same. Note that if a judgment is secure with respect to this larger set, it is secure also with respect to the set of indistinguishable runs because the former set contains the latter. This larger set depends just on the database state $db$ and the policy $sec$, not on the run or the attacker model $ATK_{\text{u}}$. Determining judgment’s security is, however, still undecidable on this larger set. We therefore employ a second under-approximation that uses query rewriting.

We rewrite the sentence $\phi$ to a sentence $\phi_{\text{rew}}$ such that if $r, i \vdash_{\text{u}} \phi$ is not secure for the user $\text{u}$, then $[\phi_{\text{rew}}]_{db} = \top$. The formula $\phi_{\text{rew}}$ is $\neg R_{\text{rew}}^\top \wedge \phi_{\text{rew}}^\top$, where $\phi_{\text{rew}}^\top$ and $\phi_{\text{rew}}^\bot$ are defined inductively over $\phi$. A formal definition of $\text{secure}$ is given in Section 6.8.3.

We now explain how we construct $\phi_{\text{rew}}^\top$ and $\phi_{\text{rew}}^\bot$. We assume that both $\phi$ and $V$ contain only views with the owner’s privileges. The extension to the general case is straightforward. First, for each table or view $o \in D \cup V$, we create additional views representing any possible projection of $o$. The extended vocabulary contains the tables in $D$, the views in $V$, and their projections. For instance, given a table $R(x, y)$, we create the views $R_x$ and $R_y$ representing respectively $\{y \mid \exists x. R(x, y)\}$ and $\{x \mid \exists y. R(x, y)\}$. Second, we compute the formula $\phi'$ by replacing each sub-formula $\exists x. R(\tau, y)$ in $\phi$ with the view $R_x(\tau)$ associated with the corresponding projection. Third, for each predicate symbol $R$ in the formula $\phi'$, we compute the sets $R_{\text{rew}}^\top$ and $R_{\text{rew}}^\bot$. The set $R_{\text{rew}}^\top$ (respectively $R_{\text{rew}}^\bot$) contains all the tables and views $K$ in the extended vocabulary such that (1) $K$ is contained in (respectively contains) $R$, and (2) the user $\text{u}$ is authorized to read $K$ in $s$, i.e., there is a grant $(\text{op}, u, (\text{SELECT}, K), u') \in s$ such that either $K' = K$ or $K$ is obtained from $K'$ through a projection. The formula $\phi_{\text{rew}}^\top$, where $v \in \{\top, \bot\}$, is defined as follows:

$$\phi_{\text{rew}}^v = \begin{cases} \bigvee_{s \in R_{\text{rew}}^\top} S(\tau) & \text{if } \phi = R(\tau) \text{ and } v = \top \\ \bigwedge_{s \in R_{\text{rew}}^\bot} S(\tau) & \text{if } \phi = R(\tau) \text{ and } v = \bot \\ \neg \psi_{\text{rew}}^v & \text{if } \phi = \neg \psi \\ \psi_{\text{rew}}^v \cdot v_{s,u} & \text{if } \phi = \psi \cdot v \text{ and } v \in \{\land, \lor\} \\ Q x. \psi_{\text{rew}}^v & \text{if } \phi = Q x. \psi \text{ and } Q \in \{\exists, \forall\} \\ \phi & \text{otherwise} \end{cases}$$

The formulae are such that if $\phi_{\text{rew}}^\top$ holds, then $\phi$ holds and if $\neg \phi_{\text{rew}}^\bot$ holds, then $\neg \phi$ holds. To compute the sets $R_{\text{rew}}^\top$ and $R_{\text{rew}}^\bot$, we check the containment between queries. Since query containment is undecidable [10], we implement a sound under-approximation of it, described in Section 6.8.3. Other sound under-approximations can be used as well.

Our $\phi_{\text{rew}}^\top$ and $\phi_{\text{rew}}^\bot$ rewritings share similarities with the low and high evaluations of Wang et al. [165]. Both try to approximate the result of a query just by looking at the authorized data. However, we use $\phi_{\text{rew}}^\top$ and $\phi_{\text{rew}}^\bot$ to determine a judgment’s security, whereas Wang et al. use evaluations to restrict the query’s results only to authorized data.

**Example 6.7.** Consider a database with three tables $S$, $R$, and $Q$, and two views $V = \{v, \text{admin}, x, y \mid S(x, y) \land (x = 1 \lor y = 3)\}, O$ and $W = \{w, \text{admin}, x \mid R(x) \lor Q(x)\}, O$. The database state $db$ is $db(S) = \{1, 2, 3\}, db(R) = \{\},$ and $db(Q) = \{4\}$. The set $U$ is $\{u, \text{admin}\}$, and the policy $sec$ is $\{(\phi, u, (\text{SELECT}, V), \text{admin}), (\phi, u, (\text{SELECT}, W), \text{admin})\}$. Let the state $s$ be $(\langle db, U, \text{sec}, \emptyset, (V, W) \rangle, c)$ and the run $r$ be $s$. We want to check the security of $r, 1 \vdash_{\text{u}} \phi$, where $\phi := (\exists y. S(2,y)) \land (-R(5) \lor \exists y. S(4,y))$ for the user $u$. Figure 6.8 depicts the database state $db$, the materializations of the views $V$ and $W$, and the materializations of the views $S_x$, $S_y$, $V_x$, and $V_y$ in the extended vocabulary. Gray indicates those tables and views that $u$ cannot read.

The rewriting process, depicted in Figure 6.8, proceeds as follows. We first rewrite the formula $\phi$ as $S_y(2) \land (-R(5) \lor V_y(4))$. The sets $S_y(2)$, $S_y(4)$, $R_{\text{rew}}^\top$, and $R_{\text{rew}}^\bot$ are respectively $\{V_y\}$, $\emptyset$, $\emptyset$, and $\{W\}$. The formulae $\phi_{\text{rew}}^\top$ and $\phi_{\text{rew}}^\bot$ are respectively $S_y(2) \land (-R(5) \lor V_y(4))$, which is equivalent to $V_y(2) \land (-W(5) \lor V_y(4))$, and $S_y(2) \land (-R(5) \lor V_y(4))$, which is equivalent to $\top$. They are both secure, as they depend only on $V$ and $W$. Furthermore, since $\phi_{\text{rew}}^\top$ holds in $s$, then $\phi$ holds as well. Finally, $\phi_{\text{rew}}$ is $\neg \phi_{\text{rew}}^\bot \land \phi_{\text{rew}}^\top$ Since $\phi_{\text{rew}}$ does not hold in $s$, it follows that $r, 1 \vdash_{\text{u}} \phi$ is secure. ■
6.5.3 Theoretical Evaluation

Our PDP provides the desired security guarantees and its data complexity, i.e., the complexity of executing \( f \) when the action, the policy, the triggers, and the views are fixed, is \( AC^0 \). This means that \( f \) can be evaluated in logarithmic space in the database’s size, as \( AC^0 \subseteq LOGSPACE \), and evaluation is highly parallelizable. Note that secure’s data complexity is \( AC^0 \) because it relies on query evaluation, whose data complexity is \( AC^0 \) \([10]\). In contrast, all other operations in \( f \) are executed in constant time in terms of data complexity. Note also that on a single processor, \( f \)’s data complexity is polynomial in the database’s size. We believe that this is acceptable because the database is usually very large, whereas the query, which determines the degree of the polynomial, is small. The proof of Theorem 6.1 is given in Appendix C.

**Theorem 6.1.** Let \( P = (M, f) \) be an extended configuration, where \( M \) is a system configuration and \( f \) is as above. The PDP \( f \) (1) provides data confidentiality with respect to \( P \), \( u \), \( ATK_u \), and \( \equiv_{P,u} \), for any user \( u \in U \), and (2) provides database integrity with respect to \( P \). Moreover, the data complexity of \( f \) is \( AC^0 \).

As the Examples 6.8 and 6.9 below show, \( f \) is more permissive than existing PDPs for those actions that violate neither database integrity nor data confidentiality.

**Example 6.8.** Our PDP is more permissive than existing mechanisms for commands of the form 
\[
\text{GRANT SELECT ON } V \text{ TO } u, \quad \text{where } V \text{ is a view with owner’s privileges, } u \text{ is a user, and the statement is issued by the view’s owner } o. \text{ Such mechanisms, in general, authorize the GRANT if } o \text{ is authorized to delegate the read permission for all tables and views that occur in } v \text{’s definition. Consider again Example 6.6. Our PDP authorizes } (\oplus, u', \langle \text{SELECT}, W \rangle, u) \text{ under the policy } S'. \text{ However, existing mechanisms reject it because } u \text{ is not directly authorized to read } V, \text{ although } u \text{ can read the underlying tables. Our PDP also authorizes all the secure GRANT statements authorized by existing mechanisms.}
\]

**Example 6.9.** Our PDP is more permissive than the mechanisms used in existing DBMSs for secure SELECT statements. Such mechanisms, in general, authorize a SELECT statement issued by a user \( u \) if \( u \) is authorized to read all tables and views used in the query. They will reject the query in Example 6.7 even though the query is secure. Furthermore, any secure SELECT statement authorized by them will be authorized by our solution as well. Also the PDP proposed by Rizvi et al. [131] rejects the query in Example 6.7 as insecure. However, our solution and the proposal of Rizvi et al. [131] are incomparable in terms of permissiveness, i.e., some secure SELECT queries are authorized by one mechanism and not by the other.
Chapter 6. Database Access Control in Modern Databases

6.5.4 Implementation

To evaluate the feasibility and security of our approach in practice, we implemented our PDP in Java. The prototype, available at [86], implements both our PDP and the operational semantics of our system model. It relies on the underlying PostgreSQL database for executing the SELECT, INSERT, and DELETE commands. Note that our prototype also handles all the differences between the relational calculus and SQL. For instance, it translates every relational calculus query into an equivalent SELECT SQL query over the underlying database. We performed a preliminary experimental evaluation of our prototype. Our experiments were run on a PC with an Intel i7 processor and 32GB of RAM. Note that we materialized the content of all the views.

Our evaluation has two objectives: (1) to empirically validate that the prototype provides the desired security guarantees, and (2) to evaluate its overhead. For (1), we ran the attacks in Section 6.1 against our prototype. As expected, our PDP prevents all the attacks. For (2), we simulated Examples 6.6 and 6.7 on database states where the number of tuples ranges from 1,000 to 100,000. Figures 6.9 and 6.10 shows the PDP’s execution time for the Examples 6.6 and 6.7 respectively. Our results show that our solution is feasible. In more detail, \( f_{\text{int}} \)'s execution time does not depend on the database size, whereas \( f_{\text{conf}} \)'s execution time does. We believe that the overhead introduced by the PDP is acceptable for a proof of concept. Even with 100,000 tuples, the PDP’s running time is under a second. In Example 6.7, \( f_{\text{conf}} \)'s execution time is comparable to the execution time of the user’s query. As Figure 6.11 shows, \( f_{\text{conf}} \)'s query rewriting time does not depend on the database’s size, whereas \( f_{\text{conf}} \)'s query execution time does.

To improve \( f_{\text{conf}} \)'s performance, one could strike a different balance between simple syntactic checks and our query rewriting solution. This, however, would result in more restrictive PDPs.

6.6 Related Work

We compare our work against two lines of research: database access control and information-flow control. Both of these have similar goals, namely preventing the leakage and corruption of sensitive information.

6.6.1 Database Access Control

Attacker Models for Database Access Control. Surprisingly, and in contrast to other areas of information security [66], there does not exist a well-defined attacker model for database access control. From the literature, we extracted the SELECT-only attacker model, where the attacker uses
6.6. Related Work

Figure 6.10: Example 6.7 – PDP execution time

Figure 6.11: Example 6.7 – $f_{conf}$’s execution time.
just SELECT commands. A number of access control mechanisms, such as [8,13,26,27,41,90,110,131,145,150,165], implicitly consider this attacker model. The boundaries of this model are blurred and the attacker’s capabilities are unclear. For instance, only a few works, such as [165], explicitly state that update commands are not supported, whereas others [13,26,27,131] ignore what the attacker can learn from update commands. Works on Inference Control [40,72,158] and Controlled Query Evaluation [36] consider a variation of the SELECT-only attacker, in which the attacker additionally has some initial knowledge about the data and can derive new information from the query’s results through inference rules. Note that while [158] supports update commands, it treats them just as a way of increasing data availability, rather than considering them as a possible attack vector.

**Discretionary Database Access Control.** Our framework builds on prior research in database access control [131,165] as well as established notions from database theory, such as determinacy [124] and instance-based determinacy [107].

Specifically, our notion of secure judgments extends instance-based determinacy from database states to runs, while data confidentiality extends existing security notions [131,165] to dynamic settings, where both the database and the policy may change. Observe that existing security notions for the Truman [165] and Non-Truman [131] models are based on SELECT-only attackers and provide no security guarantees against realistic attackers that can alter the database and the policy or use advanced SQL features. The same consideration applies to the security criteria we present in Chapter 3. Furthermore, our indistinguishability notion extends the one proposed by Wang et al. [165] (as well as our indistinguishability notion from Chapter 3) from database states to runs. Finally, our formalization of ↝auth relies on determinacy to decide whether the content of a view is fully determined by a set of other views.

Griffiths and Wade propose a PDP [83] that prevents Attacks 6.2 and 6.3 by using syntactic checks and by removing all views whose owners lack the necessary permissions. In contrast, we prevent the execution of GRANT and REVOKE operations leading to inconsistent policies.

**Mandatory Database Access Control.** Research on mandatory database access control has historically focused on Multi-Level Security (MLS) [63], where both the data and the users are associated with security levels, which are compared to control data access. Our PDP extends the SQL discretionary access control model with additional mandatory checks to provide database integrity and data confidentiality. In the following, we compare our work with the access control policies and semantics used by MLS systems.

With respect to policies, our work uses the SQL access control model, where policies are sets of GRANT statements. In this model, users can dynamically modify policies by delegating permissions. In contrast, MLS policies are usually expressed by labelling each subject and object in the system with labels from a security lattice [138]. The policy is, in general, fixed (cf. the tranquility principle [138]).

With respect to semantics, existing MLS solutions are based on the Truman model [131]; they transparently modify the commands issued by the users to restrict the access to only the authorized data. In contrast, we use the same semantics as SQL, that is, we execute only the secure commands. Namely, we adopt the Non-Truman model [131]. We refer the reader to Chapter 3 for a detailed comparison of these two access control models. Operationally, MLS mechanisms use poly-instantiation [99], which is neither supported by the relational model nor by the SQL standard, and requires ad hoc extensions [63,140]. Furthermore, the operational semantics of MLS systems differs from the standard relational semantics. In contrast, our operational semantics supports the relational model and is directly inspired by SQL.

The above differences influence how security properties are expressed. Data confidentiality, which relies on a precise characterization of security based on a possible worlds semantics, is a key component of the Non-Truman model (and SQL) access control semantics. Similarly, database integrity requires that any “write” operation is authorized according to the policy and is directly inspired by the SQL access control semantics. The security properties in MLS systems, in contrast, combine the multilevel relational semantics [63,140] with MLS and BIBA properties [138].

MLS systems prevent attacks similar to Attacks 6.4 and 6.5 using poly-instantiated tuples and triggers [140,149], whereas attacks similar to Attack 6.1 cannot be carried out because triggers with activator’s privileges are not supported [149]. The SeaView system [63], which combines discretionary access control and MLS, additionally prevents attacks similar to Attacks 6.2 and 6.3 by relying on Griffiths and Wade’s PDP [83]. However, these solutions cannot be applied to SQL databases for the aforementioned reasons.

### 6.6.2 Information-flow Control

Various authors have applied ideas from information-flow control to databases. Davis and Chen [59] study how cross-application information flows can be tracked through databases. Other researchers [57, 111,141] present languages for developing secure applications that use databases. They employ secure
type systems to track information flows through databases. However, they neither model nor prevent the attacks we identified because they do not account for the advanced database features and the strong attacker model we study in this chapter.

Schultz and Liskov [143] extend decentralized information-flow control [122] to databases, based on concepts from multi-level security [63]. They identify one attack on data confidentiality that exploits integrity constraints. Their solution relies on poly-instantiation [99] and cannot be applied to SQL databases that do not support multi-level security. Their mechanism neither prevents the other attacks we identify nor provides provable and precise security guarantees.

Several researchers have studied attacker models in information-flow control [18, 78]. Giacobazzi and Mastroeni [78] model attackers as data-flow analysers that observe the program’s behaviour, whereas Askarov and Chong [18] model attackers as automata that observe the program’s events. They both model passive attackers, who can observe, but do not influence, the program’s execution. In contrast, our attacker is active and interacts with the database.

6.7 Conclusions

Motivated by practical attacks against existing databases, we have initiated several new research directions. First, we developed the idea that attacker models should be studied and formalized for databases. Rather than being implicit, the relevant models must be made explicit, just like when analyzing security in other domains. In this respect, we presented a concrete attacker model that accounts for relevant features of modern databases, like triggers and views, and attacker inference capabilities.

Second, access control mechanisms must be designed to be secure, and provably so, with respect to the formalized attacker capabilities. This requires research on mechanism design, complemented by a formal operational semantics of databases as a basis for security proofs. We presented such a mechanism, proved that it is secure, and built and evaluated a prototype of it in PostgreSQL.

6.8 Technical Details

In Section 6.8.1, we present our full attacker model. Afterwards, we provide further details on how we enforce database integrity (Section 6.8.2) and data confidentiality (Section 6.8.3). Finally, in Section 6.8.4 we formalize our enforcement mechanism as a PDP.

6.8.1 Full Attacker Model

In this section, we formalize our attacker model $ATK_u$. Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ is a system configuration and $f$ is an $M$-PDP. Let $L$ be the P-LTS, and $u \in U$ be a user. The set $ATK_u$ is the smallest set of judgments satisfying the inference rules in Figures 6.12-6.24. With a slight abuse of notation, in the rules we use $r, i \vdash_u \phi$ to denote that this judgment holds in $ATK_u$, i.e., $r, i \vdash \phi \in ATK_u$. Note that we redefine here also the rules we presented before in Figure 6.3.

In our attacker model, we assume that the attacker knows the PDP’s implementation. Hence, he can learn information from leaks caused by the security decision. The rules in Figure 6.23 say that whenever a PDP leaks information through its security decision, then the attacker may learn any sentence that differentiates between the database states. These rules over-approximate what an attacker may infer from leaks caused by the PDP’s security decision.

In the rules, we use $|=_{fin}$ to denote the standard semantic entailment relation for first-order logic over finite models. Given a view $⟨V, ow, \{T | \phi\}, m⟩ \in VIEW_D$ and a relational calculus formula $ψ$, we denote by replace($ψ$, $⟨V, ow, \{T | \phi\}, m⟩$) the formula $ψ'$ obtained from $ψ$ by replacing all occurrences of $V(\Gamma)$ with $ϕ(\Gamma)$. Note that $ψ$ and replace($ψ$, $⟨V, ow, \{T | \phi\}, m⟩$) are semantically equivalent. Finally, given a database schema $D$, a state $s = ⟨db, U, sec, T, V, ctx⟩$, and an action $a ∈ A_{D,U} ∪ TRHG DB_D$, we denote by $user(s, a)$ the following function:

$$user(s, a) = \begin{cases} invoker(s) & \text{if } tr(s) \neq \epsilon \\ u & \text{if } tr(s) = \epsilon ∧ u \in U ∧ a ∈ A_{D,u} \end{cases}$$

In the following, we omit some details when dealing with integrity constraints. For instance, when we refer to functional dependencies of the form $\forall \mathbb{r}, g, g', \mathbb{r}, \mathbb{r}'. (R(\mathbb{r}, g, \mathbb{r}) ∧ R(\mathbb{r}, g', \mathbb{r}')) → g = g'$, we implicitly assume that $[g] = [g']$ and $[\mathbb{r}] = [\mathbb{r}']$. Furthermore, when we refer to tuples in $R$, we use the notation $[\mathbb{v}, \mathbb{w}, g]$ to stress that a tuple can be partitioned according to $\mathbb{v}$, $\mathbb{w}$, and $g$, and we implicitly assume that $[\mathbb{v}] = [\mathbb{v}]$, $[\mathbb{w}] = [\mathbb{w}]$, and $[g] = [g]$. We make similar simplifications for the inclusion dependencies.
Chapter 6. Database Access Control in Modern Databases

![Image](image.png)

Figure 6.12: Rules defining how the attacker propagates the knowledge.

![Image](image.png)

Figure 6.13: Rules regulating how information propagates in case of successful INSERT and DELETE.
6.8. Technical Details

DELETE Success - 1
\[1 < i \leq |r| \quad r^i = r^{i-1} \cdot \langle u, \text{SELECT}, \phi \rangle \cdot s = \langle db, U, \text{sec}, T, V, h, \langle \langle u, \text{SELECT}, \phi \rangle, T, \top, \langle \epsilon, \epsilon, \epsilon \rangle \rangle \quad r, i \vdash_u \phi \]

DELETE Success - 2
\[1 < i \leq |r| \quad r^i = r^{i-1} \cdot \langle u, \text{SELECT}, \phi \rangle \cdot s = \langle db, U, \text{sec}, T, V, h, \langle \langle u, \text{SELECT}, \phi \rangle, \bot, \top, \langle \epsilon, \epsilon, \epsilon \rangle \rangle \quad r, i \vdash_u \neg \phi \]

INSERT Success
\[r^i = r^{i-1} \cdot \langle u, \text{INSERT}, R, T \rangle \cdot s \quad 1 < i \leq |r| \quad s = \langle db, U, \text{sec}, T, V, h, \langle \langle u, \text{INSERT}, R, T \rangle, T, \top, \langle \epsilon, \epsilon, \epsilon \rangle \rangle \quad r, i \vdash_u R(T) \]

INSERT Success - FD
\[l \in \{i, i-1\} \quad r^i = r^{i-1} \cdot \langle u, \text{INSERT}, R, T \rangle \cdot s \quad \forall \exists \; \gamma, \xi, \phi, (\langle R(\tau, \xi, \phi) \rangle \wedge R(\tau, \xi, \phi)) \rightarrow \gamma = \gamma' \in \Gamma \setminus E \quad \top = (\tau, \emptyset) \]

INSERT Success - ID
\[l \in \{i, i-1\} \quad r^i = r^{i-1} \cdot \langle u, \text{INSERT}, R, T \rangle \cdot s \quad \forall \exists \; \gamma, \xi, \phi, (\langle R(\tau, \xi, \phi) \rangle \wedge R(\tau, \xi, \phi)) \rightarrow \gamma = \gamma' \in \Gamma \setminus E \quad \top = (\tau, \emptyset) \]

DELETE Success
\[r^i = r^{i-1} \cdot \langle u, \text{DELETE}, R, T \rangle \cdot s \quad 1 < i \leq |r| \quad s = \langle db, U, \text{sec}, T, V, h, \langle \langle u, \text{DELETE}, R, T \rangle, T, \top, \langle \epsilon, \epsilon, \epsilon \rangle \rangle \quad r, i \vdash_u \neg R(T) \]

DELETE Success - ID
\[l \in \{i, i-1\} \quad r^i = r^{i-1} \cdot \langle u, \text{DELETE}, R, T \rangle \cdot s \quad \forall \exists \; \gamma, \xi, \phi, (\langle R(\tau, \xi, \phi) \rangle \wedge R(\tau, \xi, \phi)) \rightarrow \gamma = \gamma' \in \Gamma \setminus E \quad \top = (\tau, \emptyset) \]

INSERT Exception
\[l \in \{i, i-1\} \quad r^i = r^{i-1} \cdot \langle u, \text{INSERT}, R, T \rangle \cdot s \quad E \neq \emptyset \quad r, i \vdash_u \neg R(T) \]

DELETE Exception
\[l \in \{i, i-1\} \quad r^i = r^{i-1} \cdot \langle u, \text{DELETE}, R, T \rangle \cdot s \quad E \neq \emptyset \quad r, i \vdash_u \neg R(T) \]

INSERT FD Exception
\[r^i = r^{i-1} \cdot \langle u, \text{INSERT}, R, T \rangle \cdot \gamma \in \{i, i-1\} \quad s = \langle db, U, \text{sec}, T, V, h, \langle \langle u, \text{INSERT}, R, T \rangle, T, \top, \langle \epsilon, \epsilon, \epsilon \rangle \rangle \quad (\forall \exists \; \gamma, \xi, \phi, (\langle R(\tau, \xi, \phi) \rangle \wedge R(\tau, \xi, \phi)) \rightarrow \gamma = \gamma' \in \Gamma \setminus E \quad \top = (\tau, \emptyset) \]

INSERT ID Exception
\[r^i = r^{i-1} \cdot \langle u, \text{INSERT}, R, T \rangle \cdot \gamma \in \{i, i-1\} \quad s = \langle db, U, \text{sec}, T, V, h, \langle \langle u, \text{INSERT}, R, T \rangle, T, \top, \langle \epsilon, \epsilon, \epsilon \rangle \rangle \quad (\forall \exists \; \gamma, \xi, \phi, (\langle R(\tau, \xi, \phi) \rangle \wedge R(\tau, \xi, \phi)) \rightarrow \gamma = \gamma' \in \Gamma \setminus E \quad \top = (\tau, \emptyset) \]

DELETE ID Exception
\[r^i = r^{i-1} \cdot \langle u, \text{DELETE}, R, T \rangle \cdot \gamma \in \{i, i-1\} \quad s = \langle db, U, \text{sec}, T, V, h, \langle \langle u, \text{DELETE}, R, T \rangle, T, \top, \langle \epsilon, \epsilon, \epsilon \rangle \rangle \quad (\forall \exists \; \gamma, \xi, \phi, (\langle R(\tau, \xi, \phi) \rangle \wedge R(\tau, \xi, \phi)) \rightarrow \gamma = \gamma' \in \Gamma \setminus E \quad \top = (\tau, \emptyset) \]

Insert Integrity Constraint
\[1 \leq i \leq |r| \quad s \in \text{last}(r^i).V \quad r, i \vdash_u \psi \quad \psi' = \text{replace}(\psi, v) \quad r, i \vdash_u \psi'' \]

Figure 6.14: Rules defining how the attacker extracts knowledge from the run.
Chapter 6. Database Access Control in Modern Databases

**Figure 6.15:** Rules regulating how information propagates in case of rollbacks.

\[
\begin{align*}
&1 \leq i \leq |r| \\
&\Phi \subseteq \{ \phi \mid r, i \vdash_u \phi \} \\
&\Phi \models \gamma
\end{align*}
\]

**Figure 6.16:** Rules regulating the reasoning.

**Figure 6.17:** Rules describing how the attacker learns facts about `INSERT` and `DELETE` commands.

**Figure 6.18:** Rules regulating the propagation of information through disabled triggers.
Learn $\text{INSERT FORWARD}$

$r^i = r^{i-1}.t.s$

$\text{invoker}(\text{last}(r^{i-1})) = u$

$s = (db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{INSERT}, R, \overline{t}, T, \emptyset \rangle \rangle, tr)$

$\text{secEx}(s) = \bot$

$Ex(s) = \emptyset$

$t = (id, ow, ev, R', \phi, act, m)$

$r, i \vdash_u R(\overline{t})$

Learn $\text{INSERT FD}$

$r^i = r^{i-1}.t.s$

$\text{invoker}(\text{last}(r^{i-1})) = u$

$s = (db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{INSERT}, R, \overline{t}, T, \emptyset \rangle \rangle, tr)$

$l \in \{i, i-1\}$

$\text{secEx}(s) = \bot$

$Ex(s) = \emptyset$

$t = (id, ow, ev, R', \phi, act, m)$

$r, i \vdash_u \neg \exists \overline{y}. \exists! \overline{z}. R(\overline{t}, \overline{y}, \overline{z}) \land \overline{y} \neq \overline{w}$

Learn $\text{INSERT FD - 1}$

$r^i = r^{i-1}.t.s$

$\text{invoker}(\text{last}(r^{i-1})) = u$

$s = (db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{INSERT}, R, \overline{t}, T, \emptyset \rangle \rangle, tr)$

$\overline{t} = (\overline{t}, \overline{w})$

$\text{secEx}(s) = \bot$

$t = (id, ow, ev, R', \phi, act, m)$

$r, i \vdash_u \neg \exists \overline{y}. \exists! \overline{z}. R(\overline{t}, \overline{y}, \overline{z}) \land \overline{y} \neq \overline{w}$

Learn $\text{INSERT ID}$

$r^i = r^{i-1}.t.s$

$\text{invoker}(\text{last}(r^{i-1})) = u$

$s = (db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{INSERT}, R, \overline{t}, T, \emptyset \rangle \rangle, tr)$

$l \in \{i, i-1\}$

$\text{secEx}(s) = \bot$

$Ex(s) = \emptyset$

$t = (id, ow, ev, R', \phi, act, m)$

$r, i \vdash_u \neg \exists \overline{y}. S(\overline{t}, \overline{y})$

Learn $\text{INSERT ID - 1}$

$r^i = r^{i-1}.t.s$

$\text{invoker}(\text{last}(r^{i-1})) = u$

$s = (db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{INSERT}, R, \overline{t}, T, \emptyset \rangle \rangle, tr)$

$\overline{t} = (\overline{t}, \overline{w})$

$\text{secEx}(s) = \bot$

$t = (id, ow, ev, R', \phi, act, m)$

$r, i \vdash_u \neg \exists \overline{y}. S(\overline{t}, \overline{y})$

Learn $\text{INSERT BACKWARD - 1}$

$r^i = r^{i-1}.t.s$

$\text{invoker}(\text{last}(r^{i-1})) = u$

$s = (db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{INSERT}, R, \overline{t}, T, \emptyset \rangle \rangle, tr)$

$\text{secEx}(s) = \bot$

$Ex(s) = \emptyset$

$t = (id, ow, ev, R', \phi, act, m)$

$r, i \vdash_u \neg \psi$

$r, i \vdash_u \neg \psi$

Learn $\text{INSERT BACKWARD - 2}$

$r^i = r^{i-1}.t.s$

$\text{invoker}(\text{last}(r^{i-1})) = u$

$s = (db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{INSERT}, R, \overline{t}, T, \emptyset \rangle \rangle, tr)$

$\text{secEx}(s) = \bot$

$Ex(s) = \emptyset$

$t = (id, ow, ev, R', \phi, act, m)$

$r, i \vdash_u \neg \psi$

$r, i \vdash_u \neg \psi$

Figure 6.19: Extracting knowledge from triggers – part 1.
Chapter 6. Database Access Control in Modern Databases

Learn DELETE Forward

\[ r, i - 1 \vdash_u \phi[R | R |] \mapsto \tau \in \text{last}(r^{i-1})] \quad 1 < i \leq |r| \]
\[ r^i = r^{i-1} \cdot t, s \quad \text{invoker(last(r^{i-1}) = u} \]
\[ \text{secEx}(s) = \perp \quad \text{Ex}(s) = \emptyset \quad t = (id, ow, ev, R', \phi, act, m) \]
\[ r, i \vdash_u \neg R(t) \]

Learn DELETE - ID

\[ r, i - 1 \vdash_u \phi[R | R |] \mapsto \tau \in \text{last}(r^{i-1})] \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot t, s \quad \text{invoker(last(r^{i-1}) = u} \]
\[ s = (\langle db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{DELETE}, R, T \rangle, T, \emptyset \rangle), \text{tr} \rangle) \]
\[ t \in \{i, i - 1\} \quad \text{secEx}(s) = \perp \quad \text{Ex}(s) = \emptyset \quad t = (id, ow, ev, R', \phi, act, m) \]
\[ (\forall \tau, \exists (S(\tau, \tau) \mapsto \tau) \mapsto \text{Ex}(\tau)) \in \Gamma \quad \Gamma = (\tau, \tau) \]
\[ r, i - 1 \vdash_u \forall \tau, \exists (S(\tau, \tau) \mapsto \tau) \mapsto \text{Ex}(\tau) \]

Learn DELETE Backward - 1

\[ r^i = r^{i-1} \cdot t, s \quad \text{invoker(last(r^{i-1}) = u} \quad 1 < i \leq |r| \]
\[ \text{secEx}(s) = \perp \quad \text{Ex}(s) = \emptyset \quad t = (id, ow, ev, R', \phi, act, m) \]
\[ r, i - 1 \vdash_u \psi \quad r, i \vdash_u \neg \psi \]

Learn DELETE Backward - 2

\[ r^i = r^{i-1} \cdot t, s \quad \text{invoker(last(r^{i-1}) = u} \quad 1 < i \leq |r| \]
\[ \text{secEx}(s) = \perp \quad \text{Ex}(s) = \emptyset \quad t = (id, ow, ev, R', \phi, act, m) \]
\[ r, i - 1 \vdash_u \psi \quad r, i \vdash_u \neg \psi \]

Learn GRANT/REVOKE Backward

\[ r^i = r^{i-1} \cdot t, s \quad \text{invoker(last(r^{i-1}) = u} \quad 1 < i \leq |r| \]
\[ s = (\langle db, U, \text{sec}, T, V, h, (t, \text{when}, \langle \langle u', \text{DELETE}, R, T \rangle, T, \emptyset \rangle), \text{tr} \rangle) \]
\[ \text{secEx}(s) = \perp \quad \text{Ex}(s) = \emptyset \quad t = (id, ow, ev, R', \phi, act, m) \]
\[ u', u'' \in U \quad \text{op} \in \{\oplus, \ominus, \odot\} \quad \text{last(r^{i-1}), sec} \neq \text{last(r^i), sec} \]
\[ r, i - 1 \vdash_u \phi[R | R |] \mapsto \tau \in \text{last}(r^{i-1})] \]

Figure 6.20: Extracting knowledge from triggers – part 2.
Figure 6.21: Rules for propagating knowledge through triggers.
Chapter 6. Database Access Control in Modern Databases

Trigger FD INSERT Disabled Backward

$$1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s$$

$$\text{invoker}(\text{last}(r^{i-1})) = u \quad s = (db, U, sec, T, V, h, (t, when, stmt), tr) \quad \text{secEx}(s) = \bot \quad \text{Ex}(s) = \emptyset$$

$$t = (id, ow, ev, R, \phi, act, m) \quad \text{action}(act, user(\text{last}(r^{i-1}), t), tpl(\text{last}(r^{i-1})) = (u', \text{INSERT}, R, (v, \emptyset))$$

$$r, i - 1 \vdash u \land \neg \phi[*]^R \rightarrow t pl(\text{last}(r^{i-1}))]$$

Trigger FD DELETE Disabled Backward

$$1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s$$

$$\text{invoker}(\text{last}(r^{i-1})) = u \quad s = (db, U, sec, T, V, h, (t, when, stmt), tr) \quad \text{secEx}(s) = \bot \quad \text{Ex}(s) = \emptyset$$

$$t = (id, ow, ev, R, \phi, act, m) \quad \text{action}(act, user(\text{last}(r^{i-1}), t), tpl(\text{last}(r^{i-1})) = (u', \text{DELETE}, R, (v, \emptyset))$$

$$r, i - 1 \vdash u \land \neg \phi[*]^R \rightarrow t pl(\text{last}(r^{i-1}))]$$

Trigger GRANT Disabled Backward

$$1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s$$

$$\text{invoker}(\text{last}(r^{i-1})) = u \quad s = (db, U, sec, T, V, h, (t, when, stmt), tr) \quad \text{secEx}(s) = \bot \quad \text{Ex}(s) = \emptyset$$

$$t = (id, ow, ev, R', \phi, act, m) \quad \text{action}(act, user(\text{last}(r^{i-1}), t), tpl(\text{last}(r^{i-1})) = (u', \phi, p, u')$$

$$u', u'' \in U \quad op \in \{\oplus, \oslash\} \quad (op, u', p, u') \notin \text{last}(r^{i-1}), \text{sec} \quad \text{last}(r^{i-1}), \text{sec} = \text{sec}$$

$$r, i - 1 \vdash u \land \neg \phi[*]^R \rightarrow t pl(\text{last}(r^{i-1}))]$$

Figure 6.22: Extracting knowledge from disabled triggers.

Learn from deny - actions

$$r, r', r'' \in \text{traces}(L) \quad a \in A_{D_{r''}} \quad s, s' \in \Omega_M \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot a \cdot s$$

$$r' = r'' \cdot a \cdot s' \quad r'' = r' \cdot s' \quad r'' \cdot s' \in F_{P_{r''}, r''} \quad \text{secEx}(s) \neq \text{secEx}(s') \quad \text{ex}[a]_{\text{last}(r^{i-1}), db] = \top \quad \text{ex}[a]_{\text{last}(r'), db] = \bot$$

$$r, i - 1 \vdash u \land \phi$$

Learn from deny - triggers

$$r, r', r'' \in \text{traces}(L) \quad t \in \text{TRIGGER}_{P_{r''}} \quad s, s' \in \Omega_M \quad 1 < i \leq |r| \quad r^i = r^{i-1} \cdot t \cdot s$$

$$r' = r'' \cdot t \cdot s' \quad r'' = r' \cdot s' \quad r'' \cdot s' \in F_{P_{r''}, r''} \quad \text{invoker}(\text{last}(r^{i-1})) = u \quad \text{invoker}(\text{last}(r'')) = u \quad \text{accC}(s) \neq \text{accC}(s') \lor \text{secEx}(s) \neq \text{secEx}(s') \quad \text{ex}[a]_{\text{last}(r^{i-1}), db] = \top \quad \text{ex}[a]_{\text{last}(r''), db] = \bot$$

$$r, i - 1 \vdash u \land \phi$$

Figure 6.23: Extracting knowledge from the PDP.
6.8. Technical Details

Trigger INSERT FD Exception
\[ 1 < i \leq |r| \quad r^i = r^{i-1}.t.s \quad \text{invoker}(\text{last}(r^{i-1})) = u \]
\[ s = (db, U, sec, T, V, h, (t, when, ((u', \text{INSERT}, R, T), T, T, E), tr)) \]
\[ t = (id, ow, ev, R', \phi, act, m) \quad (\forall \bar{x}, \bar{y}, \bar{z}. ((R(\bar{x}, \bar{y}, \bar{z}) \land R(\bar{x}, \bar{y}, \bar{z})) \Rightarrow \bar{y} = \bar{y}')) \in \text{Ex}(s) \]
\[ r, i - 1 \vdash u \exists \bar{x}, \exists \bar{y}. R(\bar{x}, \bar{y}, \bar{z}) \land \bar{y} \neq \bar{y}' \]

Trigger INSERT ID Exception
\[ 1 < i \leq |r| \quad r^i = r^{i-1}.t.s \quad \text{invoker}(\text{last}(r^{i-1})) = u \]
\[ s = (db, U, sec, T, V, h, (t, when, ((u', \text{INSERT}, R, T), T, T, E), tr)) \]
\[ \text{secEx}(s) = \bot \quad t = (id, ow, ev, R', \phi, act, m) \quad (\forall \bar{x}, \bar{y}, \bar{z}. ((R(\bar{x}, \bar{y}, \bar{z}) \Rightarrow \exists \bar{w}. S(\bar{x}, \bar{w})) \in \text{Ex}(s)) \]
\[ r, i - 1 \vdash u \forall \bar{x}, \bar{y}. S(\bar{x}, \bar{y}) \Rightarrow \bar{y} \neq \bar{y}' \]

Trigger DELETE ID Exception
\[ 1 < i \leq |r| \quad r^i = r^{i-1}.t.s \quad \text{invoker}(\text{last}(r^{i-1})) = u \]
\[ s = (db, U, sec, T, V, h, (t, when, ((u', \text{DELETE}, R, T), T, T, E), tr)) \]
\[ \text{secEx}(s) = \bot \quad t = (id, ow, ev, R', \phi, act, m) \quad (\forall \bar{x}, \bar{y}, \bar{z}. ((R(\bar{x}, \bar{y}, \bar{z}) \Rightarrow \exists \bar{w}. R(\bar{x}, \bar{w})) \in \text{Ex}(s)) \]
\[ r, i - 1 \vdash u \exists \bar{x}, \exists \bar{y}. R(\bar{x}, \bar{y}, \bar{z}) \land \bar{y} = \bar{y}' \]

Trigger Exception
\[ 1 < i \leq |r| \quad r^i = r^{i-1}.t.s \quad \text{invoker}(\text{last}(r^{i-1})) = u \]
\[ s = (db, U, sec, T, V, h, (t, when, ((u', \text{SELECT}, \phi, T, T, \emptyset), ((u', \text{INSERT}, R, T), res, aC, E), tr)) \]
\[ \text{secEx}(s) = \top \lor \text{Ex}(s) \neq \emptyset \quad t = (id, ow, ev, R, \phi, act, m) \]
\[ r, i - 1 \vdash u \phi[\bar{x}] \Rightarrow t \in \text{tp}(\text{last}(r^{i-1})) \]

Trigger INSERT Exception
\[ 1 < i \leq |r| \quad r^i = r^{i-1}.t.s \quad \text{invoker}(\text{last}(r^{i-1})) = u \]
\[ s = (db, U, sec, T, V, h, (t, ((u', \text{SELECT}, \phi, T, T, \emptyset), ((u', \text{INSERT}, R, T), res, aC, E), tr)) \]
\[ \text{secEx}(s) = \bot \quad \text{Ex}(s) \neq \emptyset \quad t = (id, ow, ev, R', \phi, act, m) \]
\[ r, i - 1 \vdash u \neg \text{R}(\bar{t}) \]

Trigger DELETE Exception
\[ 1 < i \leq |r| \quad r^i = r^{i-1}.t.s \quad \text{invoker}(\text{last}(r^{i-1})) = u \]
\[ s = (db, U, sec, T, V, h, (t, ((u', \text{SELECT}, \phi, T, T, \emptyset), ((u', \text{DELETE}, R, T), res, aC, E), tr)) \]
\[ \text{secEx}(s) = \bot \quad \text{Ex}(s) \neq \emptyset \quad t = (id, ow, ev, R', \phi, act, m) \]
\[ r, i - 1 \vdash u \neg \text{R}(\bar{t}) \]

Trigger Rollback INSERT
\[ n + 1 < i \leq |r| \quad s_1, s_2, \ldots, s_n \in \Omega_M \]
\[ t_1, \ldots, t_n \in \text{TRIGGER}_D \quad \text{secEx}(s_n) = \top \lor \text{Ex}(s_n) \neq \emptyset \]
\[ r^i = r^{i-1}.(u, \text{INSERT}, R, T), s_1, t_1, s_2, \ldots, s_n \quad s_n = (db, U, sec, T, V, h, (t_n, when, stmt), (e, e, e)) \]
\[ r, i - 1 \vdash u \neg \text{R}(\bar{t}) \]

Trigger Rollback DELETE
\[ n + 1 < i \leq |r| \quad s_1, s_2, \ldots, s_n \in \Omega_M \quad t_1, \ldots, t_n \in \text{TRIGGER}_D \quad \text{secEx}(s_n) = \top \lor \text{Ex}(s_n) \neq \emptyset \]
\[ r^i = r^{i-1}.(u, \text{DELETE}, R, T), s_1, t_1, s_2, \ldots, s_n \quad s_n = (db, U, sec, T, V, h, (t_n, when, stmt), (e, e, e)) \]
\[ r, i - 1 \vdash u \neg \text{R}(\bar{t}) \]

Figure 6.24: Extracting knowledge from trigger’s exceptions.
6.8.2 Enforcing Database Integrity

Here, we define the access control function \( f_{\text{int}} \), which models the \( f_{\text{int}} \) procedure described in Section 6.5. The function \( f_{\text{int}} \) is as follows:

\[
f_{\text{int}}(s, a) = \begin{cases} 
  \top & \text{if } \text{trigger}(s) = \epsilon \land s \sim_{\text{appr}} a \\
  \top & \text{if } \text{trigger}(s) = t \land t \neq \epsilon \land a = \text{trigAct}(s) \\
  \top & \text{if } \text{trigger}(s) = (id, ow, e, R, \phi, st, A) \land a = \text{trigAct}(s) \\
  \top & \text{if } \text{trigger}(s) = (id, ow, e, R, \phi, st, O) \land a = \text{trigAct}(s) \\
  \bot & \text{otherwise}
\end{cases}
\]

The function \( \text{trigCond}(s) \) (respectively \( \text{trigAct}(s) \)) returns the condition (respectively the action) associated with the trigger \( \text{trigger}(s) \). If \( \text{trigger}(s) = (id, ow, e, R, \phi, st, O) \), then \( \text{trigAct}(s) = \text{getAction}(st, ow, tpl(s)) \). If \( \text{trigger}(s) = (id, ow, e, R, \phi, st, A) \), then \( \text{trigAct}(s) = \text{getAction}(st, invoker(s), tpl(s)) \) and \( \text{trigCond}(s) = \{ \text{SELECT}, \phi[\mathfrak{M}^{R}] \mapsto \text{tpl}(s) \} \).

The relation \( \sim_{\text{appr}} \subseteq \Omega_{M} \times (A_{D,D} \cup \mathcal{T}_{\text{TRIGGER}} D) \) is the smallest relation satisfying the inference rules given in Figure 6.25. Informally, \( \sim_{\text{appr}} \) is obtained from \( \sim_{\text{auth}} \) by replacing determinacy with a sound under-approximation. As we show in Appendix C, \( \sim_{\text{appr}} \) is a sound and computable under-approximation of the relation \( \sim_{\text{auth}} \). In the rules, we use a number of auxiliary functions. The most important ones are:

(a) The \( aT \) (respectively \( aV \)) function that takes as input a database state, an operator \( op \) in \( \{ \oplus, \ominus \} \), and a user, and returns the set of tables (respectively views) that the user is authorized to read (if \( op = \oplus \)) or to delegate the read access to other users (if \( op = \ominus \)) according to our approximation of \( \sim_{\text{auth}} \). These functions are defined as follows:

\[
aT((db, U, sec, T, V, c), op, u) = \{ R \in D \mid (u = \text{admin} \land \exists u' \in U, op' \in \{ \oplus, \ominus \}, (db, U, sec, T, V, c) \sim_{\text{appr}} (op', u, (\text{SELECT}, R), u')) \lor \exists u' \in U, g \in sec, op' \in \{ \oplus, \ominus \}, g = (op', u, (\text{SELECT}, R), u') \land (db, U, sec, T, V, c) \sim_{\text{appr}} g \}
\]

\[
aV((db, U, sec, T, V, c), op, u) = \{ V \in V \cap \forall E \mathcal{W}_{D}^{\text{owner}} \mid (u = \text{admin} \land \exists u' \in U, op' \in \{ \oplus, \ominus \}, (db, U, sec, T, V, c) \sim_{\text{appr}} (op', u, (\text{SELECT}, V), u')) \lor \exists u' \in U, g \in sec, op' \in \{ \oplus, \ominus \}, g = (op', u, (\text{SELECT}, V), u') \land (db, U, sec, T, V, c) \sim_{\text{appr}} g \}
\]

(b) The \( \text{apprDet} \) function is used to determine whether a set of tables and a set of views completely determine the result of a formula \( \phi \) in all possible database states. Note that the function \( \text{apprDet} \) is a sound under-approximation of query determinacy [124].

A sound under-approximation of query determinacy

Here we define our sound under-approximation of query determinacy, implemented in the function \( \text{apprDet} \). In the following, we assume that both the formula \( \phi \) and the set of views \( V \) in the state \( s \) contain just views with owner’s privileges. This is without loss of generality since views with activator’s privileges are just syntactic sugar (they do not disclose additional information to a user other than what he is already authorized to read because they are executed under \( u \)’s privileges).

Let \( M \) be a system configuration, \( s = (db, U, sec, T, V) \) be an \( M \)-system state, and \( (v, o, q, O) \) be a view with owner’s privileges. We denote by \( \text{inline}_{M}(v, o, q, O) \) the view \( (v, o, q', O) \) where \( q' \) is obtained from \( q \) by replacing all occurrences of views in \( V \) with owner’s privileges with their definitions. Note that \( \text{inline}_{M} \) does not compute a fix-point, i.e., if a view’s definition refers to another view, the latter is not replaced with its definition.

The \( \text{apprDet} \) function takes as input a system configuration \( M \), a set of tables \( T \), a set of views \( V \) with owner’s privileges \( V \), a query \( \phi \), and a partial \( M \)-state, and returns \( \top \) if the tables and views in \( T \) and \( V \) determine the result of \( \phi \). The function is formalized below, and it relies on \( \text{extend}(M, s, V) \), which is the smallest set satisfying the following recurrence relation: \( \text{extend}(M, s, V) = V \cup \{ \text{inline}(v, s) \mid v \in \text{extend}(M, s, V) \} \).
6.8.3 Enforcing Data Confidentiality

Here, we define the PDP $f_{conf}^u(s, a)$, which models the function $f_{conf}(s, a, u)$ from Section 6.5. The PDP $f_{conf}^u$ is shown in Figure 6.26. The function is parametrized by the user $u$ against which the PDP provides data confidentiality. The mapping between the PDP $f_{conf}^u$ and the pseudo-code shown in Figure 6.7 is immediate.

The PDP $f_{conf}^u$ uses a number of auxiliary functions. Recall that the function $tr$ takes as input an $M$-state $s \in OM$ and returns the definition of the trigger that the system is executing. If the system is not executing any trigger, then $tr(s) = \epsilon$. Equivalently, $tr(s)$ is the first trigger in the sequence of triggers returned by $triggers(s)$.

The function $tDet$ takes as input a view $v = (i, o, \{\phi\}, m) \in VIEW_D$, a state $s \in OM$, and a system configuration $M = (D, \Gamma)$ and returns as output the smallest set of tables in $D$ that determines $v$, namely the smallest set $T \in 2^{|D|}$ such that $appDet(T, \emptyset, s, M)$ holds, where $appDet$ is defined in Section 6.8.2. Note that such a set is always unique.

The function $noLeak$ takes as input a state $s$, an INSERT or DELETE action $a$, and a user $u$ and it checks whether the execution of the action $a$ may leak sensitive information through the views that the user $u$ can read, as shown in Example 6.5. Note that the function $noLeak$ returns $\top$ if there is no leakage of sensitive information and returns $\bot$ if the action $a$ may leak sensitive information through the views the user $u$ can read in the state $s$. Formally, $noLeak(s, \{u', \emptyset, R, \top\}, u)$ is defined as follows:

\[
\begin{align*}
\top \quad & \text{if } u' = u \land trigger(s) = \epsilon \land \forall v \in VIEW_D. ((\{\emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset, \emptyset) \in tDet(v, s, M)) \rightarrow (\forall o \in tDet(v, s, M). (\{\emptyset, \emptyset, \emptyset, \emptyset\} \in permissions(s, u))) \\
\bot \quad & \text{otherwise}
\end{align*}
\]

We now define the $Dep$, $getInfoS$, $getInfoV$, and $getInfo$ functions. The function $Dep$ retrieves all integrity constraints that may be violated by executing an INSERT or DELETE action. Formally, $Dep(u, INSERT, R, T, \Gamma)$ returns the set containing all the formulae in $\Gamma$ of the form $\forall \tau, \gamma, \gamma', \tau', (R(\tau, \gamma, \gamma') \land R(\tau', \gamma, \gamma')) \rightarrow \gamma' = \gamma'$ or $\forall \tau, \gamma, \tau'. R(\tau, \gamma) \rightarrow \exists \tau'. S(\tau, \gamma)$, whereas $Dep(u, DELETE, R, \top, \Gamma)$ returns the set containing all the formulae in $\Gamma$ of the form $\forall \tau, \gamma, \tau'. S(\tau, \gamma) \rightarrow \exists \tau'. R(\tau, \gamma)$.

The function $getInfoS$ returns the sentence capturing what an attacker may learn from the integrity constraint $\phi$ in case an action is executed successfully. The function is defined as follows:

- $getInfoS(u, INSERT, R, (\tau, \gamma, \gamma'), \phi_{funct}^{R})$ is the formula $\exists \gamma, \tau. R(\tau, \gamma, \gamma') \land \gamma' \neq \gamma$, where $\phi_{funct}^{R}$ is a formula of the form $\forall \tau, \gamma, \gamma', \tau'. R(\tau, \gamma, \gamma') \land R(\tau', \gamma, \gamma') \rightarrow \gamma' = \gamma'$.
- $getInfoS(u, INSERT, R, (\tau, \gamma, \gamma'), \phi_{funct}^{R,S})$ is the formula $\exists \gamma, \tau. S(\tau, \gamma)$, where $\phi_{funct}^{R,S}$ is a formula of the form $\forall \tau, \gamma, \tau'. R(\tau, \gamma) \rightarrow \exists \tau'. S(\tau, \gamma)$.
- $getInfoS(u, DELETE, R, (\tau, \gamma, \gamma'), \phi_{funct}^{R})$ is the formula $\forall \tau, \gamma, \tau'. (S(\tau, \gamma) \rightarrow \exists \tau'. R(\tau, \gamma') \land \gamma' \neq \gamma)$, where $\phi_{funct}^{R}$ is a formula of the form $\exists \gamma, \tau. R(\tau, \gamma, \gamma') \rightarrow \gamma' = \gamma'$.
- $getInfoS(\emptyset, \phi) = \top$ otherwise.

The function $getInfoV$ returns the sentence capturing what an attacker may learn from the integrity constraint $\phi$ in case an action violates the constraint $\phi$. The function is as follows:

- $getInfoV(u, INSERT, R, (\tau, \gamma, \gamma'), \phi_{funct}^{R})$ is the formula $\exists \gamma, \tau. R(\tau, \gamma, \gamma') \land \gamma' \neq \gamma$, where $\phi_{funct}^{R}$ is a formula of the form $\forall \tau, \gamma, \gamma', \tau'. (R(\tau, \gamma, \gamma') \land R(\tau', \gamma, \gamma')) \rightarrow \gamma' = \gamma'$.

Observe that our under-approximation of query determinacy ignores the integrity constraints. This is always sound since if $D, \emptyset \vdash Q \rightarrow q$, then $D, \Gamma \vdash Q \rightarrow q$.
Add User
\[ u \in U \quad u' = \text{admin} \quad (db, U, sec, T, V, c) \xrightarrow{\text{add}} (u', \text{ADD USER}, u) \]

Insert-Delete
\[ g \in \text{sec} \quad R \in D \quad t \in \{\oplus, \ominus\} \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-1}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} g \quad (db, U, sec, T, V, c) \xrightarrow{\text{Insert-Delete}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} (db, U, sec, T, V, c) \xrightarrow{\text{Insert-Delete}} \]

Create View
\[ v \in \text{VIEW} \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-2}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} v \quad (db, U, sec, T, V, c) \xrightarrow{\text{Create VIEW}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Create Trigger
\[ t \in \text{TRIGGER} \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-3}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} v \quad (db, U, sec, T, V, c) \xrightarrow{\text{Create Trigger}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Insert-Delete admin
\[ R \in D \quad t \in \text{TRIGGER} \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-4}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} L \]

Revoke
\[ u, u' \in U \quad priv \in PRIV \quad s = (db, U, sec, T, V, c) \xrightarrow{\text{Grant-5}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} v \quad (db, U, sec, T, V, c) \xrightarrow{\text{Revoke}} \]

Grant-1
\[ op \in \{\oplus, \ominus\} \quad priv \in PRIV \quad g = (db, U, sec, T, V, c) \xrightarrow{\text{Grant-1}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Grant-2
\[ u \in U \quad op \in \{\oplus, \ominus\} \quad priv \in PRIV \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-2}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Grant-3
\[ u, owner \in U \quad op \in \{\oplus, \ominus\} \quad priv = (SELECT, v) \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-3}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Grant-4
\[ u, owner \in U \quad op \in \{\oplus, \ominus\} \quad priv = (SELECT, v) \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-4}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Grant-5
\[ u, owner \in U \quad op \in \{\oplus, \ominus\} \quad priv = (SELECT, v) \quad (db, U, sec, T, V, c) \xrightarrow{\text{Grant-5}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Execute Trigger-1
\[ t \in T \quad (db, U, sec, T, V, c) \xrightarrow{\text{Get Action}} t \quad (db, U, sec, T, V, c) \xrightarrow{\text{op}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Execute Trigger-2
\[ t \in T \quad (db, U, sec, T, V, c) \xrightarrow{\text{Get Action}} t \quad (db, U, sec, T, V, c) \xrightarrow{\text{op}} (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Execute Trigger-3
\[ t \in T \quad (db, U, sec, T, V, c) \xrightarrow{\text{Get Action}} t \quad (db, U, sec, T, V, c) \xrightarrow{\text{op}} \]

Figure 6.25: Definition of the \( \xrightarrow{\text{app}} \) relation.
6.8. Technical Details

\[ f_{\text{conf}}^u(s, \text{act}) = \begin{cases} f_{\text{conf}, \text{g}}^u(s, \text{act}) & \text{if } \text{act} = (u', \text{SELECT}, q) \\ f_{\text{conf}, \text{i}, \text{g}}^u(s, \text{act}) & \text{if } \text{act} = (u', \text{INSERT}, R, \Gamma) \\ f_{\text{conf}, \text{i}, \text{d}}^u(s, \text{act}) & \text{if } \text{act} = (u', \text{DELETE}, R, \Gamma) \\ f_{\text{conf}, \text{a}, \text{h}}^u(s, \text{act}) & \text{if } \text{act} = (\text{op}, u'', p, u') \land \text{op} \in \{\oplus, \oplus^*\} \\ \top & \text{if } u = \text{admin} \\ \top & \text{otherwise} \end{cases} \]

\[ f_{\text{conf}, \text{g}}^u(s, \langle u', \text{SELECT}, q \rangle) = \begin{cases} \text{secure}(u, \text{getInfo}(\text{act}), s) \land \\
\bigwedge_{\gamma \in \text{Dep}(\text{act}, \Gamma)} \text{secure}(u, \text{getInfoS}(\gamma, \text{act}), s) \\
\top & \text{if } \text{trigger}(s) = \epsilon \\
\bot & \text{otherwise} \end{cases} \]

\[ f_{\text{conf}, \text{i}, \text{g}}^u(s, \langle u', \text{INSERT}, R, \Gamma \rangle) = \begin{cases} \text{secure}(u, \text{getInfo}(\text{act}), s) \land \\
\bigwedge_{\gamma \in \text{Dep}(\text{act}, \Gamma)} \text{secure}(u, \text{getInfoS}(\gamma, \text{act}), s) \\
\bot & \text{if } \text{trigger}(s) = \epsilon \\
\top & \text{otherwise} \end{cases} \]

\[ f_{\text{conf}, \text{i}, \text{d}}^u(s, \langle u', \text{DELETE}, R, \Gamma \rangle) = \begin{cases} \text{secure}(u, \text{getInfo}(\text{act}), s) \land \\
\bigwedge_{\gamma \in \text{Dep}(\text{act}, \Gamma)} \text{secure}(u, \text{getInfoS}(\gamma, \text{act}), s) \\
\bot & \text{if } \text{trigger}(s) = \epsilon \\
\top & \text{otherwise} \end{cases} \]

\[ f_{\text{conf}, \text{a}, \text{h}}^u(s, \langle \text{op}, u'', p, u' \rangle) = \begin{cases} \bot & \text{if } u'' = u \land u'' = u \land \text{trigger}(s) = \epsilon \land p \in \{\oplus, \oplus^*\} \land p = (\text{SELECT}, O) \land (\oplus, \text{SELECT}, O) \notin \text{permissions}(s, u) \\
\bot & \text{otherwise} \end{cases} \]

\[ f_{\text{conf}, \text{g}}^u(s, \langle \text{op}, u'', p, u' \rangle) = \begin{cases} \bot & \text{if } u'' = u \land \text{trigger}(s) = \epsilon \land p = (\text{SELECT}, O) \land (\oplus, \text{SELECT}, O) \notin \text{permissions}(s, u) \\
\top & \text{otherwise} \end{cases} \]

Figure 6.26: Access control function \( f_{\text{conf}}^u \).
Conjunction: \[
\frac{\text{free} (\phi \land \psi) = \text{free} (\phi)}{\phi \land \psi \subseteq M \phi}
\]

Disjunction: \[
\frac{\text{free} (\phi) = \text{free} (\phi \lor \psi)}{\phi \lor \psi \subseteq M \phi}
\]

Projection: \[
\frac{M = (D, \Gamma), \text{free} (\phi) \neq \emptyset, 1 \leq i \leq n, \phi \subseteq M \psi}{\exists x_i, \phi \subseteq M \exists y_i \phi}
\]

Identity: \[
\frac{M = (D, \Gamma), \text{free} (\phi) = \emptyset, \gamma \subseteq M \psi}{\phi \subseteq M \psi}
\]

Inclusion: \[
\frac{\forall \pi, \tau, (R(\pi, \tau) \rightarrow \exists \pi'. S(\pi, \tau')) \in \Gamma}{\exists \pi. R(\pi, \tau) \subseteq M \exists \pi'. S(\pi, \tau')}
\]

Dependency: \[
\frac{\phi \subseteq M \gamma}{\text{free} (\phi) \neq \emptyset}
\]

Transitivity: \[
\frac{M = (D, \Gamma), \text{free} (\phi) \neq \emptyset, \text{free} (\gamma) \neq \emptyset}{\phi \subseteq M \psi}
\]

Figure 6.27: Containment rules.

- `getInfoV(u, INSERT, R, [π, τ])`, `φ_{insert}^{R, S}` is the formula `∀π, τ, S(π, τ) → τ ≠ π`, where `φ_{insert}^{R, S}` is a formula of the form `∀π, τ, R(π, τ) → π = τ`.
- `getInfoV(u, DELETE, R, [π, τ])`, `φ_{insert}^{R, S}` is the formula `∃π, S(π, τ) ∧ ∀τ, R(τ, τ) → τ = π`.
- `getInfoV(\text{act}, \phi) = \top` otherwise.

Finally, the function `getInfo`, which returns the information an attacker may learn by the successful execution of an `INSERT` or `DELETE` command, is as follows:

\[
\text{getInfo}(u, op, R, t) = \left\{ \begin{array}{ll}
\neg R(t) & \text{if } op = \text{INSERT} \\
R(t) & \text{if } op = \text{DELETE}
\end{array} \right.
\]

Checking a judgment’s security

We now define the `secure : U × RC_{bool} × \Omega_M → \{\top, \bot\}` function that determines whether a given judgment is secure. In more detail, the `secure` function is as follows:

\[
\text{secure}(u, \phi, s) = \left\{ \begin{array}{ll}
\top & \text{if } [\phi_{\text{conf}}^{u}_{\text{rel}}]^s_{\text{th}} = \bot \\
\bot & \text{otherwise}
\end{array} \right.
\]

Again, in the following, we assume that both the formula `\phi` and the set of views `V` in the state `s` contain just views with owner’s privileges. The extension to the general case is straightforward.

Before defining the `\phi_{\text{conf}}^{u}_{\text{rel}}` and `\phi_{\text{conf}}^{u}_{\text{rel}}` rewritings, we define query containment. Let `M = (D, \Gamma)` be a system configuration. Given two formulae `\phi(\pi)` and `\psi(\tau)`, we write `\phi \subseteq M \psi` to denote that `\phi` is contained in `\psi`, i.e., `∀db \in \Omega_M.[\{\pi | \phi\}] \subseteq [\{\tau | \psi\}]_{\text{db}}`. Determining whether `\phi \subseteq M \psi` holds is undecidable for the relational calculus [10]. Hence, we develop a sound, under-approximation of query containment. Figure 6.27 describes the rules defining our under-approximation. For simplicity’s sake, the rules are defined only for relational calculus formulae that do not use views. To check whether `\phi \subseteq M \psi` holds for two formulae `\phi` and `\psi` that use views, we first compute the formulae `\phi'$ and `\psi'$, obtained by replacing views’ identifiers with their definitions, and then we check whether `\phi' \subseteq M \psi'$ using the rules in Figure 6.27. This preserves containment since `\phi` and `\psi` are semantically equivalent to `\phi'$ and `\psi'`. Observe that in the rules, we assume that there is a total ordering `≤_{\text{var}}` over the set of all possible variable identifiers. This ensures that, given a formula `\phi`, there is a unique non-boolean query `φ(\pi)` associated with it, where the variables in `\pi` are those in `\text{free}(\phi)` ordered according to `≤_{\text{var}}`.

Given a table or a view `O` and a sequence of distinct integers `i = i_1 \ldots i_n` such that `1 \leq i_j \leq |O|` for all `1 \leq j \leq n`, where `0 \leq n < |O|`, the `i`-projection of `O`, denoted by `O_i`, is the formula `∃x_{i_1}, \ldots, x_{i_n}, O(x_{i_1}, \ldots, x_{i_n})`. Given a database schema `D` and a set of views `V` defined over `D`, we denote by `V_{\text{ext}}(D, V)` the extended vocabulary obtained by defining all possible projections of tables in `D` and views in `V`, i.e., for each `O \in D \cup V`, we define a predicate `O_{i_j}` for each projection `∃x_{i_1}, \ldots, x_{i_n}, O(x_{i_1}, \ldots, x_{i_n})` of `O`. Furthermore, given a relational calculus formula `\phi` over `D`, we denote by `V_{\text{ext}}(D, \phi)` the formula obtained by replacing all sub-formulae of the form `∃\pi. R(\pi, \tau)` with the predicates in `V_{\text{ext}}(D, V)` representing the corresponding projections `R_{\pi}`. Finally, we denote by `\phi_{\text{conf}}^{u}_{\text{rel}}(\phi)`, where `\phi` is a relational calculus formula over `V_{\text{ext}}(D, V)`, the formula `\phi'$ obtained by replacing all predicates associated with projections with the corresponding formulae.
Let \( S \) be a predicate in \( \text{extVocabulary}(D, V) \) and \( s \) be an \( M \)-state. We denote by \( S^+_\text{V} \) the set of all projections of tables in \( D \) and views in \( V \) that are contained in \( S \), i.e., \( S^+_\text{V} := \{ R \in \text{extVocabulary}(D, V) \mid R(\pi)_\text{V} \subseteq S(\pi) \} \). Similarly, we denote by \( S^+ \) the set of all projections of tables in \( D \) and views in \( V \) that contains \( S \), i.e., \( S^+ := \{ R \in \text{extVocabulary}(D, V) \mid S(\pi) \subseteq R(\pi) \} \). Furthermore, we denote by \( \text{AUTH}_{s,u} \) the set of all tables and views that \( u \) is authorized to read in \( s \), i.e., \( \text{AUTH}_{s,u} := \{ o \mid (s, o) \in \text{permissions}(s, u) \} \), and by \( \text{AUTH}^+_{s,u} \) the set of all the projections obtained from tables and views in \( \text{AUTH}_{s,u} \).

We are now ready to formally define the \( \phi^+_\text{V} \) and \( \phi^+ \) rewritings.

**Definition 6.7.** Let \( M = (D, \Gamma) \) be a system configuration, \( s = (dh, U, \text{sec}, T, V, c) \) be an \( M \)-state, \( u \) be a user, and \( \phi \) be a relational calculus sentence over \( \text{extVocabulary}(D, V) \).

The function \( \text{bound} \) takes as input a formula \( \phi \), a state \( s \), a user \( u \), a variable identifier \( x \), and a value \( v \) in \( \{ \top, \bot \} \). It is inductively defined as follows:

- \( \text{bound}(R(\pi), s, u, x, v) \), where \( R \) is a predicate symbol in \( \text{extVocabulary}(D, V) \), is \( \top \) iff (a) \( x \) occurs in \( \pi \), and (b) the set \( R^+ := R^+_\text{V} \cap \text{AUTH}^+_{s,u} \), is not empty.
- \( \text{bound}(\psi, s, u, x, v) := \text{bound}(\psi, s, u, x, \neg v) \), where \( \psi \) is a relational calculus formula.
- \( \text{bound}(\psi \land \gamma, s, u, x, v) := \text{bound}(\psi, s, u, x, v) \land \text{bound}(\gamma, s, u, x, v) \), where \( \psi \) and \( \gamma \) are relational calculus formulae.
- \( \text{bound}(\psi \lor \gamma, s, u, x, v) := \text{bound}(\psi, s, u, x, v) \lor \text{bound}(\gamma, s, u, x, v) \), where \( \psi \) and \( \gamma \) are relational calculus formulae.
- \( \text{bound}(\exists y. \psi, s, u, x, v) := \text{bound}(\psi, s, u, x, y) \) if \( x \neq y \), and \( \text{bound}(\exists y. \psi, s, u, x, v) := \bot \) otherwise.
- \( \text{bound}(\forall y. \psi, s, u, x, v) \), where \( \psi \) is a relational calculus formula and \( x \) is a variable identifier, is \( \exists x. \psi^T_{s,u} \) if \( \text{bound}(\psi, s, u, x, T) = \top \) and \( (\exists x. \psi^T_{s,u}) := \bot \) otherwise.
- \( (\forall x. \psi)_{s,u}^T \), where \( \psi \) is a relational calculus formula and \( x \) is a variable identifier, is \( \forall x. \psi^T_{s,u} \) if \( \text{bound}(\psi, s, u, x, T) = \top \) and \( (\forall x. \psi)^T_{s,u} := \bot \) otherwise.

The formula \( \phi^+_\text{V} \) is inductively defined as follows:

\[
R(\pi)_{s,u} := \bigvee_{S \in R^+_\text{V}} S(\pi), \quad (x = v)_{s,u} := (x = v), \quad (\top)_{s,u} := \top, \quad (\bot)_{s,u} := \bot, \quad (\neg \psi)_{s,u} := \neg \psi^+_\text{V}, \quad (\psi \land \gamma)_{s,u} := \psi^+_\text{V} \land \gamma^+_\text{V}, \quad (\psi \lor \gamma)_{s,u} := \psi^+_\text{V} \lor \gamma^+_\text{V}, \quad (\exists x. \psi)_{s,u} := \exists x. \psi^T_{s,u}, \quad (\forall x. \psi)_{s,u} := \forall x. \psi^T_{s,u}, \quad (\psi \land \gamma)_{s,u} := \psi^+_\text{V} \land \gamma^+_\text{V}, \quad (\psi \lor \gamma)_{s,u} := \psi^+_\text{V} \lor \gamma^+_\text{V}, \quad (\exists x. \psi)_{s,u} := \exists x. \psi^T_{s,u}, \quad (\forall x. \psi)_{s,u} := \forall x. \psi^T_{s,u}
\]

The formula \( \phi^+ \) is inductively defined as follows:

\[
R(\pi)_{s,u} := \bigwedge_{S \in R^+} S(\pi), \quad (x = v)_{s,u} := (x = v), \quad (\top)_{s,u} := \top, \quad (\bot)_{s,u} := \bot, \quad (\neg \psi)_{s,u} := \neg \psi^+ \text{V}, \quad (\psi \land \gamma)_{s,u} := \psi^+ \land \gamma^+, \quad (\psi \lor \gamma)_{s,u} := \psi^+ \lor \gamma^+, \quad (\exists x. \psi)_{s,u} := \exists x. \psi^T_{s,u}, \quad (\forall x. \psi)_{s,u} := \forall x. \psi^T_{s,u}, \quad (\psi \land \gamma)_{s,u} := \psi^+ \land \gamma^+, \quad (\psi \lor \gamma)_{s,u} := \psi^+ \lor \gamma^+, \quad (\exists x. \psi)_{s,u} := \exists x. \psi^T_{s,u}, \quad (\forall x. \psi)_{s,u} := \forall x. \psi^T_{s,u}
\]

Finally, we define the formula \( \phi^+ \) which represents the overall rewritten formula.

**Definition 6.8.** Let \( M = (D, \Gamma) \) be a system configuration, \( s = (dh, U, \text{sec}, T, V, c) \) be an \( M \)-state, \( u \) be a user, and \( \phi \) be a relational calculus sentence over \( D \). The formula \( \phi^+ \) is defined as

\[
\text{inlined}_{\text{V}, D}(\neg \psi^T_{s,u} \land \psi^+_\text{V}), \quad \psi := \text{extVoc}_{\text{V}, D}(\phi).
\]

\( ^2 \)With a slight abuse of notation, we write \( R(\pi) \subseteq_M S(\pi) \) instead of \( \text{inlined}_{\text{V}, D}(\nu R(\pi). \phi) \subseteq_M \text{inlined}_{\text{V}, D}(\nu S(\pi). \phi) \).
6.8.4 Enforcement Mechanism

Here, we model the PDP $f$, presented in Section 6.5, which is obtained by composing the PDPs $f_{int}$ and $f_{conf}$ presented above. The PDP $f$ is obtained by composing $f_{int}$ and $f_{conf}$ as follows:

$$f(s, act) = \text{int}(s, act) \land \text{conf}(s, act)$$

We recall that the function \text{user} takes as input an action and a state and returns the actual user executing the action. It is defined as follows, where $\text{i}$ denotes the \text{invoker} function and $\text{tr}$ denotes the \text{trigger} function.

$$\text{user}(act, s) = \begin{cases} \text{invoker}(s) & \text{if } \text{trigger}(s) \neq \epsilon \\ u & \text{if } \text{trigger}(s) = \epsilon \text{ and } act \in A_{D,u} \end{cases}$$
Chapter 7

Reconciling Database Access Control and Information-flow Control

Access control formulations suffer from a number of difficulties. First, because they are described in terms of a mechanism for enforcing security, they provide no guidance in circumstances where those mechanisms prove inadequate. Second, it is easy to construct perverse interpretations of access control policies that satisfy the letter, but not the intent of the policy, to the point of being obviously insecure.

John Rushby – Noninterference, transitivity, and channel-control security policies

Database access control (DBAC) and information-flow control (IFC) share the same goal: protecting the confidentiality and integrity of sensitive information. However, they have different foundations, security notions, and enforcement mechanisms. In this chapter, we reconcile these two, seemingly disparate, areas by developing a framework for reasoning about the security of database-backed programs. First, we reduce database access control to determining whether programs leak information, thereby providing a way of applying IFC techniques to database access control. Second, we develop a security monitor based on a novel combination of information-flow tracking with concepts from database theory like disclosure lattices and query determinacy.

Structure. We overview the relationships between DBAC and IFC in Section 7.1. In Section 7.2, we introduce our setting. We present WhileSql in Section 7.3 and our security model in Section 7.4. In Section 7.5, we present our reduction from DBAC to IFC, whereas we present our IFC mechanism for securing database-backed applications in Section 7.6. We discuss related work in Section 7.7 and draw conclusions in Section 7.8. We present some technical details in Section 7.9. The proofs of all our results are given in Appendix D.

7.1 Overview

Database access control and information-flow control have a similar objective: to protect the confidentiality and integrity of sensitive information. Yet they typically do so for different components of a system: databases and applications, respectively. Furthermore, the enforcement techniques used in these two settings are different. IFC employs techniques ranging from security type systems [135, 163] to multi-execution [64], whereas DBAC mechanisms mostly rely on query rewriting [110, 131, 150, 165].

Unfortunately, these mechanisms do not compose naturally. Even if both DBAC and IFC are in place, application-level information, such as a sensitive context of a function call that executes a query, is lost at the time of DBAC enforcement. Conversely, database-level information, such as fine-grained table-level security labels, is lost at the time of IFC enforcement, when information from the database is manipulated by the application. This poses a challenge of tackling insecurities arising at component boundaries.

More fundamentally, DBAC and IFC build on different foundations. DBAC’s foundations lie in database theory and access control, whereas IFC builds on programming language theory. Hence, the security properties studied in these two settings are quite different.

Despite their differences, IFC and DBAC are starting to overlap in their scopes. Various IFC approaches [24, 49, 50, 57, 59, 111, 143, 167] have taken databases into account. These approaches extend IFC foundations with database models and apply standard IFC techniques, such as security type systems [57, 141], symbolic execution [49] or faceted values [167], to track how information flows through databases with the goal of providing end-to-end security guarantees. In contrast, DBAC
mechanisms for databases that support features like triggers or stored procedures [140,149] face problems like preventing implicit flows, which are typical of the IFC setting.

Surprisingly, the relationship between DBAC and IFC remains largely unexplored. This gap hinders the development of both lines of research and limits the integration and re-usability of existing results and mechanisms. For instance, IFC solutions that support databases [57,59,141,167] consider only simplified database models that ignore advanced features, which have been studied in DBAC [140,149] and can be exploited by attackers. In contrast, DBAC mechanisms could benefit from existing research in IFC.

7.2 Problem Setting

This section presents the system and attacker models associated with DBAC and IFC. We motivate these models with concrete examples and discuss their security requirements.

7.2.1 System and Attacker Model

Figure 7.1 depicts our system model. The system consists of users that interact with the database either directly, by issuing SQL commands, or indirectly, by interacting with programs (e.g., web applications) that, in turn, issue commands to the database. We refer to users directly interacting with the database as internal users, and we call external users those whose interaction with the database is mediated by a program. For simplicity, we assume that each internal and external user is uniquely associated with a user account that is used to retrieve information from the database. Furthermore, we assume that external users execute all their programs using their own user accounts. Users execute commands concurrently, and a scheduler mediates the interaction with the database system.

In our system model, the security policy is defined at the database level using access control policies, which specify the access permissions of each user account for the database tables and views. The database system has a distinguished user, the administrator, who defines the database schema, the database’s integrity constraints, and the database’s initial access control policy. We assume that the users’ commands, the programs, the database schema, and the database configuration, which consists of the security policy, the views, and the triggers in the system, are publicly known.

To accommodate both DBAC and IFC in the same system model, we consider internal and external attackers. An internal attacker is a database user, different from the administrator, who directly issues SQL commands to the database and observes the results of these commands. Namely, an internal attacker is one of the internal users in Figure 7.1. His goal is learning information about the tables and views he cannot read. This reflects the usual attacker model adopted in DBAC (see, for instance, Chapter 6). We call a sequence of commands secure with respect to an internal attacker if the attacker cannot learn sensitive information by observing the results of his own commands.

We remark that our system model does not include an access control system that enforces the security policy. This allows us to study the security condition, i.e., what it means, semantically, for a sequence of commands to be secure, without focusing on the enforcement mechanism, namely an algorithm that only authorizes commands that satisfy the security condition. We sometimes refer to sequences of commands as programs.

An external attacker is an external user that cannot directly issue commands to the database, since communication between the attacker and the database is mediated by a program. An external attacker, therefore, can only observe the program’s output. He cannot observe the results of the
queries issued by the program unless these results are part of the program’s output. Securing applications with respect to this attacker model involves tracking the propagation of data across the application and database, thus enforcing the security policy in an end-to-end fashion.

The external attacker corresponds to the information-flow setting. As is standard, we say that a program is secure with respect to an external attacker if executing the program does not leak sensitive information to the attacker. For simplicity, we ignore application-level policies for external users. These can easily be modeled using database tables with appropriate security policies.

### 7.2.2 Overview of Security Conditions

By using examples, we illustrate the intuition behind our security conditions, which we formalize in Section 7.4.

**Internal Attackers.** Consider a movie rental database where the customer Bob pays for a subscription to an online media provider like Netflix. Bob is allowed to watch any movie in the table MOVIE if he has a valid subscription. For simplicity, we model the valid subscription as a read permission on MOVIE. When the subscription expires (and the policy changes), Bob should no longer be able to access the movie, even if he has previously streamed a copy of it.

Suppose the database administrator can modify the access control policy of the table MOVIE, and Bob queries the database to watch the movie. The following program should be considered insecure as Bob accesses the table MOVIE after the subscription has expired (we write $u:q$ to denote that the user $u$ executes the command $q$).

\[
\begin{align*}
\text{admin} &: \text{GRANT SELECT ON MOVIE TO Bob} \\
\text{Bob} &: \text{SELECT } \ast \text{ FROM MOVIE} \\
\text{admin} &: \text{REVOKE SELECT ON MOVIE FROM Bob} \\
\text{Bob} &: \text{SELECT } \ast \text{ FROM MOVIE}
\end{align*}
\]

The building blocks for our security condition for internal attackers are the attacker knowledge, the security policy, and the notion of an epoch. The attacker knowledge describes the set of database states consistent with the attacker’s observations, e.g., the results of database queries issued by the attacker. The security policy determines the attacker’s initial knowledge, namely the set of databases the attacker initially considers as possible. An epoch represents a portion of the program’s execution. Our security condition for internal attackers ensures that the attacker’s knowledge remains constant inside each epoch, thus satisfying the current security policy.

In our example, epochs are determined by the commands executed by Bob. In particular, the above program involves two epochs corresponding respectively to the commands (2) and (4). In the first epoch, Bob is authorized to read the table MOVIE. Hence, his initial knowledge consists of the content of the table MOVIE. Therefore, after the SELECT command, Bob’s knowledge does not change as he already knows the content of table MOVIE. In the second epoch, our condition resets Bob’s initial knowledge and forgets any information that Bob may have learned in the past. Now, Bob is not authorized to read the table MOVIE. Hence, he considers as possible any value for the table MOVIE. However, after the SELECT command, Bob learns the content of MOVIE, thus refining his knowledge and violating the current policy. Our security condition therefore considers the program as insecure.

Our security condition for internal attackers characterizes security with respect to the database state at the beginning of each epoch. This is in line with current DBAC requirements [131, 165] that interpret the security policy with respect to the current state of the database.

**External Attackers.** A classical example of an external attacker is a user interacting with a server-side application that communicates with a database through programming language constructs. The goal, here, is to track information across application-database boundaries, ensuring that the user does not learn sensitive information about the initial state of the database. In contrast to internal attackers, an external attacker can only learn information through output statements and he cannot observe the results of the queries. Moreover, the attacker has perfect recall in the sense that he remembers all prior observations.

Consider a web application allowing teaching assistants (TAs) and instructors to grade students’ assignments. The application consists of a server-side application and a database system storing the grades. The application performs a SELECT query to access the students’ grades and then outputs them to a TA.

\[
x := \text{SELECT } \ast \text{ FROM GRADE} \\
\text{out}(\text{TA, } x)
\]
Basic Types

<table>
<thead>
<tr>
<th>(Basic Types)</th>
<th>(Relation Ids)</th>
<th>(View Ids)</th>
<th>(Table Ids)</th>
<th>(Variables)</th>
<th>(Values)</th>
<th>(Formulae)</th>
<th>(Trigger Ids)</th>
<th>(User identifiers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T ∈ T</td>
<td>R ∈ T ∪ V</td>
<td>V ∈ V</td>
<td>T</td>
<td>x ∈ Vars</td>
<td>n ∈ Vals</td>
<td>ϕ ∈ RC</td>
<td>tr ∈ TR</td>
<td>u ∈ UID</td>
</tr>
</tbody>
</table>

Syntax

(User Context) uc := OWNER | INVOKER
(Privileges) p := SELECT ON R | INSERT ON T | DELETE ON T | CREATE VIEW | CREATE TRIGGER ON T
(Actions) a := INSERT e₁,...,eₙ INTO T | DELETE e₁,...,eₙ FROM T | GRANT p TO u | GRANT p TO u WITH GRANT OPTION | REVOKE p FROM u
(SQL commands) q := u | SELECT ϕ | ADD USER u | CREATE VIEW V : SELECT ϕ AS uc | CREATE TRIGGER tr ON T AFTER (INS|DEL) IF ϕ DO a AS uc
(Expression) e := n | x | ¬e | e₁ ⊕ e₂
(Statements) c := e | x ← q | x := e | out(u,e) | if e then c₁ else c₂ | while e do c | c₁ ; c₂

Figure 7.2: WhileSql’s syntax.

Assume now that while executing this application, the administrator changes the policy by revoking the read permission on GRADE.

GRANT SELECT ON GRADE TO TA
REVOKE SELECT ON GRADE FROM TA

Finally, assume that the interleaving produced by the executions of the two programs is as follows. The policy initially allows the TA to learn the grades. Then, the application retrieves the students’ grades using the SELECT query, and, immediately after that, the administrator executes the REVOKE command, thereby disallowing the TA to learn the grades. The grades are then sent to the TA through an output command. Although the query was performed at a time when it was allowed by the policy, the TA learns the grade when this is no longer allowed. Hence, this program should be considered insecure.

We propose a security condition ensuring that information release is always done in accordance with the current security policy. Since information release happens through output statements, a program is considered secure if whenever the attacker learns some information, then (i) the attacker already learned that information in the past when this was allowed by the policy, or (ii) the current security policy allows the attacker to learn that information. In this scenario, the TA learns the grade when this is no longer allowed by the policy and, since he did not learn the grade in the past, the program is considered insecure. In contrast to internal attackers, the security condition neither resets the attacker’s knowledge nor forgets any information that the attacker may have learned in the past. Furthermore, as is common in the IFC setting [24, 49, 56, 143, 167], we interpret the security policy with respect to the initial state of the database.

7.3 WhileSql

WhileSql is a simple language that captures the main features of both programming languages extended with querying constructs and procedural extensions of the SQL standard, such as Oracle’s PL/SQL or Microsoft Transact-SQL. At the same time, it simplifies some subtle aspects of their semantics, while still capturing the main security-critical features.

7.3.1 Syntax

Figure 7.2 depicts WhileSql’s syntax. Let T, V, and TR be three countably infinite sets representing table identifiers, view identifiers, and trigger identifiers. Furthermore, let Vars and Vals be countably infinite sets of variables and values. We assume that all these sets are pairwise disjoint.

As shown in Figure 7.2, a WhileSql program is an imperative program extended with querying capabilities, i.e., statements of the form x ← q. A statement x ← q executes the SQL command q and assigns its result to the variable x. WhileSql supports SQL’s core features, such as SELECT, INSERT, DELETE, GRANT, and REVOKE commands, as well as advanced database features such as triggers and views. Additionally, WhileSql programs support assignments and standard control flow statements. WhileSql also supports out(u,e) statements to output the value of the expression e to the user u.

For simplicity, we assume that all expressions are well-typed and all SQL statements refer either to tables in the database schema or to previously created views.

WhileSql builds on top of the database operational semantics from Chapter 5. Hence, it supports the same fragment of SQL supported by the database semantics. We now recall the restrictions and simplifications inherited from our operational semantics. For SELECT commands, instead of using
SQL, we rely on the relational calculus. Moreover, we support only `INSERT` and `DELETE` commands that explicitly identify the tuple to be inserted or deleted. More complex `INSERT` and `DELETE` commands, as well as `UPDATE` commands, can be simulated by combining `SELECT` queries with `INSERT` and `DELETE` commands. Finally, we support only triggers that are executed in response to `INSERT` and `DELETE` commands. We assume that a trigger’s body has the form `IF φ DO a AS uc`, where `φ` is a boolean query, `a` is an `INSERT`, `DELETE`, `GRANT`, or `REVOKE` command, and `uc` specifies whether the trigger is executed under the owner’s or the activator’s privileges.

Each SQL command either returns the query result or an error message `em ∈ EM`. Error messages indicate whether queries (or triggers) violate security constraints, like a query that is not allowed by the current security policy, or integrity constraints, such as an `INSERT` statement that violates a primary key constraint. Note that error messages are values, i.e., `EM ⊆ Vals`.

### 7.3.2 Local Semantics

Here, we define the semantics of a `WhileSql` program executed in isolation. A `WhileSql` program is defined with respect to a `system configuration` `M = ⟨D, Γ⟩`, where `D = ⟨Σ, dom⟩` is a database schema and `Γ` is a set of integrity constraints. We assume that `dom ⊆ Vals` and that only values in `dom` are used to construct queries. For simplicity, we fix a configuration `M = ⟨D, Γ⟩` for the rest of the section.

**Databases.** `WhileSql` reuses the database model and the notion of system and runtime states from Chapter 5. Here, we recall only the main concepts that we use in the rest of the chapter. We refer the reader to Chapter 5 for more details on security policies and our database model.

A `security policy` is a finite set of `GRANT` statements. Given a policy `sec` and a user `u`, we denote by `auth(sec, u)` the set of all tables and views with the owner’s privileges that `u` is authorized to read according to the `GRANT` statements in `sec`. A `system state` is a tuple `<db, U, sec, T, V>` where `db` is a database state, `U ⊆ UID` is a finite set of users, `T` is a finite set of triggers, `V` is a finite set of views, and `sec` is a security policy. Note that we lift `auth` from policies to system states, i.e., `auth(db, U, sec, T, V, u) = auth(sec, u)`. A `context ctx` describes the database’s history, the scheduled triggers that must be executed, and how to modify the database’s state in case a roll-back occurs. A `runtime state` is a tuple `<s, ctx>` where `s` is a state system and `ctx` is a context. The set of all runtime states is denoted by `Ω_M` and we denote by `ε` the empty context. In the following, we use `s` to refer to both system states and runtime states whenever this is clear from the context, and we use the notation `<s, ctx>` otherwise.

**Memories and Configurations.** A `memory m ∈ Mem` is a partial function mapping variables to values, i.e., `Mem : Vars → Vals`. A `local configuration` `<c, m, ⟨s, ctx⟩>` consists of a command `c ∈ Com`, a memory `m ∈ Mem`, and a runtime state `<s, ctx>` ∈ `Ω_M`. A local configuration is `initial` iff its context `ctx` is `ε` and the database state `s` is an initial database state as defined in Chapter 5. We denote by `Conf` the set of all configurations.

**Users.** Let `UID` be a countably infinite set representing all user identifiers. In addition to users in `UID`, we add a designated user `public` that can observe only public events (i.e., changes to the public memories and configurations). For each user `u ∈ UID`, we add a user `db(u)` that represents the internal user corresponding to `u`. Hence, the set `U` of all users is `UID ∪ {db(u) | u ∈ UID} ∪ {public}`. We define a partial order `≤` over `U` by `u_1 ≤ u_2` iff `u_1 = public, u_1 = u_2`, or `u_1 = db(u_2)`.

**Observations.** We model program-level and database-level observations. The former are generated using `out` statements while the latter are generated by queries. A program-level observation is a pair `<u, o>`, where `u ∈ UID` is a user identifier and `o ∈ Vals` is a value. A database-level observation, instead, is a 4-tuple `<u, q, o, τ>`, where `u ∈ {db(u) | u ∈ UID} ∪ {public}`, `q` is either an SQL command or a trigger, `o` is either a value or an SQL command, and `τ` is a (possibly empty) sequence of database-level observations. We use `τ` to represent the observations caused by triggers executed in response to an SQL command. We write `<u, q, o>` instead of `<u, q, o, τ>` if `τ = ε`. We denote by `Obs` the set of all observations.

**Semantics.** Given a user `u ∈ UID`, the relation `→_u ⊆ (Com × Mem × Ω_M) × Obs × (Com × Mem × Ω_M)` formalizes the small-step local operational semantics of `WhileSql` programs executed by `u`. A `run` `r` is an alternating sequence of configurations and observations that starts with an initial configuration and respects the rules defining `→_u`. Given a run `r`, we denote by `r^i`, where `i ∈ N`, the run obtained by truncating `r` at the `i`-th state. A `trace` is an element of `Obs^*`. The trace `τ` associated to a run `r`, denoted by `trace(r)`, is obtained by concatenating all observations in the run.

As noted above, the operational semantics of SQL statements relies on the operational semantics given in Chapter 5. We use the function `[q]((s, ctx), u)` (defined in Section 7.9.1) to connect `WhileSql`’s operational semantics with the database operational semantics from Chapter 5. The function `[q]((s, ctx), u)` takes as input an SQL statement `q`, a runtime state `<s, ctx>` ∈ `Ω_M`, and the user `u ∈ UID` executing the command, and it returns a tuple `<(s, ctx)', r, em, τ>`, where `<(s', ctx')> ∈ Ω_M`
is the new runtime state, \( r \) is \( q \)'s result, \( em \) is an error message, and \( \tau \) is a trace of database-level observations produced by triggers executed in response to \( q \). We also write \([e](m)\) to denote the evaluation of an expression \( e \) in memory \( m \). It is always clear from context if \([\cdot](\cdot)\) refers to query or expression evaluation.

Figure 7.3 depicts the rules defining WhileSQL’s local semantics. Most of the rules are standard. The only non-standard rules are E-QUERYOK and E-QUERYEX, which regulate the execution of SQL commands. The rule E-QUERYOK models the successful execution of SQL commands. It first replaces the free variables in the query with their actual values. Afterwards, it executes the query, which produces the database-level observation associated with the query, and stores the query result in the memory. The rule relies on the function \( \text{Usr}(u, q) \), which takes as input a user \( u \in \text{UID} \) and a query and returns the user \( db(u) \) if \( q \) is a SELECT, INSERT, or DELETE command, and public otherwise (since queries that modify the database’s configuration produce observations that are visible to all users). The rule E-QUERYEX, instead, models the failed execution of a query. The rule executes the query, retrieves the error message, and stores it in the memory.

7.3.3 Global Semantics

To model realistic scenarios, where attackers and honest users each run their own programs which may access a common database, we assume that programs do not share memory, whereas the database is shared. We now present a global semantics capturing the parallel execution of WhileSQL programs.

Schedulers. We model a scheduler as an infinite sequence of natural numbers \( S \in \mathbb{N}^\omega \). In the global semantics, we use the scheduler to determine which program has to be executed at each point in the execution.

Global Configurations. We denote the set of commands together with the executing user by \( \text{Com}_{\text{UID}} = \text{UID} \times \text{Com} \) and the set of pairs of users and memories as \( \text{Mem}_{\text{UID}} = \text{UID} \times \text{Mem} \). To model a system state where multiple WhileSQL programs run in parallel and share a common database, we introduce global configurations. A global configuration is a tuple \( (C, M, (s, ctx), S) \in \text{GConf} \), where \( C \in \text{Com}_{\text{UID}} \) is a sequence of WhileSQL programs with the executing users, \( M \in \text{Mem}_{\text{UID}} \) is a sequence of memories (one per program in \( C \)), \( (s, ctx) \in \Omega_M \) is the runtime state of the shared database, and \( S \) is a scheduler formalizing the interleaving of the programs in \( C \). We consider only configurations \( (C, M, s, S) \) such that \( |C| = |M| \) and for all \( 1 \leq i \leq |C| \), \( C(i) = (u, c) \), \( M(i) = (u', m) \), and \( u \neq u' \). Furthermore, a global state is a pair \( (M, s) \), where \( M \in \text{Mem}_{\text{UID}} \) and \( s \) is a system state.

Semantics. The relation \( \rightarrow \subseteq \text{GConf} \times \text{Obs} \times \text{GConf} \), shown in Figure 7.4, formalizes the global operational semantics of a database system that runs multiple WhileSQL programs in parallel.
Given a global configuration \((C, M, s, S)\), the global operational semantics uses the scheduler \(S\) to
select which of the programs in \(C\) to execute. This is done by extracting the first number \(n'\) from
the scheduler \(S\) and identifying the associated program \((u, c)\) and memory \((u, m)\) in \(C\) and \(M\) respectively.
The rule M-Eval-Step identifies the WhileSql program that should be executed according to
the scheduler, it executes one step of the local semantics, and it updates the global state accordingly.
The rule M-Eval-End, instead, removes the terminated programs from the global configuration.
Given a run \(r\), we denote by \(\text{conf}(r)\) the global configuration in the last state in the run. Furthermore,
\(\text{trace}(r)\) denotes the trace associated with the run \(r\), and \(\text{db}(r)\) denotes the database state in
the global configuration \(\text{conf}(r)\).

### 7.4 Security Model

Here we formalize the security conditions for internal and external attackers as presented in
Section 7.2. For consistency, we define our security conditions over general
WhileSql commands and learn information from the database iff the current database policy authorizes them
to do so. In contrast, the security condition for external attackers adopts an end-to-end interpretation
of security policies that accounts for both the database and the application. In particular, if the
program issues a query when authorized by the policy but subsequently outputs the result of that
query when this is forbidden by the policy, our condition rejects the program as insecure.

#### 7.4.1 Preliminaries

Before presenting our security conditions, we introduce some notation.

**Equivalence of memories.** Given a user \(u\), two sequences of memories \(M_1, M_2 \in \text{Mem}_{UID}\) are
\(u\)-equivalent, written \(M_1 \approx_u M_2\), iff \(|M_1| = |M_2|\) and for all \(1 \leq i \leq |M_1|,\ u_1^i = u_2^i\) and if \(u_1^1 = u,\) then \(m_1^i = m_2^i\), where \(M_1(i) = \langle u_1^i, m_1^i \rangle\) and \(M_2(i) = \langle u_2^i, m_2^i \rangle\).

**Equivalence of databases.** Given two database states \(db\) and \(db'\) and a set \(S\) of tables and views,
we say that \(db\) and \(db'\) are \(S\)-equivalent, written \(db \approx_S db'\), iff the content of all tables and views
(with owner’s privileges) in \(S\) is the same in \(db\) and \(db'\). For the indistinguishability notions between
system states, we reuse data-indistinguishability from Chapter 6. Given a user \(u\), two system states
\(s = \langle db, U, sec, T, V \rangle\) and \(s' = \langle db', U', sec', T', V' \rangle\) are \(u\)-equivalent, written \(s \approx_u s'\), iff \((1)\ U = U',\)
(2) \(sec = sec'\), (3) \(T = T'\), (4) \(V = V'\), and (5) \(db \approx_{\text{auth}(sec, u)} db'\).

**Equivalence of global states.** Two global states \((M, s)\) and \((M', s')\) are \(u\)-equivalent for a user \(u \in \text{UID}\),
written \((M, s) \approx_u (M', s')\), iff \(M \approx_u M'\) and \(s \approx_u s'\). Furthermore, we write \((M, s) \approx_{\text{db}(u)} (M', s')\) iff \((M, s) \approx_u (M', s')\). We denote by \([[(M, s)]_u]\) the set of all global states \((M', s')\)
such that \((M', s') \approx_u (M, s)\).

**Equivalence of traces.** The projection of a trace \(\tau\) for a user \(u\), written \(\tau|_u\), is as follows: \(\epsilon_u = \epsilon,\)
\((\langle u', o, \tau' \rangle)|_u = \langle u', o, \tau' \rangle\) if \(u' \leq_u u\), \((\langle u', q, o, \tau'' \rangle - \tau')|_u = \langle u', q, o, \tau'' \rangle - \tau'|_u\) if \(u' \leq_u u\) or there is a \(u'' \in\) \(\text{users}(\tau')\) such that \(u'' \leq_u u\) and \((\langle u', o, \tau' \rangle)|_u = \langle u', o, \tau'' \rangle - \tau'|_u\), otherwise,
where \(\text{users}(\tau)\) is the set of all users appearing in the observations in \(\tau\). Finally, we say that two
traces \(\tau_1\) and \(\tau_2\) are \(u\)-progress-insensitive-equivalent, for a user \(u\), written \(\tau_1 \sim_u \tau_2\), iff \(\tau_1|_u \approx \tau_2|_u\) or \(\tau_2|_u \approx \tau_1|_u\).

**Knowledge.** Following [20], we semantically characterize what an attacker can infer from an
execution as the set of global states that are consistent with an observed trace.
Abusing notation, we report only

\[ (\langle M_0, s_0 \rangle, S, \tau) \] of a user \( u \) for a global state \( \langle M_0, s_0 \rangle \), a
sequence of programs \( C \), a scheduler \( S \), and a trace \( \tau \) is \( \{ (M, s) \mid s \simeq_{u} s_0 \land M \simeq_{u} M_0 \land \forall ctx', \tau', C', M', s', S' \mapsto \tau \simeq_{u} \tau' \} \).

The knowledge of an attacker \( u \) is the set of initial global states that the attacker cannot dis-
tinguish based on the observation of the trace \( \tau_{nu} \). Thus, a smaller knowledge set indicates that \( u \) has a more precise knowledge. The definition of the attacker’s knowledge is progress-insensitive, that is we ignore information leaks due to the progress of computation, e.g., program divergence and termination [19]. We achieve this by requiring that any execution starting from a \( u \)-equivalent global state only produces traces \( \tau' \) that are progress-insensitive equivalent with the original trace \( \tau \).

7.4.2 Internal Attackers

To properly account for internal attackers, we introduce the notion of epoch, which represents a portion of a run. An internal attacker forgets the accumulated knowledge each time a new epoch starts. We use epochs to model the non-permanent release of information as is common in database access control. Intuitively, by forgetting all accumulated knowledge, we prevent any release of information whenever that information is considered sensitive, although the same information might have been non-sensitive in the past or might become non-sensitive in the future.

**Epochs.** We formalize epochs using an **epoch predicate**, which is a relation over \( \text{GIConf} \times \text{GIConf} \) that, for each pair of (consecutive) global configuration \( g_c \) and \( g_c' \), specifies whether transitioning from \( g_c \) to \( g_c' \) starts a new epoch. Given an epoch predicate \( E \) and a (terminating) run \( r \), we denote by \( \{E[r]\} \) the set of epochs defined by \( E \) over the run \( r \). Namely, \( \{E[r]\}(C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n + 1) = \{C_1 \rightarrow 1 C_{j+1} \} \cup \{E[r](C_{j+1} \rightarrow 1 C_{j+1} \rightarrow \ldots \rightarrow C_{n+1}) \mid 1 \leq j \leq n \) is the smallest natural number such that \( (C_j, C_{j+1}) \in E \) (if such number exists), and \( \{E[r]\}(C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_{n+1}) = \{C_1 \rightarrow \ldots \rightarrow C_{n+1} \} \) if there is no \( j \in \{1, \ldots, n\} \) such that \( (C_j, C_{j+1}) \in E \).

We also introduce selection predicates. These identify a subset of the epochs in a run that satisfies a given condition. Formally, a **selection predicate** is a set of epochs. Given an epoch predicate \( E \), a (terminating) run \( r \), and a selection predicate \( S \), \( \{E, S\}[r] \) denotes the set \( \{E[r]\} \cap S \).

To illustrate the concept of epochs, we now introduce epoch and selection predicates capturing the attacker model of Chapter 6, where users do not know the commands executed by other users. We capture this through **user-based epochs**, which represent portions of a run where a user continuously interacted with the system, without interleaved commands executed by other users. Note that this is the notion of epoch used in Section 7.2. We model this using the epoch predicate \( E_{user} \), which splits runs into “user sessions”, and the selection predicate \( S_{user}^{E} \), which selects only user’s sessions. Formally, the epoch predicate \( E_{user} \) is defined as follows: \( [(C, M, s, n, S), (C', M', s', n', S')] \in E_{user} \) if \( C(1 + (n \mod |C|)) = (u, c), C'(1 + (n' \mod |C'|)) = (u', c') \), and \( u \neq u' \). Additionally, the selection predicate \( S_{user}^{E} \), for a user \( u \), is defined as follows: \( [(C_1, M_1, s_1, n_1, S_1), \ldots, (C_k, M_k, s_k, n_k, S_k)] \in S_{user}^{E} \) if for all \( 1 \leq i \leq k \), there is a command \( c \) such that \( C_i(1 + (n_i \mod |C_i|)) = (u, c) \).

**Security Condition.** We now define our security condition for internal attackers. Intuitively, this notion corresponds to an attacker that “forgets” all knowledge whenever the epoch changes. Otherwise, an attacker’s knowledge remains constant inside each epoch, thus satisfying the security policy at the beginning of the epoch.

**Definition 7.2.** A sequence of programs \( C \in \text{Comp}_{1JD}^{\tau} \) is **secure** with respect to an **internal attacker** \( u \) in \( UID \) for a scheduler \( S \), a system state \( s_0 \), a sequence of memories \( M_0 \in \text{Mem}_{1JD}^{\tau} \), an epoch predicate \( E \), and a selection predicate \( S \) if whenever \( r = (\langle C, M_0, (s_0, c), S \rangle \rightarrow^{\tau} \langle C_n, M_n, (s_n, ctx_n), S_n \rangle) \) for all epochs \( (C_1, M_1, (s_1, ctx_1), S_1) \rightarrow^{\tau} \langle C_j, M_j, (s_j, ctx_j), S_j \rangle \) in \( \{E, S\}[r] \), then \( K_{\alpha^{\alpha}(\langle M_i, s_i \rangle, C_i, S_i, \tau)} = [\langle M_i, s_i \rangle]_{\alpha^{\alpha}} \).

**Example 7.1.** Consider (a variant of) the program from Section 7.2 where the commands are scheduled as follows:

\begin{align*}
\text{GRANT SELECT ON MOVIE TO Bob} & \quad (1) \\
x := \text{SELECT} & \exists x. \text{MOVIE}(x) \quad (2) \\
\text{REVOKE SELECT ON MOVIE FROM Bob} & \quad (3) \\
\text{y := SELECT} & \exists y. \text{MOVIE}(y) \quad (4)
\end{align*}

Suppose Bob is an attacker who wants to learn whether there are any movies in the table MOVIE. Abusing notation, we report only Bob’s memory and write \( m \) instead of \( \langle \text{Admin}, m_{\text{admin}} \rangle \cdot \langle \text{Bob}, m \rangle \). Furthermore, we denote by \( C_i \) the code at the \( i \)-th step in the execution.
Consider an initial local state \( (m, s) \) such that \( m(x) = \bot \) and the table MOVIE is empty in \( s \). There are two user-based epochs for the user Bob: the former corresponds to the execution of line (2), and the latter corresponds to the execution of line (4). In the first epoch, since the policy allows Bob to learn the data in the table MOVIE, we have \( K_{db(bob)}((m, s_1), C, S, \tau) = \{ (m, s) \} \) where \( m(x) = \bot \) and \( s_1 = s \). Similarly, \( \{(m, s)\}_{db} = \{ (m, s) \} \). The equality holds both before and after the execution of line (2); the execution of the statement in line (2) does not change Bob's knowledge since he can read the table MOVIE. In the second epoch, the security policy prevents Bob from learning the data in the table MOVIE, hence \( \{(m, s_3)\}_{db} = \{ (m, s_3), (m, s_3'), (m, s_3''), \ldots \} \), where \( s_3, s_3', s_3'' \) are all possible database states and \( s_3 = s \). In contrast, after the \( SELECT \) statement in line (4), Bob learns that the table MOVIE is empty. Namely, \( K_{db(bob)}((m, s_3), C, S_3, \tau) = \{ (m, s_3) \} \). Since the two sets are different, the program is rejected as insecure.

### 7.4.3 External Attackers

We now formalize our security condition for external attackers. The condition is based on an attacker-centric view of security. It keeps track of the attacker’s knowledge during a program execution and ensures that the knowledge complies with the current security policy. The security condition is inspired by existing conditions for information-flow in the presence of dynamic policies [18,39].

**Allowed knowledge.** We interpret security policies with respect to the initial global states. The allowed knowledge \( A_{u,sec} \) determines the set of initial global states that a user \( u \) considers possible given a security policy \( sec \). Given a sequence of memories \( M_0 \in \text{Mem}_{uid} \), a system state \( s_0 \approx (db_0, U_0, sec_0, T_0, V_0) \), a security policy \( sec \), and a user \( u \), we define the set \( A_{u,sec}(M_0, s_0) \) as \( \{ (M, s) \mid s \approx_{sec, u} s_0 \wedge M \approx_{sec} M_0 \} \), where \( (db', U', sec', T', V') \approx_{sec, u} (db'', U'', sec'', T'', V'') \) iff \( db' \approx_{auth(\text{sec}, u)} db'' \). We call \( A_{u,sec}(M_0, s_0) \) the allowed knowledge since it represents the knowledge of the initial global state that the user \( u \) is permitted to learn given the policy \( sec \).

**Security Condition.** We now introduce our security condition.

**Definition 7.3.** A sequence of programs \( C \in \text{Com}_{uid} \) is secure with respect to an external attacker \( u \in UID \), a system state \( s_0 \), and a sequence of memories \( M_0 \in \text{Mem}_{uid} \) iff whenever \( r = (C, M_0, (s_0, c), S) \approx_{r} (C', M', (s', c'tx), S') \), then for all \( 1 \leq i \leq \#r \), \( K_u((M_0, s_0), (C, S, trace(r^{-i})) \cap A_{u,sec}(M_0, s_0) \subseteq K_u((M_0, s_0), (C, S, trace(r))) \), where the database in \( r \)'s \((i-1)\)-th configuration is \( (db, U, sec, T, V) \).

In a nutshell, the shell ensures that the attacker’s knowledge after observing \( trace(r') \) is at most as precise as the previous knowledge combined with the allowed knowledge of the execution \( r'^{-1} \), i.e., the knowledge increase resulting from observing \( trace(r') \) is allowed by the current policy.

**Example 7.2.** Consider (a variant of) the program from Section 7.2 and a scheduler \( S \) that interleaves the commands as follows.

\[
\text{GRANT SELECT ON GRADE to Student} \quad (1) \\
\text{x := SELECT } \exists x. \text{GRADE}(x) \quad (2) \\
\text{REVOKE SELECT ON GRADE from Student} \quad (3) \\
\text{out(Student, x)} \quad (4)
\]

The user Student is an attacker who wants to learn whether there are any grades in the table GRADE. We now show that the program is insecure with respect to Definition 7.3. Abusing notation, we report only Student's memory, and we write \( m \) instead of \( \langle \text{Admin}, m_{admin} \rangle \cdot \langle \text{Student}, m \rangle \).

Consider an initial memory \( m_0 \) such that \( m_0(x) = \bot \) and an initial database state \( s_0 \) such that the table GRADE is empty. We write \( K_{\text{Student}}((m_0, s_0), C, S, \tau_0) \) and \( A_{\text{Student},sec}(m_0, s_0) \) to denote, respectively, the attacker’s knowledge and the allowed knowledge after executing the commands up to line (4), where \( 0 \leq i \leq 4 \). Let \( r \) be the program execution. Initially, since the Student has made no observations, we have \( K_{\text{Student}}((m_0, s_0), C, S, \tau_0) = \text{All} \), where \( \text{All} \) is the set of all possible initial database states. Similarly, assuming that the Student is not allowed to access the database initially, we have \( A_{\text{Student},sec}(m_0, s_0) = \text{All} \). The attacker’s knowledge does not change until the output statement is executed, since the observations produced by the policy changes are independent of \( s_0 \).

As a result, the inclusion relation from Definition 7.3 holds trivially for the execution \( r^3 \), since the allowed knowledge set can only make the left-hand side smaller. In particular, we have \( K_{\text{Student}}((m_0, s_0), C, S, \tau_3) = \text{All} \) and \( A_{\text{Student},sec}(m_0, s_0) = \text{All} \) due to the \text{REVOKE} statement in line (3). However, after executing the output statement, the attacker learns that the database was initially empty, i.e., \( K_{\text{Student}}((m_0, s_0), C, S, \tau_4) = \{ (m_0, s_0) \} \). Thus, \( K_{\text{Student}}((m_0, s_0), C, S, trace(r^4)) \cap A_{\text{Student},sec}(m_0, s_0) \not\subseteq K_{\text{Student}}((m_0, s_0), C, S, trace(r^4)) \). Hence, the program is rejected as insecure.
7.4.4 Discussion

The security conditions for internal and external attackers are incomparable. Namely, there are programs that are secure for one condition and insecure for the other one. We now illustrate these differences using examples.

Consider a simple variation of Example 7.1, where each SQL statement is immediately followed by an output statement printing the query’s result. This program is insecure for internal attackers for the same motivations outlined in Example 7.1. Despite that, the program is secure for external attackers. The second SELECT query would not modify the attacker’s knowledge, since the attacker learned the content of the table MOVIE when this was allowed by the policy.

Consider now a variation of Example 7.2, where the output statement is replaced by the statement if \( x \) then \( \{ \text{out}(\text{Student}, x) ; y \leftarrow \text{INSERT} 1 \text{ INTO} R \} \) else \( \{ \text{out}(\text{Student}, x) \} \). This program is insecure for external attackers for the same motivations outlined in Example 7.2. The program is, however, secure according to our condition for internal attacker (instantiated with user-based epochs), even though the INSERT commands allows the internal attacker to learn the value of \( x \). This follows from the fact that the SELECT and the INSERT commands are in different epochs.

7.5 From Database Access Control to Information-flow Control

The security model in Section 7.4 focuses on information-flow control. Given programs \( c_1, \ldots, c_n \) executed by the users \( u_1, \ldots, u_n \) and an attacker \( \text{atk} \), our security conditions can be used to determine whether the parallel execution of \( c_1, \ldots, c_n \) leaks information to \( \text{atk} \). This approach relies on two key assumptions: (1) the users \( u_1, \ldots, u_n \) interacting with the system are known beforehand, and (2) the executed programs \( c_1, \ldots, c_n \) are fixed and known to all users. Neither of these assumptions, however, hold for database access control, where any user can potentially interact with the system and issue arbitrary commands. As a result, neither the users \( u_1, \ldots, u_n \) nor the code of the programs \( c_1, \ldots, c_n \), i.e., the queries, are known beforehand. Furthermore, each user may have only a limited knowledge of other users’ commands. To overcome these restrictions, we now present a reduction that constructs a database access control mechanism starting from an information-flow enforcement mechanism for WhileSQL programs.

7.5.1 Preliminaries

We first introduce some terminology and notation.

**Progress-sensitive security.** To properly capture existing DBAC security notions, we focus on the progress-sensitive versions of our security conditions. The progress-sensitive knowledge \( PK_u((M_0, s_0), C, \tau) \) of a user \( u \in \mathcal{U} \) for a global state \((M_0, s_0)\), programs \( C \), a scheduler \( \mathcal{S} \), and a trace \( \tau \) is \( \{ (M, s) \mid s \simeq_u s_0 \land M \simeq_u M_0 \land \exists\text{ctx}' \cdot \tau', C', M', s', \mathcal{S}' \cdot \langle C, M, (s, \epsilon), \mathcal{S} \rangle \xrightarrow{\text{mem}} \langle C', M', (s', \text{ctx}') \rangle, \mathcal{S}' \land \tau[d_u(a)] = \tau'[d_u(a)] \} \). The progress-sensitive variant of Definition 7.2 (respectively Definition 7.3) is obtained by replacing the knowledge \( K_u \) with progress-sensitive knowledge \( PK_u \).

**Enforcement Mechanisms.** An enforcement mechanism is a function \( \text{ifEnf} \) that takes as inputs a sequence of programs \( C \in \text{Com}_{\mathcal{U}ID} \), a sequence of memories \( M \in \text{Mem}_{\mathcal{U}ID} \), a scheduler \( \mathcal{S} \), a user \( u \), and a system state \( s_0 \) and returns as output a security decision \( d \in \{ \top, \bot \} \), where \( \top \) stands for “secure” and \( \bot \) stands for “insecure”. We say that \( \text{ifEnf} \) is sound for a security condition iff the condition holds whenever \( \text{ifEnf} \) returns \( \top \).

**Database Access Control and Auxiliary Functions.** Following Chapter 5, we define a database access control mechanism as a function \( \text{dbEnf} \) taking as input a system state \( s \), a context \( \text{ctx} \) (representing the past history), an SQL command \( q \in \mathcal{Q} \), and a user \( u \in \mathcal{U} \) and returning a security decision in \( \{ \top, \bot \} \), where \( \top \) stands for “authorized” and \( \bot \) stands for “denied”.

According to the operational semantics defined in Chapter 5, a database access control mechanism can be invoked also during the execution of triggers. We assume that there are (1) a function \( \text{inTrigger} \) that inspects the context \( \text{ctx} \) and determines whether the mechanism has been invoked during the processing of a trigger, and (2) a function \( \text{sessUser} \) extracting from \( \text{ctx} \) the user \( u' \) in whose session the query \( q \) has been executed. These functions can be easily constructed on top of the auxiliary functions given in Chapter 5. Finally, for simplicity, we assume that the database operational semantics always stores in the current context \( \text{ctx} \) the initial system state, and we denote the initial state stored in \( \text{ctx} \) by \( \text{init}(\text{ctx}) \). Observe that the operational semantics in Chapter 5 can be trivially extended to store the initial state in the context, without affecting the results we presented in Chapters 5 and 6.

**Simple, query-only programs.** A query-only program is a program that contains only sequences of database queries that do not contain free program variables, i.e., statements of the form \( x \leftarrow q \) such that \( \text{vars}(q) = \emptyset \), or \( \text{skip} \) statements. Furthermore, a program is simple in case it contains exactly one statement. Hence, a simple query-only program is either \( \text{skip} \) or \( x \leftarrow q \). Finally, given
a sequence of simple query-only programs, we denote by \( \#_{\text{qs}}(C) \) the number of statements in \( C \) that are not skip, i.e., the number of query statements in \( C \).

**Additional definitions on states and configurations.** We say that a global configuration \( \langle C, M, s, S \rangle \) is \( (E, S) \)-single-epoch, for an epoch predicate \( E \) and a selection predicate \( S \), if \( [E, S](r) = \{ r \} \), where \( r \) is the longest run obtained by starting from \( (C, M, s, S) \).

We say that a global state \( \langle M, (s, ctx) \rangle \) is reachable iff there is a context \( ctx' \) such that \( (s, ctx') \) is reachable in the LTS defined by the database semantics in Chapter 5, \( \text{triggers}(ctx) = \epsilon \), and \( \text{triggers}(ctx') = \epsilon \).

**Safe programs.** Let \( C \) be a sequence of simple, query-only programs, \( (M, s) \) be a global state, and \( S \) be a scheduler. We say that \( C \) is safe for \( (M, s) \) and \( S \) iff \( (C, M, (s, \epsilon), S) \nrightarrow^* (\epsilon, \epsilon, (s', ctx'), S) \).

Furthermore, we say that \( C \) is safe for \( S \) and a set of global states \( S \) iff it is safe for any \( (M, s) \) in \( S \).

### 7.5.2 Reduction

We now present our reduction that builds a DBAC mechanism \( db\text{Enf} \) starting from an enforcement mechanism \( if\text{Enf} \). In the following, we represent commands as pairs \( \langle u, q \rangle \), where \( u \in UID \) is the user executing the command and \( q \) is the executed query. Furthermore, we denote by \( \langle u, q \rangle \) the query that is given as input to \( db\text{Enf} \). In a nutshell, the reduction extracts from the context \( ctx \) all commands \( (u_1, q_1), \ldots, (u_n, q_n) \) that have already been executed and the initial system state \( s_0 \). We associate to each command \( \langle u, q \rangle \) a WHILESQL program \( c_1 \). Furthermore, we construct the program \( c \) associated with the current query \( \langle u, q \rangle \). Then, \( db\text{Enf} \) authorizes \( \langle u, q \rangle \) iff \( (1) \) \( if\text{Enf} \) says that the programs \( c_1, \ldots, c_n \) are secure for the internal user \( u \), the initial state \( s_0 \), and the memories \( \langle u_1, m_{u_1} \rangle, \ldots, \langle u_n, m_{u_n} \rangle(\langle u, m \rangle) \), where \( \text{dbEnf}(m\langle x \rangle) = 0 \) for all \( x \in Vars \), and \( (2) \) the scheduler does not result in producing a security exception.

#### Reduction 7.1

Let \( s, ctx, q, \) and \( u \) be respectively the system state, the context, the query, and the user given as input to the DBAC mechanism. From the context \( ctx \), we extract the sequence \( L \) of all the commands that have been executed from the start of the interaction. Note that \( L \) contains only the commands directly issued by the users, not the commands executed as part of triggers. We represent each command as a pair \( \langle u, q \rangle \), where \( u \) is the user executing the command and \( q \) is the SQL command itself. Let \( L' \) be the sequence of commands obtained by conditionally appending to \( L \) the current query, i.e., \( L' = L \cup \langle u, q \rangle \) if \( \text{~inTrigger}(ctx) \) and \( L' = L \) otherwise. We now construct a WHILESQL program \( c_1 \in Com_{UID} \) and a memory \( m_i \in Mem_{UID} \) for each query in \( L' \): for each \( 1 \leq i \leq |L'| \), the program \( c_i \) is \( \langle u_i, s_i \rangle \), where \( s_i(i) = \langle u_i, q_i \rangle \), \( x_i \) is a fresh variable, and the statement \( s_i \) is \( x_i \leftarrow q_i \) in case \( (1) \) \( i = |L'| \) and \( \text{~inTrigger}(ctx) \), or \( (2) \) the \( i \)-th query in the history has been successfully executed (i.e., without security exceptions occurring when executing the command or the triggers executed in response to the command) and \( s_i = \text{skip} \) otherwise, and the memory \( m_i \) is \( \langle u_i, Ax \in Vars, 0 \rangle \). The scheduler \( S \) is as follows: \( S = 0^\omega \). The access control mechanism \( db\text{Enf}(s, ctx, q, u) \) returns \( \top \) iff \( (1) \) \( if\text{Enf}(c_1, \ldots, c_n) \) is \( \top \), \( \text{sessUser}(ctx, init(ctx)) = \top \), and \( (2) \) \( \langle c_1, \ldots, c_n, m_1, \ldots, m_{|L'|}, (\text{init}(ctx), \epsilon), S \rangle \nrightarrow^* (\epsilon, \epsilon, (s', ctx'), S') \) and the last observation in \( \tau \) does not contain security exceptions.

To illustrate, consider an history consisting of two queries \( \langle u_1, q_1 \rangle \) and \( \langle u_2, q_2 \rangle \), where the former is authorized and the latter is not authorized, a system state \( s \), and a context \( ctx \) such that \( \text{inTrigger}(ctx) = \bot \). Furthermore, assume that the user \( u \) issues the query \( q \). Our reduction first constructs the programs \( c_1 = \langle u_1, x_1 \leftarrow q_1 \rangle, c_2 = \langle u_2, \text{skip} \rangle \), and \( c = \langle u, x \leftarrow q \rangle \). Then, it authorizes \( \langle u, q \rangle \) iff \( (1) \) \( if\text{Enf} \) says that the sequential execution of the programs \( c_1, c_2 \) is secure for \( u \) given the memories \( \langle u_1, m_{u_1} \rangle, \langle u_2, m_{u_2} \rangle, \text{and} \) the state \( \text{init}(ctx) \), and \( (2) \) the last query (or any of the associated triggers) does not throw security exceptions. The second condition ensures that the WHILESQL’s semantics, which directly embeds the checks on database integrity, agrees with the database semantics from Chapter 5, which does not contain these checks.

### 7.5.3 Security Analysis

Here, we connect our security condition for internal attackers to the data confidentiality notion from Chapter 6.

**Stability.** A key property for enforcement mechanisms is stability. Informally, an enforcement mechanism \( if\text{Enf} \) is u-stable iff \( if\text{Enf} \)’s security decision depends only on the traces produced by the user-based epochs associated with the user \( u \) but it does not depend on the last statement’s result. Stability ensures that \( if\text{Enf} \)’s security decision does not leak information.

Two runs \( r_1 \) and \( r_2 \) are u-epoch-equivalent, where \( u \in UID \), iff there is a bijection \( \mu : [E^{user}, S^{user}][r_1] \rightarrow [E^{user}, S^{user}][r_2] \) such that \( (1) \) \( \mu \) preserves the ordering of epochs, and \( (2) \) for all \( e_1 \in [E^{user}, S^{user}][r_1] \), the initial configurations in \( e_1 \) and \( \mu(e_1) \) are indistinguishable for \( u \) and the
traces in $C_1$ and $\mu(e_1)$ are the same. We say that an enforcement mechanism $\text{ifEnf}$ is $u$-stable iff $\text{ifEnf}(C, M, S, u, s) = \text{ifEnf}(C', M', s', u, s')$ whenever: (a) $C$ terminates starting from state $\langle M, s \rangle$ and scheduler $S$ and produces run $r_1$ iff $C'$ terminates starting from state $\langle M', s' \rangle$ and scheduler $S'$ and produces run $r_2$, (b) the last statement executed in $r_1$ and $r_2$ is the same, and (c) if $r_1$ terminates, then $r_1'$ and $r_2'$ are $u$-epoch-equivalent, where $r_1'$ (respectively $r_2'$) is the run obtained from $r_1$ (respectively $r_2$) by replacing the observation produced by the last statement with $\epsilon$. Namely, an enforcement mechanism $\text{ifEnf}$ is $u$-stable iff $\text{ifEnf}$'s security decision depends only on the traces produced by the epochs associated with the user $u$ (but it does not depend on the last statement's result).

**Main Result.** The main result of this section is that a sound and stable enforcement mechanism for internal attackers can be used to construct a provably secure DBAC mechanism with respect to the data confidentiality property and the attacker model from Chapter 6. We prove Theorem 7.1 in Appendix D.

**Theorem 7.1.** Let $\text{ifEnf}$ be an enforcement mechanism for WHILESQL programs and $\text{dbEnf}$ be the mechanism constructed using Reduction 7.1. If for all $u \in \text{UID}$, (1) $\text{ifEnf}$ is sound for the progress-sensitive variant of Definition 7.2 given $E^\text{user}$ and $S^\text{user}_u$, and (2) $\text{ifEnf}$ is $u$-stable, then $\text{dbEnf}$ provides data confidentiality with respect to the attacker model from Chapter 6.

We remark that Theorem 7.1 presents a way of constructing a provably secure DBAC mechanism by re-using existing IFC techniques for internal attackers; it is related neither to the security condition for external attackers (Definition 7.3) nor to the security monitor from Section 7.6. Furthermore, the theorem highlights the importance of stability, which prevents leaks through the IFC's security decision, for the resulting DBAC mechanism's security. In Section 7.5.5, we show how this result can be extended to IFC mechanisms for external attackers.

### 7.5.4 Extending the results to the Truman Model

The operational semantics from Chapter 5 targets the Non-Truman model for DBAC, as do the SQL standard and many existing database systems. Therefore, both Reduction 7.1 and Theorem 7.1 apply only to the Non-Truman model setting.

We now extend our results to the Truman model. To do so, we develop a technique for deriving a Truman model mechanism from one in the Non-Truman model. Since the Truman model semantics has been defined only for SELECT queries [131, 165] and Truman and Non-Truman semantics agree for boolean queries (see Chapter 3), we focus only on non-boolean SELECT queries.

**Reduction 7.2.** Let $s = \langle db, U, sec, T, V \rangle$, $ctx$, $q$, and $u$ be the system state, the context, the non-boolean SELECT query, and the user given as input to the database access control mechanism respectively. Furthermore, let $NTdbEnf$ be a database access control mechanism under the Non-Truman model semantics. The Truman model mechanism $TdbEnf$ can be constructed as follows: (1) the mechanism executes the query $q = \{\tau | \varphi\}$ over the database $db$ and retrieves the set of tuples $T$ produced as result, and (2) $TdbEnf$ returns the set of tuples $\{t \in T \mid NTdbEnf(s, ctx, SELECT \varphi, t, u) \models \top\}$, namely it returns only the authorized tuples in $q$’s result.

By combining Reductions 7.1 and 7.2, it follows that stability and soundness are enough to ensure that the resulting DBAC mechanism is secure. We prove this result in Appendix D.

**Theorem 7.2.** Let $\text{ifEnf}$ be an enforcement mechanism for WHILESQL programs and $\text{dbEnf}$ be the database access control mechanism for the Truman model semantics constructed using Reductions 7.1 and 7.2. If for all user $u \in \text{UID}$, (1) $\text{ifEnf}$ is sound with respect to the progress-sensitive variant of Definition 7.2 given $E^\text{user}$ and $S^\text{user}_u$, and (2) $\text{ifEnf}$ is $u$-stable, then $\text{dbEnf}$ is secure and sound for the Truman model, i.e., it satisfies Definitions 3.8 and 3.18.

### 7.5.5 From internal attackers to external attackers

Reduction 7.1 constructs a provably secure DBAC mechanism starting from a sound IFC mechanism for internal attackers. Despite that, most of the existing IFC mechanisms target external attackers, not internal ones. These IFC mechanisms cannot, therefore, be used with our construction. To address this, we present a way of using IFC mechanisms that target external attackers to determine the security of simple query-only programs with respect to internal attackers. By combining this with Reduction 7.1, we can, therefore, construct provably secure DBAC mechanisms using sound IFC mechanisms for external attackers as a building block.

Security for internal attackers requires that the attacker’s knowledge remains constant inside each epoch. Lemma 7.1 states that, for the sequential scheduler, simple query-only programs, and user-based epochs, if the program corresponding to each epoch is secure, then so is the original program.
Lemma 7.1. Let $S$ be the sequential scheduler $0^\infty$, $C \in \text{Com}_{\text{UID}}$ be a sequence of simple query-only programs, $s$ be a system state, $M \in \text{Mem}_{\text{UID}}$ be a sequence of memories, and $u \in \text{UID}$ be a user. Furthermore, let $r$ be the longest run obtained starting from $(C, M, (s, c), S)$. If for all epochs $(C_m, M_m, (s, c), S)$ in $[E_{\text{user}}, S_{\text{user}}]^{c_{r_1}}(r)$, (1) the sequence of programs $C_i^{[r_1]}$ is secure with respect to $u$ for $S$, $s$, $M_i^{[r_1]}$, $E_{\text{user}}$, $S_{\text{user}}$ for the progress-sensitive variant of Definition 7.2, and (2) $\mathcal{R}(C_m, M_m, (s, c), S_i^{[r_1]})$ is a $(E_{\text{user}}, S_{\text{user}})$-single-epoch configuration, then $C$ is secure with respect to $u$ for $S$, $s$, $M$, $E_{\text{user}}$, and $S_{\text{user}}$ for the progress-sensitive variant of Definition 7.2, where $\mathcal{R}(C, M, s, S, m)$ denotes the global configuration $(C^m, M^m, s, S)$.

Before connecting security for internal and external attackers, we introduce a transformation on global configurations. The transformation ensures that the runs that start from the transformed configuration (1) leak the same information as those starting from the original configuration, and (2) do not modify the access control policy associated with the original tables and views. This guarantees that the allowed knowledge used in the security condition for external attackers does not contribute to a program's security.

Definition 7.4. Let $(D, \Gamma)$ be a system configuration and $(C, M, (\langle db, U, sec, T, V \rangle, ctx), S)$ be a global configuration such that $\langle db, U, sec, T, V \rangle$ is a database state in $\Omega_{D,T}$ and $C$ is a sequence of simple query-only programs that do not contain ADD USER commands. Furthermore, let $D'$ be the database schema obtained by extending $D$ with a predicate symbol $TMP_R$ for each predicate symbol $R$ in $D$. We denote by $\mathcal{T}(C, M, (\langle db, U, sec, T, V \rangle, ctx), S))$ the global configuration $(C', M', (\langle db', U', sec', T', V' \rangle, ctx'), S')$ over the system configuration $(D', \Gamma)$ defined as follows:

- the sequence of programs $C' \in \text{Com}_{\text{UID}}^{c_{r_1}+1}$ is as follows:
  - $C'(0) = \langle \text{admin}, x_{1,1} \leftarrow \text{REVOKE TMP}_{R_1} \text{ FROM } u_{1,1}, \ldots, x_{1,n_1} \leftarrow \text{REVOKE TMP}_{R_{n_1}} \text{ FROM } u_{1,1}, \ldots, x_{1,m_1} \leftarrow \text{REVOKE TMP}_{R_{m_1}} \text{ FROM } u_{1,1}, \ldots, x_{m_1,n_1} \leftarrow \text{REVOKE TMP}_{R_{m_1,n_1}} \text{ FROM } u_{m_1,n_1} \rangle$, where $u_{1,1}, \ldots, u_{m_1,n_1}$ are all users in $U$ and for each user $u_i$, $\{ R_R_{1,i}, \ldots, R_{R_{n_1,i}} \}$ is the set containing all relation schemas $R_i$ in $D$ such that there is no $\text{GRANT } \langle op, u_i, (\text{SELECT}, R_j), \text{admin} \rangle \in \text{sec}$, where $op \in \{ \oplus, \ominus \}$. i.e., $\{ R_{R_{1,i}}, \ldots, R_{R_{n_1,i}} \} = \{ R_j \mid R_j \in D \land \exists \text{op } \in \{ \oplus, \ominus \}, \langle op, u_i, (\text{SELECT}, R_j), \text{admin} \rangle \in \text{sec} \}$,
  - $C'(i) = (\langle u_{i-1,1}, \text{CREATE TRIGGER } tr \text{ ON } T \text{ AFTER } ev \text{ IF } \varphi \text{ DO GRANT SELECT ON } TMP_R \text{ TO } u \text{ AS } uc \rangle$ if $c_{i-1} = \text{CREATE TRIGGER } tr \text{ ON } T \text{ AFTER } ev \text{ IF } \varphi \text{ DO GRANT SELECT ON } R \text{ TO } u \text{ AS } uc$, where $C(i-1) = (\langle u_{i-1,1}, c_{i-1} \rangle)$,
  - $C'(i) = (\langle u_{i-1,1}, \text{CREATE TRIGGER } tr \text{ ON } T \text{ AFTER } ev \text{ IF } \varphi \text{ DO GRANT SELECT ON } R \text{ TO } u \text{ WITH GRANT OPTION AS uc} \rangle$ if $c_{i-1} = \text{CREATE TRIGGER } tr \text{ ON } T \text{ AFTER } ev \text{ IF } \varphi \text{ DO GRANT SELECT ON } R \text{ TO } u \text{ WITH GRANT OPTION AS uc}$, where $C(i-1) = (\langle u_{i-1,1}, c_{i-1} \rangle)$,
  - $C'(i) = (\langle u_{i-1,1}, \text{CREATE TRIGGER } tr \text{ ON } T \text{ AFTER } ev \text{ IF } \varphi \text{ DO REVOKE SELECT ON } TMP_R \text{ FROM } u \text{ AS uc} \rangle$ if $c_{i-1} = \text{CREATE TRIGGER } tr \text{ ON } T \text{ AFTER } ev \text{ IF } \varphi \text{ DO REVOKE SELECT ON } R \text{ FROM } u \text{ AS uc}$, where $C(i-1) = (\langle u_{i-1,1}, c_{i-1} \rangle)$,
  - $C'(i) = (\langle u_{i-1,1}, \text{GRANT SELECT ON } TMP_R \text{ TO } u \rangle$ if $c_{i-1} = \text{GRANT SELECT ON } R \text{ TO } u$, where $C(i-1) = (\langle u_{i-1,1}, c_{i-1} \rangle)$,
  - $C'(i) = (\langle u_{i-1,1}, \text{GRANT SELECT ON } TMP_R \text{ TO } u \text{ WITH GRANT OPTION} \rangle$ if $c_{i-1} = \text{GRANT SELECT ON } R \text{ TO } u \text{ WITH GRANT OPTION}$, where $C(i-1) = (\langle u_{i-1,1}, c_{i-1} \rangle)$,
  - $C'(i) = (\langle u_{i-1,1}, \text{REVOKE SELECT ON } TMP_R \text{ FROM } u \rangle$ if $c_{i-1} = \text{REVOKE SELECT ON } R \text{ FROM } u$, where $C(i-1) = (\langle u_{i-1,1}, c_{i-1} \rangle)$,
  - $C'(i) = (\langle u_{i-1,1}, \text{CREATE VIEW } V : q \text{ AS } u \text{ CREATE VIEW } TMP_V : q' \text{ AS } u \rangle$ if $c_{i-1} = \text{CREATE VIEW } V : q \text{ AS } u$, where $C(i-1) = (\langle u_{i-1,1}, c_{i-1} \rangle)$ and $q'$ is the query obtained from $q$ by replacing each predicate symbol $R$ with $TMP_R$, and
  - $C'(i) = (\langle i \rangle)$ otherwise,
- the sequence of memories $M' \in \text{Com}_{\text{UID}}^{c_{r_1}+1}$ is $(\langle \text{admin}, m_0 \rangle, M)$, where $m_0$ is the memory where all values are set at 0,
- the runtime states is as follows:
  - $db'(R) = db(R)$ and $db'(TMP_R) = \emptyset$ for all $R$ in $D$,
  - $U' = U$,
  - $\text{sec}' = \text{sec} \cup \{ \oplus, u, (\text{SELECT}, TMP_R), \text{admin} \} | u \in U \land R \in D \land \exists \text{op } \in \{ \oplus, \ominus \}, \langle \text{op}, u, (\text{SELECT}, R), \text{admin} \rangle \in \text{sec}, \langle \text{op}, u', (\text{SELECT}, R), \text{admin} \rangle \in \text{sec} \}$,
  - $T = \{ \text{transTrigger}(t) \mid t \in T \}$, where $\text{transTrigger}(t) = t$ in case $t$'s action is not a GRANT or REVOKE command involving a SELECT privilege and $\text{transTrigger}((\langle id, e, e, R, \varphi, \langle \text{op}, u, (\text{SELECT}, R), m \rangle = (id, e, e, R, \varphi, \langle \text{op}, u, (\text{SELECT}, R), m \rangle)$ otherwise,
  - $V' = V \cup \{ \text{transView}(v) \mid v \in V \}$, where $\text{transView}((\langle id, o, q, m \rangle = (\langle \text{trans}, o, q, m \rangle)$ and $q'$ is the query obtained from $q$ by replacing each predicate symbol $R$ with $TMP_R$, and
  - $\text{ctx}' = \text{ctx}$,
  - the scheduler $S'$ is $S$.

\[\Box\]
We also introduce an inverse transformation \( T^{-1} \) that allows one to revert the effects of the transformation on a system state. The inverse transformation is defined as follows: \( T^{-1}(\langle db, U, sec, T, V \rangle) = \langle db', U, sec', T', V' \rangle \), where \( db' \) is obtained by dropping all relation schemas of the form \( TMP_R \), \( sec' \) is obtained by (1) removing from \( sec \) all GRANTs associated with SELECT privileges involving predicate symbols not of the form \( TMP \), and (2) replacing all occurrences of predicate symbols of the form \( TMP_R \) with \( R \), \( T' \) is obtained by replacing all predicate symbols of the form \( TMP_R \) with \( R \) in the triggers in \( T \), and \( V' \) is obtained by dropping all views that refer to predicate symbols of the form \( TMP_R \).

Lemma 7.2 connects the security conditions for external and internal attackers for a single epoch. Observe that the lemma works for a slightly modified version of the security condition for external attackers, which supports also database-level users.

**Lemma 7.2.** Let \( S \) be the sequential scheduler \( 0^\infty \), \( C \in Com_{UID} \) be a sequence of simple query-only programs (without ADD USER commands), \( s \) be a system state, \( M \in Mem_{UID} \) be a sequence of memories, \( u \in UID \) be a user, and \( (C', M', (s', \epsilon), S') \) be the configuration \( T(\langle C, M, (s, \epsilon), S \rangle) \). If \( (C, M, (s, \epsilon), S) \) is \((E^{user}, S_{user}^{user})\)-single-epoch and the sequence of programs \( C' \) is secure with respect to the user \( db(u) \) for \( S', s' \), and \( M' \) according to the progress-sensitive variant of Definition 7.3 (extended to handle users in \( UID \cup \{db(u) \mid u \in UID\} \)), then the sequence of programs \( C \) is secure with respect to the attacker \( u \) for \( S, s, M, E^{user}, \) and \( S_{user}^{user} \) according to the progress-sensitive variant of Definition 7.2.

We are now ready to introduce Reduction 7.3, which shows how to use an IFC mechanism for external attackers to construct an IFC mechanism for internal attackers when restricted to the sequential scheduler and simple query-only programs.

**Reduction 7.3.** Let \( S \) be the sequential scheduler \( 0^\infty \), \( C \in Com_{UID} \) be a sequence of simple query-only programs (without ADD USER commands), \( s \) be a system state, \( M \in Mem_{UID} \) be a sequence of memories, \( u \in UID \) be a user, and \( ifEnf \) be an IFC mechanism for external attackers. The reduction works as follows. We compute the longest run \( r \) that can be obtained by starting from \( (C, M, (s, \epsilon), S) \) and the set of epochs \((E^{user}, S_{user}^{user})(r)\). Then, the IFC mechanism \( ifEnf' \) for internal attackers returns \( \top \) iff for all epochs \((C_i, M_i, (s_i, ctx_i), S_i) \overrightarrow{s_m} (C_j, M_j, (s_j, ctx_j), S_j) \) in \((E^{user}, S_{user}^{user})(r)\), \( ifEnf(C', M', S, u, s') = \top \), where \( T((C_i^{[\epsilon s_i]}_i, M_i^{[\epsilon s_i]}_i, (s_i, ctx_i), S_i)) = \langle C', M', (s', ctx'), S' \rangle \).

Finally, Theorem 7.3 states that Reduction 7.3 can be used to construct a sound IFC mechanism for internal attackers using a sound IFC mechanism for external attackers as a building block.

**Theorem 7.3.** Let \( ifEnf \) be an IFC mechanism that is sound with respect to the progress-sensitive variant of Definition 7.3 (extended to handle users in \( UID \cup \{db(u) \mid u \in UID\} \)) and \( ifEnf' \) be the mechanism constructed in Reduction 7.3. Then, \( ifEnf' \) is a sound progress-sensitive IFC mechanism for internal attackers (for user-based epochs and simple query-only programs that do not contain ADD USER commands).

**Proof.** The soundness of the reduction follows from Lemma 7.1, which establish the soundness of considering the single epochs separately, and Lemma 7.2, which establish the soundness of using a mechanism for external attackers to check security against internal attackers for a single epoch.

We remark that Theorem 7.3 refers to a slightly extended version of the progress-sensitive variant of Definition 7.3 that supports also users of the form \( db(u) \), where \( u \in UID \). Our results can be immediately extended also to conditions that supports only users in \( UID \) by replacing each query statement \( x \leftarrow q \) executed by a user \( u \) with \( x \leftarrow q; \text{out}(u, x) \). Indeed, this transformation ensures that all database-level observations are turned into program-level observations, thereby preserving the attacker’s knowledge.

### 7.5.6 From progress-sensitive to progress-insensitive security

Theorems 7.1-7.3 refer to the progress-sensitive variants of Definitions 7.2 and 7.3. Our results, however, can be lifted to the progress-insensitive setting.

Theorem 7.4 states that the progress-sensitive and progress-insensitive definitions coincide for external attackers for safe programs.

**Theorem 7.4.** Let \( S \) be the sequential scheduler \( 0^\infty \), \( u \) be a user, \( C \in Com_u \) be a sequence of simple query-only programs, and \( (M_0, s_0) \) be a reachable global state such that \( C \) is safe for \( S \) and \( [(M_0, s_0)]_u \). Then, \( C \) is secure with respect to Definition 7.3 for \( S, M, s, \) and \( u \) iff \( C \) is secure with respect to the progress-sensitive variant of Definition 7.3 for \( S, M, s, \) and \( u \).

Theorem 7.5 states that the progress-sensitive and progress-insensitive definitions coincide for internal attackers whenever the programs associated with the single epochs are safe.
Theorem 7.5. Let $S$ be the sequential scheduler $0^\infty$, $u$ be a user, $C \in \text{Conf}_\varnothing$ be a sequence of simple query-only programs, and $(M_0, s_0)$ be a reachable global state such that $|C| = |M|$ and for all $1 \leq i \leq |M|$, $M(i) = \langle u_i, m_i \rangle$ and $C(i) = \langle u_i, c_i \rangle$. Furthermore, let $r$ be the longest run obtained starting from $(\langle C, M_0, (s_0, e), S \rangle)$. If $C_i^{[r]} | S_j$ is safe for $S$ and $(\langle M_i^{[r]} | s_i \rangle) | s$ for all $(\langle C_i, M_i, (s_i, ctx_i), S_i \rangle) \in [E^\text{user}, S^\text{user}]^\ast (r)$, then $C$ is secure with respect to Definition 7.2 for $S$, $M$, $s$, $u$, $E^\text{user}$, and $S^\text{user}$ iff $C$ is secure with respect to the progress-sensitive variant of Definition 7.2 for $S$, $M$, $s$, $u$, $E^\text{user}$, and $S^\text{user}$.

We remark that the programs produced by Reductions 7.1 and 7.2 satisfy the following conditions: (1) each query statement $q$ produces an output according to the WHILESql’s semantics, (2) the resulting programs are simple query-only programs, and (3) the programs associated to each epoch are safe with respect to the epoch’s initial state. Hence, Theorems 7.4 and 7.5 can be used to extend our results to the progress-insensitive setting.

7.5.7 A note on integrity

In this section, we used a sound IFC mechanism as a building block for constructing a provably secure DBAC mechanism that satisfies the data confidentiality notion from Chapter 6. Our results, however, do not extend to database integrity. Even though variants of non-interference can be used to ensure integrity for programs, the database integrity notion from Chapter 6 directly accounts for the semantics of SQL access control. As a result, database integrity introduces restrictions, such as those associated with GRANT and REVOKE commands, that have no immediate formalization in terms of non-interference.

7.6 Enforcing end-to-end security

This section presents a monitor that provably secures WHILESql programs with respect to external attackers. To achieve end-to-end security across the database and applications, the monitor must keep track of dependencies at the database level (between queries) and at the program level (between variables) and ensure that the information released by output statements complies with the current security policy. To this end, we combine dynamic information-flow tracking with query determinacy [124] and disclosure lattices [26]. We refer the reader to Chapter 2 for an introduction to these concepts.

We leverage disclosure lattices to reason about the security of database queries [26]. Given two sets of queries $Q_1$ and $Q_2$, disclosure lattices provide a precise model for answering the following questions: (i) Does $Q_1$ reveal more information than $Q_2$? (ii) What combined information is revealed by $Q_1$ and $Q_2$? (iii) What common information is revealed by $Q_1$ alone and $Q_2$ alone? Observe that a security policy defines a set of database tables and views that a user is authorized to read. Hence, policies can be seen as sets of database queries, that is elements of a disclosure lattice. In the following, we fix a system configuration $(D, \Gamma)$, where $D = (\Sigma, \text{dom})$.

7.6.1 Preliminaries

Before presenting our security monitor, we define some preliminary concepts.

Predicate queries. A predicate query is a query of the form $T(\overline{v})$ where $T$ is a relation schema in $D$ and $\overline{v} \in \text{dom}^{|T|}$. A predicate query represents a single tuple in the database. We denote by $RC^{\text{pred}}$ the set of all predicate queries.

Query Support. Let $q$ be a query and $Q$ be a set of predicate queries. We say that $Q$ minimally determines $q$, denoted $\text{minDet}_{D,T}(Q, q)$, if $D, \Gamma \vdash Q \rightarrow q$, and there is no $Q' \subset Q$ such that $D, \Gamma \vdash Q' \rightarrow q$. The support of $q$, denoted $\text{supp}_{D,T}(q)$, is $\{Q \in 2^{RC^{\text{pred}}} | \text{minDet}_{D,T}(Q, q)\}$. Informally, the support of $q$ contains all (minimal) sets of tuples that determine $q$. For instance, the query $T(1) \land R(2)$ is minimally determined by $\{T(1), R(2)\}$. Hence, its support is $\{\{T(1), R(2)\}\}$. For simplicity, we write $\text{supp}(q)$ instead of $\text{supp}_{D,T}(q)$ when the system configuration $(D, \Gamma)$ is clear from the context.

Well-formed integrity constraints. We say that a set $\Gamma$ of integrity constraints is well-formed iff for all predicate queries $q \in RC^{\text{pred}}$, $\text{supp}_{D,T}(q) = \{\{q\}\}$. In the following, we consider only well-formed system configurations $(D, \Gamma)$. Observe that common integrity constraints used in practice, such as primary key and foreign key constraints, are well-formed. An example of integrity constraint that is not well-formed is $T(1) \leftrightarrow R(2)$. Indeed, $\text{supp}_{D,T(1) \leftrightarrow R(2)}(T(1)) = \{T(1), \{R(2)\}\}$. 


7.6.2 Security monitor

Our enforcement mechanism for external attackers is a dynamic security monitor. It keeps track of how information flows across the programs and the database, and it terminates the execution whenever a statement may leak information. For simplicity, we assume there is a single attacker, represented by the user identifier ATK. Our mechanism can be generalized to support multiple attackers. We also assume that each user is associated with at most one program and that different programs use disjoint sets of variable identifiers. In the following, we denote by $\text{Vars}_{\text{ATK}}$ the set of variables occurring in $\text{ATK}$’s program and by $\text{sec}_{\text{u}}$ the initial security policy.

Security Lattice. Our security monitor uses the determinacy-based disclosure lattice to track information. To handle both queries and memory variables, we extend the database schema $D$ with a propositional symbol $\text{MEM}_{x}$ for each variable $x \in \text{Vars}$ occurring in the monitored WHILESql programs. We denote by $D^{\text{ext}}$ the extended database schema. Formally, our security lattice is the disclosure lattice $\langle \mathcal{L}, \subseteq, \sqcup, \sqcap, \perp, \top \rangle$ defined over $D^{\text{ext}}$, where $\subseteq = \subseteq_{D^{\text{ext}}}$, $\perp = \perp_{D^{\text{ext}}}$, and $\top = \top_{D^{\text{ext}}}$

Mapping Queries to Labels. Our security monitor uses the determinacy-based disclosure lattice to abstractly represent the information initially stored in the program memories, which does not come from the database.

Monitor States. A monitor state $\Delta$ is a function $\text{Vars} \cup \text{RC}^{\text{pred}} \cup \{ \text{pc}_{u} \mid u \in \text{UID} \} \rightarrow \mathcal{L}$ that associates each variable identifier and predicate query with a security label. The monitor state also keeps track of the security level associated with each program’s security context. Since each user $u$ is executing at most one program, we formalize the security contexts using identifiers of the form $\text{pc}_{u}$, where $u \in \text{UID}$ is the user executing the program. For example, $\Delta(\text{pc}_{\text{bob}})$ captures the security level of the condition if Bob’s program is executing one of the branches of an If statement. We lift $\Delta$ to expressions:

$$\Delta(e) = \bigcup_{e \in \text{vars}(e)} \Delta(x)$$

where $e$ is an expression and $\text{vars}(e)$ is $e$’s free variables.

The monitor’s initial state $\Delta_{0}$ is as follows: (a) for each variable $x$, if $x \in \text{Vars}_{\text{ATK}}$, then $\Delta_{0}(x) = \text{MEM}_{x}$ and $\Delta_{0}(x) = \perp$ otherwise, (b) for all $T(\tau) \in \text{RC}^{\text{pred}}$, then $\Delta_{0}(T(\tau)) = cl(T(\tau))$, and (c) for all $u \in \text{UID}$, $\Delta_{0}(\text{pc}_{u}) = \perp$.

Mapping Users to Labels. The function $L_{\text{u}}$ maps users to labels in our security lattice. Since we are interested in end-to-end security guarantees, we associate the attacker ATK with the set of tables and views he is authorized to read according to the current security policy and to the initial policy $\text{sec}_{0}$ along with the labels associated with all the variables in ATK’s program. Formally, $L_{\text{u}}(s, u) = \top$ for any $u \neq \text{ATK}$ and $L_{\text{u}}(s, \text{ATK}) = cl(\text{auth}(s, \text{ATK}) \cup \text{auth}(\text{sec}_{0}, \text{ATK})) \cup \bigcup_{x \in \text{Vars}_{\text{ATK}}} \text{MEM}_{x}$. To illustrate, given a security policy $\text{sec}_{0}$ stating that the attacker can read the table $T$ but not the table $R$ and ignoring the labels associated with variables, $L_{\text{u}}(s, \text{ATK}) = \bigcup_{x \in \text{dom}(T)} cl(T(\tau))$, i.e., the attacker is authorized to know anything contained in $T$.

The mapping of queries and users to labels from the disclosure lattice allows one to reason about information disclosure. For instance, if the above attacker observes the result of the query $q = T(1) \land R(2)$, this violates the security policy. In fact, $L_{\text{u}}(\Delta, q) \not\subseteq L_{\text{u}}(s, \text{ATK})$ since $cl(T(1), R(2)) \not\subseteq \bigcup_{x \in \text{Vars}(T)} cl(T(\tau))$.

Expansion Process. To correctly handle triggers, our monitor rewrites each SQL command into WHILESql statements encoding the triggers’ execution. We do so using the $\text{expand}(s, m, u, x \leftarrow q)$ function, which takes as input a runtime state $s$, a memory $m$, the user $u$ executing the command, and a command $x \leftarrow q$, and produces as output a sequence of statements modeling the command’s execution (together with all the triggers executed in response to $q$). We formalize the $\text{expand}$ function in Section 7.9.3.

Additional Queries and Statements. Our monitor extends the WHILESql syntax and semantics with two designated queries $T \oplus \mathcal{E}$ and $T \ominus \mathcal{E}$, and five designated statements $\text{asuser}(u', c)$, $\text{dbout}(u, v, o, \tau)$, $\|x \leftarrow q\|$, $\|c\|$, and $\text{set pc to l}$. The $T \oplus \mathcal{E}$ (respectively $T \ominus \mathcal{E}$) query inserts (respectively deletes) tuples from the database without firing triggers. The $\text{asuser}(u', c)$ statement is used to execute the command $c$ as the user $u'$ (inside the session of the user $u$ executing the attacker’s $c$ command).

The $\text{dbout}(u, v, o, \tau)$ statement, instead, is used to print database-level events, e.g., policy changes. Finally, we use $\|x \leftarrow q\|$ to denote queries that have already been through the expansion process. All the above queries and statements are used during the expansion process.

The main goal of our work is studying the interplay between programs and databases. Therefore, to avoid timing leaks caused by executing multiple programs in parallel, the monitor’s semantics...
7.6. Enforcing end-to-end security

F-Assign
\[ \text{nsu}(x, \text{pc}_u) \xrightarrow{\Delta'} = \Delta[x \mapsto \Delta(\text{pc}_u) \cup \Delta(e)] \]
\[ \langle \Delta, x := e, m, s \rangle \xrightarrow{u} \langle \Delta', \varepsilon, m[x \mapsto [e](m)], s \rangle \]

F-Expand
\[ \text{expand}(s, x, q, u) \]
\[ \langle \Delta, x \leftarrow q, m, s \rangle \xrightarrow{u} \langle \Delta, [e], m, s \rangle \]

F-Select
\[ \{v_1, \ldots, v_n\} = \text{vars}(\varphi) \quad \varphi' = \varphi[v_1 \mapsto [v_1](m), \ldots, v_n \mapsto [v_n](m)] \]
\[ q = \text{SELECT} \quad \langle [q](s, u) = \langle s', r, \varepsilon, \varepsilon \rangle, L_Q(\Delta, \varphi) \cup \bigcup_{v \in \text{vars}(\varphi)} \Delta(v) \rangle \text{nsu}(x, \text{pc}_u) \]
\[ \langle \Delta, [x \leftarrow \text{SELECT} \varphi], m, s \rangle \xrightarrow{u} \langle \Delta[x \mapsto \Delta(\text{pc}_u) \cup \ell_e], \varepsilon, m[x \mapsto r], s' \rangle \]

F-UpdateDatabaseOk
\[ \mathcal{T} = ([e_1](m), \ldots, [e_n](m)) \quad \otimes \in \{\oplus, \odot\} \quad q = T \otimes \mathcal{T} \]
\[ [q](s, u) = \langle s', r, \varepsilon, \varepsilon \rangle \quad \ell_e = \bigcup_{1 \leq i \leq n} \Delta(e_i) \quad \text{nsu}(T(\mathcal{T}), \text{pc}_u) \quad \ell_e \subseteq \Delta(T(\mathcal{T})) \quad \text{nsu}(x, \text{pc}_u) \]
\[ \langle \Delta, [x \leftarrow T \otimes \{e_1, \ldots, e_n\}], m, s \rangle \xrightarrow{u} \langle \Delta[T(\mathcal{T}) \mapsto \Delta(\text{pc}_u) \cup \ell_e], x \mapsto \Delta(\text{pc}_u) \cup \ell_e], \varepsilon, m[x \mapsto r], s' \rangle \]

F-UpdateConfigurationOk
\[ \{v_1, \ldots, v_n\} = \text{vars}(q') \quad q' = q[v_1 \mapsto [v_1](m), \ldots, v_n \mapsto [v_n](m)] \]
\[ \text{isCfgCmd}(q') \quad [q'](s, u) = \langle s', r, \varepsilon, \varepsilon \rangle \quad \ell_{cmd} = \bigcup_{1 \leq i \leq n} \Delta(v_i) \]
\[ \ell_{cmd} \subseteq c(l(\text{auth(seco}, \text{ATK})) \quad \Delta(\text{pc}_u) \subseteq c(l(\text{auth(seco}, \text{ATK}))) \quad \text{nsu}(x, \text{pc}_u) \]
\[ \langle \Delta, [x \leftarrow q'], m, s \rangle \xrightarrow{u} \langle \Delta[x \mapsto \Delta(\text{pc}_u) \cup \ell_{cmd}], \varepsilon, m[x \mapsto r], s' \rangle \]

F-Out
\[ \Delta(e) \cup \Delta(\text{pc}_u) \subseteq L_U(s, u') \]
\[ \langle \Delta, \text{out}(u', \varepsilon), m, s \rangle \xrightarrow{u} \langle \Delta', \varepsilon, m, s \rangle \]

F-IfTrue
\[ \begin{array}{l}
\{e\}(m) = \text{tt} \\
\begin{array}{l}
\ell' = \{c_1; \text{set pc to } \Delta(\text{pc}_u)\}
\Delta' = \Delta[pc_u \mapsto \Delta(e) \cup \Delta(\text{pc}_u)]
\end{array}
\end{array} \]
\[ \langle \Delta, \text{if then } c_1 \text{ else } c_2, m, s \rangle \xrightarrow{u} \langle \Delta', c', m, s \rangle \]

Figure 7.5: Security monitor – Selected Rules.
executes branching statements atomically, i.e., without interleaving the execution of other programs whenever a program is executing a branching statement. To do so, we introduce statements of the form $[c]$, which denote that $c$ should be executed atomically. Finally, we use statements of the form set $pc$ to $l$, where $l \in L$, to update the label associated with the executing program's context.

**Relaxed no-sensitive upgrade checks.** Our enforcement mechanism is a dynamic security monitor. A common technique for preventing implicit leaks of sensitive information in this setting is using *no-sensitive upgrade* (NSU) checks [22, 169]. Intuitively, a NSU check restricts the changes only to the variables whose label is at least that of the current execution's context, i.e., only for variables $x$ such that $\Delta(pc_u) \subseteq \Delta(x)$. This guarantees that changes to “low” variables never happen in “high” execution contexts. NSU checks, however, are rather restrictive: they may block executions that do not leak information. This is particularly relevant for security lattices with many labels, such as our disclosure lattice, where many labels are simply unrelated and NSU checks often fail.

To address this, we propose a simple relaxation of NSU checks that still prevents leaks of sensitive information. In particular, our relaxed NSU checks exploit the fact that the initial security policy $sec_0$ can be used to determine which labels can be considered as permanently “low”, no matter how the policy is modified during the execution. In more detail, given a variable $x$, the relaxed NSU check is defined as follows: $\Delta(pc_u) \subseteq c(l(auth(sec_0, ATK))) \vee \Delta(pc_u) \subseteq \Delta(x)$, where $sec_0$ is the initial security policy. Our relaxed NSU check is satisfied whenever the standard NSU check is. Additionally, our relaxed NSU check allows flows of informations whenever the security context $pc_u$ is permanently low, i.e., $\Delta(pc_u) \subseteq c(l(auth(sec_0, ATK)))$. In the following, we denote by $nsu(x, pc)$ the predicate $\Delta(pc) \subseteq c(l(auth(sec_0, ATK))) \lor \Delta(pc) \subseteq \Delta(x)$, where $x$ is either a variable identifier or a predicate query, $pc$ is an identifier of the form $pc_u$, and $u$ is a user identifier.

To illustrate, our relaxed NSU check allows changes to a variable $x$ whose label is, say, $c(l(T(1)))$ in an execution context such that $\Delta(pc_u) = c(l(V(2)))$ whenever the attacker $ATK$ is authorized to read both the table $T$ and $V$ with respect to the initial policy $sec_0$. A standard NSU check, instead, would have prevented the assignment since $c(l(V(1))) \not\subseteq c(l(T(1)))$.

**Enforcement Rules.** Figure 7.5 presents selected rules associated with the execution of queries. We present the full operational semantics of our monitor in Section 7.9.2.

The rule $F$-Assign updates the monitor state whenever there is an assignment. The rule prevents leaks using relaxed NSU checks. The rule $F$-Out ensures that the monitor produces only secure output events. In particular, it prints to the user $u'$ the value associated with the expression $e$ only if the security labels associated with the printed information and with the program counter are authorized to flow to $u'$, i.e., $\Delta(e) \cup \Delta(pc_u) \subseteq L_u(s, u')$. The rule $F$-If-True, instead, executes the then branch $c_1$ in an if statement and updates the labels associated with $pc_u$ based on the label associated with the if’s condition. The rule relies on the set $pc$ to $l$ command to reset the label of $pc_u$ when leaving the then branch. Also note that the rule encapsulates both the then branch $c_1$ and the set $pc$ to $l$ statement inside an atomic statement $[c_1]$ to prevent internal timing channels caused by the scheduler. We remark that the above rules implement rather standard dynamic information-flow tracking [134].

The rule $F$-Expand ensures that triggers as well as constraint checking is desugared into normal WHILESQL code that is handled by the other rules. The rule uses the $expand$ function, which we formalize in Section 7.9.3. The F-Select rule ensures, using relaxed NSU checks, that query results are stored only in variables with the proper security labels. The rule, finally, updates the label associated with the variable storing the query’s result to correctly propagate the flow of information.

The rule $F$-UpdateConfigurationOk handles configuration commands, i.e., GRANT, REVOKE, ADD USER, and CREATE commands. Since changes to the configuration are visible to $ATK$, the rule ensures that such changes are performed only in contexts that are initially low for the attacker, i.e., $\Delta(pc_u) \subseteq c(l(auth(sec_0, ATK)))$. Furthermore, the rule prevents leaks of sensitive information using the free variables in the commands using the check $l_{cmd} \subseteq c(l(auth(sec_0, ATK)))$. The rule also uses relaxed NSU checks to ensure that query results are stored only in variables with the proper security labels. The rule uses the predicate $isCfgCmd(q)$, which returns $\top$ iff $q$ is a configuration command. Finally, the rule $F$-UpdateDatabaseOk handles queries that modify the database’s content. The rule ensures that there are no changes to security labels based on secret information using relaxed NSU checks. Furthermore, the rule keeps track of the labels associated with the information stored in the database by updating the monitor’s state $\Delta$.

Observe that in WHILESQL, policy changes generate public events. This eliminates leaks through authorization channels [17], and no additional checks for such leaks (cf. channel context bounds in [18]) are needed.

**Soundness.** Theorem 7.6, proven in Appendix D, states that our enforcement mechanism is sound for external attackers, i.e., it satisfies Definition 7.3 with $\Rightarrow$ as the underlying evaluation relation.
Theorem 7.6. For all sequences of programs $C \in \text{Conf}_{\text{ID}}$, schedulers $S$, sequences of memories $M \in \text{Mem}_{\text{ID}}$, and system states $s$, whenever $r = (\Delta_0, C, M, (s, \epsilon), S) \xrightarrow{r_i} (\Delta', C', M', (s', clx'), S')$, then for all $1 \leq i \leq n$, $K_{\Delta_i}(M, s, C, S, \text{trace}(r^{i-1})) \cap A_{\text{ATK}, \text{sync}}(M, s) \subseteq K_{\Delta_i}(M, s, C, S, \text{trace}(r^i))$, where $K_{\Delta_i}$ refers to Definition 7.1 with $\rightarrow$ as evaluation relation and the database in $r$'s $(i-1)$-th configuration is $(db, U, sec, T, V)$.

Example 7.3. Let $T, V, Z$ be three tables, $t$ be the trigger defined by the following command:

```
CREATE TRIGGER t ON T AFTER INS IF V(1) DO INSERT 1 INTO Z AS admin,
```

and $s$ be a system state containing $t$. The trigger is executed whenever a tuple is inserted in the table $T$, and $t$ inserts 1 in $Z$ if V(1) holds in the database state. The statement $x \leftarrow \text{INSERT 2 INTO T}$ is expanded as follows (provided all operations are authorized by the policy and there are no integrity constraints):

```
y \leftarrow \text{SELECT V(1)}
```

if $y$ then

```
x \leftarrow ||T \oplus 2||
```

```
z \leftarrow ||Z \oplus 1||
```

dbout((db(\text{ATK})), ||T \oplus 2||, x, \epsilon)

else

```
x \leftarrow ||T \oplus 2||
```

dbout((db(\text{ATK})), ||T \oplus 2||, x, \epsilon)

Suppose the attacker $\text{ATK}$ executes $x \leftarrow \text{INSERT 2 INTO T}; w \leftarrow \text{SELECT Z(1)}; \text{out(\text{ATK}, w)}$ from an initial database state $s_0$ where the tables $T$ and $Z$ are empty and the table $V$ contains a single record with value 1. We illustrate the behavior of the monitor for the security policy where $\text{ATK}$ cannot read $V$ but can read and modify $T$ and $Z$. Intuitively, the program is insecure since the presence of 1 in $Z$ depends (implicitly) on the presence of 1 in $V$, which $\text{ATK}$ cannot read. Consider the program execution with the initial state $s_0$ as above, and initial monitor state $\Delta_0$ such that $\Delta_0(x) = \text{MEM}_x$, $\Delta_0(w) = \text{MEM}_w$, $\Delta_0(y) = \Delta_0(z) = \top$ (since $y$ and $z$ do not occur in $\text{ATK}$’s program), and $\Delta_0(\text{pc}_{\text{ATK}}) = \bot$. The attacker’s label is $L_0(s_0, \text{ATK}) = \bigcup_{v \in \text{Vals}} c(l(T(v)) \cup \bigcup_{v \in \text{Vals}} c(Z(v))) \cup c(\{\text{MEM}_x, \text{MEM}_w\})$. The monitor would apply the rules $F\text{-EXPAND}$ (explained above), $F\text{-SELECT}$, $F\text{-IFTRUE}$, $F\text{-UPDATEDATABASEOK}$ (twice), $F\text{-DBOUT}$ (not shown), $F\text{-SETPC}$ (not shown), $F\text{-SELECT}$, and $F\text{-OUT}$. The evaluation of the first $\text{SELECT}$ statement yields $\Delta' = \Delta_0[y \mapsto \Delta(V(1)) \cup \bot]$, i.e., $\Delta'(y) = c(V(1))$. Observe that the $\text{SELECT}$ statement is successful $(\text{nsu}(x, \text{pc}_{\text{ATK}}))$ holds given that $\Delta_0(\text{pc}_{\text{ATK}}) = \bot$. The evaluation of the boolean condition $y$ yields $\Delta' = \Delta[y \mapsto c(V(1)), \text{pc}_{\text{ATK}} \mapsto c(V(1))]$. For the subsequent database update, the monitor checks whether $\text{nsu}(T(2), \text{pc}_{\text{ATK}})$ holds for the monitor state $\Delta'$. Namely, the monitor checks whether $c(V(1)) \subseteq l_0 \lor c(V(1)) \subseteq c(T(2))$ holds, where $l_0 = \bigcup_{v \in \text{Vals}} c(T(v)) \cup \bigcup_{v \in \text{Vals}} c(Z(v))$. Since this is not the case, the monitor stops the execution and correctly prevents the leakage.

### 7.6.3 A note on implementations

While most of the rules defining our monitor’s semantics can be easily implemented in practice, there are some points that are worth discussing.

**Representing the monitor state.** The monitor state $\Delta$ associates to each predicate query $q$ a label $l \in \mathcal{L}$. However, the set of predicate queries is infinite. To address this, one can explicitly keep track only of the labels of those queries that have been involved in $\text{INSERT}$ or $\text{DELETE}$ operations. This allows one to always store only a finite number of labels, without affecting the monitor’s security. Furthermore, each label $l$ may consist of an infinite set of predicate queries. Instead of storing the predicate queries, one can just store the finite set of queries $Q$ associated with the label $l$.

**Determining the relation between labels.** Given two labels $l$ and $l'$, determining whether $l \subseteq l'$ requires to determine whether a set of queries $Q$ determines another set of queries $Q'$. However, query determinacy is undecidable for the relational calculus [124]. To address this, one can use a sound under-approximation of determinacy to decide whether $Q$ determines $Q'$, such as the one we presented in Chapter 6. This defines a sound under-approximation $\subseteq_{ax}$ of the relation $\subseteq$.

**Computing the $L_Q$ function.** Computing the result of $L_Q(\Delta, q)$ requires to compute the set $\text{supp}_{\text{D}, q}(\text{q})$. This is challenging for two reasons: (1) the notion of $\text{supp}_{\text{D}, q}(\text{q})$ relies on query determinacy, and (2) $\text{supp}_{\text{D}, q}(\text{q})$ may contain an infinite amount of queries. The first problem can be addressed by employing a sound over-approximation of query determinacy. Differently from $\subseteq_{ax}$,
computing $\text{supp}_{P,F}(q)$ requires an over-approximation to preserve the monitor’s security. To address the second problem one can represent sets of predicate queries using non-boolean queries.

**Approximations and precision.** While the above points do not impact the monitor’s security, they can influence its precision and performance. In general, better under- and over-approximations of determinacy may lead to more precise monitors but may reduce the monitor’s performance. For example, one may employ relatively simple syntactic under- and over-approximations of determinacy, such as the one we presented in Chapter 6. This allows the monitor to efficiently compute all the security checks. However, these approximations often lead to imprecise results. For instance, the under-approximation from Chapter 6 is not able to determine that $cl(T(1)) \subseteq cl((R(1) \lor \neg R(1)) \land T(1))$. Alternatively, one may rely on more complex approximations, such as using automated theorem provers, or even exact solutions for fragments of the relational calculus [124, 127]. While these approaches may greatly increase the monitor’s precision, they usually have a high computational complexity.

### 7.7 Related Work

**Database Access Control.** As we already discussed in Chapters 3 and 6, many security conditions have been proposed for SELECT-only attackers [26, 27, 131, 165]. These conditions are usually built on top of well-known database problems, such as determinacy [124], instance-based determinacy [107], and certain answer [9], and they target a setting where both the database’s content and the policy are static. Our work, instead, considers a stronger attacker model where malicious users may dynamically modify the policy as well as exploiting advanced database features. In this respect, the attacker model considered in this chapter is similar to the one from Chapter 6. We refer the reader to Section 6.6 for an in-depth discussion of DBAC attacker models.

The security conditions we reviewed above often refer to secure DBAC mechanisms. They mix security and enforcement concerns. In contrast, our condition characterize the security of a given sequence of commands only in terms of the attacker’s knowledge. As a result, our condition provides a purely semantic characterization of database security in dynamic settings, where both the security policy and the database’s content may change.

Bender et al. [26, 27] introduce disclosure lattices to reason about the security of SELECT queries in the presence of fine-grained security policies. Specifically, they use a disclosure lattice based on query homomorphisms limited to a restricted class of queries (filter-project queries) [26, 27]. In contrast, we exploit a disclosure lattice based on query determinacy to track how information flows across programs and databases. Observe also that our security condition for internal attackers may serve as a baseline for justifying soundness of their policy enforcement algorithms.

Since the work we presented in this chapter is based on the operational semantics from Chapter 5 and relies on an attacker model similar to the one from Chapter 6, we refer the reader to Section 6.6 for a detailed comparison with Mandatory Access Control research.

**Information-flow Control for database-backed applications.** Until recently, little work has been done on IFC for database-backed applications [24, 49, 50, 57, 96, 143, 167]. In contrast to existing studies, our work has the following distinguishing features: (1) a realistic model of database systems, which accounts for advanced database constructs like triggers, views, and dynamic policies, (2) a security monitor combining information-flow tracking with disclosure lattices, and (3) a soundness proof of security for a realistic database model. Existing works either consider simple database models or provide informal soundness arguments.

A recent trend is to integrate database query mechanisms directly inside programming languages, see, for instance, Microsoft’s Language Integrated Queries (LINQ) [3]. In response to this, information-flow approaches have been proposed for languages supporting LINQ [24, 141], Schoepf et al. [141] propose SeLinq, a secure type system for a subset of the F# language extended with language integrated queries. Their work builds on the LINQ’s foundations of Cheney et al. [46], which supports only SELECT queries. Furthermore, they assume that both the database content and the security policy are static and they support only column-level policies over the database. Balliu et al. [24], instead, present JsLinq, a framework for securing multi-tier applications. Their work too builds on the work of Cheney et al. [46]. Hence, it is subject to similar restrictions as SeLinq, namely it supports only SELECT queries and column-level policies. Observe that JsLinq supports declassification through escape hatches, which is similar to our use of GRANT commands to delegate read privileges. In contrast to these approaches, our work adopts a realistic database model, which accounts for advanced features, and we support dynamic policies. Additionally, we combine dynamic monitoring with disclosure lattices to track fine-grained tuple-level dependencies, which SeLinq and JsLinq cannot track.

Corcoran et al. [57] present SeLinks, an extension of the Links programming language [56] that allows a database and a web application to collaborative enforce security policies using dependent types. Internally, SeLinks builds on top of the Fable system [155]. In contrast to our work, SeLinks...
7.7. Related Work

targets a rather simple database model that does not support security-critical features like integrity
constraints and triggers, which can be used to leak sensitive information (see Chapter 6).

Sif, developed by Chong et al. [50], is a software framework that allows developers to build secure
web applications. Programmers annotate variables and other data sources with fine-grained labels,
and Sif automatically tracks information flows across the web application to guarantee the absence
of leaks of sensitive information. Sif, however, only provides a limited database support: it requires
the database interface to be annotated with Sif annotations at column-level. Fabric [114], instead,
is a Jif extension for building secure distributed systems. While it considers persistent storage, it
does not provide direct support for databases and for tracking information through them. Compared
to these approaches, our work supports complex dynamic policies defined through GRANT and REVOKE
commands and it tracks dependencies between variables and database tuples.

Schultz and Liskov [143] extend decentralized information-flow control to track information across
database boundaries. Their work is based on concepts from multi-level security and it relies on poly-
instantiation. In contrast to our work, Schultz and Liskov’s approach cannot be directly applied to
most SQL databases, which do not support multi-level security approaches (see Section 6.6 for a
detailed discussion of this topic).

Chlipala [49] introduces UrFlow, a static analysis information-flow tool for the Ur/Web pro-
gramming language. Similarly to our security policies, UrFlow support policies specified using SQL
queries. Huang et al. [96], instead, present WeBSsARI, a tool that combines static and runtime
analysis to secure database-backed PHP applications. Both UrFlow and WeBSsARI consider a
simplistic database model. In contrast, our work targets a realistic database model that supports
advanced security-critical features, like triggers and dynamic policies. Additionally, our security
monitor provide precise security guarantees.

Yang et al. [167] present Jacqueline, a framework for policy-agnostic programming for database-
backed applications. The system combines dynamic information-flow control and faceted values to
secure programs interacting with databases. The database model considered in [167] lacks many of
the advanced features that are supported by our model. Observe also that Jacqueline’s goal is
executing programs in a secure way, not detecting security violations.

Griffin et al. [79] present Hails, a framework for developing web applications that supports
mandatory access control policies. DiTaint [59], instead, enhances database data types with one-
bit taint information. Compared to our approach, these works lack formal security justifications and
advanced database features.

Security Conditions. Our security conditions from Section 7.4 are inspired by existing knowledge-
based conditions for IFC [18,21,23]. Our condition for external attackers assumes perfect recall
attackers and considers dynamic policies. The condition is similar to existing knowledge-based notions
that use escape hatches for declassification, such as [21]. However, while escape hatches are often
disclosed permanently [21], we support both GRANT and REVOKE commands. We refer the reader to
Broberg et al. [39] for an extensive survey of IFC and dynamic policies.

Our security condition for internal attackers, instead, targets attackers that reset their knowledge
about the system’s state whenever the epoch changes. In this respect, the notion of non-interference
between updates introduced by Hicks et al. [94], which requires that non-interference holds between
policy updates, can be seen as a special case of our security condition instantiated with permission-
based epochs. Permission-based epochs represent portions of a run where the policy does not change.
We formalize them using the predicate $E_{perm}^u$, where $u \in UID$, defined as follows: $(C, M, (s, ctx),$
$S), (C’, M’, (s’, ctx’), S’)) \in E_{perm}^u$ iff $auth(s, u) \neq auth(s’, u)$.

Our security conditions in Section 7.4 are progress-insensitive. There are different flavors of
progress-insensitivity in the literature. For instance, Hedin and Sabelfeld [93] present a security con-
dition that differentiate between divergence and termination. Our security conditions, instead, follow
state-of-the-art conditions for dynamic policies [18,39] by not differentiating between divergence and
termination. As a result, our conditions ignore leaks caused by the output trace’s length. Observe,
however, that our condition is subject to brute-forcing leaks similar to those for [93], with known
information-theoretic bounds [19]. Note, however, that the results in Section 7.5 can be applied to
both progress-sensitive and progress-insensitive techniques.

Label Models. The universal lattice introduced by Hunt and Sands [98] is similar in spirit to our
disclosure lattice. The universal lattice allows one to express dependencies between variables,
where the lattice’s elements are sets of variables and the order relationship is set containment.
In contrast, our disclosure lattice allows one to reason about complex dynamic policies in databases
and applications. Observe also that the universal lattice can be embedded inside our disclosure
lattice, by using predicate symbols of the form MEM$_x$ for each variable $x$ (see Section 7.6). By
directly combining disclosure lattices with dynamic information-flow tracking, we can track fine-
grained dependencies between variables and queries. Note that these dependencies would be lost
using simpler label models, such as the standard “high” and “low” lattice.
Bridging DBAC and IFC. As we already mentioned, there are several IFC approaches that support databases [24, 49, 50, 57, 96, 143, 167]. Despite that, only little work has been done on bridging access control and information-flow control. To the best of our knowledge, John Rushby’s paper [133] has been one of the few to formally characterize non-interference and access control in the same framework. Rushby presented three properties that any system monitored by a secure access control mechanism should satisfy. Informally, these three requirements are: (1) the output produced by a user’s action should depend only on the values the user is authorized to read, (2) whenever a user’s action changes the system’s state, the resulting state must depend only on the values the user is authorized to read, and (3) whenever a user’s action changes the system’s state, the user should possess the corresponding write privilege. Rushby proved that, under specific conditions, a system that satisfies these requirements also satisfies non-interference. Our work can be seen as concretizing Rushby’s ideas for a specific setting, namely database access control, and execution model, i.e., the database operational semantics from Chapter 5. In this respect, we present a reduction for constructing secure DBAC mechanisms using IFC techniques as a building block, and we provide a semantic characterization (based only on the attacker’s knowledge) of our data confidentiality condition from Chapter 6 in terms of non-interference.

7.8 Conclusions

In this chapter, we developed common foundations for database access control and information-flow control, two previously unconnected research areas. These foundations rely on WhileSql, a simple imperative language with querying capabilities that builds on top of our database operational semantics from Chapter 5. We used these foundations to improve both DBAC and IFC.

Specifically, with respect to DBAC, we leveraged our security conditions to develop a general technique for constructing provably secure access control mechanisms from sound IFC enforcement mechanisms. This allows the DBAC community to benefit from existing IFC techniques and tools. Additionally, our security condition for internal attackers provides a semantic characterization for our data confidentiality notion from Chapter 6.

With respect to IFC, instead, we have developed a novel enforcement mechanism that monitors how information flows inside applications and across database boundaries. Our monitor combines dynamic information flow tracking with advanced DBAC concepts like disclosure lattices and query determinacy. As a result, it can track complex dependencies that depend on the database’s content and it supports advanced features like dynamic policies and triggers.

7.9 Technical Details

Here we present some technical details about WhileSql and our enforcement mechanism.

7.9.1 From WhileSql to the database operational semantics of Chapter 5

The WhileSql’s operational semantics from Section 7.3 builds on top of the database operational semantics formalized in Chapter 5. In particular, in the WhileSql semantics, the execution of the database commands is delegated to the function $[q](s, u)$, which takes as input an SQL statement $q$, a runtime state $s \in \Omega_M$, and the user $u \in UID$ executing the command, and it returns a tuple $\langle s', r, em, \tau \rangle$, where $s' \in \Omega_M$ is the new runtime state, $r$ is $q$’s result, $em$ is an error message, and $\tau$ is a trace of observation representing the public observations produced by the triggers activated by the query $q$. The function $[q](s, u)$ is defined in Figures 7.7–7.9 and it relies on the transition relation $\rightarrow_f$ from Chapter 5, where $f$ is a PDP. In the following, we instantiate $f$ to be the PDP $f_{int}$ developed in Chapter 5. This PDP ensures the integrity of the database, e.g., by avoiding unauthorized changes, but it does not provide confidentiality guarantees. Furthermore, we re-use various functions from Chapter 5, e.g., we reuse the functions res, Ex, and secEx to extract the outcomes of the command’s execution from a runtime state, and we lift the auth function from system states to runtime states, i.e., $auth(s, ctx)$ is simply $auth(s, u)$.

We remark that $[q](s, ctx, u)$ guarantees that the trigger transaction, which has been defined in Chapter 5, in the resulting runtime state $(s', ctx')$ is always $\epsilon$. This, combined with the fact that the context for initial states is $\epsilon$, ensures the correct execution of queries. As a consequence of this, $[q](s, ctx, u) = [q](s, ctx', u)$ for any two contexts $ctx, ctx'$ such that the trigger transaction is $\epsilon$.

For SELECT queries, $[q](s, u)$ is defined only for boolean queries because the operational semantics in Chapter 5 supports only boolean queries. We refer the reader to Chapter 5 for a discussion on how to handle non-boolean queries.

For INSERT and DELETE queries, $[q](s, u)$ relies on the function toTrace, defined in Figure 7.6, that takes as input an INSERT or DELETE command $q$ and a trace of the database execution consisting
7.9. Technical Details

The rules for the global semantics. The rules for the local operational semantics of our dynamic monitor. Figure 7.15 presents the security monitor that is the user that fired the trigger, and the identifier depending on the trigger's definition, which privileges the triggers is to be executed (i.e., the invoker depending on the trigger's condition). Note that the variables associated with the trigger's identifier, act is t's action, invk is the user that fired the trigger, and owner is the trigger's owner. Therefore, if the original command contains program variables, then cond and act may both contain program variables. Observe that the triggers function can be implemented on top of the functions we defined in Chapter 5. Given a trigger represented as \( t = (t, u, cond, act, invk, owner) \), we denote by \( id(t) \) the identifier \( id \), by \( user(t) \) the user \( u \), by \( cond(t) \) the condition \( cond \), by \( act(t) \) the action \( act \), by \( invoker(t) \) the invoker \( invk \), and by \( owner(t) \) the user \( owner \).

**Instrumented Commands.** An instrumented command is a pair \((c,r)\) such that \( c \) is an INSERT command, a DELETE command, or a trigger (represented as specified above using a 4-tuple), and

\[
\text{toTrace}(q, \epsilon) = \epsilon
\]

\[
\text{toTrace}(q, s_1 \xrightarrow{f} s_2 \xrightarrow{f} \ldots \xrightarrow{f} s_{n+1}) = \text{toTrace}(q, s_1 \xrightarrow{f} s_2 \xrightarrow{f} \ldots \xrightarrow{f} s_{n+1})
\]

\[
\begin{cases}
\langle \text{public}, t, \text{GRANT} \ p \ TO \ u \rangle & \text{if } res(s') = T \land acC(s') = T \\
\langle \text{public}, t, \text{GRANT} \ p \ TO \ u \text{ WITH GRANT OPTION} \rangle & \text{if } res(s') = T \land acC(s') = T \\
\langle \text{public}, t, \text{REVOKE} \ p \ FROM \ u \rangle & \text{if } res(s') = T \land acC(s') = T \\
\epsilon & \text{otherwise}
\end{cases}
\]

**Definition of the toTrace function.** The functions \( res \) and \( acC \) are defined in Chapter 5, the \( action \) function takes as input a trigger and returns its action. Only of triggers (throwing neither security nor integrity exceptions) and extracts a trace of public observations \((\text{public}, q, r)\) associated with the \( \text{GRANT} \) and \( \text{REVOKE} \) statements executed by the triggers in the trace. It also relies on the functions \( acC \) and \( acA \) that take as input a runtime state and retrieve the access control decisions associated with the trigger’s condition and the trigger’s action. We refer the reader to Chapter 5 for a formalization of these functions.

### 7.9.2 Enforcement Operational Semantics

Here we provide the full operational semantics of our security monitor. In the following, \( ATK \) denotes the user identifier associated with the attacker and \( sec0 \) denotes the initial security policy. Furthermore, we call extended any WHILESQL program \( c \) that possibly contains the additional commands introduced in Section 7.6.

**Preliminaries.** The auxiliary function \( \text{atomic}(c) \) takes as input an extended WHILESQL program and returns \( T \) if there are \( c', c'' \) such that \( c = [c'] \) or \( c = [c''] \). The auxiliary function \( \text{query}(c) \) takes as input an extended WHILESQL program and returns \( T \) if there are \( x, q \) such that \( c = x \leftarrow q \) or \( c = [[x \leftarrow q]] \). Finally, Figure 7.10 illustrates how the queries of the form \( t \cap \tau \), where \( \cap \in \{\oplus, \ominus\} \), are handled by the underlying database.

**Enforcement Rules.** Given a user \( u \in UID \), the relation \( \sim_u \), shown in Figures 7.11–7.14, formalizes the local operational semantics of our dynamic monitor. Figure 7.15 presents the security monitor rules for the global semantics. The rules F-EVAL-STEP and F-EVAL-END are similar to the WHILESQL semantics. Additionally, the monitor uses the F-ATOMIC-STATEMENT rule to handle the atomic execution of code. Observe that the atomic execution does not consume the scheduler.

### 7.9.3 Expansion Process

Here we illustrate our expansion process for queries.

**Extracting triggers.** Our expansion process uses the function \( \text{triggers} : \Omega_M \times Q \times \mathcal{U} \to \{\text{TRIGGER} \times \mathcal{U} \times Q \times \mathcal{U} \times \mathcal{U}\} \) that provides an interface to the database and returns the triggers in the form of tuples \((t, u, \text{cond}, \text{act}, \text{invk}, \text{owner})\), where \( t \) is the trigger’s identifier, \( u \) specifies the user under which privileges the triggers is to be executed (i.e., \( u \) is either the trigger’s owner or the trigger’s invoker depending on the trigger’s definition), \( \text{cond} \) specifies \( t \)’s \text{WHEN} condition, \( \text{act} \) is \( t \)’s action, \( \text{invk} \) is the user that fired the trigger, and \( \text{owner} \) is the trigger’s owner. Therefore, if the original command contains program variables, then \( \text{cond} \) and \( \text{act} \) may both contain program variables. Observe that the triggers function can be implemented on top of the functions we defined in Chapter 5. Given a trigger represented as \( t = (t, u, \text{cond}, \text{act}, \text{invk}, \text{owner}) \), we denote by \( id(t) \) the identifier \( id \), by \( user(t) \) the user \( u \), by \( cond(t) \) the condition \( cond \), by \( act(t) \) the action \( act \), by \( invoker(t) \) the invoker \( invk \), and by \( owner(t) \) the user \( owner \).
\[
\begin{align*}
\text{[SELECT] } \\
&\phi(s, u) = \begin{cases} \\
(s', \text{res}(s'), s, s) & \text{if } s \xrightarrow{\langle u, \text{SELECT}, \phi \rangle} s' \wedge \neg \text{SecEx}(s') \\
(s', t, \langle \text{SecEx}, \emptyset \rangle, s) & \text{if } s \xrightarrow{\langle u, \text{SELECT}, \phi \rangle} s' \wedge \text{SecEx}(s') \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{[ADD_USER] } u' \quad (s, u) = \begin{cases} \\
(s', \text{res}(s'), s, s) & \text{if } s \xrightarrow{\langle u, \text{ADD_USER}, u' \rangle} s' \wedge \neg \text{SecEx}(s') \\
(s', t, \langle \text{SecEx}, \emptyset \rangle, s) & \text{if } s \xrightarrow{\langle u, \text{ADD_USER}, u' \rangle} s' \wedge \text{SecEx}(s') \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{[CREATE_OBJECT] } \quad (s, u) = \begin{cases} \\
(s', \text{res}(s'), s, s) & \text{if } s \xrightarrow{\langle u, \text{CREATE_OBJECT} \rangle} s' \wedge \neg \text{SecEx}(s') \\
(s', t, \langle \text{SecEx}, \emptyset \rangle, s) & \text{if } s \xrightarrow{\langle u, \text{CREATE_OBJECT} \rangle} s' \wedge \text{SecEx}(s') \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{[GRANT] } p \to u' \quad (s, u) = \begin{cases} \\
(s', \text{res}(s'), s, s) & \text{if } s \xrightarrow{\langle \oplus, u, p, u' \rangle} s' \wedge \neg \text{SecEx}(s') \\
(s', t, \langle \text{SecEx}, \emptyset \rangle, s) & \text{if } s \xrightarrow{\langle \oplus, u, p, u' \rangle} s' \wedge \text{SecEx}(s') \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{[GRANT*] } p \to u' \quad (s, u) = \begin{cases} \\
(s', \text{res}(s'), s, s) & \text{if } s \xrightarrow{\langle \ominus, u, p, u' \rangle} s' \wedge \neg \text{SecEx}(s') \\
(s', t, \langle \text{SecEx}, \emptyset \rangle, s) & \text{if } s \xrightarrow{\langle \ominus, u, p, u' \rangle} s' \wedge \text{SecEx}(s') \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{[REVOKE] } p \text{ FROM } u' \quad (s, u) = \begin{cases} \\
(s', \text{res}(s'), s, s) & \text{if } s \xrightarrow{\langle \ominus, u, p, u' \rangle} s' \wedge \neg \text{SecEx}(s') \\
(s', t, \langle \text{SecEx}, \emptyset \rangle, s) & \text{if } s \xrightarrow{\langle \ominus, u, p, u' \rangle} s' \wedge \text{SecEx}(s') \\
\end{cases}
\end{align*}
\]

Figure 7.7: Definition of the \([q](s, u)\) function – part 1.

\[
\begin{align*}
\langle s', t, e, e \rangle & \quad \text{if } s \xrightarrow{\langle u, \text{INSERT}, T \rangle} s' \wedge \text{triggers}(s') = e \\
\langle s', t, e, \tau \rangle & \quad \text{if } \exists s_1, \ldots, s_n \in \Omega T, \exists t_1, \ldots, t_n \in \text{TIGGER}.
\end{align*}
\]

\[
\begin{align*}
\langle s', t, \langle \text{IntEx}, e(s') \rangle, e \rangle & \quad \text{if } s \xrightarrow{\langle u, \text{INSERT}, T \rangle} s' \wedge \text{SecEx}(s') \neq \emptyset \wedge \text{triggers}(s') = e \\
\langle s', t, \langle \tau, \text{IntEx}, e(s') \rangle, \tau \rangle & \quad \text{if } \exists s_1, \ldots, s_n \in \Omega T, \exists t_1, \ldots, t_{n-1} \in \text{TIGGER}.
\end{align*}
\]

\[
\begin{align*}
[q](s, u) = \begin{cases} \\
\langle s', t, e, e \rangle & \quad \text{if } s \xrightarrow{\langle u, \text{INSERT}, T \rangle} s' \wedge \text{triggers}(s') = e \\
\langle s', t, e, \tau \rangle & \quad \text{if } \exists s_1, \ldots, s_n \in \Omega T, \exists t_1, \ldots, t_n \in \text{TIGGER}.
\end{cases}
\end{align*}
\]

Figure 7.8: Definition of the \([q](s, u)\) function – part 2. Note that \(q = \text{INSERT} \land \text{INTO} \ T\).
### 7.9. Technical Details

\[
\begin{align*}
(s', r, e, e) & \quad \text{if } s \stackrel{\langle \text{DELETE}, \emptyset \rangle}{\rightarrow} f s' \wedge \text{secEx}(s') \\
(s', r, e, \tau) & \quad \text{if } \exists s_1, \ldots, s_n \in \Omega_M. \ \exists t_1, \ldots, t_n \in \text{TRIGGER.} \\
& \quad s \stackrel{\langle \text{DELETE}, \emptyset \rangle}{\rightarrow} f s_1 \xrightarrow{t_1} f s_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} f s_n \xrightarrow{t_n} f s' \\
& \quad r = \text{res}(s_1) \wedge \text{Ex}(s_1) = \emptyset \wedge \neg \text{secEx}(s_1) \wedge \\
& \quad \bigwedge_{1 \leq i < n} \left( \text{Ex}(s_{i+1}) = \emptyset \wedge \alpha A(s_{i+1}) \wedge \alpha C(s_{i+1}) \right) \\
& \quad \neg \text{triggers}(s') = e \wedge \neg \text{secEx}(s') \wedge \text{Ex}(s') = \emptyset \\
& \quad \tau = \text{toTrace}(\langle \text{DELETE}, \emptyset \rangle, s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} f s') \\

[q](s, u) = \\
\end{align*}
\]

- \( [T \oplus \emptyset](s, u) = \) \[
\begin{align*}
\{ (s'', \text{res}(s'), e, e) & \quad \text{if } s \stackrel{\langle \text{INSERT}, \emptyset \rangle}{\rightarrow} f s' \wedge \neg \text{secEx}(s') \wedge s = (db, U, S, T, V, ctx) \wedge \\
& \quad s'' = \{db[R \oplus \emptyset], U, S, T, V, ctx) \}
\end{align*}
\]
- \( [T \oplus \emptyset](s, u) = \) \[
\begin{align*}
\{ (s', \text{res}(s'), e, e) & \quad \text{if } s \stackrel{\langle \text{DELETE}, \emptyset \rangle}{\rightarrow} f s' \wedge \neg \text{secEx}(s') \wedge s = (db, U, S, T, V, ctx) \wedge \\
& \quad s' = \{db[R \oplus \emptyset], U, S, T, V, ctx) \}
\end{align*}
\]

\[7.19\] Definition of the \([q](s, u)\) function – part 3. Note that \( q = \text{DELETE} \) \( \emptyset \) FROM \( T \).

\[7.20\] Definition of the \([q](s, u)\) function – part 4.
Chapter 7. Reconciling Database Access Control and Information-flow Control

F-Skip

\[ \Delta, \text{skip}, m, s \leadsto u \Delta, \varepsilon, m, s \]

F-Assign

\[ [c; \text{set pc to } \Delta pc_u] \quad \Delta = \Delta x \mapsto \Delta pc_u \]

\[ \Delta, x := c, m, s \leadsto u \Delta, \varepsilon, m[x \mapsto e(m)], s \]

F-OUT

\[ \Delta(e) \cup \Delta pc_u \subseteq L_u(s, u') \]

\[ \Delta, \text{out}(u', e), m, s \leadsto u \Delta, \varepsilon, m, s \]

F-Seq

\[ \Delta(c1 ; c2, m, s) \leadsto u \Delta, \varepsilon, m, s \]

F-IfTrue

\[ \Delta, \text{if } e \text{ then } c1 \text{ else } c2, m, s \leadsto u \\Delta, \varepsilon, m, s \]

F-IfFalse

\[ \Delta, \text{if } e \text{ then } c \text{ else } \epsilon, m, s \leadsto u \\Delta, \varepsilon, m, s \]

F-WhileTrue

\[ \Delta(c; \text{while } e \text{ do } \epsilon, m, s) \leadsto u \Delta, \varepsilon, m, s \]

F-WhileFalse

\[ \Delta(c; \text{while } e \text{ do } \epsilon, m, s) \leadsto u \Delta, \varepsilon, m, s \]

F-ExpandedCode

\[ \Delta, c, m, s \leadsto u \Delta, \varepsilon, m, s \]

F-RemoveExpandedCode

\[ \Delta(c), m, s \leadsto u \Delta, \varepsilon, m, s \]

F-UpdateLabels

\[ \Delta, \text{set pc to } I, m, s \leadsto u \Delta pc_u \mapsto I, \varepsilon, m, s \]

F-AsUser

query(c)

\[ \Delta(pc_u \mapsto \Delta pc_u), c, m, s \leadsto u \Delta', c', m', s' \]

F-DbOut

\[ \text{vars}(v) = \{v_1, \ldots, v_n\} \quad \text{vars}(o) = \{o_1, \ldots, o_n\} \quad \tau = \tau_1 \cdot \ldots \cdot \tau_n \]

\[ \text{vars}(\tau(1)) = \{k^n_1, \ldots, k^n_{m_1}\} \quad \ldots \quad \text{vars}(\tau(n)) = \{k^n_{m_1}, \ldots, k^n_{m_n}\} \]

\[ o' = o|o_1 \mapsto [o_1](m), \ldots, o_n \mapsto [o_n](m) \quad \tau'(i) = \tau(i)(k^n_1 \mapsto [k^n_1](m), \ldots, k^n_{m_1} \mapsto [k^n_{m_1}](m)) \]

\[ u'' = \begin{cases} u' & \text{if } u' \neq \text{ATK} \land u' \neq \text{public} \land \tau \land \text{ATK} = \epsilon \\ \text{ATK} & \text{otherwise} \end{cases} \quad \Delta pc_u \cup \ell_{db} \subseteq L_u(s, u'') \]

\[ \Delta, \text{dbout}(u', v, o, \tau), m, s \leadsto u', \varepsilon, m, s \]

Figure 7.11: Security monitor – local operational semantics for assignments, print, and control flow statements.

Figure 7.12: Security monitor – local operational semantics for atomic statements.
F-Expand
\[ c_e = \text{expand}(s, x, q, u) \]
\[ \langle \Delta, x \leftarrow q, m, s \rangle \leadsto_a \langle \Delta, [c_e], m, s \rangle \]

F-Select
\[ \{v_1, \ldots, v_n\} = \text{vars}(\varphi) \quad \varphi' = \varphi[v_1 \mapsto [v_1](m), \ldots, v_n \mapsto [v_n](m)] \]
\[ q = \text{SELECT} \varphi \quad [q](s, u) = \langle s', r, \epsilon, \epsilon \rangle \quad \ell_x = L_Q(\Delta, \varphi) \cup \bigsqcup_{v \in \text{vars}(\varphi)} \Delta(v) \quad \text{nsu}(x, pc_u) \]
\[ \langle \Delta, \|x \leftarrow \text{SELECT} \varphi\|, m, s \rangle \leadsto_a (\Delta[x \mapsto \Delta(pc_u) \sqcup \ell_x], \epsilon, m[x \mapsto r], s') \]

F-UpdateDatabaseOk
\[ q = \langle \{e_1, \ldots, e_n\}(m) \rangle \otimes \langle T, \bar{T} \rangle \]
\[ q \circ (s, u) = \langle s', r, \epsilon, \epsilon \rangle \quad \ell_x = \bigsqcup_{1 \leq i < n} \Delta(e_i) \quad \text{nsu}(T(\bar{T}), pc_u) \quad \ell_x \subseteq T(\bar{T}) \quad \text{nsu}(x, pc_u) \]
\[ \langle \Delta, \|x \leftarrow T \otimes \{e_1, \ldots, e_n\}\|, m, s \rangle \leadsto_a (\Delta[T(\bar{T}) \mapsto \Delta(pc_u) \sqcup \ell_x], x \mapsto \Delta(pc_u) \sqcup \ell_x, \epsilon, m[x \mapsto r], s') \]

F-UpdateConfigurationOk
\[ \{v_1, \ldots, v_n\} = \text{vars}(q) \quad q' = q[v_1 \mapsto [v_1](m), \ldots, v_n \mapsto [v_n](m)] \]
\[ \text{isCfgCmd}(q') \quad [q'](s, u) = \langle s', r, \epsilon, \epsilon \rangle \quad \ell_{cmd} = \bigsqcup_{1 \leq i < n} \Delta(u_i) \]
\[ \ell_{cmd} \subseteq \text{cl(auth(\text{sec0}, \text{ATK}))} \quad \Delta(pc_u) \subseteq \text{cl(auth(\text{sec0}, \text{ATK}))} \quad \text{nsu}(x, pc_u) \]
\[ \langle \Delta, \|x \leftarrow q\|, m, s \rangle \leadsto_a (\Delta[x \mapsto \Delta(pc_u) \sqcup \ell_{cmd}], \epsilon, m[x \mapsto r], s') \]

\textbf{Figure 7.14:} Security monitor – local operational semantics for database operations.

F-Eval-Step
\[ \forall i \in \{1, \ldots, |C|\}, u' \in \text{UID}, C(i) \neq \langle u', \epsilon \rangle \quad |C| = |M| \]
\[ n = 1 + (n' \mod |C|) \quad C(n) = (u, c) \quad M(n) = (u, m) \quad \langle \Delta, c, m, s \rangle \leadsto_u (\Delta', c', m', s') \quad \text{atomic}(c') \quad C' = C(1) \ldots C(n-1) \cdot (u', c') \cdot C(n+1) \ldots C(|C|) \]
\[ M' = M(1) \ldots M(n-1) \cdot (u', m') \cdot M(n+1) \ldots M(|C|) \]
\[ \langle \Delta, C, m, s, n' : S \rangle \leadsto (\Delta', C', M', s', S) \]

F-Atomic-Statement
\[ \forall i \in \{1, \ldots, |C|\}, u \in \text{UID}, C(i) \neq \langle u', \epsilon \rangle \quad |C| = |M| \]
\[ n = 1 + (n' \mod |C|) \quad C(n) = (u, c) \quad M(n) = (u, m) \quad \langle \Delta, c, m, s \rangle \leadsto_u (\Delta', c', m', s') \quad \text{atomic}(c') \quad C' = C(1) \ldots C(n-1) \cdot (u', c') \cdot C(n+1) \ldots C(|C|) \]
\[ M' = M(1) \ldots M(n-1) \cdot (u', m') \cdot M(n+1) \ldots M(|C|) \]
\[ \langle \Delta, C, m, s, n' : S \rangle \leadsto (\Delta', C', M', s', S) \]

\textbf{Figure 7.15:} Security monitor – global operational semantics.
r ∈ {ok, dis, secEx, ex}. Note that both commands and triggers may contain program variables. Given an instrumented command \( \langle c, r \rangle \), \( \text{first}(\langle c, r \rangle) = c \) and \( \text{second}(\langle c, r \rangle) = r \).

**Constructing the execution paths.** Let \( \mathcal{T} \) be a sequence of commands and triggers. We denote by \( \text{paths}(\mathcal{T}) \) the following function:

\[
\text{paths}(\epsilon) = \emptyset \\
\text{paths}(c) = \{ \langle o, \text{ok} \rangle, \langle c, \text{secEx} \rangle, \langle c, \text{ex} \rangle \} \text{ where } c \text{ is an SQL command} \\
\text{paths}(t) = \{ \langle t, \text{ok} \rangle, \langle t, \text{dis} \rangle, \langle t, \text{secEx} \rangle, \langle t, \text{ex} \rangle \} \text{ where } t \text{ is a trigger} \\
\text{paths}(t\mathcal{T}) = \text{merge} (\text{paths}(t), \text{paths}(\mathcal{T})) \\
\text{merge}(S_1, S_2) = \{ s_1 \cdot s_2 \mid s_1 \in S_1 \land s_2 \in S_2 \land \neg \exists s'_1. o. (s_1 = s'_1 \cdot \langle o, \text{ex} \rangle \lor s_1 = s'_1 \cdot \langle o, \text{secEx} \rangle) \}
\]

**Configuration-consistent Execution Paths.** The execution paths computed through the \( \text{paths} \) function may, in general, contain unfeasible paths. For instance, they may contain commands terminating in a security exception even though this may not happen given the current security policy. To take this into account, we define the notion of a configuration-consistent execution path. Note that there is no analogous of configuration-consistent path for integrity exceptions caused by \text{INSERT} and \text{DELETE} commands. The reason is that these exceptions depend on the database content and are directly handled by the expansion process.

In the following, let \( s \) be a runtime state, \( m \) be a memory, \( u \) be a user, \( \mathcal{T} \) be a sequence of instrumented commands (i.e., an execution path), \( c \) be a database command, and \( t \) be a trigger. Furthermore, given a command (or trigger) \( o \) containing program variables, we denote by \( o(m) \) the command (or trigger) obtained from \( o \) by replacing all program variables with the corresponding values in \( m \). The configuration-consistency relation is defined as follows:

1. \( s, m, u \models^{<q>}(c, \text{ok}) \text{ if } \text{allowed}(s, u, c(m)) \).
2. \( s, m, u \models^{<q>}(c, \text{ex}) \text{ if } \text{allowed}(s, u, c(m)) \) and \( c \) is an \text{INSERT} or \text{DELETE} command.
3. \( s, m, u \models^{<q>}(c, \text{secEx}) \text{ if } \neg \text{allowed}(s, u, c(m)) \).
4. \( s, m, u \models^{<q>}(t, \text{ok}) \text{ if } \text{allowed}(s, u, t(m)) \).
5. \( s, m, u \models^{<q>}(t, \text{dis}) \text{ if } \text{allowed}(s, u, t(m)) \) and \( t \)'s action is an \text{INSERT} or \text{DELETE} command.
6. \( s, m, u \models^{<q>}(t, \text{secEx}) \text{ if } \neg \text{allowed}(s, u, t(m)) \).
7. \( s, m, u \models^{<q>}(t, \text{ex}) \).
8. \( s, m, u \models^{<q>}(o, r), \mathcal{T} \text{ if } \mathcal{T} \neq \epsilon, s, m, u \models^{<q>}(o, r), r \neq \text{dis}, \text{ and } \text{apply}(s, o(m)), m, u \models^{<q>} \mathcal{T} \).
9. \( s, m, u \models^{<q>}(o, \text{dis}), \mathcal{T} \text{ if } \mathcal{T} \neq \epsilon, s, m, u \models^{<q>}(o, \text{dis}) \) and \( s, m, u \models^{<q>} \mathcal{T} \).

The above definition relies on the functions \text{apply} and \text{allowed}, which are formalized in Figures 7.16 and 7.17. Let \( \mathcal{T} \) be a sequence of commands and triggers (possibly containing program variables), \( s \) be a database state, \( m \) be a memory, and \( u \) be a user. We denote by \( \text{consPaths}(\mathcal{T}, s, m, u) \) the set of all configuration-consistent paths derivable from \( \mathcal{T} \). Formally, \( \text{consPaths}(\mathcal{T}, s, m, u) = \{ \mathcal{T} \in \text{paths}(\mathcal{T}) \mid s, m, u \models^{<q>} \mathcal{T} \} \).

**Weakest precondition for database updates.** We now introduce weakest preconditions for \text{INSERT} and \text{DELETE} operations on databases. In the following, we assume that constants are not used inside predicate symbols. E.g., the formula \( T(x) \) is expressed as \( \exists x. T(x) \land x = x \). Let \( \phi \) be a relational calculus sentence that does not refer to views (for formulae that refer to views, one can first replace the views with their definitions and later compute the weakest precondition). Furthermore, we denote by \( T \oplus \pi \) (respectively \( T \ominus \pi \)) an insertion (respectively deletion) operation on the database. Note that \( \pi \) may contain program variables. The weakest precondition of \( \phi \) given \( T \oplus \pi \), written \( \text{wp}(\phi, T \oplus \pi) \), is obtained by replacing all occurrences of \( T(\pi) \) with \( (T(\pi) \lor \pi = \pi) \). Similarly, the weakest precondition of \( \phi \) given \( T \ominus \pi \), written \( \text{wp}(\phi, T \ominus \pi) \), is obtained by replacing all occurrences of \( T(\pi) \) with \( \neg (T(\pi) \land \pi = \pi) \).

**Weakest precondition for execution paths.** In Figure 7.18, we extend weakest preconditions from single \text{INSERT} and \text{DELETE} commands to sequences of instrumented commands.

**Expansion procedure.** Finally, the expansion procedure is shown in Figures 7.19 and 7.20. In the figures, \( s \) is a database state, \( m \) is a memory, \( u \) is a user identifier, and \( x \leftarrow q \) is an SQL command. Without loss of generality, we assume that \( x \not\in \text{free}(q) \) (if this is not the case, one can just introduce an additional temporary variable). Furthermore, given a list \( c_1 \ldots c_n \) of WhileSQL statements, we denote by \( \{c_1 \ldots c_n\} \) the statement \( c_1; c_2; \ldots; c_n \). Similarly, given a list of WhileSQL expressions \( e_1 \ldots e_n \), we denote by \( \forall (e_1 \ldots e_n) \) the expression \( e_1 \land e_2 \land \ldots \land e_n \).
allowed(s, u, SELECT ϕ) = \{ \top \text{ if } s \xrightarrow{(u, \text{SELECT}, ϕ)} f s' \land \neg \text{secEx}(s') \}
                                  \bot \text{ if } s \xrightarrow{(u, \text{SELECT}, ϕ)} f s' \land \text{secEx}(s')

allowed(s, u, GRANT p TO u') = \{ \top \text{ if } s \xrightarrow{(s, u, p, u')} f s' \land \neg \text{secEx}(s') \}
                                  \bot \text{ if } s \xrightarrow{(s, u, p, u')} f s' \land \text{secEx}(s')

allowed(s, u, GRANT p TO u' WITH GRANT OPTION) = \{ \top \text{ if } s \xrightarrow{(s, u, p, u')} f s' \land \neg \text{secEx}(s') \}
                                                  \bot \text{ if } s \xrightarrow{(s, u, p, u')} f s' \land \text{secEx}(s')

allowed(s, u, REVOKE p FROM u') = \{ \top \text{ if } s \xrightarrow{(s, u, p, u')} f s' \land \neg \text{secEx}(s') \}
                                                  \bot \text{ if } s \xrightarrow{(s, u, p, u')} f s' \land \text{secEx}(s')

allowed(s, u, INSERT \text{ ? INTO } T) = \{ \top \text{ if } s \xrightarrow{(s, u, \text{INSERT}, \text{?}, T)} f s' \land \neg \text{secEx}(s') \}
                                                  \bot \text{ if } s \xrightarrow{(s, u, \text{INSERT}, \text{?}, T)} f s' \land \text{secEx}(s')

allowed(s, u, DELETE \text{ ? FROM } T) = \{ \top \text{ if } s \xrightarrow{(s, u, \text{DELETE}, \text{?}, T)} f s' \land \neg \text{secEx}(s') \}
                                                  \bot \text{ if } s \xrightarrow{(s, u, \text{DELETE}, \text{?}, T)} f s' \land \text{secEx}(s')

allowed(s, u, CREATE obj) = \{ \top \text{ if } s \xrightarrow{(s, u, \text{CREATE}, \text{obj})} f s' \land \neg \text{secEx}(s') \}
                                                  \bot \text{ if } s \xrightarrow{(s, u, \text{CREATE}, \text{obj})} f s' \land \text{secEx}(s')

allowed(s, u, ADD USER u') = \{ \top \text{ if } s \xrightarrow{(s, u, \text{ADD USER}, u')} f s' \land \neg \text{secEx}(s') \}
                                                  \bot \text{ if } s \xrightarrow{(s, u, \text{ADD USER}, u')} f s' \land \text{secEx}(s')

allowed(s, u, t) = \text{allowed}(s, \text{owner}(t), \text{act}(t)) \text{ if } t \text{ is a trigger with owner’s privileges}

allowed(s, u, t) = \text{allowed}(s, \text{invoker}(t), \text{act}(t)) \land \text{allowed}(s, \text{owner}(t), \text{act}(t))
                                                  \text{ if } t \text{ is a trigger with activator’s privileges}

Figure 7.16: allowed function for the expansion process.

apply(s, u, SELECT ϕ) = s
apply(s, u, GRANT p TO u') = s’ where \( s \xrightarrow{(s, u', p, u)} f s' \)
apply(s, u, GRANT p TO u' WITH GRANT OPTION) = s’ where \( s \xrightarrow{(s, u', p, u)} f s' \)
apply(s, u, REVOKE p FROM u') = s’ where \( s \xrightarrow{(s, u', p, u)} f s' \)
apply(s, u, INSERT \text{ ? INTO } T) = s
apply(s, u, DELETE \text{ ? FROM } T) = s
apply(s, u, CREATE obj) = s’ where \( s \xrightarrow{(s, u, \text{CREATE}, \text{obj})} f s' \)
apply(s, u, ADD USER u') = s’ where \( s \xrightarrow{(s, u, \text{ADD USER}, u')} f s' \)
apply(s, u, t) = \text{apply}(s, \text{user}(t), \text{act}(t)) \text{ if } t \text{ is a trigger}

Figure 7.17: apply function for the expansion process. Note that we are interested only in changes to the database configuration, not to the database state. Therefore, the function does not update the database on INSERT and DELETE commands.
\[
wp(\phi, \epsilon) = \phi \\
wp(\phi, \overline{\mathcal{P}} \cdot ic) = wp(wp(\phi, ic), \overline{\mathcal{P}})
\]

where ic is an instrumented command and \(\overline{\mathcal{P}}\) is a sequence of instrumented commands

\[
wp(\phi, (\text{INSERT} T \text{ INTO } \overline{\mathcal{P}}, \text{ok})) = wp(\phi, T \oplus \overline{\mathcal{P}})
\]

\[
wp(\phi, (\text{INSERT} T \text{ INTO } \overline{\mathcal{P}}, \text{ex})) = wp(\phi, T \oplus \overline{\mathcal{P}})
\]

\[
wp(\phi, (\text{DELETE} T \text{ FROM } \overline{\mathcal{P}}, \text{ok})) = wp(\phi, T \ominus \overline{\mathcal{P}})
\]

\[
wp(\phi, (\text{DELETE} T \text{ FROM } \overline{\mathcal{P}}, \text{ex})) = wp(\phi, T \ominus \overline{\mathcal{P}})
\]

\[
wp(\phi, (c, r)) = \phi \text{ where } c \text{ is a SELECT, GRANT, REVOKE, CREATE, or ADD USER command}
\]

\[
wp(\phi, (t, ok)) = wp(\phi, \text{act}(t)) \text{ where } t \text{ is a trigger}
\]

\[
wp(\phi, (t, \text{ex})) = wp(\phi, \text{act}(t)) \text{ where } t \text{ is a trigger}
\]

\[
wp(\phi, (t, \text{secEx})) = \phi \text{ where } t \text{ is a trigger}
\]

\[
wp(\phi, (t, \text{dis})) = \phi \text{ where } t \text{ is a trigger}
\]

\[
\text{Figure 7.18: Weakest precondition for sequences of instrumented commands.}
\]

**Expansion Procedure.**

\[
\text{expand}(s, m, u, x \leftarrow q) = \text{decls}(s, m, u, x \leftarrow q) \land \text{body}(s, m, u, x \leftarrow q)
\]

**Shorthands.**

\[
\tilde{t}_q = \text{triggers}(s, u, q) \quad \text{first}(a, b) = a \quad \overline{\mathcal{EP}}_{s, m, u, q} = \text{toList} (\text{consPaths}(q, \tilde{t}_q, s, u))
\]

\[
\Gamma = \{\gamma_1, \ldots, \gamma_n\} \text{ are the integrity constraints} \quad \overline{T}_\Gamma = \text{toList}(2^{\{\gamma_1, \ldots, \gamma_n\}})
\]

\[
\text{throwsEx}(\overline{T}) = \exists o. (\overline{T}(\overline{\mathcal{P}})) = (o, \text{ex}) \lor \overline{T}(\overline{\mathcal{P}}) = (o, \text{secEx}) \quad \text{secEx}(\overline{T}) = \exists o. \overline{T}(\overline{\mathcal{P}}) = (o, \text{secEx})
\]

\[
isCfgCmd(q) = T \text{ iff } q \text{ is a GRANT, REVOKE, ADD USER, or CREATE command}
\]

\[
isInsDel(q) = T \text{ iff } q \text{ is an INSERT or DELETE command}
\]

\[
isSelect(q) = T \text{ iff } q \text{ is a SELECT command}
\]

\[
isTrigger(o) = T \text{ iff } o \text{ is a trigger}
\]

**Computing the expansion's auxiliary declarations.**

\[
\text{decls}(s, m, u, x \leftarrow q) = ((\text{map}(g_{s, m, u, q}, \overline{\mathcal{EP}}_{s, m, u, q})))
\]

\[
g_{s, m, u, q}(\overline{T}) = h_{s, m, u, q}(\tilde{t}(1), \ldots, h_{s, m, u, q}(\overline{T}))
\]

\[
h_{s, m, u, q}(\tilde{t}(i)) = |x_{i, \gamma_1} \leftarrow \text{SELECT } wp(\gamma_1, \tilde{t})|; \ldots; |x_{i, \gamma_n} \leftarrow \text{SELECT } wp(\gamma_n, \tilde{t})|
\]

where \(\tilde{t}(i)\) is not a trigger and \(x_{i, \gamma_1}, \ldots, x_{i, \gamma_n}\) are fresh variables

\[
h_{s, m, u, q}(\tilde{t}(i)) = |x_{i, \text{cond}} \leftarrow \text{SELECT } wp(\phi, \tilde{t}^{-1})|; |x_{i, \gamma_1} \leftarrow \text{SELECT } wp(\gamma_1, \tilde{t})|; \ldots; |x_{i, \gamma_n} \leftarrow \text{SELECT } wp(\gamma_n, \tilde{t})|
\]

where \(\tilde{t}(i)\) is a trigger, \(\phi\) is \(\tilde{t}(i)\)'s condition, and \(x_{i, \text{cond}}, x_{i, \gamma_1}, \ldots, x_{i, \gamma_n}\) are fresh variables

\[
\text{Figure 7.19: Expansion process – 1.}
\]
Computing the expansion’s body.

\[
body(s, m, u, x \leftarrow q) = \{ (map(d_{s,m,u,x,q}, E_{s,m,u,x,q})) \}
\]

\[
d_{s,m,u,x,q}(\bar{t}) = \begin{cases} 
\text{ok} & \text{if } cond_{s,m,u,x,q}(\bar{t}) \text{ then body}_{s,m,u,x,q}(\bar{t}) \text{ else skip} \\
\end{cases}
\]

\[
cond_{s,m,u,x,q}(\bar{t}) = \land \{ \text{map}(c_{s,m,u,x,q}, \bar{\bar{t}}, 1, \ldots, |\bar{\bar{t}}|) \}
\]

\[
c_{s,m,u,x,q}(\bar{t})(i) = \bar{x}_{i,\gamma_1} \land \ldots \land \bar{x}_{i,\gamma_n} \text{ where } \bar{t}(i) = (c, \text{ok}) \text{ and } c \text{ is not a trigger}
\]

\[
c_{s,m,u,x,q}(\bar{t})(i) = \lnot(\bar{x}_{i,\gamma_1} \land \ldots \land \bar{x}_{i,\gamma_n}) \text{ where } \bar{t}(i) = (c, \text{ex}) \text{ and } c \text{ is not a trigger}
\]

\[
c_{s,m,u,x,q}(\bar{t})(i) = \top \text{ where } \bar{t}(i) = (c, \text{secEx}) \text{ and } c \text{ is not a trigger}
\]

\[
c_{s,m,u,x,q}(\bar{t})(i) = \bar{x}_{i,\text{cond}} \land \bar{x}_{i,\gamma_1} \land \ldots \land \bar{x}_{i,\gamma_n} \text{ where } \bar{t}(i) = (t, \text{ok}) \text{ and } t \text{ is a trigger}
\]

\[
c_{s,m,u,x,q}(\bar{t})(i) = \bar{x}_{i,\text{cond}} \land \lnot(\bar{x}_{i,\gamma_1} \land \ldots \land \bar{x}_{i,\gamma_n}) \text{ where } \bar{t}(i) = (t, \text{ex}) \text{ and } t \text{ is a trigger}
\]

\[
c_{s,m,u,x,q}(\bar{t})(i) = \lnot\bar{x}_{i,\text{cond}} \text{ where } \bar{t}(i) = (t, \text{dis}) \text{ and } t \text{ is a trigger}
\]

\[
body_{s,m,u,x,q}(\bar{t}) = \begin{cases} 
x = (\text{SecEx}, \emptyset); o_{s,m,u,x,q}(\bar{t}) & \text{if } \bar{t}(1) = (\langle \text{o, secEx} \rangle) \text{ and } o \text{ is not a trigger} \\
x = (\text{id(first(\bar{t}))) \text{, SecEx}, \emptyset); o_{s,m,u,x,q}(\bar{t}) & \text{if } \bar{t}(1) = (\langle \text{o, secEx} \rangle) \text{ and } o \text{ is a trigger} \\
\{ (\text{map}(c_{s,m,u,x,q}, \bar{\bar{t}}, 1, \ldots, |\bar{\bar{t}}|)) \} \text{ if } \bar{t}(1) = (\langle \text{o, ex} \rangle) \\
\{ (\text{map}(d_{s,m,u,x,q}, \bar{\bar{t}}, 1, \ldots, |\bar{\bar{t}}|)) \} \text{ o_{s,m,u,x,q}(\bar{t}) otherwise} \\
\end{cases}
\]

\[
b_{s,m,u,x,q}(\bar{t})(i) = \langle |x \leftarrow T \odot \bar{\bar{t}}| \rangle \text{ where } \bar{t}(i) = (\langle \text{INSERT T INTO } \emptyset, \text{ok} \rangle)
\]

\[
b_{s,m,u,x,q}(\bar{t})(i) = \langle |x \leftarrow T \rhd \bar{\bar{t}}| \rangle \text{ where } \bar{t}(i) = (\langle \text{DELETE T FROM } \emptyset, \text{ok} \rangle)
\]

\[
b_{s,m,u,x,q}(\bar{t})(i) = \langle |x \leftarrow t\bar{\bar{t}}| \rangle \text{ where } \bar{t}(i) = (\langle \text{INSERT } \emptyset \text{ INTO } T, \text{ok} \rangle)
\]

\[
b_{s,m,u,x,q}(\bar{t})(i) = \text{asuser}(\text{user}(t), |y \leftarrow T \circ \bar{\bar{t}}|) \text{ where } \bar{t}(i) = (\langle \text{t, ok} \rangle), t \text{ is a trigger},
\]

\[
act(t) = \langle \text{INSERT T INTO } \emptyset, \text{y is a fresh variable} \rangle
\]

\[
b_{s,m,u,x,q}(\bar{t})(i) = \text{asuser}(\text{user}(t), |y \leftarrow T \circ \bar{\bar{t}}|) \text{ where } \bar{t}(i) = (\langle \text{t, ok} \rangle), t \text{ is a trigger},
\]

\[
act(t) = \langle \text{DELETE T FROM } \emptyset, \text{y is a fresh variable} \rangle
\]

\[
b_{s,m,u,x,q}(\bar{t})(i) = \text{asuser}(\text{user}(t), |y \leftarrow \text{act}(t)) \text{ where } \bar{t}(i) = (\langle \text{t, ok} \rangle), t \text{ is a trigger},
\]

\[
\text{isInsDel}(\text{act}(t)) = \bot, \text{y is a fresh variable}
\]

\[
b_{s,m,u,x,q}(\bar{t})(i) = \text{skip} \text{ where } \bar{t}(i) = (\langle \text{t, dis} \rangle)
\]

\[
e_{s,m,u,x,q}(\bar{t})(\Theta) = \begin{cases} 
\bigwedge_{\gamma_1} \bar{x}_{i,\gamma_1} \land \bigwedge_{\gamma_1} \bar{x}_{i,\gamma_1} \text{ then } e'_{s,m,u,x,q}(\bar{t})(\Theta) \text{ else skip} \\
\end{cases}
\]

\[
e'_{s,m,u,x,q}(\bar{t})(\Theta) = \begin{cases} 
dbout(d_{b,u}, q, \langle \text{IntEx}, \Theta \rangle, \text{extr}(\bar{t})); x = (\langle \text{IntEx}, \Theta \rangle) & \text{if } |\bar{t}| = 1 \\
dbout(d_{b,u}, q, \langle \text{id(first(\bar{t}))), \text{IntEx}, \Theta, \text{extr}(\bar{t}) \rangle; x = (\langle \text{id(first(\bar{t}))), \text{IntEx}, \Theta \rangle) & \text{if } |\bar{t}| > 1 \\
\end{cases}
\]

\[
o_{s,m,u,x,q}(\bar{t}) = \begin{cases} 
dbout(d_{b,u}, q, \langle \text{SecEx}, \emptyset \rangle, \text{extr}(\bar{t})); x = (\langle \text{SecEx}, \emptyset, \text{extr}(\bar{t}) \rangle) & \text{if } \text{isInsDel}(q) \land \neg \text{throwsEx}(\bar{t}) \\
\end{cases}
\]

\[
extr(\bar{t}) = \begin{cases} 
\langle \text{public}, \text{id(t)}, \text{act}(t) \rangle \rhd \text{extr}(\bar{t}) & \text{if } \text{isTrigger(t)} \land r = \text{ok} \land \text{isCfgCmd}(\text{act}(t)) \\
\text{extr}(\bar{t}) & \text{otherwise}
\end{cases}
\]

Figure 7.20: Expansion process – 2.
Part IV

Conclusions
Chapter 8

Conclusion

I hate endings. Just detest them. Beginnings are definitely the most exciting, middles are perplexing and endings are a disaster. . . . The temptation towards resolution, towards wrapping up the package, seems to me a terrible trap. Why not be more honest with the moment? The most authentic endings are the ones which are already revolving towards another beginning. That's genius.

Sam Shepard

Preventing disclosure of sensitive data is an extremely challenging task. It often requires a careful configuration of multiple enforcement mechanisms. It may also involve reasoning about security at different levels, from restricting access to the database's content to enforcing end-to-end security requirements on the applications interacting with the database. Therefore, insecure systems may result from mistakes introduced during several phases of the security engineering process. For instance, security vulnerabilities may be introduced by inaccurate security requirements, by misconfigured enforcement mechanisms, or by mechanisms that fail at providing the desired security guarantees.

In this context, provably secure enforcement mechanisms represent an indispensable component for database security. By providing precise security guarantees and provably preventing attacks, they significantly simplify the process of securing databases. Indeed, provably secure mechanisms free the security engineers from the difficult task of verifying whether a given mechanism, which is often an off-the-shelf component, correctly provides the desired security guarantees. Hence, the security engineers can focus on their main duties, namely eliciting the security requirements and formalizing them using security policies.

In this thesis, we developed solid foundations for access control and inference control for database systems. We used these foundations to build and verify practical and provably secure enforcement mechanisms. Specifically, we investigated DBAC and DBIC with respect to three different classes of attackers of increasing strength. First, we provided theoretical results clarifying the exact limits of DBAC with respect to SELECT-only attackers. Second, we designed a practical and provably secure enforcement mechanism that secures databases against SELECT-only attackers with probabilistic reasoning capabilities. Finally, we developed a framework for reasoning about database security against attackers that execute SELECT queries, modify the database’s content and configuration, and exploit advanced database features like triggers and views. We used this framework to build and verify a provably secure enforcement mechanism for this class of attackers. We also used it as a starting point for bridging DBAC with application-level end-to-end security in the form of IFC, and we showed that each area of research can benefit from reusing concepts and techniques developed by the other.

The work in this thesis opens up a number of future research directions, which we outline below.

Efficient algorithms for probabilistic inference. As shown by our work in Chapter 4, dedicated inference engines are an essential component for building practical enforcement mechanisms that effectively secure databases against selected classes of attackers, as the runtime complexity of general-purpose inference is intractable. In this respect, there are two main open research directions:

1. Tractable inference procedures are needed for other fragments of ProbLog. This would allow security engineers to model larger classes of attacker models. We remark that there are various classes of probabilistic models with tractable inference other than poly-tree BNs. For instance, inference runs in \( \text{PTime} \) in the network's size also for Bayesian Networks with bounded junction-trees [108]. While bounded junction-trees Bayesian Networks cannot be encoded as acyclic ProbLog programs, an approach similar to the one we adopted in Chapter 4, i.e., dedicated syntactic criteria together with a compilation to Bayesian Networks, may be used to derive other classes of ProbLog programs where inference is tractable.
Approximate inference algorithms are needed to handle programs that do not belong to fragments with efficient inference procedures. To be useful for security purposes, an approximate algorithm should return an interval \([a, b]\) containing the actual probability, as this would allow an enforcement mechanism to check both lower and upper bounds on an attacker’s belief. While naive approximate algorithms often have a very low precision, further research on the topic may lead to approximate algorithms with a reasonable efficiency and precision. A particularly interesting research direction here is investigating how approximate algorithms may benefit from exact algorithms, e.g., by reusing exact algorithms as sub-routines while analyzing portions of the overall program.

From a security standpoint, another open challenge is extending our framework to dynamic settings where the database and the policy change over time. In this respect, one would need to (1) lift attacker beliefs from databases states to traces, and (2) specify how operations modifying the database’s content and configuration affect these beliefs.

**Attacker models as first-class citizens.** We developed the idea that attacker models should be studied and formalized for databases. Rather than being implicit, the relevant models must be made explicit, just like when analyzing security in other domains. The \(\text{SELECT-only attacker model, the class of attackers expressible using AtkLog, and the attacker model we developed in Chapter 6 are just a few examples of attackers. Reasoning about database security requires, however, a large library of attacker models, each one capturing an attacker with different capabilities (attackers that can interact only through applications, attackers with direct database access, attackers with external knowledge, and so on). Studying and formalizing realistic classes of attackers is an important research direction for database security, which unfortunately has received only little attention so far. We remark that this library of attackers should be complemented with a family of provably secure enforcement mechanisms. In this way, a security engineer could just select an enforcement mechanism that provides the desired guarantees against a given attacker model.**

**Integration between DBAC and IFC.** Our work in Chapter 7 opens research opportunities for both access control and information-flow control. For DBAC, our reduction allows the development of provably secure DBAC mechanisms starting from IFC solutions. From a theoretical standpoint, an interesting open problem is extending our reduction to work for larger fragments of SQL. On a more practical side, it would be interesting to use our reduction to construct practical DBAC mechanisms using state-of-the-art IFC techniques. For IFC, instead, it would be interesting to study how other common IFC techniques, such as secure type systems or secure multi-executions, could be extended and benefit from a closer integration with databases.
Part V

Appendices
Appendix A

Proofs for Chapter 3

Proof of Theorem 3.3. We prove the decidability of $\text{AGREE}^{\text{ERC}}$ using Lemma 3.1, where the fragments are $F = \text{ERC}$ and $F' = \text{BSRRC}$. We divide the proof in four steps. First, we define some preliminary notation. Next, we provide a formula $\phi'_{\text{INDIST}(S,\phi)}$ not in the $\text{BSRRC}$ fragment that encodes the indistinguishability relation. Afterwards, we show that $\phi'_{\text{INDIST}(S,\phi)}$ can be equivalently rewritten as a $\text{BSRRC}$-formula $\phi_{\text{INDIST}(S,\phi)}$. Finally, we prove that the conditions 2 and 3 of Lemma 3.1 are met, and therefore we derive the decidability result.

Step 1. Let $D$ be a database schema, $S$ be an ERC-security policy over $D$, and $db$ be a state in $\Omega_D$. Let $\bar{t}$ be a tuple, and $y \equiv (y_1, \ldots, y_n)$ be a tuple of free variables such that $[\bar{t}] = [\bar{y}]$. The formula $\theta_{\phi,S}(\bar{y}, \bar{t})$, where $S$ is an ERC-policy, $\phi$ is a non-boolean ERC-query such that there is a constraint for $q$ in $S$, $\bar{t}$ is a tuple of variables, and $\bar{t}$ is a tuple of values in $\text{dom}$ such that $[\bar{y}] = [\bar{t}]$, is defined as follows:

$$\theta_{\phi,S}(\bar{y}, \bar{t}) := \bigwedge_{i \in \{1, \ldots, n\}} (y_i \mapsto \bar{t}(i)) \land \bigwedge_{i \in \{1, \ldots, n\}} \neg (\phi_S(y_i, \bar{t})).$$

The formula $\theta_{\phi,S}(\bar{y}, \bar{t})$ enforces that the tuple $\bar{t}$ associated with the values of the variables $\bar{y}$ is such that $\text{mask}_{S,q}(\bar{t}) = \bar{t}$, where $q$ is a non-boolean ERC-query.

Given a tuple of variables $\bar{t}$, we denote by $\bar{y}_{\bar{t}}$ the tuple of variables $y_1, \ldots, y_n$.

Step 2. The formula $\phi'_{\text{INDIST}(S,\phi)}$ encoding the indistinguishability relation is as follows:

$$\psi_S(\phi_{\bar{y}}, \phi_{\bar{t}}) := \bigvee_{(q, \phi)} |\text{Auth}_{S,q}(\phi_{\bar{y}})| (db) \bigwedge_{\bar{y}_{\bar{t}}} (\psi[\bar{y} \mapsto \bar{y}] \land \psi_{\phi/S}(\bar{y})).$$

$$\gamma_{\phi,S,\bar{t}} := \bigvee_{\phi_{\bar{y}}} |\text{card}_{S,\phi}(\phi_{\bar{y}})| (db) \bigwedge_{\bar{t}_{\phi}} (\phi[\bar{t} \mapsto \bar{y}] \land \psi_{\phi/S}(\bar{t})).$$

$$\phi'_{\text{INDIST}(S,\phi)} := \bigwedge_{(q, \phi) \in \text{ROW}} (\psi_S(q, \phi) \land \bigwedge_{\bar{t}_{\phi}} \gamma_{q,S,\bar{t}}).$$

Observe that $|\text{Auth}_{S,q}(db)| = \Sigma_{\phi \in \text{INDIST}(S,\phi)} |\text{card}_{S,\phi}(\phi)| (db)$ for any security policy $S$ and any non-boolean query $q$ such that $S$ contains a constraint for $q$. Therefore, although in $\gamma_{q,S,\bar{t}}$ we used counting quantifiers with the $\geq$ operator, the whole formula $\phi'_{\text{INDIST}(S,\phi)}$ can be satisfied iff all the counting quantifiers are satisfied exactly with equality. Furthermore, note also that if $S = (\emptyset, \emptyset)$, then $\phi'_{\text{INDIST}(S,\phi)} := \top$. The formula $\phi'_{\text{INDIST}(S,\phi)}$ encodes the indistinguishability relation. The encoding, however, is not in the $\text{BSRRC}$ fragment.

Correctness: We now prove the correctness of the encoding $\phi'_{\text{INDIST}(S,\phi)}$. Let $D$ be a database schema, $S = (\text{ROW}, \text{COL})$ be a security policy, and $db, db'$ be two states in $\Omega_D$. We now show that $\phi'_{\text{INDIST}(S,\phi)}|_{\phi} = \top$ iff $db \equiv_S db'$. Without loss of generality, we assume that $S \neq (\emptyset, \emptyset)$. If this is not the case, the claim trivially holds.

$(\Rightarrow)$. We show that $\phi'_{\text{INDIST}(S,\phi)}|_{\phi} = \top$ implies $db \equiv_S db'$. Let $(q, \phi) \in \text{ROW}$ be a constraint. From $\phi'_{\text{INDIST}(S,\phi)}|_{\phi} = \top$, it follows that $\psi_S(q, \phi) \land \bigwedge_{\phi \in \text{INDIST}(S,\phi)} \gamma_{q,S,\phi}(\phi) = \top$. From $\psi_S(q, \phi)|_{\phi} = \top$, it follows that there are at least $\text{card}_{S,\phi}(\phi)$ tuples $\bar{t}$ in $\text{Auth}_{S,q}(db)$ such that $\text{mask}_{S,\phi} \circ (\bar{t}) = \bar{t}$. Note also that, as before, $|\text{Auth}_{S,q}(db)| = \Sigma_{\phi \in \text{INDIST}(S,\phi)} |\text{card}_{S,\phi}(\phi)| (db)$, and therefore, for each $\bar{t} \in \text{Ind}_{S,q}(db)$, there are exactly $\text{card}_{S,\phi}(\phi)$ tuples $\bar{t}$ in $\text{Auth}_{S,q}(db)$ such that $\text{mask}_{S,\phi} \circ (\bar{t}) = \bar{t}$. From this, it follows that there is a bijection $f$ from $\text{Auth}_{S,q}(db)$ to $\text{Auth}_{S,q}(db')$ such that $\text{mask}_{S,\phi} \circ (f) = \bar{t}$. Hence, $\phi'_{\text{INDIST}(S,\phi)}|_{\phi} = \top$ implies $db \equiv_S db'$.

$(\Leftarrow)$. We show that $db \equiv_S db'$ implies $\phi'_{\text{INDIST}(S,\phi)}|_{\phi} = \top$. Let $(q, \phi) \in \text{ROW}$ be a constraint. From $db \equiv_S db'$, it follows that $|\text{Auth}_{S,q}(db)| = |\text{Auth}_{S,q}(db')|$, and therefore $\psi_S(q, \phi)|_{\phi} = \top$. From $db \equiv_S db'$, it also follows that for each $\bar{t} \in \text{Ind}_{S,q}(db)$, there are exactly $\text{card}_{S,\phi}(\phi)$ tuples $\bar{t}$ in $\text{Auth}_{S,q}(db)$ such that $\text{mask}_{S,\phi} \circ (\bar{t}) = \bar{t}$. Therefore, for each $\bar{t} \in \text{Ind}_{S,q}(db)$, there are exactly $\text{card}_{S,\phi}(\phi)$ tuples $\bar{t}$ in $\text{Auth}_{S,q}(db')$ such that $\text{mask}_{S,\phi} \circ (\bar{t}) = \bar{t}$. Hence, $\phi'_{\text{INDIST}(S,\phi)}|_{\phi} = \top$ implies $db \equiv_S db'$.
Appendix A. Proofs for Chapter 3

\[ \text{Auth}_{S,q}(d^\prime) \text{ such that } \text{mask}_{S,q,d}(\pi) = \top, \text{ and thus } [\forall_{t \in \text{Indist}(d)} \gamma_{q,S,d,t}]^{d^\prime} = \top. \text{ From } [\psi_{S,q,a}]^{d^\prime} = \top \text{ and } [\forall_{t \in \text{Indist}(d)} \gamma_{q,S,d,t}]^{d^\prime} = \top, \text{ it follows that } [\phi']^{\text{Indist}(S,a)}]^{d^\prime} = \top. \text{ Hence, } d^\prime \equiv_{S} d^\prime \text{ implies } [\phi']^{\text{Indist}(S,a)}]^{d^\prime} = \top. \]

**Step 3.** We now show that there is a formula \( \phi_{\text{Indist}(S,a)} \in BSRRC \) that is equivalent to \( \phi'_{\text{Indist}(S,a)} \). Let \( D \) be a database schema, \( S \) be a security policy over \( D \), \( s \) be a state, \( q \) be a non-boolean query such that there is a constraint on \( q \) in \( S \), and \( t \) be a tuple. We first show that there is a BSRRC formula equivalent to \( \psi_{S,q,a} \). Afterwards, we show that there is a BSRRC formula equivalent to \( \gamma_{q,S,d,t} \). Finally, we show that, under some weak assumptions, \( \phi \land \psi \in BSRRC \) if \( \phi, \psi \in BSRRC \). In the following, we denote logical equivalence by \( \equiv \). Our proof mainly exploits the equivalence \( (Q x, \phi) \text{ op } \psi \equiv Q x, (\phi \text{ op } \psi) \), where \( Q \in \{\forall, \exists\} \) and \( \text{op} \in \{\land, \lor\} \), if \( x \) is not a free variable in \( \psi \). Without loss of generality, we assume that the quantified variables used in the authorization constraints and in the queries have unique variable identifiers.

We now show that all the formulae of the form of \( \psi_{S,q,a} \) can be equivalently rewritten as BSRRC formulae if \( S \) is an ERC-policy. Let \( m \) be a natural number and \( \exists \gamma. \psi(\gamma, \bar{x}, \bar{y}) \) and \( \exists \gamma. \phi(\gamma, \bar{x}) \) be two ERC formulae. Figure A.1 shows that all the formulae of the form of \( \psi_{S,q,a} \) can be equivalently rewritten as BSRRC formulae if \( S \) is an ERC-policy.\(^1\) Note that the last formula in Figure A.1 is in the BSRRC fragment. Since \( \psi_{S,q,a} \) has the same form as the formula \( \psi \) used in the derivation, it follows that \( \psi_{S,q,a} \) can be equivalently rewritten as a BSRRC-formula.

Let \( \psi \) and \( \phi \) be two formulae in the BSRRC fragment such that the free variables of \( \psi \) are not used or quantified as variables in \( \phi \) and vice versa. Then, \( \psi \land \phi \in BSRRC \) as shown by the following derivation.

\[
\psi \land \phi = \exists \gamma. \forall \bar{y}. \gamma'(\bar{y}, \bar{x}, \bar{z}) \land \exists \gamma. \forall \bar{x}. \phi'(\bar{y}, \bar{x}, \bar{z}) \\
= \exists \gamma. (\forall \bar{y}. \gamma'(\bar{y}, \bar{x}, \bar{z}) \land \forall \bar{x}. \phi'(\bar{y}, \bar{x}, \bar{z})) \\
= \exists \gamma. \exists \gamma. (\forall \bar{y}. \gamma'(\bar{y}, \bar{x}, \bar{z}) \land \forall \bar{x}. \phi'(\bar{y}, \bar{x}, \bar{z})) \\
= \exists \gamma. \exists \gamma. \forall \bar{y}. (\forall \bar{y}. \gamma'(\bar{y}, \bar{x}, \bar{z}) \land \forall \bar{x}. \phi'(\bar{y}, \bar{x}, \bar{z})) \\
= \exists \gamma. \forall \bar{y}. \gamma.(\gamma'(\bar{y}, \bar{x}, \bar{z}) \land \phi'(\bar{y}, \bar{x}, \bar{z})) \\
= \exists \gamma. \forall \bar{y}. \gamma.(\psi'(\gamma, \bar{x}, \bar{y}) \land \phi'(\gamma, \bar{x}, \bar{y}))
\]

In a similar way, we can prove that \( \psi \lor \phi \in BSRRC \) for all \( \psi, \phi \in BSRRC \) such that the free variables of \( \psi \) are not used as quantified variables in \( \phi \) and vice versa.

To prove that \( \gamma_{q,S,a,t} \) can be rewritten as an equivalent BSRRC formula, we first show, in Figure A.2, that given an ERC-security policy \( S \), all formulae of the form of \( \theta_{q,S}(\bar{y}, \bar{t}) \) can be equivalently rewritten as BSRRC formulae with free variables \( \bar{y} \). The last formula in Figure A.2 is in the BSRRC fragment. Observe that \( \psi', \psi_{S,q} ', \theta_{q,S}(\bar{y}, \bar{t}) \) are BSRRC formulae, and that the quantified variables in one of the formulae are not free in the others (and vice versa). The conjunction of them is again a BSRRC formula \( \delta(\bar{y}) \), and therefore

\[
\gamma_{q,S,d,t} = \exists \gamma. \forall \bar{y}. \delta(\bar{y}) \\
= \exists \gamma_{1}, \ldots, \gamma_{m}. (\bigwedge_{i \in \{1, \ldots, m\}} \delta_{i}(\bar{y}) \land \bigwedge_{i,j \in \{1, \ldots, m\} \land i \neq j} \bar{y}_{i} \neq \bar{y}_{j})
\]

Both \( \bigwedge_{i \in \{1, \ldots, m\}} \delta_{i}(t) \) and \( \bigwedge_{i,j \in \{1, \ldots, m\} \land i \neq j} \bar{y}_{i} \neq \bar{y}_{j} \) are BSRRC formulae (and quantified variables in one of the formulae are not free variables in the other one), and therefore also their conjunction is a BSRRC formula. Moreover, since a BSRRC formula is of the form \( \exists \gamma. \forall \bar{y}. \phi(\bar{y}, \bar{z}) \), then also the formula \( \exists \gamma_{1}, \ldots, \gamma_{m}. \exists \gamma. \forall \bar{y}. \bigwedge_{i} \phi(\bar{y}, \bar{z}, \bar{z}) \) is a BSRRC formula, and therefore \( \gamma_{q,S,d,t} \in BSRRC \).

Since \( \psi_{S,q,a} \) and \( \gamma_{q,S,d,t} \) can be equivalently rewritten as BSRRC formulae, and since \( \phi'_{\text{Indist}(S,a)} \) is obtained from \( \psi_{S,q,a} \) and \( \gamma_{q,S,d,t} \) only by conjunctions, there is a BSRRC-formula \( \phi_{\text{Indist}(S,a)} \) that is equivalent to \( \phi'_{\text{Indist}(S,a)} \) in the BSRRC fragment.

**Step 4.** Finally, we prove that for any ERC formula \( \psi \) and any ERC-policy \( S \) then:
1. \( \phi_{\text{Indist}(S,a)} \land \psi \in BSRRC \), and
2. \( \phi_{\text{Indist}(S,a)} \land \lnot \psi \in BSRRC \).

The first case is simple since both \( \phi_{\text{Indist}(S,a)} \) and \( \psi \) are BSRRC sentences, and therefore also their conjunction can be rewritten as a BSRRC sentence. In the second case we can apply the following

\(^1\) In case \( m = 0 \), the resulting formula \( \exists \gamma. \forall \bar{y}. \phi(\bar{y}, \bar{z}) \) can be rewritten as an equivalent formula of the form \( \forall \gamma. \exists \bar{y}. \lnot \phi(\bar{y}, \bar{z}) \) which is in the BSRRC fragment.
ψ = ∃m,m. (∃i.ψ(i, x) ∧ ∃y.φ(y, x))
≡ ∃x₁, ..., xₘ.
( \( \bigwedge_{i \in \{1, \ldots, m\}} (\exists i.ψ(i, x_i) ∧ \exists y.φ(y, x_i)) \) ∧ \( \bigwedge_{i, j \in \{1, \ldots, m\} ∧ i \neq j} x_i ≠ x_j \) ∧ \( \forall x_{m+1}. (¬(∃ i.ψ(i, x_{m+1}) ∧ ∃ y.φ(y, x_{m+1})) \) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \)
≡ ∃x₁, ..., xₘ. (∃y₁, ..., yₘ, z₁, ..., zₘ.
( \( \bigwedge_{i \in \{1, \ldots, m\}} (ψ(z_i, x_i) ∧ φ(y_i, z_i)) \) ∧ \( \bigwedge_{i, j \in \{1, \ldots, m\} ∧ i \neq j} x_i ≠ x_j \) ∧ \( \forall x_{m+1}. (¬(∃ i.ψ(i, x_{m+1}) ∧ ∃ y.φ(y, x_{m+1})) \) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \)
≡ ∃x₁, ..., xₘ. (∃y₁, ..., yₘ, z₁, ..., zₘ.
( \( \bigwedge_{i \in \{1, \ldots, m\}} (ψ(z_i, x_i) ∧ φ(y_i, z_i)) \) ∧ \( \bigwedge_{i, j \in \{1, \ldots, m\} ∧ i \neq j} x_i ≠ x_j \) ∧ \( \forall x_{m+1}. (¬(∃ i.ψ(i, x_{m+1}) ∧ ∃ y.φ(y, x_{m+1})) \) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \)
≡ ∃x₁, ..., xₘ. (∃y₁, ..., yₘ, z₁, ..., zₘ.
( \( \bigwedge_{i \in \{1, \ldots, m\}} (ψ(z_i, x_i) ∧ φ(y_i, z_i)) \) ∧ \( \bigwedge_{i, j \in \{1, \ldots, m\} ∧ i \neq j} x_i ≠ x_j \) ∧ \( \forall x_{m+1}. (¬(∃ i.ψ(i, x_{m+1}) ∧ ∃ y.φ(y, x_{m+1})) \) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \)
≡ ∃x₁, ..., xₘ. (∃y₁, ..., yₘ, z₁, ..., zₘ.
( \( \bigwedge_{i \in \{1, \ldots, m\}} (ψ(z_i, x_i) ∧ φ(y_i, z_i)) \) ∧ \( \bigwedge_{i, j \in \{1, \ldots, m\} ∧ i \neq j} x_i ≠ x_j \) ∧ \( \forall x_{m+1}. (¬(∃ i.ψ(i, x_{m+1}) ∧ ∃ y.φ(y, x_{m+1})) \) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \)
≡ ∃x₁, ..., xₘ. (∃y₁, ..., yₘ, z₁, ..., zₘ. ∀xₘ₊₁. yₘ. (¬ψ(i, xₘ₊₁) ∨ ¬φ(yₘ, xₘ₊₁)) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \)
≡ ∃x₁, ..., xₘ. (∃y₁, ..., yₘ, z₁, ..., zₘ. ∀xₘ₊₁. yₘ. (¬ψ(i, xₘ₊₁) ∨ ¬φ(yₘ, xₘ₊₁)) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \)
≡ ∃x₁, ..., xₘ. (∀xₘ₊₁. yₘ. (¬ψ(i, xₘ₊₁) ∨ ¬φ(yₘ, xₘ₊₁)) ∨ \( \bigvee_{i \in \{1, \ldots, m\}} x_i = x_{m+1} ) \).

Figure A.1: Derivation for the ψ_{S, q, s} formula.
\[ \psi(\mathbf{y}, \mathbf{t}) = \bigwedge_{i \in \{1, \ldots, n\}} \exists \mathbf{y}. (\phi_i(\mathbf{y}, \mathbf{t}) \land \psi(i) = \mathbf{i}(i)) \land \bigwedge_{j \in \{1, \ldots, m\}} \neg \exists \mathbf{y}. \phi_j(\mathbf{y}, \mathbf{t}) \]

\[ \Rightarrow \exists \mathbf{y}_1, \ldots, \mathbf{y}_n (\bigwedge_{i \in \{1, \ldots, n\}} \phi_i(\mathbf{y}_i, \mathbf{t}) \land \psi(i) = \mathbf{i}(i)) \land \bigwedge_{j \in \{1, \ldots, m\}} \forall \mathbf{y} \neg \phi_j(\mathbf{y}, \mathbf{t}) \]

\[ \Rightarrow \exists \mathbf{y}_1, \ldots, \mathbf{y}_n (\bigwedge_{i \in \{1, \ldots, n\}} \phi_i(\mathbf{y}_i, \mathbf{t}) \land \psi(i) = \mathbf{i}(i)) \land \bigwedge_{j \in \{1, \ldots, m\}} \forall \mathbf{y} \neg \phi_j(\mathbf{y}, \mathbf{t}) \]

\[ \Rightarrow \exists \mathbf{y}_1, \ldots, \mathbf{y}_n, \forall \mathbf{z}_1, \ldots, \mathbf{z}_m (\bigwedge_{i \in \{1, \ldots, n\}} \phi_i(\mathbf{y}_i, \mathbf{t}) \land \psi(i) = \mathbf{i}(i)) \land \bigwedge_{j \in \{1, \ldots, m\}} \neg \phi_j(\mathbf{z}_j, \mathbf{t}) \]

Figure A.2: Derivation for the \( \theta_{K,S}(\mathbf{y}, \mathbf{t}) \) formula.

derivation:
\[ \gamma = \exists \mathbf{r}. \forall \mathbf{y}. \phi(\mathbf{r}, \mathbf{y}) \land \neg (\exists \mathbf{r}. \psi(\mathbf{r})) \]
\[ \Rightarrow \exists \mathbf{r}. \forall \mathbf{y}. \phi(\mathbf{r}, \mathbf{y}) \land \forall \mathbf{y} \neg \psi(\mathbf{r}) \]
\[ \Rightarrow \exists \mathbf{r}. (\forall \mathbf{y}. \phi(\mathbf{r}, \mathbf{y}) \land \forall \mathbf{y} \neg \psi(\mathbf{r})) \]
\[ \Rightarrow \exists \mathbf{r}. \forall \mathbf{y}. \exists \mathbf{z}. (\phi(\mathbf{r}, \mathbf{y}) \land \neg \psi(\mathbf{z})) \]

Therefore we can rewrite \( \phi_{INDIST(S,\alpha)} \land \neg \psi \) to an equivalent BSRRC formula.

Given a non-boolean query \( q \), a state \( db \), and a security policy \( S \), the sets \( Auth_{S,\alpha}(db) \) and \( Ind_{S,\alpha}(db) \) are finite and can be computed just by evaluating the constraints in \( S \) for all the tuples in \( [q]^d \). Similarly, we can trivially compute the value \( card_{S,\alpha,\theta}(\mathbf{t}) \) for any \( \mathbf{t} \in Ind_{S,\alpha}(db) \). Note that given a state \( s \) and a security policy \( S \), we can implement a computable algorithm that produces the formula \( \phi_{INDIST(S,\alpha)} \). It is easy to see that the rewriting from \( \phi_{INDIST(S,\alpha)} \) to the BSRRC-formula \( \phi_{INDIST(S,\alpha)} \) can be done in a systematic way in a finite number of operations. Therefore, we can implement a computable algorithm that, given a state \( db \), an ERC-policy \( S \), and an ERC-formula \( \psi \), produces the BSRRC-formulae \( \phi_{INDIST(S,\alpha)} \land \psi \), and \( \phi_{INDIST(S,\alpha)} \land \neg \psi \).

Since all the conditions of Lemma 3.1 are satisfied, then \( AGREE^{ERC} \) is decidable.

**Proof of Theorem 3.7.** We apply Lemma 3.3 to the ERC fragment, i.e., \( F = F' = ERC \). From Theorem 3.3, it follows that \( AGREE^{ERC} \) is decidable. We therefore need only an encoding of \( \phi_{T,\psi(\mathbf{r})} \) for any finite multi-set \( T \) and any \( \psi(\mathbf{r}) \in ERC \). In the following, given a multi-set of tuples \( T \) in \( M \) and a tuple \( \mathbf{t} \in T \), let \( K_{\mathbf{T},\mathbf{t}} \) be the multi-set \( \{ \mathbf{t} ': \mathbf{t} ' \in T \land \mathbf{t} ' \subseteq \mathbf{t} \} \).

Let \( T \) be a multi-set of masked and unmasked tuples, and let \( \psi(\mathbf{r}) \in ERC \) be a formula with free variables \( \mathbf{r} \) such that \( |\mathbf{r}| = |\mathbf{t}| \) for all \( \mathbf{t} \in T \). The formula \( \phi_{T,\psi(\mathbf{r})} \) is given by:
\[ \phi_{T,\psi(\mathbf{r})} := \bigwedge_{\mathbf{r} \in T} \exists \mathbf{z} \in |K_{\mathbf{T},\mathbf{t}}| \exists \mathbf{x}. (\left( \psi(\mathbf{x}) \land \bigwedge_{i \in \{1, \ldots, |\mathbf{r}|\} \psi(i) = \mathbf{i}(i) \right) \land \bigwedge_{j \in \{1, \ldots, |\mathbf{r}|\} \psi(j) = \mathbf{j}(j) \}) \]

Note that \( \phi_{T,\psi(\mathbf{r})} \) can be equivalently rewritten as an ERC-sentence as follows:
\[ \phi_{T,\psi(\mathbf{r})} := \bigwedge_{\mathbf{r} \in T} \exists \mathbf{z} \exists \mathbf{x}. (\left( \psi(\mathbf{x}) \land \bigwedge_{i \in \{1, \ldots, |\mathbf{r}|\} \psi(i) = \mathbf{i}(i) \right) \land \bigwedge_{j \in \{1, \ldots, |\mathbf{r}|\} \psi(j) = \mathbf{j}(j) \}) \]

In the formula above, \( |\mathbf{r}| = |\mathbf{r}| \) for all \( \mathbf{r} \in \{1, \ldots, |K_{\mathbf{T},\mathbf{t}}|\} \). Note that if \( T = \emptyset \), then the encoding is \( \top \) since \( \emptyset \subseteq K \) for any \( K \in N \).

Given a finite multi-set \( T \), and a tuple \( \mathbf{t} \in T \), the multi-set \( K_{\mathbf{T},\mathbf{t}} \) is finite and can be computed trivially. Therefore, given a multi-set \( T \) and an ERC-formula \( \psi(\mathbf{r}) \), we can implement a computable algorithm that produces the ERC-formula \( \phi_{T,\psi(\mathbf{r})} \). Since all the conditions of Lemma 3.3 are satisfied, then \( SUBSUME^{ERC} \) is decidable.

**Correctness:** Here we prove the correctness of the encoding \( \phi_{T,\psi(\mathbf{r})} \) given above. We now show that given a state \( db \in \Omega_D \), \( \phi_{T,\psi(\mathbf{r})} \) is \( \top \) iff \( T \subseteq \{ [\mathbf{r} | \psi(\mathbf{r})] \} \).
Appendix A. Proofs for Chapter 3

163

Proof of Theorem 3.10. We first introduce some additional results that we use in the proof of Theorem 3.10. There is no encoding of \(\varphi_{T,\psi(T)}\) in the ERC fragment, but there is an encoding in an extension of the ERC fragment called \(\text{ERC}^+\). We first introduce this extension, and afterwards we show that \(\text{AGREE}^{\text{ERC}^+}\) is decidable for all ERC-policies, i.e., the problem is decidable for all database schemas, all ERC-policies, and all ERC\(^+-\)-queries.

We first introduce the ERC fragment, the dual of the ERC-fragment.

Definition A.1. ERC: Let \(D\) be a database schema. The formula \(\varphi(T) := \forall \mathfrak{T}. \psi(\mathfrak{T}, \emptyset)\) is an ERC-formula over \(D\) iff \(\psi\) is a quantifier-free RC formula over \(D\).

Then, the ERC\(^+\)-fragment is defined as follows:

Definition A.2. ERC\(^+\): Let \(D\) be a database schema. An ERC\(^+\)-formula over \(D\) is inductively defined as follows.

1. \(\phi\) is an ERC\(^+\)-formula over \(D\), where \(\phi\) is an ERC-formula over \(D\).
2. \(\varphi \text{ op } \psi\) is an ERC\(^+\)-formula over \(D\), where \(\varphi\) is an ERC\(^+\)-formula over \(D\), \(\psi\) is an ERC\(^+\)-formula over \(D\), and \(\text{op} \in \{\land, \lor, \\exists\}\).

Observe that ERC \(\subset\) ERC\(^+\) \(\subset\) BSRRC. As a result, the encoding presented in Theorem 3.3 can be used to prove the decidability of the AGREE problem for the ERC\(^+\)-fragment. Note that this result holds just for ERC-policies, not for all ERC\(^+\)-policies.

Theorem A.1. AGREE\(^{\text{ERC}^+}\) is decidable for all ERC-policies.

Proof. The steps from 1 to 3 are the same as in the proof of Theorem 3.3 because we consider only ERC-policies. We just change slightly the fourth step.

Step 4: We prove that for any ERC\(^+\)-sentence \(\gamma\) and any ERC-policy \(S\) then:

1. \(\phi_{\text{INDIST}(S,d)} \land \gamma \in \text{BSRRC}\), and
2. \(\phi_{\text{INDIST}(S,d) \land \neg \gamma} \in \text{BSRRC}\).

From the fact that \(\gamma\) is of the form \(\phi \text{ op } \psi\), where \(\phi \in \text{ERC}\) and \(\psi \in \overline{\text{ERC}}\), it follows that both \(\gamma\) and \(\neg \gamma\) are in BSRRC because \(\phi\) is a sentence of the form \(\exists \mathfrak{T}. \varphi(\mathfrak{T})\) and \(\psi\) is a sentence of the form \(\forall \mathfrak{T}. \varphi(\mathfrak{T})\). From this and \(\phi_{\text{INDIST}(S,d)} \in \text{BSRRC}\), it follows that \(\phi_{\text{INDIST}(S,d) \land \gamma} \in \text{BSRRC}\), and \(\phi_{\text{INDIST}(S,d) \land \neg \gamma} \in \text{BSRRC}\).

It is easy to see that we can implement a computable algorithm that, given a state \(db\), an ERC-security policy \(S\), and an ERC\(^+\)-formula \(\psi\), produces the BSRRC-formulae \(\phi_{\text{INDIST}(S,d)} \land \psi\), and \(\phi_{\text{INDIST}(S,d) \land \neg \psi}\).

Since all the conditions of Lemma 3.1 are met, AGREE\(^{\text{ERC}^+}\) is decidable for all ERC-policies.

We are now ready to prove that EQUAL\(_{\text{ERC}}\) is decidable (i.e., Theorem 3.10). In this proof, we apply Lemma 3.4 to the ERC and ERC\(^+\) fragments, i.e., \(F = \text{ERC}\) and \(F' = \text{ERC}^+\). Let \(\psi(\mathfrak{T})\) be an ERC-formula with free variables \(\mathfrak{T}\) and \(T\) be a set of tuples such that \([\mathfrak{T}] = \overline{T}\) for all \(\mathfrak{T} \in T\). For EQUAL\(_{\text{ERC}}\) the encoding is as follows:

\[
\begin{align*}
\text{AT\_LEAST}_{T,\psi(\mathfrak{T})} & := \bigwedge_{\mathfrak{T} \in T} \exists \mathfrak{T}. (\psi(\mathfrak{T}) \land \mathfrak{T} = \overline{\mathfrak{T}}) \\
\text{AT\_MOST}_{T,\psi(\mathfrak{T})} & := \forall \mathfrak{T}. (\psi(\mathfrak{T}) \Rightarrow \bigvee_{\mathfrak{T} \in T} \mathfrak{T} = \overline{\mathfrak{T}}) \\
\phi_{T,\psi(\mathfrak{T})} & := \text{AT\_LEAST}_{T,\psi(\mathfrak{T})} \land \text{AT\_MOST}_{T,\psi(\mathfrak{T})}
\end{align*}
\]

Note that if \(T = \emptyset\), then \(\phi_{T,\psi(\mathfrak{T})} := \forall \mathfrak{T}. \neg \psi(\mathfrak{T}) \in \text{ERC}\).

Observe that \(\text{AT\_LEAST}_{T,\psi(\mathfrak{T})}\) can be equivalently rewritten as an ERC-formula, whereas \(\text{AT\_MOST}_{T,\psi(\mathfrak{T})}\) can be equivalently rewritten as an \(\overline{\text{ERC}}\)-formula, and therefore \(\phi_{T,\psi(\mathfrak{T})}\) can be equivalently rewritten as an ERC\(^+\)-sentence. Moreover, it is easy to see that given an ERC-formula
ψ(π) and a set of tuples T we can compute the encoding φ_{T,ψ(π)}. Therefore from Lemma 3.4 and Theorem A.1, it follows that EQUALERC is decidable.

*Correctness:* We now prove the correctness of the encoding φ_{T,ψ(π)} given above.

(⇒). We prove that [φ_{T,ψ(π)}]^{db} = ⊤ implies T = [{π | ψ(π)}]^{db}. From [φ_{T,ψ(π)}]^s = ⊤, it follows that [AT\_LEAST_{T,ψ(π)}]^{db} = ⊤ and [AT\_MOST_{T,ψ(π)}]^{db} = ⊤. From the former, it follows that 

\[ T \subseteq [{\{π | ψ(π)\}}]^{db} \]

and from the latter, it follows that 

\[ T \supseteq [{\{π | ψ(π)\}}]^{db} \]

Hence, 

\[ T = [{\{π | ψ(π)\}}]^{db}. \]

(⇐). We prove that 

\[ T = [{\{π | ψ(π)\}}]^{db} \]

implies [φ_{T,ψ(π)}]^{db} = ⊤. From 

\[ T = [{\{π | ψ(π)\}}]^s \]

it follows that 

\[ [AT\_LEAST_{T,ψ(π)}]^s = ⊤ \] and \[ [AT\_MOST_{T,ψ(π)}]^s = ⊤ \]. Hence, 

\[ [φ_{T,ψ(π)}]^s = ⊤. \]
Appendix B

Proofs for Chapter 4

B.1 Acyclicity of the ground graph

In this section we show that the ground graph of an acyclic program is a forest of poly-trees.

B.1.1 Proofs about Annotations

Here we prove that the ordering, disjointness, and uniqueness annotations capture the desired semantic properties.

Lemma B.1. Let \( \langle \Sigma, \text{dom} \rangle \) be a database schema such that \( \text{dom} \) is a finite domain and \( p \) be a \( \langle \Sigma, \text{dom} \rangle \)-ProbLog program. Furthermore, let \( \prec \) be the following relation over pairs of predicate symbols in \( \Sigma \): \( a \prec b \) iff \( b \in \text{reach}(a) \) and \( a \not\in \text{reach}(b) \). The relation \( \prec \) is a strict partial order and it is well-founded.

Proof. Let \( \langle \Sigma, \text{dom} \rangle \) be a database schema such that \( \text{dom} \) is a finite domain and \( p \) be a \( \langle \Sigma, \text{dom} \rangle \)-ProbLog program. Furthermore, let \( \prec \) be the following relation over pairs of predicate symbols in \( \Sigma \): \( a \prec b \) iff \( b \in \text{reach}(a) \) and \( a \not\in \text{reach}(b) \).

\( \prec \) is a strict partial order. To show that \( \prec \) is a strict partial order we must show that \( \prec \) is irreflexive and transitive. For irreflexivity, let \( a \) be a predicate symbol and assume, for contradiction’s sake, that \( a \prec a \) holds. From this, \( a \in \text{reach}(a) \) and \( a \not\in \text{reach}(a) \), leading to a contradiction. For transitivity, let \( a, b, c \) be three predicate symbols such that \( a \prec b \) and \( b \prec c \). From \( a \prec b \), \( b \in \text{reach}(a) \) and \( a \not\in \text{reach}(b) \). From \( b \prec c \), \( c \in \text{reach}(b) \) and \( b \not\in \text{reach}(c) \). From \( b \in \text{reach}(a) \) and \( c \in \text{reach}(b) \), it follows that \( c \in \text{reach}(a) \) (since \( \text{reach}(b) \subseteq \text{reach}(a) \)). From \( b \prec c \), \( \text{reach}(c) \subseteq \text{reach}(b) \). From this and \( a \not\in \text{reach}(c) \), it follows that \( a \not\in \text{reach}(c) \). From \( c \in \text{reach}(a) \) and \( a \not\in \text{reach}(c) \), \( a \prec c \).

\( \prec \) is well-founded. To show that \( \prec \) is well-founded, it is enough to show that there is no infinite descending sequence of elements. Assume, for contradiction’s sake, that this is not the case. Namely, there is an infinite sequence of predicate symbols \( a_0, a_1, \ldots \) such that for all \( i \in \mathbb{N} \), \( a_{i+1} \prec a_i \). We now consider the first \( n+1 \) elements, where \( n \) is the number of predicate symbols in \( \Sigma \). Then, we have the descending chain \( a_{n+1} \prec a_n \prec \cdots \prec a_1 \prec a_0 \). Observe that all \( n+1 \) elements have to be distinct (due to the irreflexivity and transitivity of \( \prec \)). This however contradicts the fact that we have at most \( n \) different predicate symbols.

\( \square \)

Proposition B.1. Let \( \langle \Sigma, \text{dom} \rangle \) be a database schema such that \( \text{dom} \) is a finite domain and \( p \) be a \( \langle \Sigma, \text{dom} \rangle \)-ProbLog program. If we can derive the annotation \( \text{ORD}(A) \) from \( p \), then if \( \text{pr}_1(\pi_1, \pi_2), \text{pr}_2(\pi_1, \pi_3), \ldots, \text{pr}_{n-1}(\pi_{n-1}, \pi_n) \) are in \( \text{ground}(p) \) and \( \{\text{pr}_1, \ldots, \text{pr}_{n-1}\} \subseteq A \), then \( \pi_1 \neq \pi_n \).

Proof. Let \( \langle \Sigma, \text{dom} \rangle \) be a database schema such that \( \text{dom} \) is a finite domain and \( p \) be a \( \langle \Sigma, \text{dom} \rangle \)-ProbLog program. Furthermore, we assume that (1) we can derive \( \text{ORD}(A) \) from \( p \), (2) \( \text{pr}_1(\pi_1, \pi_2), \ldots, \text{pr}_{n-1}(\pi_{n-1}, \pi_n) \) are in \( \text{ground}(p) \), and (3) \( \{\text{pr}_1, \ldots, \text{pr}_{n-1}\} \subseteq A \). Let \( \prec \) be the strict partial order over predicates in \( \Sigma \) defined in Lemma B.1. We lift \( \prec \) over sets of predicate symbols as follows: \( A \prec B \) iff there is a bijection \( \mu \) from \( A \) to \( B \) such that (1) there exists \( a \in A \) such that \( a \prec \mu(a) \), (2) \( a \prec \mu(a) \) or \( a = \mu(a) \) for all \( a \in A \), and (3) \( A \neq B \). We now prove, by induction over \( \prec \), that the transitive closure of the unions of the relations induced by \( \text{pr}_1, \ldots, \text{pr}_{n-1} \) over \( \text{ground}(p) \) is a strict partial order over \( \text{dom}^{\{1, \ldots, n\}} \). From this, it follows that if \( \text{pr}_1(\bar{\pi}_1, \bar{\pi}_2), \text{pr}_2(\bar{\pi}_2, \bar{\pi}_3), \ldots, \text{pr}_{n-1}(\bar{\pi}_{n-1}, \bar{\pi}_n) \) are in \( \text{ground}(p) \), then \( \bar{\pi}_1 \neq \bar{\pi}_n \).

Base Case. For the base case, assume that for all \( a \in A \), there is no predicate symbol \( a' \) (different from \( a \)) such that \( a' \prec a \). From this and \( \text{ORD}(A) \) can be derived from \( p \), it follows that (1) for all \( pr \in A \), there is no rule \( r \) in \( p \) such that \( \text{pred(head(r))} = pr \) and \( \text{body(r)} \neq \emptyset \), and (2) the transitive closure of the relation \( R = \bigcup_{pr \in A} \{(\pi, \tau) \mid \exists r \in p. ((\text{head(r)} = pr(\pi, \tau) \lor \exists r'. \text{head(r)} = r'.\text{pr}(\tau, \tau)) \land \text{body(r)} = \emptyset) \} \wedge |\pi| = |\tau| = |pr|/2 \} \) is strict partial order. From this and \( \{(\pi, \tau) \mid \exists r \in A. \text{pr}(\tau, \tau) \in \text{ground}(p) \cap |\tau| = |\tau| = |pr|/2 \} \subseteq R \), it follows that the transitive closure of the union of the relations induced by \( pr \) over \( \text{ground}(p) \) is a strict partial order over \( \text{dom}^{\{1, \ldots, n\}} \).
**Induction Step.** Assume that the claim holds for all $A' \prec A$. We now prove that it holds also for $A$. From the fact that $ORD(A)$ can be derived from $p$, it follows that there is an annotation $ORD(A')$ and two distinct predicate symbols $pr \in \Sigma$ and $pr' \in A'$ such that

1. $ORD(A')$ can be derived from $p$,
2. $|pr| = |pr'|$,
3. $A = (A' \setminus \{pr'\}) \cup \{pr\}$,
4. $pr' \notin \bigcup_{a \in A} \text{reach}(a)$, and
5. for all rules $r$ in $p$ such that $\text{pred}(\text{head}(r)) = pr$, there are sequences of variables $\tau$ and $\gamma$ such that $\text{head}(r) = pr(\tau, \gamma)$, $pr'(\tau, \gamma) \in \text{body}(r)$, and $|\tau| = |\gamma|$.

There are two cases:

- There are no rules in $p$ such that $\text{pred}(\text{head}(r)) = pr$. In this case, the claim trivially holds from (1) and the induction hypothesis.

- There exists at least one rule in $p$ such that $\text{pred}(\text{head}(r)) = pr$. From this and (5), $\text{head}(r) = pr(\tau, \gamma)$ and $pr'(\tau, \gamma) \in \text{body}(r)$. Hence, $pr' \prec pr$ (since $pr' \notin \text{reach}(pr)$ and $pr \in \text{reach}(pr')$).

From this, (3), and (4), it follows that $A' \prec A$. We now prove that also the relation induced by $A$ over $\text{ground}(p)$ is a strict partial order over $\text{dom}^{\text{pr}}$. Since $pr \in A$, $A = (A' \setminus \{pr\}) \cup \{pr\}$, $pr_1 \in A'$, and $ORD(A)$ can be derived from $p$, it follows that for all rules $r$ in $p$ such that $\text{pred}(\text{head}(r)) = pr$, there are sequences of variables $\tau$ and $\gamma$ and an $i \in \mathbb{N}$ such that $\text{head}(r) = pr(\tau, \gamma)$, $\text{body}(r, i) = pr_1(\tau, \gamma)$, and $|\tau| = |\gamma|$. From this, it follows that the $\{\tau, \gamma \mid pr(\tau, \gamma) \in \text{ground}(p)\} \subseteq \{\tau, \gamma \mid pr_1(\tau, \gamma) \in \text{ground}(p)\}$ from $A' \prec A$ and the induction hypothesis, it follows that the transitive closure of $\bigcup_{pr' \in A'}(\tau, \gamma) \to (\tau, \gamma) \in \text{ground}(p)$ is a strict partial order over $\text{dom}^{\text{pr}}$. This completes the proof of our claim.

This completes the proof. 

**Proposition B.2.** Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain and $p$ be a $(\Sigma, \text{dom})$-ProbLog program. If $\text{DIS}(pr, pr')$ can be derived from $p$, then there is no tuple $\tau$ such that $pr(\tau) \in \text{ground}(p)$ and $pr'(\tau) \in \text{ground}(p)$.

**Proof.** Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain and $p$ be a $(\Sigma, \text{dom})$-ProbLog program. Furthermore, we assume that $\text{DIS}(pr, pr')$ can be derived from $p$. Furthermore, let $\prec$ be the strict partial order defined in Lemma B.1. We lift $\prec$ to pairs of predicate symbols as follows: $(a, b) \prec (a', b')$ if (1) $a = a'$ and $b \prec b'$, (2) $a \prec a'$ and $b = b'$, or (3) $a \prec a'$ and $b \prec b'$. We prove, by induction over $\prec$, that whenever we can derive $\text{DIS}(pr, pr')$ from $p$, there is no tuple $\tau$ such that $pr(\tau) \in \text{ground}(p)$ and $pr'(\tau) \in \text{ground}(p)$.

**Base Case.** Let $(pr, pr')$ be a pair in $\Sigma^2$ such that there is no $(pr_1, pr'_1) \prec (pr, pr')$ and $\text{DIS}(pr, pr')$ can be derived from $p$. From this, it follows that (1) there are no rules $r \in p$ such that $\text{body}(r) \neq \emptyset$ and $\text{pred}(\text{head}(r)) = pr$ or $\text{pred}(\text{head}(r)) = pr'$, and (2) the sets $A = \{\tau \mid pr(\tau) \in p \lor \exists v. v :: pr(\tau) \in p\}$ and $A' = \{\tau \mid \exists p. \text{head}(r) = pr(\tau) \lor \exists v. v :: pr'(\tau) \in p\}$ are disjoint. From this, it follows that there is no tuple $\tau$ such that $pr(\tau) \in \text{ground}(p)$ and $pr'(\tau) \in \text{ground}(p)$.

**Induction Step.** Assume that there is no tuple $\tau$ such that $pr_1(\tau) \in \text{ground}(p)$ and $pr'_1(\tau) \in \text{ground}(p)$ for any $(pr_1, pr'_1) \prec (pr, pr')$ such that $\text{DIS}(pr_1, pr'_1)$ can be derived from $p$. We now prove that there is no tuple $\tau$ such that $pr(\tau) \in \text{ground}(p)$ and $pr'(\tau) \in \text{ground}(p)$. There are three cases:

- There are no rules $r \in p$ such that $\text{body}(r) \neq \emptyset$ and $\text{pred}(\text{head}(r)) = pr$, the annotation $\text{DIS}(pr, pr'_1)$ can be derived from $p$, $pr'_1 \notin \text{reach}(pr')$, $pr' \neq pr'_1$, and for all rules $r \in p$ such that $\text{head}(r) = pr'(\tau)$, $pr_1(\tau) \in \text{body}(r)$. From this, $(pr, pr'_1) \prec (pr, pr')$. From $(pr, pr'_1) \prec (pr, pr')$, $\text{DIS}(pr, pr')$ can be derived from $p$, and the induction’s hypothesis, it follows that there is no tuple $\tau$ such that $pr(\tau) \in \text{ground}(p)$ and $pr'_1(\tau) \in \text{ground}(p)$.

- There are no rules $r \in p$ such that $\text{body}(r) \neq \emptyset$ and $\text{pred}(\text{head}(r)) = a'$, the annotation $\text{DIS}(pr_1, pr')$ can be derived from $p$, $pr_1 \neq \text{reach}(pr)$, $pr \neq pr_1$, and for all rules $r \in p$ such that $\text{head}(r) = pr(\tau)$, $pr_1(\tau) \in \text{body}(r)$. From this, $(pr_1, pr') \prec (pr, pr')$. From $(pr_1, pr') \prec (pr, pr')$, $\text{DIS}(pr_1, pr')$ can be derived from $p$, and the induction’s hypothesis, it follows that there is no tuple $\tau$ such that $pr_1(\tau) \in \text{ground}(p)$ and $pr'(\tau) \in \text{ground}(p)$.

- The annotation $\text{DIS}(pr_1, pr'_1)$ can be derived from $p$, $(pr_1, pr'_1) \cap \text{reach}(pr) \cup \text{reach}(pr') = \emptyset$, $pr_1 \neq pr$, $pr'_1 \neq pr'$, and for all rules $r \in p$, if $\text{head}(r) = pr(\tau)$, $pr_1(\tau) \in \text{body}(r)$ and if...
Proposition B.3. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain and $p$ be a $(\Sigma, \text{dom})$-Prolog program. If UNQ$(p, K)$ can be derived from $p$, then for all tuples $pr(\tau), pr(\tau') \in$ ground($p$), if $\tau(i) = \tau'(i)$ for all $i \in K$, then $\tau = \tau'$. We prove, by induction over $\prec$, that for all annotations $UNQ(p, K)$ that can be derived from $p$, and all tuples $pr(\tau), pr(\tau') \in$ ground($p$), if $\tau(i) = \tau'(i)$ for all $i \in K$, then $\tau = \tau'$. Without loss of generality, in the following we assume that $K \neq \{1, \ldots, |pr|\}$. If this is not the case, then our claim holds trivially.

**Base Case.** Let $pr$ be a predicate such that there is no $pr' \prec pr$. From this and $UNQ(p, K)$ can be derived from $p$, it follows that there are no rules $r$ in $p$ such that $\text{body}(r) \neq \emptyset$ and $\text{head}(r) = pr$, and for all $\tau, \tau'$ in $A = \{\tau \mid pr(\tau) \in \text{p} \lor \exists \tau'' : pr(\tau'') \in \text{p}\}$, if $\tau(i) = \tau'(i)$ for all $i \in K$, then $\tau = \tau'$. From this and $pr(\tau), pr(\tau') \in$ ground($p$), if $\tau(i) = \tau'(i)$ for all $i \in K$, then $\tau = \tau'$.

**Induction Step.** Assume that the claim holds for any $pr' \prec pr$ and $K \subseteq \{1, \ldots, |pr|\}$ such that $UNQ(p, K)$ can be derived from $p$ (we denote this induction hypothesis as (A)). We now prove that the claim holds for $pr$ as well. Without loss of generality, we assume that there is at least one rule $r$ such that $\text{body}(r) \neq \emptyset$ and $\text{head}(r) = pr$ (otherwise the proof is trivial). Assume, for contradiction’s sake, that there are two ground atoms $pr(\tau), pr(\tau') \in$ ground($p$) such that $\tau(i) = \tau'(i)$ for all $i \in K$ but $\tau \neq \tau'$. There are two cases:

- $pr(\tau)$ and $pr(\tau')$ are generated by two ground instances of the same rule $r$. From $\tau \neq \tau'$, it follows that there is $j \notin K$ such that $\tau(j) \neq \tau'(j)$. From this and the fact that both atoms are generated by the same rule $r$, it follows that there is a variable $x$ such that $x \in \{\tau(i) \mid i \notin K\}$ and $x = \tau(j)$, where $\tau = \text{args(\text{head}(r))}$. From this, it follows that $x \in \bigcup_{i \in \text{bound(\text{head}(r)), K}\text{body}^+(r), \mu^i} \text{vars}(l)$. We claim that the value of $x$ is determined by the values of the variables whose positions are in $K$. From this and the fact that $\tau$ and $\tau'$ agree on all values whose positions are in $K$, it follows that $\tau = \tau'$, leading to a contradiction.

We now prove our claim that the value of $x$ is determined by the values of the variables whose positions are in $K$. The value of the variable $x$ is determined by the grounding of one of the atoms in $\text{bound(\text{head}(r), K, \text{body}^+(r), \mu)}$. We first slightly modify the definition of bound. We denote by $bd^h(h, K, L, \mu)$ the function $\bigcup_{b(\tau'y) \in L} b(y) \land h \neq \text{reach}(pr) \land h \neq \text{pr} \land b(\tau'y)$ and by $bd^h(h, K, L, \mu)$, $u(\tau'y, \mu(b)) \subseteq u(\text{args}(h), K)$

where $i > 0$, the function $bd^{i-1}(h, K, L, \mu)$ $\bigcup_{b(\tau'y) \in L} b(y) \land h \neq \text{reach}(pr) \land h \neq \text{pr} \land b(\tau'y)$.

It is easy to see that (1) $\text{bound}(\text{head}(r), K, \text{body}^+(r), \mu^i) = \bigcup_{b(\tau'y) \in L} b(y) \land h \neq \text{reach}(pr) \land h \neq \text{pr} \land b(\tau'y)$ and (2) the fixpoint is always reached in a finite number of steps (bounded by $|\text{body}^+(r)|$). We now prove by induction on $i$ that the groundings of the literals in $bd^i(h, K, L, \mu^i)$, such as $bd^i(h, K, L, \mu)$, is always determined by the values of the variables in $u(\text{args}(\text{head}(r)), K)$. From this, our claim immediately follows. For the base case, let $i = 0$. Then, for any literal $b(\tau'y)$ in $bd^0(h, K, L, \mu)$, it follows that (a) $b \notin \text{reach}(pr)$, (b) $b \neq \text{pr}$, and (c) $u(\tau'y, \mu(b)) \subseteq u(\text{args}(\text{head}(r)), K)$. From this, $L = \text{body}^+(r)$, and $\text{pred}(\text{head}(r)) = pr$, it follows that $b \prec pr$. From this, $UNQ(b, \mu(b))$ can be derived from $p$, and the induction’s hypothesis (A), it follows that the variables in $UNQ(b, \mu(b))$ uniquely determine the grounding of $b(\tau'y)$. From this and $u(\tau'y, \mu(b)) \subseteq u(\text{args}(\text{head}(r)), K)$, it follows that the variables in $u(\text{args}(\text{head}(r)), K)$ uniquely determine the grounding of $b(\tau'y)$. For the induction’s step, assume that the claim hold for all $j < i$ (we denote this induction hypothesis as (A)). Let $b(\tau'y) \in bd^i(h, K, L, \mu^i)$ (without loss of generality, $b(\tau'y) \notin bd^{i-1}(h, K, L, \mu^i)$). From this, it follows that $b \prec pr$ (since $b \notin \text{reach}(pr)$, $b \neq \text{pr}$, and $pr \in \text{reach}(b)$ and there is an $i' \in \text{bound}^{-1}(h, K, L, \mu)$ such that $u(\tau'y, \mu(b)) \subseteq \text{vars}(l')$. From the induction hypothesis (A), it follows that the grounding of $l'$ is directly determined by the grounding of the variables in $u(\text{args}(\text{head}(r)), K)$. Furthermore, from $b \prec pr$, $UNQ(b, \mu(b))$ can be derived from $p$, and the induction hypothesis (A), it follows that the values of the variables in $u(\tau'y, \mu(b))$ uniquely determine the grounding of $b(\tau'y)$. Therefore, the values of the variables in $u(\text{args}(\text{head}(r)), K)$ uniquely determine the grounding of $b(\tau'y)$. This completes the proof of the claim.
• $pr(\pi)$ and $pr(\pi)$ are generated by ground instances of two different rules $r_1$ and $r_2$. From this, $UNQ(pr, K)$ can be derived from $p$, and $K \neq \{1, \ldots, |pr|\}$, it follows that there is a value $i \in K$ such that $\text{args(head}(r_1))(i) \in \text{dom}$, $\text{args(head}(r_2))(i) \in \text{dom}$, and $\text{args(head}(r_1))(i) \neq \text{args(head}(r_2))(i)$. Therefore, there is an $i \in K$ such that $\pi(i) \neq \pi(i)$. This contradicts the fact that $\pi(i) = \pi(i)$ for all $i \in K$.

This completes the proof of our claim. □

B.1.2 Proofs about Propagation Maps

Here we prove some results about propagation maps.

Lemma B.2. Let $\langle \Sigma, \text{dom} \rangle$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $\langle \Sigma, \text{dom} \rangle$-PROBLOG program, $r$ be a rule in $p$, and $l$ be the $i$-th literal in $\text{body}(r)$. Furthermore, let $\mu$ be the (r,l)-vertical map. Given a rule $r' \in \text{ground}(p, r)$, then $\overline{b}(j) = \overline{b}(\mu(j))$ for any $j$ such that $\mu(j)$ is defined, where $\overline{h} = \text{args(head}(r'(i)))$ and $\overline{b} = \text{args(body}(r', i))$.

Proof. Let $\langle \Sigma, \text{dom} \rangle$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $\langle \Sigma, \text{dom} \rangle$-PROBLOG program, $r$ be a rule in $p$, and $l$ be the $i$-th literal in $\text{body}(r)$. Furthermore, let $\mu$ be the (r,l)-vertical map and $r' \in \text{ground}(p, r)$. From the definition of $\text{ground}(p, r)$, it follows that there is an assignment $\Theta$ from variables to elements in $\text{dom}$ such that $r' = r\Theta$. From this, $\overline{b}(j) = \Theta(\text{args}(l(\mu(j))))$ and $\overline{h}(\mu(j)) = \Theta(\text{args}(\text{head}(r))(\mu(j)))$. Note that from the definition of (r,l)-vertical map it follows that $\text{args}(l(j)) = \text{args}(\text{head}(r))(\mu(j))$. From this, $\overline{b}(j) = \overline{h}(\mu(j))$ whenever $\mu(j)$ is defined. □

Lemma B.3. Let $\langle \Sigma, \text{dom} \rangle$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $\langle \Sigma, \text{dom} \rangle$-PROBLOG program, $r$ be a rule in $p$, $l$ be the $i$-th literal in $\text{body}(r)$, and $l'$ be the $j$-th literal in $\text{body}(r)$. Furthermore, let $\mu$ be the (r,l,l')-horizontal map. Given a rule $r' \in \text{ground}(p, r)$, then $\overline{b}(k) = \overline{b}(\mu(k))$ for any $k$ such that $\mu(k)$ is defined, where $\overline{b} = \text{args(body}(r', i))$ and $\overline{b} = \text{args(body}(r', j))$.

Proof. Let $\langle \Sigma, \text{dom} \rangle$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $\langle \Sigma, \text{dom} \rangle$-PROBLOG program, $r$ be a rule in $p$, $l$ be the $i$-th literal in $\text{body}(r)$, and $l'$ be the $j$-th literal in $\text{body}(r)$. Furthermore, let $\mu$ be the (r,l,l')-horizontal map and $r' \in \text{ground}(p, r)$. From the definition of $\text{ground}(p, r)$, it follows that there is an assignment $\Theta$ from variables to elements in $\text{dom}$ such that $r' = r\Theta$. From this, $\overline{b}(j) = \Theta(\text{args}(l(\mu(j))))$ and $\overline{b}(\mu(j)) = \Theta(\text{args}(l'(\mu(j))))$. Note that from the definition of (r,l,l')-vertical map it follows that $\text{args}(l(j)) = \text{args}(l'(\mu(j)))$. From this, $\overline{b}(j) = \overline{b}(\mu(j))$ whenever $\mu(j)$ is defined. □

Proposition B.4. Let $\langle \Sigma, \text{dom} \rangle$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $\langle \Sigma, \text{dom} \rangle$-PROBLOG program, $P = pr_1 \overrightarrow{r_1,i_1} \cdots \overrightarrow{r_{n-1},i_{n-1}} \overrightarrow{r_n}$ be a directed path in $\text{graph}(p)$, and $\nu : N \rightarrow N$ be a mapping. Furthermore, let $P' = a_1 \overrightarrow{r_1,s_1,i_1} a_2 \overrightarrow{r_2,s_2,i_2} \cdots a_{n-1} \overrightarrow{r_{n-1},s_{n-1},i_{n-1}} a_n$ be a directed path in $\text{gg}(p)$ corresponding to $P$. If $P$ $\nu$-downward links to $l$, where $l$ is the $k$-th literal in $r_j$, then $\overline{b}(m) = \overline{b}(\nu(m))$ whenever $\nu(m)$ is defined, where $\overline{b} = \text{args(body}(s_1,i_1))$ and $\overline{b} = \text{args(body}(s_j,k))$.

Proof. Let $\langle \Sigma, \text{dom} \rangle$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $\langle \Sigma, \text{dom} \rangle$-PROBLOG program, $P = pr_1 \overrightarrow{r_1,i_1} \cdots \overrightarrow{r_{n-1},i_{n-1}} \overrightarrow{r_n}$ be a directed path in $\text{graph}(p)$, and $\nu : N \rightarrow N$ be a mapping. Furthermore, let $P' = a_1 \overrightarrow{r_1,s_1,i_1} a_2 \overrightarrow{r_2,s_2,i_2} \cdots a_{n-1} \overrightarrow{r_{n-1},s_{n-1},i_{n-1}} a_n$ be a directed path in $\text{gg}(p)$ corresponding to $P$. Assume that $P$ $\nu$-downward links to $l$, where $l$ is the $k$-th literal in $r_j$. From this, it follows that the function $\mu := \mu' \circ \mu_j \circ \cdots \circ \mu_1$ satisfies $\mu(m) = \nu(m)$ for all $m$ for which $\nu(m)$ is defined, where for $1 \leq h \leq j$, $\mu_h$ is the vertical map connecting $\text{body}(r_h, i_h)$ and $r_h$, and $\mu'$ is the horizontal map connecting $\text{body}(r_{j+1}, l_{j+1})$ with $l$. By repeatedly applying Lemma B.2 to the rules in $P'$, we have that $\overline{b}(m) = \overline{b}(\nu(m))$ whenever $\nu(m)$ is defined, where $\overline{b} = \text{args(body}(s_1,i_1))$, $\overline{b} = \text{args(body}(s_j,k))$, and $\overline{b} = \text{args(body}(s_j,k)) + \phi = \mu_j \circ \cdots \circ \mu_1$. Moreover, by applying Lemma B.3, we have that $\overline{b}(m) = \overline{b}(\nu(m))$ whenever $\nu(m)$ is defined, where $\overline{b} = \text{args(body}(s_j,k))$ and $\overline{b} = \text{args(body}(s_j,k))$. Therefore, we have that $\overline{b}(m) = \overline{b}(\nu(m))$ whenever $\nu(m)$ is defined (by composing the previous results). From this and $\mu(m) = \nu(m)$ for all $m$ for which $\nu(m)$ is defined, it follows that $\overline{b}(m) = \overline{b}(\nu(m))$ whenever $\nu(m)$ is defined. □

Proposition B.5. Let $\langle \Sigma, \text{dom} \rangle$ be a database schema, $p$ be a $\langle \Sigma, \text{dom} \rangle$-PROBLOG program, $P = pr_1 \overrightarrow{r_1,i_1} \cdots \overrightarrow{r_{n-1},i_{n-1}} \overrightarrow{r_n}$ be a directed path in $\text{graph}(p)$, and $\nu : N \rightarrow N$ be a mapping. Furthermore, let $P' = a_1 \overrightarrow{r_1,s_1,i_1} a_2 \overrightarrow{r_2,s_2,i_2} \cdots a_{n-1} \overrightarrow{r_{n-1},s_{n-1},i_{n-1}} a_n$ be a directed path in $\text{gg}(p)$ corresponding to $P$. If $P$ $\nu$-upward links to $l$, where $l$ is the $k$-th literal in $r_j$, then $\overline{b}(m) = \overline{b}(\nu(m))$ whenever $\nu(m)$ is defined, where $\overline{b} = \text{args(head}(r_{n-1}))$ and $\overline{b} = \text{args(head}(r_{j-1}))$. □
Proof. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $(\Sigma, \text{dom})$-PROBLOG program, $P = pr_1 \rightarrow_{r_1, t_1} \ldots \rightarrow_{r_{n-1}, t_{n-1}} pr_n$ be a directed path in $\text{graph}(p)$, and $\nu : \mathbb{N} \rightarrow \mathbb{N}$ be a mapping. Furthermore, let $P' = a_1 \rightarrow_{r_1, t_1} a_2 \rightarrow_{r_2, t_2} \ldots \rightarrow_{r_{n-1}, t_{n-1}} a_n$ be a directed path in $\text{graph}(p)$ corresponding to $P$. Assume that $P$ $\nu$-upward links to $l$, where $l$ is the $k$-th literal in $r_j$. From this, it follows that the function $\mu := \mu_1^{-1} \circ \mu_2^{-1} \circ \ldots \circ \mu_n^{-1}$ satisfies $\mu(k) = \nu(k)$ for all $k$ for which $\nu(k)$ is defined, where $\mu_k$ is the $(r_k, t_k)$-vertical map, for $j < h \leq n - 1$, and $\mu'$ is the $(r_j, t_j)$-vertical map. By repeatedly applying Lemma B.2 to the rules in $P'$, we obtain that $b_{j+1}(\phi^{-1}(m)) = b_{j}(m)$ whenever $\phi^{-1}(m)$ is defined, where $b_{j+1} = \text{args(body(s_{j+1}, t_{j+1}))}$, $b_0 = \text{args(head(s_{j-1}))}$, and $\phi^{-1} = \mu_j^{-1} \circ \ldots \circ \mu_n^{-1}$. Furthermore, by applying Lemma B.2 to the $(r_j, t_j)$-vertical map, we have that $\overline{\text{args}}(\nu^{-1}(m)) = b_{j+1}(m)$ whenever $\nu^{-1}(m)$ is defined, where $b_{j+1} = \text{args(head(s_j))}$ and $\overline{\text{args}} = \text{args(body(s_j, k))}$. From this and $\mu(m) = \nu(m)$ for all $m$ for which $\nu(m)$ is defined, it follows that $\overline{b}_c(m) = \overline{\text{args}}(\nu(m))$ whenever $\nu(m)$ is defined. □

B.1.3 Proofs about Connected Rules

We now prove that, for strongly connected rules, the grounding of a rule’s head uniquely determines the grounding of the rule’s body (Proposition B.6), whereas for weakly connected rules the grounding of one of the atoms in the body determines the rule’s grounding (Proposition B.7).

Proposition B.6. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $(\Sigma, \text{dom})$-PROBLOG program, $r$ be a rule in $p$, and $T$ be the template containing all annotations that can be derived from $p$. If $r$ is strongly connected for $T$, then for all $r_1, r_2 \in \text{ground}(p, r)$, if $\text{head}(r_1) = \text{head}(r_2)$, then $r_1 = r_2$.

Proof. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $(\Sigma, \text{dom})$-PROBLOG program, $r$ be a rule in $p$, and $T$ be the template containing all annotations that can be derived from $p$. Furthermore, we assume that there are join trees $J_1, \ldots, J_n$ such that (a) the trees cover all literals in $\text{body}(r)$, and (b) for each $1 \leq i \leq n$, $J_i$ is strongly connected for $T$. Finally, let $r_1, r_2 \in \text{ground}(p, r)$ be two ground rules such that $\text{head}(r_1) = \text{head}(r_2)$. Assume, for contradiction’s sake, that $r_1 \neq r_2$. Therefore, there is a position $1 \leq i \leq |\text{body}(r)|$ such that $\text{body}(r_1, i) \neq \text{body}(r_2, i)$. Let $J = \langle N, E, \text{root}, \lambda \rangle$ be one of the join trees that cover $l = \text{body}(r_i)$. We claim that the grounding of $\text{head}(r)$ determines the grounding of all literals in $J$. From this, it follows that $\text{body}(r_1, i) = \text{body}(r_2, i)$ leading to a contradiction.

We now prove our claim that the grounding of $\text{head}(r)$ determines the grounding of all literals in $J$. Let $V(J, i)$ be the set of all the nodes in $J$ at distance at most $i$ from the root $\text{root}$. Furthermore, we denote by $\text{ground}(r, r', J, i)$ the set $\{\text{body}(r', j) \mid \text{body}(r, j) \in V(J, i)\}$. We prove, by induction on $i$, that for all $i$ and all ground rules $r_1$ and $r_2$ instances of $r$, if $\text{head}(r_1) = \text{head}(r_2)$, then $\text{ground}(r_1, r_2, J, i) = \text{ground}(r_2, r_1, J, i)$. From this, it follows that the grounding of $\text{head}(r)$ determines the grounding of all literals in $J$.

Base case. For $i = 0$, there is a $j$ such that $V(J, 0) = \{\text{body}(r, j)\}$. From this, it follows that $\text{ground}(r, r_1, J, 0) = \{\text{body}(r_1, j)\}$ and $\text{ground}(r_2, r_2, J, 0) = \{\text{body}(r_2, j)\}$. Furthermore, $\text{args}(J, \text{body}(r_1, j), \varnothing) = \emptyset$. There are two cases depending on whether $\text{body}(r, j)$ is a positive literal or not:

1. If $\text{body}(r, j)$ is a positive literal of the form $a(T)$, then $\text{body}(r_1, j) = a(T_1)$ and $\text{body}(r_2, j) = a(T_2)$. From the fact that $J$ is strongly connected for $T$, it follows that there is a set of variables $K \subseteq \{x \mid x_1(T) \in \text{support}(a(T))\}$ such that $UNQ(a, K) \in T$. From $\text{support}(a(T)) = \text{vars(\text{head}(r))} \cup \{x \mid (x = c) \in \text{str}(r) \wedge c \in \text{dom}\}$ and $\text{head}(r_1) = \text{head}(r_2)$, it follows that the values assigned to the variables associated to the indexes in $K$ are the same in $r_1$ and $r_2$. From this, $UNQ(a, K) \in T$, $T$ has been derived from $p$, $\{a(T_1), a(T_2)\} \subseteq \text{ground}(p)$, and Proposition B.3, it follows $\varnothing = \{a(T_1), a(T_2)\}$ and $\text{ground}(r_1, r_1, J, 0) = \text{ground}(r_2, r_2, J, 0)$.

2. If $\text{body}(r, j)$ is a negative literal of the form $\neg a(T)$, then $\text{body}(r_1, j) = \neg a(T_1)$ and $\text{body}(r_2, j) = \neg a(T_2)$. From the fact that $J$ is strongly connected for $T$, it follows that $\text{vars(\neg a(T))} \subseteq \text{support}(\neg a(T))$. From this, $\text{support}(\neg a(T)) = \text{vars(\text{head}(r))} \cup \{x \mid (x = c) \in \text{str}(r) \wedge c \in \text{dom}\}$, and $\text{head}(r_1) = \text{head}(r_2)$, it follows that the values of the variables in $\text{support}(\neg a(T))$ are the same in $r_1$ and $r_2$. From this, it follows that $\varnothing = \{a(T_1), a(T_2)\}$ and $\text{ground}(r_1, r_1, J, 0) = \text{ground}(r_2, r_2, J, 0)$.

Induction Step. Assume that for all $j < i$ and all ground rules $r_1, r_2 \in \text{ground}(p, r)$, if $\text{head}(r_1) = \text{head}(r_2)$, then $\text{ground}(r_1, r_1, J, j) = \text{ground}(r_2, r_2, J, j)$. We now show that $\text{ground}(r_1, r_1, J, i) = \text{ground}(r_2, r_2, J, i)$. Assume, for contradiction’s sake, that this is not the case, namely $\text{ground}(r_1, r_1, J, i) \neq \text{ground}(r_2, r_2, J, i)$. From the definition of $\text{ground}(r, r', J, i)$, it follows that $\text{ground}(r_1, r_1, J, i) = \text{ground}(r_1, r_1, J, i - 1) \cup \{\text{body}(r_1, j) \mid \text{body}(r, j) \in V(J, i - 1)\}$ and $\text{ground}(r_2, r_2, J, i) = \text{ground}(r_2, r_2, J, i - 1) \cup \{\text{body}(r_2, j) \mid \text{body}(r, j) \in V(J, i - 1)\}$. From this, the induction’s hypothesis,
170

Appendix B. Proofs for Chapter 4

and ground(r, r1 , J, i) 6= ground(r, r2 , J, i), it follows that {body(r1 , j) | body(r, j) ∈ V (J, i) \ V (J,
i − 1)} 6= {body(r2 , j) | body(r, j) ∈ V (J, i) \ V (J, i − 1)}. Therefore, there is a j such that body(r,
j) ∈ V (J, i) \ V (J, i − 1) and body(r1 , j) 6= body(r2 , j). There are two cases, depending on whether
body(r, j) is a positive literal:
1. If body(r, j) = a(x), then body(r1 , j) = a(c1 ), body(r2 , j) = a(c2 ), and c1 6= c2 . From the fact
that J is strongly connected for T , it follows that there is a set of variables K ⊆ {i | x(i) ∈
support(a(x))} such that
S UNQ(a, K) ∈ U . From support(a(x)) = vars(head(r)) ∪ {x | (x = c) ∈
cstr(r) ∧ c ∈ dom} ∪ l∈V (J,i−1) vars(l), ground(r, r1 , J, i − 1) = ground(r, r2 , J, i − 1) (from
the induction’s hypothesis), and head(r1 ) = head(r2 ), it follows that the values assigned to the
variables associated to the indexes in K are the same in r1 and r2 . From this, UNQ(a, K) ∈ T ,
T has been derived from p, {a(c1 ), a(c2 )} ⊆ ground(p), and Proposition B.3, it follows that
c1 = c2 leading to a contradiction.
2. If body(r, j) = ¬a(x), then body(r1 , j) = ¬a(c1 ), body(r2 , j) = ¬a(c2 ), and c1 6= c2 . From the
From
fact that J is strongly connected for T , it follows that vars(¬a(x)) ⊆ support(a(x)).
S
support(a(x)) = vars(head(r)) ∪ {x | (x = c) ∈ cstr(r) ∧ c ∈ dom} ∪ l∈V (J,i−1) vars(l),
ground(r, r1 , J, i − 1) = ground(r, r2 , J, i − 1) (from the induction’s hypothesis), it follows that
the values assigned to the variables in support(a(x)) are the same in r1 and r2 . From this, it
follows that c1 = c2 leading to a contradiction.
Since both cases lead to a contradiction, ground(r, r1 , J, i) = ground(r, r2 , J, i).
Proposition B.7. Let hΣ, domi be a database schema such that dom is a finite domain, p be a
(Σ, dom)-ProbLog program, r be a rule in p, and T be the template containing all annotations
derived from p. If r is weakly connected for T , then for all r1 , r2 ∈ ground(p, r), if there is an
1 ≤ i ≤ |body(r)| such that body(r1 , i) = body(r2 , i), then r1 = r2 .
Proof. Let hΣ, domi be a database schema such that dom is a finite domain, p be a (Σ, dom)ProbLog program, r be a rule in p, and T be the template containing all annotations derived from
p. Furthermore, we assume that r is weakly connected for T , namely there is a join tree J = hN, E,
root, λi such that (a) J is weakly connected for T , (b) N ⊆ body + (r), and (c) all literals in body(r)\N
are (r, U, N )-strictly guarded. Let r1 , r2 ∈ ground(p, r) be two ground rules such that there is an
1 ≤ i ≤ |body(r)| such that body(r1 , i) = body(r2 , i), and let l be the literal body(r, i). There are two
cases:
1. body(r, i) ∈ N . Given a node n in a join tree J, we denote by adjacent(n, i), where i ∈ N, the
sub-tree obtained by considering only the nodes reachable from n using at most i edges. We
claim that for all j and all nodes in adjacent(l, j), the corresponding ground atoms in r1 and
r2 are the same. From this and the fact that there is a j such that adjacent(l, j) = J, it follows
that for all i ∈ {j | body(r, j) ∈ N }, body(r1 , i) = body(r2 , i). From this and the fact that
the literals in body(r) \ N are (r, T , N )-strictly guarded, it follows
that for all i ∈ {j | body(r,
S
j) ∈ body(r) \ N }, body(r1 , i) = body(r2 , i) (since vars(l) ⊆ l0 ∈N ∩body+ (r) vars(l0 ) ∪ {x | (x =
c) ∈ cstr(r) ∧ c ∈ dom} for any literal l in body(r) \ N ). Therefore, body(r1 ) = body(r2 ).
2. body(r, i) 6∈ N . From this and the fact that r is weakly connected, it follows that there a j
such that body(r, j) = a(x), body(r, j) ∈ N , and a UNQ(a, K) ∈ T such that {x(i) | i ∈ K} ⊆
vars(body(r, i)). From this, T has been derived from p, body(r1 , i) = body(r2 , i), {body(r1 , j),
body(r2 , j)} ⊆ ground(p), and Proposition B.3, it follows that body(r1 , j) = body(r2 , j). We
proved above that if body(r1 , j) = body(r2 , j) and body(r, j) ∈ N , then body(r1 ) = body(r2 ).
Therefore, body(rS
1 ) = body(r2 ).
From vars(head(r)) ⊆ l∈body+ (r) vars(l) and body(r1 ) = body(r2 ), it follows that head(r1 ) = head(r2 ).
Therefore, r1 = r2 .
We now prove, by induction on j, that for all j and all nodes in adjacent(l, j), the corresponding
ground atoms in r1 and r2 are the same.
Base Case. The sub-tree adjacent(l, 0) contains only the node l = body(r, i). Since body(r1 , i) =
body(r2 , i), the claim holds for the base case.
Induction step. Assume now that the claim holds for all j 0 < j. We now prove that the claim holds
also for j. The sub-tree adjacent(l, j) is obtained by extending adjacent(l, j − 1) with either edges of
L

L

1
2
the form n1 −−→
n01 , where n01 ∈ adjacent(l, j − 1), or n02 −−→
n2 , where n02 ∈ adjacent(l, j − 1). In
both cases, from the induction hypothesis, the ground literals corresponding to n01 and n02 are l10 and
l20 and they are the same in r1 and r2 . From the definition of weakly connected join tree, there are
variables K1 ⊆ L1 and K2 ⊆ L2 such that UNQ(pred(n1 ), K1 ) ∈ T and UNQ(pred(n2 ), K2 ) ∈ T .
From this, T has been derived from p, the fact that the value of K1 and K2 are fixed by l10 and l20 ,
and Proposition B.3, it follows that the ground atoms corresponding to n1 and n2 are the same in
r1 and r2 (because the values in K1 and K2 determines all values in n1 and n2 ).


B.1.4 Acyclicity Proof

We now prove our first key result, namely that the ground graph associated to an acyclic ProbLog program is a forest of poly-trees, i.e., its undirected version does not contain simple cycles (which are cycles without repetitions of edges and vertices other than the starting and ending vertices).

Theorem B.1. Let (Σ, dom) be a database schema such that dom is a finite domain, and p be a (Σ, dom)-acyclic ProbLog program. The graph gg(p) is a forest of poly-trees, i.e., its undirected version does not contain simple cycles.

Proof. Let (Σ, dom) be a database schema such that dom is a finite domain, and p be a (Σ, dom)-acyclic ProbLog program. We prove that the undirected version of gg(p) is acyclic. Assume, for contradiction’s sake, that this is not the case, namely there is a simple cycle C := n_{1} → n_{2} → ... → n_{k} → n_{1} in the undirected version of gg(p). There are two cases: (1) there is a directed simple cycle in gg(p) that directly corresponds to C, or (2) there are n directed and reversed paths P_{1}, ..., P_{n} in gg(p) that induce a simple cycle C in the undirected version of gg(p) (and P_{1}, ..., P_{n} does not correspond to any directed simple cycle in gg(p)). From the first case, it follows that there is a directed cycle in graph(p), whereas from the second case, it follows that there is an undirected cycle in graph(p).

Directed Cycle. Assume that C := n_{1} → n_{2} → ... → n_{k} → n_{1} directly corresponds to a directed cycle in gg(p). Then, the cycle has form a_{1}(τ_{1}) → a_{2}(τ_{2}) ... a_{n}(τ_{n}) → a_{n+1}(τ_{n+1}), where n = k and a_{n+1}(τ_{n+1}) = a_{1}(τ_{1}). From this, it follows that there are rules r_{1}, ..., r_{n} and ground rules s_{1}, ..., s_{n} such that: (1) there is a directed cycle a_{1} → a_{2} → ... → a_{n} in p’s dependency graph graph(p), and (2) for all 1 ≤ i ≤ n, s_{i} ∈ ground(p, r_{i}) and head(s_{i}) = o_{i+1}(c_{i+1}).

From this, it follows that p’s dependency graph graph(p) contains a directed cycle C’ := a_{1} → a_{2} → ... → a_{n} → a_{1}. Note that C’ may contain loops (i.e., it is not simple). Since p is acyclic and C’ is a directed cycle in graph(p), it follows that there is a directed unsafe structure S that covers C’ and is T-guarded, where T is the template containing all annotations that can be derived from p. Without loss of generality, we assume that S = C'. From S covers C’ and S is T-guarded, it follows that there is an ordering annotation ORD(O) ∈ T such that there are integers 1 ≤ y_{1} < y_{2} < ... < y_{k} = n, literals o_{1}(τ_{1}), ..., o_{k}(τ_{k}) (where o_{i} ∈ O and |τ_{i}| = |o_{i}|), a non-empty set K ⊆ {1, ..., |pr|}, and a bijection ν : K → {1, ..., |n|/2} such that for each

0 ≤ k < e, (1) pr_{yk} → a_{1}(τ_{1}) → a_{2}(τ_{2}) → ... → a_{n}(τ_{n}) → pr_{yk+1} ν-downward connects to o_{k+1}(τ_{k+1}), and

(2) pr_{yk+1} → a_{1}(τ_{1}) → a_{2}(τ_{2}) → ... → a_{n}(τ_{n}) → pr_{yk+1} ν'-upward connects to o_{k+1}(τ_{k+1}), which ν-downward connects to o_{k+1}(τ_{k+1}), for each 0 ≤ k < e, it follows that (1) ground atoms o_{1}(b_{1}), ..., o_{k}(b_{k+1}) are in ground(p), and (2) for all 1 ≤ w ≤ |pr|/2, both b_{1}(w) = args(body(s_{1}, i_{1}))(ν'(w)) and b_{k+1}(w) = args(head(s_{k+1}))(ν'(w)) hold. From this, ORD(O) ∈ T, T has been derived from p, and Proposition B.1, it follows that b_{1} ≠ b_{k+1}. From this and b_{1}(w) = args(body(s_{1}, i_{1}))(ν'(w)) and b_{k+1}(w) = args(head(s_{k+1}))(ν'(w)) for all 1 ≤ w ≤ |pr|/2, it follows that body(s_{1}, i_{1}) ≠ head(s_{k+1}). This contradicts head(s_{k})(a_{1}(τ)) and body(s_{1}, i_{1}) = a_{1}(τ) (which directly follows from the existence of the cycle C).

Undirected Cycle. Assume that C does not directly correspond to any directed simple cycle in gg(p). From this, it follows that there are n directed and reversed paths P_{1}, ..., P_{n} in gg(p) such that P_{1}, ..., P_{n} correspond to the simple cycle C in the undirected version of gg(p). From this, it follows that P_{1}, ..., P_{n} form an undirected cycle in gg(p). Therefore:

1. For 1 ≤ j ≤ k such that P_{j} is a directed path, there is a directed path D_{j} := a_{1}(τ_{1}) → a_{2}(τ_{2}) → ... → a_{n}(τ_{n}) → a_{1}(τ_{1}), in p’s dependency graph graph(p), where P_{j} is a_{1}(τ_{1}) → a_{2}(τ_{2}) → ... → a_{n}(τ_{n}) → a_{1}(τ_{1}).

2. For 1 ≤ j ≤ k such that P_{j} is a reversed path, there is a reversed path D_{j} := a_{1}(τ_{1}) → a_{2}(τ_{2}) → ... → a_{n}(τ_{n}) → a_{1}(τ_{1}), in p’s dependency graph graph(p), where P_{j} is a_{1}(τ_{1}) → a_{2}(τ_{2}) → ... → a_{n}(τ_{n}) → a_{1}(τ_{1}).

3. D_{1}, ..., D_{n} form an undirected cycle in graph(p).

4. D_{1}, ..., D_{n} is not a directed cycle. Indeed, if D_{1}, ..., D_{n} is a directed cycle, then either C would correspond to a directed simple cycle in gg(p) (contradicting our assumption that C does not directly correspond to any directed simple cycle in gg(p) or C would contain loops (contradicting our assumption that C is a simple cycle).
Since $p$ is acyclic and $D_1, \ldots, D_n$ forms an undirected cycle in $\text{graph}(p)$ that is not a directed cycle, it follows that there is an unsafe structure $S$ that covers $D_1, \ldots, D_n$ and is $T$-guarded, where $T$ is the set of all annotations derived from $p$. We assume that there are values $i, a, b, c$ such that:

1. $S = (D_a, D_b, D_c, U)$,
2. $a = i, b = (i + 1)\%n$, and $c = (i + 2)\%n$ (i.e., $P_a$, $P_b$, and $P_c$ are adjacent in the cycle), and
3. $U$ is the undirected path containing all $D_i$'s that are different from $D_a$, $D_b$, and $D_c$.

Note that the previous assumption is without loss of generality. Since $S$ covers $C$, we can always pick the paths in $gg(p)$ inducing $C$ in such a way that they match the guarded structure $S$.

Let $P_a$ be $a_1(\pi_1) \rightarrow a_2(\pi_2) \rightarrow a_3(\pi_3) \rightarrow \ldots \rightarrow a_{n_u}(\pi_{n_u})$, $P_b$ be $b_1(\bar{b}_1) \rightarrow b_2(\bar{b}_2) \rightarrow b_3(\bar{b}_3) \rightarrow \ldots \rightarrow b_{n_u}(\bar{b}_{n_u})$, and $P_c$ be the path $c_1(\nu_1) \rightarrow c_2(\nu_2) \rightarrow c_3(\nu_3) \rightarrow \ldots \rightarrow c_{n_u}(\nu_{n_u})$. From $S = (D_a, D_b, D_c, U)$ and $S$ covers $C$, it follows that $a_1(\pi_1) = b_1(\bar{b}_1)$ and $b_{n_u}(\bar{b}_{n_u}) = c_{n_u}(\nu_{n_u})$. From $S = (D_a, D_b, D_c, U)$, $S$ covers $C$, and $S$ is $T$-guarded, there are two cases:

1. $(D_a, D_b)$ is $T$-head guarded. Therefore, $D_a$ and $D_b$ are non-empty. There are two cases:

   (a) $D_a = D_b$. From this, it follows that (1) $P_a$ and $P_b$ are directed paths in $gg(p)$, (2) $a_1(\pi_1) = b_1(\bar{b}_{1})$, (3) $n_u = n_a$, and (4) $D_a$ and $D_b$ are head-connected. From this and $(D_a, D_b)$ is head-guarded, it follows that the rules in $D_a$ and $D_b$ are weakly connected for $T$. We claim that $s_k = s_k'$ for any $1 \leq h \leq n_u$. From this, it follows that $P_a = P_b$. This contradicts the fact that $P_1, \ldots, P_a$ induces a simple cycle in the undirected version of $gg(p)$ (because the cycle is not simple).

   We now prove, by induction on $d$, that $s_d = s_d''$ and $a_d(\pi_d) = b_d(\bar{b}_d)$.

   **Base Case.** Assume that $d = 1$. From $r_1 = r_1'$, $r_1$ is weakly connected for $T$, $\text{body}(s_1, \nu_1) = \text{body}(s_1', \nu_1')$, $s_{d} \in \text{ground}(p, r_d)$, $s_d'' \in \text{ground}(p, r_d')$, and Proposition B.7, it follows that $s_1 = s_1'$. From this, $a_1(\pi_1) = \text{head}(s_1)$, and $b_1(\bar{b}_1) = \text{head}(s_1')$, it follows that $a_1(\pi_1) = b_1(\bar{b}_1)$.

   **Induction Step.** Assume that the claim holds for all $d' < d$. We now prove that $s_d = s_d''$ and $a_d(\pi_d) = b_d(\bar{b}_d)$ hold as well. From the induction’s hypothesis, it follows that $\text{head}(s_d') = \text{head}(s_d'')$. From this and $gg(p)$'s definition, it follows that $\text{body}(s_d, \nu_1) = \text{body}(s_d', \nu_1')$. From this, $r_d' = r_d''$, $s_d'' \in \text{ground}(p, s_d'')$, and Proposition B.4.7, it follows that $s_d = s_d''$. From this, $a_d(\pi_d) = \text{head}(s_d')$, and $b_d(\bar{b}_d) = \text{head}(s_d'')$, it follows that $a_d(\pi_d) = b_d(\bar{b}_d)$.

   (b) $D_a \neq D_b$. From this and $(D_a, D_b)$ are head-guarded, it follows that there is an annotation $DIS(pr, pr') \in T$ a set $K \subseteq \{1, \ldots, a_1\}$, and a bijection $\nu: K \rightarrow \{1, \ldots, |pr'|\}$ such that $D_a \nu$-downward links to $pr$ and $D_b \nu$-downward links to $pr'$. From $D_a \nu$-downward links to $pr$, $P_a$ is a ground instance of $D_a$, and Proposition B.4.4, it follows that there is a positive literal $pr(\pi)$ in the body of one of the ground rules such that $a_1(\pi_1)(k) = \nu(\nu(K))$ for any $k \in K$. From this and the definition of $ground$, it follows that $pr(\pi) \in ground(p)$. From $D_a \nu$-downward links to $pr'$, $P_b$ is a ground instance of $D_b$, and Proposition B.4.4, it follows that there is a positive literal $pr'(\pi')$ in the body of one of the ground rules such that $b_1(\bar{b}_1)(k) = \nu'(\nu(K))$ for any $k \in K$. From this and the definition of $ground$, it follows that $pr'(\pi') \in ground(p)$. Finally, from the fact that $P_b$ and $P_a$ are head-connected, it follows that $a_1(\pi_1) = b_1(\bar{b}_1)$. From this, $a_1(\pi_1)(k) = \nu(\nu(K))$ for any $k \in K$, and $b_1(\bar{b}_1)(k) = \nu'(\nu(K))$ for any $k \in K$, it follows that $\nu = \nu'$. From this, $pr(\pi) \in ground(p)$ and $pr'(\pi) \in ground(p)$. This contradicts Proposition B.2, since $DIS(pr, pr') \in T$ and $T$ has been derived from $p$.

2. $(D_b, D_c)$ is $T$-tail guarded. Therefore, $D_b$ and $D_c$ are non-empty. There are two cases:

   (a) $D_b = D_c$. From this, it follows that (1) $P_a$ and $P_b$ are directed paths in $gg(p)$, (2) $c_{n_u}(\pi_{n_u}) = b_{n_u}(\bar{b}_{n_u})$, (3) $n_u = n_c$, and (4) $D_b$ and $D_c$ are tail-connected. From this and $(D_b, D_c)$ is tail-guarded, it follows that (1) $D_a$ and $D_b$ are tail-connected, and (2) the rules in $D_b$ and $D_c$ are strongly connected for $T$. We claim that $s_d' = s_d''$ for any $1 \leq h \leq n_c$. From this, it follows that $P_a = P_b$. This contradicts the fact that $P_1, \ldots, P_a$ induces a simple cycle in the undirected version of $gg(p)$ (because the cycle is not simple).

   We now prove, by induction on $d$, that $s''_{n_u - d} = s_{n_u - d}$ and head($s''_{n_u - d}$) = head($s_{n_u - d}$).

   **Base Case.** Assume $d = 0$. From $r''_{n_u} = r_{n_u}'$, $r''_{n_u}$ is strongly connected for $T$, $\text{head}(s''_{n_u}) = \text{head}(s_{n_u}') = c_{n_u}(\pi_{n_u})$, $s''_{n_u} \in \text{ground}(p, r''_{n_u})$, $s_d' \in \text{ground}(p, r_d')$, and Proposition B.6.6, it follows that $s_1 = s_1'$. From this, $a_1(\pi_1) = \text{head}(s_1)$, and $b_1(\bar{b}_1) = \text{head}(s_1')$, it follows that $b_{n_u}(\bar{b}_{n_u}) = c_{n_u}(\pi_{n_u})$.

   **Induction Step.** Assume that the claim holds for all $d' < d$. We now prove that $s''_{n_u - d} = s_{n_u - d}$ and $c_{n_u - d}(\pi_{n_u - d}) = b_{n_u - d}(\bar{b}_{n_u - d})$ hold as well. From the induction’s hypothesis, it follows that $s'_{n_u - d + 1} = s_{n_u - d + 1}$. From this and $gg(p)$'s definition, it follows that
B.1. Acyclicity of the ground graph

\[ \text{head}(s'_{n_k-d}, i_{n_k-d}) = \text{head}(s'_{n_k-d}, i_{n_k-d}) \]. From this, \( r''_{n_k-d} = r'_{n_k-d}, s''_{n_k-d} = s'_{n_k-d}, r''_{n_k-d} \) is strongly connected for \( T \), \( s''_{n_k-d} \in \text{ground}(p, r''_{n_k-d}), s''_{n_k-d} \in \text{ground}(p, s_{n_k-d}) \), and Proposition B.6, it follows that \( s''_{n_k-d} = s'_{n_k-d} \). From this, \( c_{n_k-d}(s''_{n_k-d}) = c_{d}(s'_{n_k-d}) \), and \( b_{n_k-d}(b_{n_k-d}) = \text{head}(s'_{n_k-d}) \), it follows that \( c_{n_k-d}(b_{n_k-d}) = b_{n_k-d} \). From this and \( (D_b, D_c) \) is tail-guarded, it follows that there is an annotation \( \text{DIS}(pr, pr') \in T \), a set \( K \subset \{1, \ldots, |pr|\} \), and a bijection \( \nu : K \rightarrow \{1, \ldots, |pr|\} \), such that \( D_b \nu \)-upward links to \( pr \) and \( D_c \nu \)-upward links to \( pr' \). From \( D_b \nu \)-upward links to \( pr, P_b \) is a path in the ground graph corresponding to \( D_b \), and Proposition B.5, it follows that a positive literal pr(\( T \)) in the body of one of the ground rules such that \( b_{n_k}((b_{n_k})(k) = \text{DIS}(K)) \) is any \( k \in K \). From this and the definition of \text{ground}, it follows that \( pr(T) \in \text{ground}(p) \). From \( D_b \nu \)-upward links to \( pr', P_c \) is a path in the ground graph corresponding to \( D_c \), and Proposition B.5, it follows that a positive literal \( pr'(\( T \)) in the body of one of the ground rules such that \( c_{n_k}((c_{n_k})(k) = \text{DIS}(K)) \) is any \( k \in K \). From this and the definition of \text{ground}, it follows that \( pr'(\( T \)) \in \text{ground}(p) \). Finally, from the fact that \( P_b \) and \( P_c \) are tail-connected, it follows that \( b_{n_k}((b_{n_k})(k) = \text{DIS}(K)) \) is any \( k \in K \), and \( c_{n_k}((c_{n_k})(k) = \text{DIS}(K)) \) is any \( k \in K \), it follows that \( \nu = \text{DIS}(K) \). Therefore, \( pr(T) \in \text{ground}(p) \) and \( pr'(\( T \)) \in \text{ground}(p) \). This contradicts Proposition B.2 since \( \text{DIS}(pr, pr') \in T \) and \( T \) has been derived from \( p \).

This completes the proof of our claim.

\[ \square \]

B.1.5 Auxiliary Results

Here we prove two auxiliary results that help in establishing that programs are acyclic. In particular, Proposition B.8 states pre-conditions that allows reducing the guardedness of a complex undirected structure to the guardedness of simpler structures. Similarly, Proposition B.9 states pre-conditions that allows reducing the guardedness of a complex directed structure to the guardedness of a sequence of simpler structures. Note that Proposition B.9 can be easily extended to support (1) different forms of cycle combination, and (2) combinations of non-self-loop cycles.

Proposition B.8. Let \( p \) be a ProbLog program, \( T \) be the template containing all annotations that can be derived from \( p \), \( C \) be an undirected cycle in \( \text{graph}(p) \), \( S = (D_1, D_2, D_3, U) \) be an undirected unsafe structure, and \( U_1 ' \) and \( U_2 ' \) be undirected cycles in \( \text{graph}(p) \). If (1) \( C \) is equivalent to \( D_1 U_1 ' U_2 ' D_3 \), (2) \( S \) is \( T \)-guarded, then there is an undirected unsafe structure that covers \( C \) and is \( T \)-guarded.

Proof. Let \( p \) be a ProbLog program, \( T \) be the template containing all annotations that can be derived from \( p \), \( C \) be an undirected cycle in \( \text{graph}(p) \), and \( S = (D_1, D_2, D_3, U) \) be an undirected structure, and \( U_1 ' \) and \( U_2 ' \) be undirected cycles in \( \text{graph}(p) \). We assume that (1) \( C \) is equivalent to \( D_1 U_1 ' U_2 ' D_3 \), (2) \( S \) is \( T \)-guarded. We define the undirected unsafe structure \( S' = (D_1, D_2, D_3, (U_1 ' U_2 ')) \). From our assumption \( C \) is equivalent to \( D_1 U_1 ' U_2 ' D_3 \). Thus, \( S' \) covers \( C \). Furthermore, since \( S \) and \( S' \) agree on all directed paths, which are the only ones that determine whether an undirected structure is guarded, it follows that \( S' \) is \( T \)-guarded.

In Proposition B.9, use the notion of a directed cycle \( C \) guarded for a set of predicates \( O \) and a mapping \( \nu \). This notion is similar to the notion of guarded directed unsafe structure restricted \( O \) and \( \nu \).

Proposition B.9. Let \( p \) be a ProbLog program, \( C_1 = \text{pr}_1(r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow r_n \rightarrow r_{n-1} \rightarrow r_{n-2} \rightarrow \ldots \rightarrow r_2 \rightarrow r_1) \) be a directed cycle in \( \text{graph}(p) \), \( O \) be a set of predicate symbols, \( \text{pr}_1 \) a predicate symbol such that for all \( o \in O, |o| = |o|, K \) be a non-empty set \( K \subseteq \{1, \ldots, |pr_1|\} \), and \( \nu \) be a predicate symbol such that for all \( o \in O, |o| = |o|, K \) be a non-empty set \( K \subseteq \{1, \ldots, |pr_1|\} \), \( \nu \) be a bijection \( \nu : K \rightarrow \{1, \ldots, |o|\} \) and \( \nu \) be the bijection \( \nu(i) = \nu(x) + |o|/2 \) for all \( 1 \leq i \leq |o|/2 \). Furthermore, we assume that (1) \( C_1 \) is guarded for \( O \) and \( \nu \), and (2) \( \text{pr}_1(r_n \rightarrow r_{n-1} \rightarrow \ldots \rightarrow r_2 \rightarrow r_1) \) is guarded for \( O \) and \( \nu \).

Proof. Let \( p \) be a ProbLog program, \( C_1 = \text{pr}_1(r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow r_n \rightarrow r_{n-1} \rightarrow r_{n-2} \rightarrow \ldots \rightarrow r_2 \rightarrow r_1) \) be a directed cycle in \( \text{graph}(p) \), \( O \) be a set of predicate symbols, \( \text{pr}_1 \) a predicate symbol such that for all \( o \in O, |o| = |o|, K \) be a non-empty set \( K \subseteq \{1, \ldots, |pr_1|\} \), \( \nu \) be a bijection \( \nu(i) = \nu(x) + |o|/2 \) for all \( 1 \leq i \leq |o|/2 \). Furthermore, we assume that (1) \( C_1 \) is guarded for \( O \) and \( \nu \), and (2) \( \text{pr}_1(r_n \rightarrow r_{n-1} \rightarrow \ldots \rightarrow r_2 \rightarrow r_1) \) is guarded for \( O \) and \( \nu \).

First, we rewrite \( C_1, \text{pr}_1(r_n \rightarrow r_{n-1} \rightarrow \ldots \rightarrow r_2 \rightarrow r_1) \) as \( \text{pr}_1(r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow r_n \rightarrow r_{n-1} \rightarrow r_{n-2} \rightarrow \ldots \rightarrow r_2 \rightarrow r_1) \). From \( C_1 \) is guarded for \( O_1 \), it follows that there are integers \( 1 \leq y_1 < y_2 < \ldots < y_k \leq n \) such that \( y_k = n \) and literals \( o_1(\overline{x}_1), \ldots, o_k(\overline{x}_k) \) (where \( o_i \in O \) and \( |\overline{x}_i| = |o_i| \)) such that for each \( 0 \leq k < e \), \( \text{pr}_1(r_{y_k-1} \rightarrow r_{y_k-2} \rightarrow \ldots \rightarrow r_{y_{k+1}-1} \rightarrow r_{y_{k+1}}) \) for each \( 0 \leq k < e \), \( \text{pr}_1(r_{y_k-1} \rightarrow r_{y_k-2} \rightarrow \ldots \rightarrow r_{y_{k+1}-1} \rightarrow r_{y_{k+1}}) \) for each \( 0 \leq k < e \), \( \text{pr}_1\).
Note that Algorithm 2 produces a forest of poly-trees and not just a single poly-tree because some predicates symbols may be independent. For instance, the program consisting of the rules \( a(x) \leftarrow b(x) \) and \( c(x) \leftarrow d(x) \) corresponds to two poly-trees (one per rule).

**Proposition B.10.** Let \( (\Sigma, \text{dom}) \) be a database schema such that \( \text{dom} \) is a finite domain, \( p \) be a \((\Sigma, \text{dom})\)-relaxed acyclic ProbLog program, and \( W \) be a witness for \( p \)’s acyclicity. The Bayesian Network \( \langle N, E, \text{CPT} \rangle \) produced by Algorithm 2 on input \( p \) is a forest of poly-trees.

**Proof.** Let \( (\Sigma, \text{dom}) \) be a database schema such that \( \text{dom} \) is a finite domain, \( p \) be a \((\Sigma, \text{dom})\)-relaxed acyclic ProbLog program, and \( W \) be a witness for \( p \)’s acyclicity. Furthermore, let \( BN = \langle N, E, \text{CPT} \rangle \) be the Bayesian Network produced by Algorithm 2 on input \( p \). There is a one-to-one mapping from paths in the ground graph \( gg(\alpha(\beta_N(p))) \) and paths in the Bayesian Network produced by Algorithm 2. Namely, there is a path from \( a(\tau) \) to \( a'(\tau') \) in \( gg(\alpha(\beta_N(p))) \) iff there is a path from \( X[a(\tau)] \) to \( X[a'(\tau')] \) in \( BN \). Assume that there is a cycle in the undirected version of \( BN \). From this, it follows that there is an undirected cycle in \( gg(\alpha(\beta_N(p))) \). This, however, contradicts Theorem B.1 (since \( p \) is relaxed acyclic, then \( \alpha(\beta_N(p)) \) is acyclic and there are no undirected cycles in \( gg(\alpha(\beta_N(p))) \)).

**B.2 Encoding’s acyclicity**

Here we prove that Algorithm 2 produces a Bayesian Network that is a forest of poly-trees. Note that Algorithm 2 produces a forest of poly-trees and not just a single poly-tree because some predicates symbols may be independent. For instance, the program consisting of the rules \( a(x) \leftarrow b(x) \) and \( c(x) \leftarrow d(x) \) corresponds to two poly-trees (one per rule).

**Proposition B.10.** Let \( (\Sigma, \text{dom}) \) be a database schema such that \( \text{dom} \) is a finite domain, \( p \) be a \((\Sigma, \text{dom})\)-relaxed acyclic ProbLog program, and \( W \) be a witness for \( p \)’s acyclicity. The Bayesian Network \( \langle N, E, \text{CPT} \rangle \) produced by Algorithm 2 on input \( p \) is a forest of poly-trees.

**Proof.** Let \( (\Sigma, \text{dom}) \) be a database schema such that \( \text{dom} \) is a finite domain, \( p \) be a \((\Sigma, \text{dom})\)-relaxed acyclic ProbLog program, and \( W \) be a witness for \( p \)’s acyclicity. Furthermore, let \( BN = \langle N, E, \text{CPT} \rangle \) be the Bayesian Network produced by Algorithm 2 on input \( p \). There is a one-to-one mapping from paths in the ground graph \( gg(\alpha(\beta_N(p))) \) and paths in the Bayesian Network produced by Algorithm 2. Namely, there is a path from \( a(\tau) \) to \( a'(\tau') \) in \( gg(\alpha(\beta_N(p))) \) iff there is a path from \( X[a(\tau)] \) to \( X[a'(\tau')] \) in \( BN \). Assume that there is a cycle in the undirected version of \( BN \). From this, it follows that there is an undirected cycle in \( gg(\alpha(\beta_N(p))) \). This, however, contradicts Theorem B.1 (since \( p \) is relaxed acyclic, then \( \alpha(\beta_N(p)) \) is acyclic and there are no undirected cycles in \( gg(\alpha(\beta_N(p))) \)).

**B.3 Encoding’s Correctness**

Here we prove the correctness of our encoding.

**B.3.1 Terminology and Notation**

Before presenting our correctness proof, let \( p \) be a relaxed acyclic ProbLog program, \( W \) be a witness for \( p \), and \( BN = \langle N, E, \text{CPT} \rangle \) be the Bayesian Network produced by Algorithm 2 having \( p \) and \( W \) as inputs. We say that the kernel of \( BN \), denoted \( K(BN) \), is the set of variables \( X[r, \emptyset, a(\tau)] \in N \mid \exists \tau. r = v(a(\tau)) \). Note that all nodes \( n \in K(BN) \) are boolean random variables by construction, namely \( D(n) = \{ \top, \bot \} \). Given a BN-total assignment \( \nu \), we say that \( \nu \) is consistent iff for all variables \( n \in N \setminus K(BN) \), \( \text{cpt}(n)(\nu(P_1), \ldots, \nu(P_m), \nu(n)) = 1 \), where \( p(n) = \{ P_1, \ldots, P_m \} \). Furthermore, we say that \( \nu \) is a model for a state \( s \), written \( \nu \models s \), iff \( a(\tau) \in s \iff \nu(X[a(\tau)]) = a(\tau) \uparrow_p \).

Let \( r \) be a rule whose head is \( h \) and whose body consists of literals \( l_1, \ldots, l_k \) and \( \langle \nu_1, \ldots, \nu_r, \nu_{r+1} \rangle \) be a tuple, where \( k \leq n \). We say that \( s \) matches \( \langle \nu_1, \ldots, \nu_r, \nu_{r+1} \rangle \) iff the following conditions hold: (1) \( \nu_{r+1} = h^+ \), (2) for all \( 1 \leq i \leq k \), if \( l_i \in \text{body}^+(r) \), then \( \nu_i = l_i \uparrow_p \), and (3) for all \( 1 \leq i \leq k \), if \( l_i \in \text{body}^-(r) \), then \( \nu_i = \bot \) or \( \nu_i \neq l_i \uparrow_p \).

**B.3.2 Exact Grounding**

We now introduce the exact grounding of a ProbLog program \( p \) given a \( p \)-probabilistic assignment. The exact grounding encodes the ProbLog semantics.

Let \( p \) be a ProbLog program and \( f \) be a \( p \)-probabilistic assignment. Furthermore, let \( \mu \) be the mapping from predicate symbols in \( p \) to \( \mathbb{N} \): \( \mu(a) = \max_{t \in \text{paths}(p,a)}(\nu(t)) \), where \( \text{paths}(p,a) \) is the set
of all directed paths that ends in a in p’s dependency graph and the weight of each path is

\[
w(t) = \begin{cases} 
0 & \text{if } t = \epsilon \\
 w(t') & \text{if } t = a_1 f_1 \cdots f_n a_{n+1} \\
1 + w(t') & \text{if } t = a_1 f_1 \cdots f_n a_{n+1}
\end{cases}
\]

We remark that \( \mu \) is a valid stratification for the program \( p \). From more details about logic programming with stratified negation we refer the reader to [10].

The functions \( g_f(p, j, i) \), \( g_f(p, j) \), and \( g_f(p) \) are defined as follows:

\[
g_f(p, 0, 0) = \{ \sigma | \sigma \in p \land \mu(a) = 0 \} \\
g_f(p, 0, i) = g_f(p, 0, 0, i) \\
g_f(p, j, 0) = g_f(p, j - 1) \\
g_f(p, j, i) = g_f(p, j - 1) \\
g_f(p, j) = \bigcup_{i \in \mathbb{N}} g_f(p, j, i) \\
g_f(p) = \bigcup_{j \in \mathbb{N}} g_f(p, j)
\]

Furthermore, the functions \( g_f(p, r, j, i) \), \( g_f(p, r, j) \), and \( g_f(p, r) \) are defined as follows:

\[
g_f(p, r, 0, 0) = \emptyset \\
g_f(p, r, 0, i) = g_f(p, r, 0, 0, i) \\
g_f(p, r, j, 0) = g_f(p, r, j - 1) \\
g_f(p, r, j, i) = g_f(p, r, j - 1) \\
g_f(p, r, j) = \bigcup_{i \in \mathbb{N}} g_f(p, r, j, i) \\
g_f(p, r) = \bigcup_{j \in \mathbb{N}} g_f(p, r, j)
\]

Proposition B.11 follows directly from \( g_f \)'s definition and the semantics of stratified logic programs.

**Proposition B.11.** For any ProbLog program \( p \) and any \( p \)-probabilistic assignment \( f \), \( g_f(p) = |\text{instance}(p, f)| \).
B.3.3 Auxiliary results about safe annotated disjunctions

Given a program $p$, a safe CPT schema $(\pi_H, \pi_V, \mu)$, and a predicate $pr$ such that $\mu(pr) \neq \emptyset$, each partition in $\pi_H(p, pr)$ identifies a row in the CPT associated with a ground atom (whose predicate is $pr$), whereas the first sequence produced by $\pi_V$ identifies the variables determining the CPT’s result. We now present some results showing the correctness of the requirements for the safety of CPT-schemas.

Proposition B.12 states that whenever two ground rules have the same head once we remove the constants through the transformation $\beta$, the corresponding original rules belong to the same set $RR$ in $\pi_H(p, pr)$.

**Proposition B.12.** Let $p$ be a $(\Sigma, dom)$-PROBLOG program, $(\pi_H, \pi_V, \mu)$ be an acyclicity witness for $p$, and $pr$ be a predicate symbol in $\Sigma$ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1,\ldots,[pr]\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $\text{pred}(\text{head}(r_1)) = pr$ and $\text{pred}(\text{head}(r_2)) = pr$, $r'_1 \in \text{ground}(p, r_1)$ and $r'_2 \in \text{ground}(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in $row_1$ and $row_2$, where $\pi_V(r_1) = \langle row_1, sel_1, psow_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psow_2 \rangle$. If $\mu(\text{head}(r'_1))_{\pi_K'} = \mu(\text{head}(r'_2))_{\pi_K'}$, then there is an $RR \in \pi_H(p, pr)$ such that $r_1, r_2 \in \bigcup_{R \in RR} R$.

**Proof.** Let $p$ be a $(\Sigma, dom)$-PROBLOG program, $(\pi_H, \pi_V, \mu)$ be an acyclicity witness for $p$, and $pr$ be a predicate symbol in $\Sigma$ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1,\ldots,[pr]\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $\text{pred}(\text{head}(r_1)) = pr$ and $\text{pred}(\text{head}(r_2)) = pr$, $r'_1 \in \text{ground}(p, r_1)$ and $r'_2 \in \text{ground}(p, r_2)$ be the corresponding ground rules. Finally, let $gRow_1$ and $gRow_2$ be the ground versions of the literals in $row_1$ and $row_2$, where $\pi_V(r_1) = \langle row_1, sel_1, psow_1 \rangle$ and $\pi_V(r_2) = \langle row_2, sel_2, psow_2 \rangle$. We assume that $\mu(\text{head}(r'_1))_{\pi_K'} = \mu(\text{head}(r'_2))_{\pi_K'}$. Assume, for contradiction’s sake, that there is no $RR \in \pi_H(p, pr)$ such that $r_1, r_2 \in \bigcup_{R \in RR} R$. From this and $r_1, r_2 \in p$, it follows that there are $RR_1, RR_2 \in \pi_H(p, pr)$, $R_1 \in RR_1$, and $R_2 \in RR_2$ such that $r_1 \in R_1$, $r_2 \in R_2$, and $R_1 \neq RR_1$. From this and requirement $(4.b)$ of schema safety, it follows that there is an annotated disjunction DIS(a, b) that can be derived from $p$ such that $\mu(\text{head}(r'_1))_{\pi_K'} \in \text{body}^+(r_1)$ and $\mu(\text{head}(r'_2))_{\pi_K'} \in \text{body}^+(r_2)$. From this, requirement $(4.a)$ of schema safety, and $\mu(\text{head}(r'_1))_{\pi_K'} = \mu(\text{head}(r'_2))_{\pi_K'}$, it follows that $\mu(\text{head}(r'_1))_{\pi_K'} \in \text{body}^+(r_1)$ and $\mu(\text{head}(r'_2))_{\pi_K'} \in \text{body}^+(r_2)$. From this and the fact that both literals are positive, it follows that both $\mu(\text{head}(r'_1))_{\pi_K'}$ and $\mu(\text{head}(r'_2))_{\pi_K'}$ are in $\text{ground}(p)$. From Proposition B.2 and DIS(a, b) can be derived from $p$, however, it follows that $\mu(\text{head}(r'_1))_{\pi_K'}$ or $\mu(\text{head}(r'_2))_{\pi_K'}$ are not in $\text{ground}(p)$, leading to a contradiction.

Proposition B.13 states that whenever we have two ground rules with the same head, they are either the same rule or they represent different rows in the CPT.

**Proposition B.13.** Let $p$ be a $(\Sigma, dom)$-PROBLOG program, $(\pi_H, \pi_V, \mu)$ be an acyclicity witness for $p$, and $pr$ be a predicate symbol in $\Sigma$ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$ and $K' = \{1,\ldots,[pr]\} \setminus K$. For any two distinct rules $r_1, r_2 \in p$ such that $\text{pred}(\text{head}(r_1)) = pr$ and $\text{pred}(\text{head}(r_2)) = pr$, all $r'_1 \in \text{ground}(p, r_1)$ and $r'_2 \in \text{ground}(p, r_2)$, if $\mu(\text{head}(r'_1))_{\pi_K} = \mu(\text{head}(r'_2))_{\pi_K}$ and $\mu(\text{head}(r'_1))_{\pi_K'} = \mu(\text{head}(r'_2))_{\pi_K'}$, then either $r'_1 = r'_2$ or there are $RR \in \pi_H(p, pr)$, $R_1 \in RR$, and $R_2 \in RR$ such that $r_1 \in R_1$, $r_2 \in R_2$, and $R_1 \neq R_2$.

**Proof.** Let $p$ be a $(\Sigma, dom)$-PROBLOG program, $(\pi_H, \pi_V, \mu)$ be an acyclicity witness for $p$, and $pr$ be a predicate symbol in $\Sigma$ such that $\mu(pr) \neq \emptyset$. Furthermore, let $K = \mu(pr)$, $K' = \{1,\ldots,[pr]\} \setminus K$, $r_1, r_2 \in p$ be two rules such that $\text{pred}(\text{head}(r_1)) = pr$ and $\text{pred}(\text{head}(r_2)) = pr$, and $r'_1 \in \text{ground}(p, r_1)$, and $r'_2 \in \text{ground}(p, r_2)$. Assume that $\mu(\text{head}(r'_1))_{\pi_K} = \mu(\text{head}(r'_2))_{\pi_K}$ and $\mu(\text{head}(r'_1))_{\pi_K'} = \mu(\text{head}(r'_2))_{\pi_K'}$. Observe that $(\pi_H, \pi_V, \mu)$ is a safe CPT schema. There are two cases:

- $r_1 = r_2$. From this and $\mu(\text{head}(r'_1))_{\pi_K} = \mu(\text{head}(r'_2))_{\pi_K}$ and $\mu(\text{head}(r'_1))_{\pi_K'} = \mu(\text{head}(r'_2))_{\pi_K'}$, it follows that the ground rules have the same heads. From this and the requirement (1) of schema safety, $r_1$ is strongly connected. From this and Proposition B.6, it follows that $r'_1 = r'_2$.

- $r_1 \neq r_2$. From $r_1 \neq r_2$, $\mu(\text{head}(r'_1))_{\pi_K'} = \mu(\text{head}(r'_2))_{\pi_K'}$, and Proposition B.12, it follows that there are $RR \in \pi_H(p, pr)$, $R_1 \in RR$, and $R_2 \in RR$ such that $r_1 \in R_1$ and $r_2 \in R_2$. From this, the requirements (2) and (3.a.i.B) of schema safety, $r_1 \neq r_2$, and $\mu(\text{head}(r'_1))_{\pi_K} = \mu(\text{head}(r'_2))_{\pi_K'}$, it follows that $r_1$ and $r_2$ cannot be in the same partition, i.e., $R_1 \neq R_2$. This completes the proof of our claim.

Proposition B.14 states that whenever two distinct ground rules represent the same row in the CPT, they produce different values (i.e., the head is different) and they are part of the same set $R$. 


Proposition B.14. Let \( p \) be a \((\Sigma, \text{dom})\)-\textsc{Problog} program, \((\pi_H, \pi_V, \mu)\) be an acyclicity witness for \( p \), and \( pr \) be a predicate symbol in \( \Sigma \) such that \( \mu(pr) \neq \emptyset \). Furthermore, let \( K = \mu(pr) \) and \( K' = \{1, \ldots, |pr|\} \setminus K \), \( r_1, r_2 \in p \) be two rules such that \( \text{pred} (\text{head}(r_1)) = pr \) and \( \text{pred} (\text{head}(r_2)) = pr \), \( r_1' \in \text{ground}(p, r_1) \) and \( r_2' \in \text{ground}(p, r_2) \) be the corresponding ground rules. Finally, let \( g\text{Row}_1 \) and \( g\text{Row}_2 \) be the ground versions of the literals in \( \text{row}_1 \) and \( \text{row}_2 \), where \( \pi_V (r_1) = \langle \text{row}_1, \text{sel}_1, \text{psel}_1 \rangle \) and \( \pi_V (r_2) = \langle \text{row}_2, \text{sel}_2, \text{psel}_2 \rangle \). If \( \text{gRow}_1(i) = \text{gRow}_2(i) \) for all \( 1 \leq i \leq \min(|\text{row}_1|, |\text{row}_2|) \), \( \text{arg} (\text{head}(r_1')) |_{K'} = \text{arg} (\text{head}(r_2')) |_{K'} \), and \( r_1' \neq r_2' \), then \( \text{arg} (\text{head}(r_1')) |_{K} \neq \text{arg} (\text{head}(r_2')) |_{K} \) and there exists an \( RR \in \pi_H(p, pr) \) and an \( R \in RR \) such that \( r_1, r_2 \in R \).

Proof. Let \( p \) be a \((\Sigma, \text{dom})\)-\textsc{Problog} program, \((\pi_H, \pi_V, \mu)\) be an acyclicity witness for \( p \), and \( pr \) be a predicate symbol in \( \Sigma \) such that \( \mu(pr) \neq \emptyset \). Furthermore, let \( K = \mu(pr) \) and \( K' = \{1, \ldots, |pr|\} \setminus K \), \( r_1, r_2 \in p \) be two rules such that \( \text{pred} (\text{head}(r_1)) = pr \) and \( \text{pred} (\text{head}(r_2)) = pr \), \( r_1' \in \text{ground}(p, r_1) \) and \( r_2' \in \text{ground}(p, r_2) \) be the corresponding ground rules. Finally, let \( g\text{Row}_1 \) and \( g\text{Row}_2 \) be the ground versions of the literals in \( \text{row}_1 \) and \( \text{row}_2 \), where \( \pi_V (r_1) = \langle \text{row}_1, \text{sel}_1, \text{psel}_1 \rangle \) and \( \pi_V (r_2) = \langle \text{row}_2, \text{sel}_2, \text{psel}_2 \rangle \). Assume that \( \text{gRow}_1(i) = \text{gRow}_2(i) \) for all \( 1 \leq i \leq \min(|\text{row}_1|, |\text{row}_2|) \). If \( \text{arg} (\text{head}(r_1')) |_{K'} = \text{arg} (\text{head}(r_2')) |_{K'} \), and \( r_1' \neq r_2' \), then \( \text{arg} (\text{head}(r_1')) |_{K} \neq \text{arg} (\text{head}(r_2')) |_{K} \) and there exists an \( RR \in \pi_H(p, pr) \) and an \( R \in RR \) such that \( r_1, r_2 \in R \).

Proposition B.15 states that whenever we fix the head and the part of the ground rules representing the row, then we have fully determined the rule.

Proposition B.15. Let \( p \) be a \((\Sigma, \text{dom})\)-\textsc{Problog} program, \((\pi_H, \pi_V, \mu)\) be an acyclicity witness for \( p \), and \( pr \) be a predicate symbol in \( \Sigma \) such that \( \mu(pr) \neq \emptyset \). Furthermore, let \( K = \mu(pr) \) and \( K' = \{1, \ldots, |pr|\} \setminus K \), \( r_1, r_2 \in p \) be two rules such that \( \text{pred} (\text{head}(r_1)) = pr \) and \( \text{pred} (\text{head}(r_2)) = pr \), \( r_1' \in \text{ground}(p, r_1) \) and \( r_2' \in \text{ground}(p, r_2) \) be the corresponding ground rules. Finally, let \( g\text{Row}_1 \) and \( g\text{Row}_2 \) be the ground versions of the literals in \( \text{row}_1 \) and \( \text{row}_2 \), where \( \pi_V (r_1) = \langle \text{row}_1, \text{sel}_1, \text{psel}_1 \rangle \) and \( \pi_V (r_2) = \langle \text{row}_2, \text{sel}_2, \text{psel}_2 \rangle \). If \( \text{gRow}_1(i) = \text{gRow}_2(i) \) for all \( 1 \leq i \leq \min(|\text{row}_1|, |\text{row}_2|) \) and \( \text{head}(r_1') = \text{head}(r_2') \), then \( r_1' = r_2' \).

Proof. Let \( p \) be a \((\Sigma, \text{dom})\)-\textsc{Problog} program, \((\pi_H, \pi_V, \mu)\) be an acyclicity witness for \( p \), and \( pr \) be a predicate symbol in \( \Sigma \) such that \( \mu(pr) \neq \emptyset \). Furthermore, let \( K = \mu(pr) \) and \( K' = \{1, \ldots, |pr|\} \setminus K \), \( r_1, r_2 \in p \) be two rules such that \( \text{pred} (\text{head}(r_1)) = pr \) and \( \text{pred} (\text{head}(r_2)) = pr \), \( r_1' \in \text{ground}(p, r_1) \) and \( r_2' \in \text{ground}(p, r_2) \) be the corresponding ground rules. Finally, let \( g\text{Row}_1 \) and \( g\text{Row}_2 \) be the ground versions of the literals in \( \text{row}_1 \) and \( \text{row}_2 \), where \( \pi_V (r_1) = \langle \text{row}_1, \text{sel}_1, \text{psel}_1 \rangle \) and \( \pi_V (r_2) = \langle \text{row}_2, \text{sel}_2, \text{psel}_2 \rangle \). Assume that \( \text{gRow}_1(i) = \text{gRow}_2(i) \) for all \( 1 \leq i \leq \min(|\text{row}_1|, |\text{row}_2|) \) and \( \text{head}(r_1') = \text{head}(r_2') \). From \( \text{head}(r_1') = \text{head}(r_2') \) and Proposition B.13, it follows that either \( r_1' = r_2' \) or there is a \( RR \in \pi_H(p, pr) \) such that \( R_1 \in RR, R_2 \in RR, r_1 \in R_1, r_2 \in R_2, \) and \( R_1 \neq R_2 \). Assume, for contradiction’s sake, that \( r_1' \neq r_2' \). Then, it must be the case that there is a \( RR \in \pi_H(p, pr) \) such that \( R_1 \in RR, R_2 \in RR, r_1 \in R_1, r_2 \in R_2, \) and \( R_1 \neq R_2 \). From this and requirement (3.b.iii), there is an \( 1 \leq i \leq \min(|\text{row}_1|, |\text{row}_2|) \) such that \( r_1 \) and \( r_2 \) differ in at least a literal. This, however, contradicts \( \text{gRow}_1(i) = \text{gRow}_2(i) \) for all \( 1 \leq i \leq \min(|\text{row}_1|, |\text{row}_2|) \). Hence, \( r_1' = r_2' \).

B.3.4 Auxiliary results about Relaxed Acyclic Programs

Proposition B.16 states a simple fact about relaxed acyclic programs.

Proposition B.16. Let \( p \) be a \((\Sigma, \text{dom})\)-\textsc{Problog} program, \( W = \langle \pi_V, \pi_H, \mu \rangle \) be an acyclicity witness for \( p \), and \( pr \) be a predicate symbol in \( \Sigma \). If \( \mu(pr) \neq \emptyset \), then each ground atom \( pr(\bar{\text{c}}) \) can be derived only from one rule in \( \alpha(3\text{V}(p)) \).

Proof. Let \( p \) be a \((\Sigma, \text{dom})\)-\textsc{Problog} program, \( W = \langle \pi_V, \pi_H, \mu \rangle \) be an acyclicity witness for \( p \), and \( pr \) be a predicate symbol in \( \Sigma \). Furthermore, assume that \( \mu(pr) \neq \emptyset \). Assume for contradiction’s sake, that there are two distinct rules \( r \) and \( r' \) from which we can derive \( pr(\bar{\text{c}}) \). There are two cases:
• $r$ and $r'$ are derived from rules $r_1$ and $r_2$ such that there are two distinct $RR_i, RR_j \in \pi_R(p, p')$, $r_1 \in \bigcup_{RR_i} R$, and $r_2 \in \bigcup_{RR_j} R$. From this and requirement (4.b) of schema safety, it follows that there is a disjoint annotation $DIS(a, b)$ (that can be derived from $p$) and indexes $i, j$ such that $a(\text{args}(\text{head}(r)))_{\downarrow ig(p)}$ and $b(\text{args}(\text{head}(r')))_{\downarrow ig(p')}$ are positive literals in $r_1$ and $r_2$. From requirement (4.a) of schema safety and the fact that $\text{head}(r) = \text{head}(r')$, it follows that $a(\text{head}(r))_{\downarrow ig(p)} = a(\text{head}(r'))_{\downarrow ig(p')}$ and $b(\text{head}(r))_{\downarrow ig(p')} = b(\text{head}(r'))_{\downarrow ig(p')}$. Furthermore, from requirements (4.b), (4.c), and the transformation's definition, it follows that $a(\text{args}(\text{head}(r)))_{\downarrow ig(p)}$ and $b(\text{args}(\text{head}(r')))_{\downarrow ig(p'p)}$ are positive literals in $r$ and $r'$. From this and $a(\text{head}(r))_{\downarrow ig(p)} = a(\text{head}(r'))_{\downarrow ig(p')}$, it follows that $a(\text{args}(\text{head}(r)))_{\downarrow ig(p)}$ and $b(\text{args}(\text{head}(r')))_{\downarrow ig(p')}$ belong to $\text{ground}(p)$. This, however, contradicts $a(\text{args}(\text{head}(r)))_{\downarrow ig(p)}$ or $b(\text{args}(\text{head}(r')))_{\downarrow ig(p')}$ not being in $\text{ground}(p)$, which follows from $\text{DIS}(a, b)$, and Proposition B.2.

• $r$ and $r'$ are derived from rules $r_1$ and $r'_1$ such that there is $RR \in \pi_{RR}(p, p')$, $r'_1 \in \bigcup_{RR} R$, and $r'_2 \in \bigcup_{RR} R$. From this and requirement (3.c), $r$ and $r'$ are the same rule, leading to a contradiction.

This completes the proof of our claim. □

Proposition B.17 states a simple results connecting the original program $p$ and its transformed version $\alpha(\beta_W(p))$.

**Proposition B.17.** Let $\text{dom}$ be a finite domain, $(\Sigma, \text{dom})$ be a database schema, $p$ be a $(\Sigma, \text{dom})$-relaxed acyclic ProbLog program, $W$ be a witness for $p$, and $p'$ be the program $\alpha(\beta_W(p))$. Then, (1) $a \in \text{ground}(p)$ implies $a_{\uparrow p} \in \text{ground}(p')$, and (2) $s \in \text{ground}(p, r)$ implies $s_{\uparrow p} \in \text{ground}(p', r_{\uparrow p})$.

**Proof.** Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $(\Sigma, \text{dom})$-relaxed acyclic ProbLog program, $W$ be a witness for $p$, and $p'$ be the program $\alpha(\beta_W(p))$. We now prove our first claim. Let $a \in \text{ground}(p)$. From this, it follows that there is an $i$ such that $a \in \text{ground}(p, i)$. We now prove our claim by induction on $i$. The base case is $a \in \text{ground}(p, 0)$. From this, it follows that $a$ is either a ground atom or a probabilistic ground atom. From this and the definition of $\alpha$ and $\beta$, $a_{\uparrow p} = a$. From this, it follows that $a \in p'$ and, therefore, $a \in \text{ground}(p')$.

For the induction step, assume that our claim holds for all $j < i$, we now show that it holds also for $i$. The only interesting case is $a \in \text{ground}(p, i) \setminus \text{ground}(p, i - 1)$. From this, it follows that there is a rule $r$ and a ground rule $s$ such that all $\text{body}(s) \subseteq \text{ground}(p, i - 1)$. From this, $r_{\uparrow p} \in p'$, the fact that the transformation does not introduce new positive literals, and the induction hypothesis, it follows that $\{b_{\downarrow p} \mid b \in \text{body}(s)\} \subseteq \text{ground}(p')$. From this and $\text{ground}$'s definition, it follows that $a_{\uparrow p} \in \text{ground}(p')$. The proof of our second claim is similar to the first one. □

### B.3.5 Auxiliary Lemmas

We are now ready to prove some auxiliary lemmas for the encoding’s correctness.

**Lemma B.4.** Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p_0$ be a $(\Sigma, \text{dom})$-relaxed acyclic ProbLog program, $W$ be a witness for $p_0$, $p$ be the transformed program $\alpha(\beta_W(p))$, $BN = (N, E, \text{CPT})$ be the Bayesian Network generated by Algorithm 2 having $p_0$ and $W$ as input. Furthermore, let $r$ be a rule in $p$ and $D$ be the function associating to each predicate symbol $p$ its domain (see Algorithm 2 for $D$’s definition). Finally, let $(v_1, \ldots, v_{n+1})$ a tuple in $D(a_1) \times \cdots \times D(a_n) \times \text{D}(\text{pred}\{\text{head}(r)\})$, where $n = |\text{body}(r)|$ and $a_i = \text{pred}(\text{body}(r, i))$ for $1 \leq i \leq |\text{body}(r)|$. The following statements hold:

1. If $v_{n+1} \neq \bot$, then $\text{satisfiable}(r, (v_1, \ldots, v_{n+1}), D, W, p) = \top$ iff there exists a rule $r' \in [r]_{p, W}$ such that $r'$ matches $(v_1, \ldots, v_n, v_{n+1})$.
2. $\text{satisfiable}(r, (v_1, \ldots, v_n), D, W, p) = \bot$ iff there are a $v_{n+1} \neq \bot$ and a rule $r' \in [r]_{p, W}$ that match $(v_1, \ldots, v_n, v_{n+1})$.
3. If $v_{n+1} \neq \bot$ and $v_{n+1} \neq \top$, then there is at most one rule $r' \in [r]_{p, W}$ such that $r'$ matches $(v_1, \ldots, v_n, v_{n+1})$.

**Proof.** Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p_0$ be a $(\Sigma, \text{dom})$-relaxed acyclic ProbLog program, $W$ be a witness for $p_0$, $p$ be the transformed program $\alpha(\beta_W(p))$, $BN = (N, E, \text{CPT})$ be the Bayesian Network generated by Algorithm 2 having $p_0$ and $W$ as input. Furthermore, let $r$ be a rule in $p$ and $D$ be the function associating to each predicate symbol $p$ its domain (see Algorithm 2 for $D$’s definition). Finally, let $(v_1, \ldots, v_{n+1})$ a tuple in $D(a_1) \times \cdots \times D(a_n) \times \text{D}(\text{pred}\{\text{head}(r)\})$, where $n = |\text{body}(r)|$ and $a_i = \text{pred}(\text{body}(r, i))$ for $1 \leq i \leq |\text{body}(r)|$.

**Statement 1.** Assume that $v_{n+1} \neq \bot$. Then, the $\text{satisfiable}$ procedure returns $\top$ iff there exists a rule $r' \in [r]_{p, W}$ such that $\text{filter}(\text{head}(r'), D, W) = v_{n+1}$ and for all $1 \leq i \leq |\text{body}(r')|$, $v_i \in \text{filter}(\text{body}(r', i), D, W)$. Observe that $(1) \text{filter}(\text{head}(r'), D, W) = v_{n+1}$ iff $h_{\uparrow p} = v_{n+1}$, (2) for any positive literal,
Assume that \( v_{n+1} = \perp \). By inspecting the code of \textit{satisfiable} it is easy to see that in case \( v_{n+1} = \perp \) the function is equivalent to \(-\bigvee_{v_{n+1} \in D(\text{pred}(head(r'))) \setminus \{ \perp \}} \textit{satisfiable}(r, \{v_1, \ldots, v_{n+1}\}, D, W, p)\). From this, \textit{satisfiable}(r, \{v_1, \ldots, v_{n+1}\}, D, W, p) = \top if \bigwedge_{v_{n+1} \in D(\text{pred}(head(r'))) \setminus \{ \perp \}} \textit{satisfiable}(r, \{v_1, \ldots, v_{n+1}\}, D, W, p) holds. From this, \textit{satisfiable}(r, \{v_1, \ldots, v_{n+1}\}, D, W, p) = \perp if there exists a \( v_{n+1} \neq \perp \) such that \textit{satisfiable}(r, \{v_1, \ldots, v_{n+1}\}, D, W, p) holds. From this (and the proof of the previous claim), \textit{satisfiable}(r, \{v_1, \ldots, v_{n+1}\}, D, W, p) = \perp if there exists a \( v_{n+1} \neq \perp \) and a rule \( r' \in [r]_{\text{rp}, W} \) that matches \( \{v_1, \ldots, v_{n+1}\} \).

**Statement 3.** Assume that \( v_{n+1} \neq \perp \) and \( v_{n+1} \neq \top \). Assume, for contradiction’s sake, that there are two distinct rules \( r', r'' \in [r]_{\text{rp}, W} \) such that \( \pi = \{v_1, \ldots, v_{n+1}\} \) matches \( r' \) and \( r'' \). Let \( p_1 \) be the symbol \( \text{pred}(head(r')) \) (note that \( \text{pred}(head(r')) = \text{pred}(head(r'')) \)). From \( r' \) matches \( \pi \), it follows that (1) \( head(r') \uparrow_1 = v_{n+1} \), (2) for all \( 1 \leq i \leq |\text{body}(r')| \), if \( l_i \in \text{body}^+(r') \), then \( \pi \uparrow_i = \pi \uparrow_1 \), and (3) for all \( 1 \leq i \leq |\text{body}(r')| \), if \( l_i \in \text{body}^-(r') \), then \( \pi \uparrow_i = \perp \) or \( \pi \uparrow_1 \). Similarly, from \( r'' \) matches \( \pi \), it follows that (1) \( head(r'') \uparrow_1 = v_{n+1} \), (2) for all \( 1 \leq i \leq |\text{body}(r'')| \), if \( l_i \in \text{body}^+(r'') \), then \( \pi \uparrow_i = \pi \uparrow_1 \), and (3) for all \( 1 \leq i \leq |\text{body}(r'')| \), if \( l_i \in \text{body}^-(r'') \), then \( \pi \uparrow_i = \perp \) or \( \pi \uparrow_1 \). From this and \( v_{n+1} \notin \{\top, \perp\} \), it follows that \( \text{args}(head(r')) \uparrow_{\pi} = v_{n+1} \) and \( \text{args}(head(r'')) \uparrow_{\pi} = v_{n+1} \). From \( r', r'' \in [r]_{\text{rp}, W} \) and requirement (3.c) of schema safety, it follows that there are \( RR \in \pi \pi \text{h}(p, p') \), \( R_1 \in \text{RR} \), and \( R_2 \in \text{RR} \) such that \( r' \in R_1 \) and \( r'' \in R_2 \) otherwise \( r \) and \( r' \) would result in different rules in the transformed program. There are two cases:

1. \( R_1 = R_2 \). From this and requirement (3.a.iii) of schema safety, \( \text{args}(head(r')) \uparrow_{\pi} = \text{args}(head(r'')) \uparrow_{\pi} \). This contradicts \( \text{args}(head(r')) \uparrow_{\pi} = v_{n+1} \) and \( \text{args}(head(r'')) \uparrow_{\pi} = v_{n+1} \).

2. \( R_1 \neq R_2 \). From this and requirement (3.b.iii) of schema safety, there exists an \( 1 \leq i \leq \min(\text{row}(r'), \text{row}(r'')) \), where \( \pi \uparrow_i (r') = (\text{row}(r'), \text{sel}_i, \text{sw}_1) \) and \( \pi \uparrow_i (r'') = (\text{row}(r''), \text{sel}_i, \text{sw}_2) \), such that:

\[ \mu(\text{pred}(\text{row}(r_1))) = \emptyset, \text{row}_1(i) \text{ and row}_2(i) \text{ positive literals and args}(\text{row}(r_1)) \uparrow_{K_1, i} = \text{args}(\text{row}(r_2)) \uparrow_{K_2, i}, \text{ where } K_1, i = \mu(\text{pred}(\text{row}(r_1))) \text{ and } K_2, i = \mu(\text{pred}(\text{row}(r_2))) \text{ or} \]

\[ \mu(\text{pred}(\text{row}(r_2))) = \emptyset, \text{row}_1(i) \text{ a positive literal and row}_2(i) \text{ a negative one or vice versa}. \]

In the first case, there is an \( 1 \leq i \leq \min(|\text{body}(r')|, |\text{body}(r'')|) \) such that \( \text{body}(r', i) \) and \( \text{body}(r'', i) \) are positive literals and \( \text{body}(r', i) \uparrow_{\pi} = \text{body}(r'', i) \uparrow_{\pi} \). This, however, contradicts that \( v_1 = \text{body}(r', i) \uparrow_{\pi} \) and \( v_1 = \text{body}(r'', i) \uparrow_{\pi} \). In the second case, there is an \( 1 \leq i \leq \min(|\text{body}(r')|, |\text{body}(r'')|) \) such that \( \text{body}(r', i) \) is a positive literal and \( \text{body}(r'', i) \) is a negative one (or vice versa). Hence, \( \text{body}(r', i) \uparrow_{\pi} = \top \) and \( \text{body}(r'', i) \uparrow_{\pi} = \perp \) (or vice versa). This, again contradicts \( v_1 = \text{body}(r', i) \uparrow_{\pi} \) and \( v_1 = \text{body}(r'', i) \uparrow_{\pi} \).

This completes the proof of our third claim.

\[ \square \]
X[a(1)] ∈ N. The base case is as follows. If a(1) ∈ g(f(p, 0, 0), then there is a rule r such that either r = a(1) or r = w:a(1) and a(1) ∈ g(f(p, r, 0, 0)). If r = a(1), then ν[X[r, 0, a(1)] = ⊤ due to ν’s consistency, g(f) ⊆ ground(p, r). Proposition B.17, r⊥p = r, and a(1)⊥p = a(1). If r = w:a(1), then f(w:a(1)) = ⊤ follows from a(1) ∈ g(f(p, 0, 0)). From this, g(f(p) ⊆ ground(p), Proposition B.17, r⊥p = r, a(1)⊥p = a(1), and ν[X[w:a(1), 0, a(1)] = f(w:a(1)), it follows that there is a variable X[r, 0, a(1)] ∈ N such that ν[X[r, 0, a(1)] = ⊤. From ν[X[r, 0, a(1)] = ⊤, BN’s construction, and ν’s definition, it follows that there are nodes X[r, a(1)], X[a(1)] ∈ N such that ν[X[r, a(1)]] = ⊤ and ν[X[a(1)]] = ⊤ as well (note that r⊥p = r, and a(1)⊥p = a(1) in this case). For the induction step, we assume that our claim holds for all i′ < i. The only interesting case is when a(1) ∈ g(f(p, 0, i−1) \ g(f(p, 0, i−1). If this, then it follows that there is a rule p such that a(1) ← b1, ..., bm ∈ g(f(p, r, 0, i). From this, it follows that b1, ..., bm ∈ g(f(p, 0, i−1) ⊆ ground(p, r), Proposition B.17, and the induction hypothesis (1), it follows that there are nodes X[b1⊥p], ..., X[bm⊥p] ∈ N and ν[X[b1⊥p]] = b1⊥p, ..., ν[X[bm⊥p]] = bm⊥p. From this, body−(r) = 0, g(f(p, r, 0, i) ⊆ g(f(p, r) ⊆ ground(p, r), Proposition B.17, and BN’s construction, it follows that if there is a ground X[r⊥p, {b1⊥p, ..., bm⊥p}], a(1)⊥p] ∈ N such that cpt(X[r⊥p, {b1⊥p, ..., bm⊥p}], a(1)⊥p)](ν[X[b1⊥p]], ..., ν[X[bm⊥p]], k) = 1 iff k = a(1)⊥p (this follows from the definitions of cpt, satisfiable, and Lemma B.4). As a result, ν[X[r⊥p, {b1⊥p, ..., bm⊥p}], a(1)⊥p)] = a(1)⊥p, since ν is consistent. From this and BN’s construction, it follows that there are nodes X[r⊥p, a(1)⊥p] X[a(1)⊥p] ∈ N such that ν[X[r⊥p, a(1)⊥p)] = ⊤ and ν[X[a(1)⊥p)] = a(1)⊥p. This completes the proof of the if direction.

(⇒). To prove that if ν[X[a(1)⊥p)] = a(1)⊥p, then a(1) ∈ g(f(p, 0, i), for some i, we prove a stronger claim. Namely, if ν[X[a(1)⊥p)] = a(1)⊥p, then a(1) ∈ g(f(p, 0, depth(X[a(1)⊥p)]), where depth(n) = 0 if ancestors(n) does not contain any variable of the form X[b(0)] and depth(n) = 1 + maxν′∈ancestors(n) depth(n′) otherwise. We prove our claim by induction on depth(X[a(1)⊥p)] (we denote the induction hypothesis as (Δ)). The base case is as follows. If depth(X[a(1)⊥p)] = 0, then from ν[X[a(1)⊥p)] = ⊤ and ν’s consistency, it follows that there must be a rule r, of the form r = w:a(1) or r = a(1), such that X[r, a(1)] ∈ N and ν[X[r, a(1)] = ⊤ (note that, in this case, r⊥p = r and a(1)⊥p = a(1)). From this and ν’s consistency, there is a variable X[r, 0, a(1)] ∈ N such that ν[X[r, 0, a(1)] = ⊤. If r = a(1), then a(1) ∈ g(f(0, 0, 0)) by definition. If r = w:a(1), then from ν[X[r, 0, a(1)] = ⊤ and ν[X[r, 0, a(1)] = f(w:a(1)), it follows that f(w:a(1)) = ⊤. From this, it follows that a(1) ∈ g(f(p, 0, 0)). For the induction step, we assume that our claim holds for all random variables of depth less than k. From ν[X[a(1)⊥p)] = a(1)⊥p and ν’s consistency, it follows that there is a rule r ∈ P such that ν[X[r, a(1)]] = a(1)⊥p, and ν’s consistency, there is a set of positive ground literals f = {b1, ..., bm} such that X[r, {b1⊥p, ..., bm⊥p}, a(1)] ∈ N and ν[X[r, {b1⊥p, ..., bm⊥p}, a(1)] = a(1)]. From this and BN’s construction (cf. the cpt and satisfiable procedures and Lemma B.4), it follows that there is a rule r′ ∈ P such that (1) r′ ∈ r⊥p, and (2) the values assigned by ν to the random variables associated to the atom in I produce a grounding s′ of r′ that satisfies all constraints and is consistent (namely, the body of s′ does not contain both an atom and its negation and multiple copies of the same CPT-like atom in r′ are assigned to the same value in r). From this, BN’s definition, and μ(a) = 0, it follows that ν[X[bm⊥p]] = b⊥p for all b ∈ I. From this and the induction hypothesis (Δ), it follows that b ∈ g(f(p, 0, depth(X[a(1)])) − 1 for all b ∈ I. From this, there is a rule r′ ∈ P such that (1) r′ ∈ r⊥p, and (2) r is satisfied by {b1, ..., bm}, i.e., there is a grounding s′ of s′ obtained by using a subset of the literals in {b1, ..., bm} and adding repeated occurrences of the literals if needed. From this, it follows that s is in g(f(p, 0, 0, depth(X[a(1)⊥p)]). From this, it follows that a(1) ∈ g(f(p, 0, depth(X[a(1)⊥p)]). This completes the proof the only if direction.

Induction Step. For the induction step, we assume that b(1) ∈ g(f(p, μ(b)), if ν[X[b(1)⊥p)] = a(1)⊥p, holds for all such that μ(b) < μ(a). We now prove that a(1) ∈ g(f(p, μ(a)) if ν[X[a(1)⊥p)] = a(1)⊥p, as well. In the following, let k be μ(a). We prove separately the two directions, namely (1) if a(1) ∈ g(f(p, μ(a), i), for some i, then ν[X[a(1)⊥p)] = a(1)⊥p, and (2) if ν[X[a(1)⊥p)] = a(1)⊥p, then a(1) ∈ g(f(p, μ(a), i), for some i. From this, it follows that a(1) ∈ g(f(p, μ(a)) if ν[X[a(1)⊥p)] = a(1)⊥p.

(⇒). We prove, by induction on i, that if a(1) ∈ g(f(p, k, i), for some i, then ν[X[a(1)⊥p)] = a(1)⊥p (we denote the induction hypothesis associated to this proof as (4)). From a(1) ∈ g(f(p)) ⊆ ground(p) and Proposition 17, then X[a(1)⊥p] ∈ N. The base case is as follows. Assume that a(1) ∈ g(f(p, k, 0). From this and a(1) ∈ a(1) ⊆ g(f(p, k−1) (since μ(a) > k − 1), it follows that there is a rule r such that either r = a(1) or r = w:a(1) and a(1) ∈ g(f(p, r, k−1), if r = a(1), then there is a node X[r, 0, a(1)] ∈ N such that ν[X[r, 0, a(1)] = ⊤ due to ν’s consistency, g(f(p, r) ⊆ ground(p, r), r⊥p = r, a(1)⊥p = a(1), and Proposition B.17. If r = w:a(1), then f(w:a(1)) = ⊤ follows from a(1) ∈ g(f(p, k, 0)). From this, g(f(p, r) ⊆ ground(p, r), Proposition 17, and ν[X[r, 0, a(1)] = f(w:a(1)), it follows that there is a node X[r, 0, a(1)] ∈ N such that ν[X[r, 0, a(1)] = ⊤. From ν[X[r, 0, a(1)] = ⊤, BN’s definition, and μ(a)’s definition, it follows that there are nodes X[r, a(1)], X[a(1)] ∈ N such that ν[X[r, a(1)] = ⊤ and ν[X[a(1)] = ⊤ as well (note that r⊥p = r, and a(1)⊥p = a(1)). For
the induction step, we assume that our claim holds for all $i' < i$. Assume that $a(\tau) \in g \nu i$. The only interesting case is when $a(\tau) \in g \nu i \setminus g \nu i \setminus 1$. From this, it follows that there is rule $r$ such that $r' = a(\tau) \iff b_1 \ldots b_m$ and $r' \in g \nu r \setminus i$. Furthermore, we denote by $b_1 \ldots b_m$ the atoms $\text{pos}(b_1) \ldots \text{pos}(b_m)$. From this, it follows that $b_1 \in \text{body}^+(r')$ and $b_2 \notin g \nu i \setminus k - 1$ for all $b_2 \in \text{body}^-(r)$). From $b_1 \in g \nu i \setminus k - 1$ for all $b_1 \in \text{body}^+(r)$, $g \nu r \subseteq \text{ground}(p, r)$, Proposition B.17, and the induction’s hypothesis (\(\star\)), it follows that there is a node $X[b_1 \nu] \in N$ such that $\nu(X[b_1 \nu]) = b_1 \nu \uparrow$, for all $b_1 \in \text{body}^+(r')$. From $b_2 \notin g \nu i \setminus k - 1$ for all $b_2 \in \text{body}^-(r')$, it follows that $b_2 \notin g \nu (p, \mu(\text{pred}(b_2)))$ for all $b_2 \in \text{body}^-(r')$. From this, $r' \in g \nu (p, r, k, i)$, $g \nu r \subseteq \text{ground}(p, r)$, Proposition B.17, $\mu(\text{pred}(b_2)) < \mu(a)$ for all $b_2 \in \text{body}^+(r')$, and the induction’s hypothesis (\(\star\)), it follows that there is a node $X[b_1 \nu] \in N$ such that $\nu(X[b_1 \nu]) \neq b_2 \nu \uparrow$ for all $b_2 \in \text{body}^-(r')$. From $r' \in g \nu (p, r, k, i)$, $g \nu r \subseteq \text{ground}(p, r)$, Proposition B.17, $\mu(\text{pred}(b_2)) < \mu(a)$ for all $b_2 \in \text{body}^+(r')$, and $BN$’s construction (cf. the $\text{cpt}$ and $\text{satisfiable}$ procedures and Lemma B.4), there is $X[r \nu p, b_1 \nu, \ldots, b_m \nu, a(\tau) \nu] \in N$ such that $\text{cpt}(X[r \nu p, b_1 \nu, \ldots, b_m \nu]) = a(\nu)(\nu) \nu \nu$. As a result, $\nu(X[r \nu p, b_1 \nu, \ldots, b_m \nu]) = \uparrow$ since $\nu$. From this and $BN$’s construction, it follows that there are variables $X[r \nu p, a(\nu)], X[a(\nu)] \in N$ such that $\nu(X[r \nu p, a(\nu)]) = a(\uparrow)$ and $\nu(X[a(\nu)]) = a(\uparrow)$. This completes the proof of the if direction.

(\(
\Rightarrow\n\)). To prove that if $\nu(X[a(\nu)]) = a(\uparrow)$, then $a(\nu) \in g \nu (p, k, i)$, for some i, we prove a stronger claim. Namely, if $\nu(X[a(\nu)]) = a(\uparrow)$, then $a(\nu) \in g \nu (p, k, \text{depth}(X[a(\nu)])$, where $\text{depth}(n)$ is as above. We prove our claim by induction on $\text{depth}(X[a(\nu)])$ (we denote the induction hypothesis associated to this proof as (\(\blacklozenge\))). The base case is as follows. If $\text{depth}(X[a(\nu)]) = 0$, then from $\nu(X[a(\nu)]) \subseteq \top$ and $\nu$’s consistency, it follows that there must be a rule $r$, of the form $r = a(\tau)$ or $r = a(\nu)$, such that $X[r, a(\tau)] \in N$ and $\nu(X[r, a(\tau)]) = \top$ (note that $a(\nu) \nu = \nu$ and $a(\uparrow) = a(\nu)$). From this and $a(\nu)$’s consistency, there is a variable $X[r, \emptyset, a(\nu)] \in N$ such that $\nu(X[r, \emptyset, a(\nu)]) = \top$. If $r = a(\tau)$, then $a(\nu) \in g \nu (p, k, 0)$ by definition. If $r = a(\tau)$, then from $\nu(X[r, \emptyset, a(\nu)]) = \top$ and $\nu(X[r, \emptyset, a(\nu)]) = \nu(a(\tau))$, it follows that $f(:=a(\tau)) = \top$. From this, it follows that $a(\nu) \in g \nu (p, k, 0)$. For the induction step, we assume that our claim holds for all random variables of depth less than $k$. We now show that it holds also for a variable $a(\nu) \nu$ such that $\text{depth}(X[a(\nu)]) = \uparrow$. From $\nu(X[a(\nu)]) = a(\uparrow)$ and $\nu$’s consistency, it follows that there is a rule $r' \in \uparrow$ such that $X[r', a(\nu)] \subseteq N$ and $\nu(X[r', a(\nu)]) = a(\uparrow)$. From this and $\nu$’s consistency, there is a set of ground atoms $I = (b_1, \ldots, b_m)$ such that $X[r', b_1 \nu, \ldots, b_m \nu, a(\nu)] \subseteq N$, $\nu(X[r', b_1 \nu, \ldots, b_m \nu, a(\nu)]) = a(\nu) \nu$, and $X[b \nu] \in N$ for all $b \in I$. From this and $BN$’s construction (cf. the $\text{cpt}$ and $\text{satisfiable}$ procedures and Lemma B.4), it follows that there is a rule $r \in \uparrow$ such that (1) $r \in \uparrow$, and (2) the values assigned by $\nu$ to the random variables associated to the atoms in $I$ produce a grounding $s$ of $r$ that satisfies all constraints and is consistent (namely, the body of $s$ does not contain both an atom and its negation and multiple copies of the same $\text{CPT}$-like atom in $r' \in \uparrow$ assigned to the same value in $r$). Let $I^+$ be the atoms in $I$ that are assigned to positive literals in $s$ and $I^-$ be the atoms in $I$ that are assigned to negative literals (note that these two sets are disjoint). From $\nu(X[r', b_1 \nu, \ldots, b_m \nu, a(\nu)]) = a(\uparrow)$ and $\nu$’s consistency, it follows that $\nu(X[b \nu]) = a(b) \nu$ for all $b \in I^+$ and $\nu(X[b \nu]) = b \nu$ for all $b \in I^-$. From $\nu(X[b \nu]) = b \nu$ and $\nu(X[b \nu]) < \nu(X[r', a(\nu)])$ for all $b \in I^+$ and the induction’s hypothesis (\(\blacklozenge\)), it follows that $b \in g \nu (p, k, \text{depth}(X[b \nu]))$ for all $b \in I^+$. From this, the induction’s hypothesis, and $\nu(X[b \nu]) < \nu(X[r', a(\nu)])$, it follows that $b \in g \nu (p, k, i - 1)$ for all $b \in \text{body}^+(r')$. From $\nu(X[b \nu]) = b \nu$ and $\nu(X[b \nu]) < \nu(a)$ for all $b \in I^+$. From this and $\mu(\text{pred}(b_2)) < \mu(a)$, it follows that $b \notin g \nu (p, \mu(\text{pred}(b_2)))$ for all $b \in I^+$. From this, it follows that $r' \in g \nu (p, r, k, i)$. This completes the proof of the only if direction.

Lemma B.6. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $(\Sigma, \text{dom})$-relaxed acyclic $\text{PROLOG}$ program, $W$ be a witness for $p$, $s$ be a $(\Sigma, \text{dom})$-structure, and $BN = (N, E, \text{CPT})$ be the Bayesian Network generated by Algorithm 2 having $p$ and $W$ as input. There is a $\text{probabilistic}$ assignment $f$ such that $\text{prob}(f) = k$ and $g \nu p = s$ iff there is a $BN$-total assignment $\nu$ such that $[BN][\nu] = k$ and $\nu$ is consistent, and $\nu \models s$.

Proof. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $p$ be a $(\Sigma, \text{dom})$-relaxed acyclic $\text{PROLOG}$ program, $W$ be a witness for $p$, $s$ be a $(\Sigma, \text{dom})$-structure, and $BN = (N, E, \text{CPT})$ be the Bayesian Network generated by Algorithm 2 having $p$ and $W$ as input. We now prove both directions of our claim.
\( \Rightarrow \). Let \( f \) be a \( p \)-probabilistic assignment \( f \) such that \( \text{prob}(f) = k \) and \( g_f(p) = s \). Furthermore, let \( \nu \) be the following BN-total assignment:

\[
\nu(n) = \begin{cases} 
\{ f(v:a) \} & \text{if } n = X[v:a], a(\emptyset), a(\emptyset) \\
\top & \text{if } n = X[a], a(\emptyset) \\
k & \text{if } k \in D(n) \land p(n) = \{ P_1, \ldots, P_m \} \\
\land \nu(P_1) = v_1 \land \ldots \land \nu(P_m) = v_m \\
\land \text{cpl}(n)(v_1, \ldots, v_m, k) = 1
\end{cases}
\]

The assignment \( \nu \) is well-defined since (1) BN is a forest of poly-trees (see Proposition B.10), and (2) for all nodes of the form \( X[v:a], a(\emptyset), a(\emptyset) \) and \( X[a], a(\emptyset), a(\emptyset) \), \( D(n) = \{ \top, \bot \} \) by construction (cf. Algorithm 2). Furthermore, the assignment \( \nu \) is consistent by construction. Indeed, for all variables \( n \in N \setminus K(BN) \), the corresponding entry in the CPT is 1.

We now prove that \( \nu \models s \). From Lemma B.5, \( \nu \)'s consistency, and \( f(v:a) = \nu(X[v:a], \emptyset, a(\emptyset)) \) for all \( v:a \in p \), it follows that \( a(\emptyset) \in g_f(p) \) iff \( \nu(X[a(\emptyset)]_p) = \nu(a(\emptyset))_p \). From this and \( g_f(p) = s \), it follows \( \nu \models s \).

Finally, we show that \( \text{prob}(BN)[\nu] = \text{prob}(f) \). In more detail, \( \text{prob}(BN)[\nu] = \left[ \left( \prod_{n \in K(BN)} \text{cpl}(n) \right) \cdot \left( \prod_{n \in N \setminus K(BN)} \text{cpl}(n) \right) \right](\nu(\emptyset)) \) further more, since \( \nu \) is consistent and the CPTs associated with the variables in \( N \setminus K(BN) \) are deterministic, \( \text{prob}(BN)[\nu] \) can be simplified as \( \left[ \prod_{n \in K(BN)} \text{cpl}(n) \right](\nu(\emptyset)) \). From this and BN’s definition, it follows that \( \text{prob}(BN)[\nu] = \prod_{n \in K(BN)} \text{cpl}(n) \). And from this and \( \nu \)'s definition, it follows that \( \text{prob}(BN)[\nu] = \left[ \left( \prod_{n \in K(BN)} \text{cpl}(n) \right) \cdot \left( \prod_{n \in N \setminus K(BN)} \text{cpl}(n) \right) \right](\nu(\emptyset)) \) which is equivalent to \( \text{prob}(f) \).

We still have to prove that \( g_f(p) = s \). From Lemma B.5, \( \nu \)'s consistency, \( f(v:a) = \nu(X[v:a], \emptyset, a(\emptyset)) \), \( \emptyset, a(\emptyset)) \), it follows that \( a(\emptyset) \in g_f(p) \) iff \( \nu(X[a(\emptyset)]_p) = \nu(a(\emptyset))_p \). From this and \( \nu \models s \), it follows \( g_f(p) = s \).

\( \blacksquare \)

### B.3.6 Proof of the main result

Theorem B.2 shows the correctness of our encoding.

**Theorem B.2.** Let \( (\Sigma, \text{dom}) \) be a database schema such that \( \text{dom} \) is a finite domain, \( p \) be a \( (\Sigma, \text{dom}) \)-relaxed acyclic PROLOG program, \( W \) be a witness for \( p \), BN be the Bayesian Network generated by Algorithm 2 having \( p \) and \( W \) as input, and \( s \) be a \( (\Sigma, \text{dom}) \)-structure. Furthermore, let \( \nu \) be the BN-partial assignment such that (1) for any ground atom \( a(\emptyset) \), \( \nu(X[a(\emptyset)]_p) = a(\emptyset) \) iff \( a(\emptyset) \in s \), and (2) \( \nu(v) \) is undefined otherwise. Then, \( [p](s) = [BN](\nu) \).

**Proof.** Let \( (\Sigma, \text{dom}) \) be a database schema such that \( \text{dom} \) is a finite domain, \( p \) be a \( (\Sigma, \text{dom}) \)-relaxed acyclic PROLOG program, \( W \) be a witness for \( p \), BN be the Bayesian Network generated by Algorithm 2 having \( p \) and \( W \) as input, and \( s \) be a \( (\Sigma, \text{dom}) \)-structure. Furthermore, let \( \nu \) be the BN-partial assignment such that (1) for any ground atom \( a(\emptyset) \), \( \nu(X[a(\emptyset)]_p) = a(\emptyset) \) iff \( a(\emptyset) \in s \), and (2) \( \nu(v) \) is undefined otherwise.

The probability \( [p](s) \) is \( \sum_{f \in \mathcal{M}(p, s)} \text{prob}(f) \), where \( \mathcal{M}(p, s) \) is the set of all assignments \( f \) such that \( \text{instance}(p, f) = s \). Equivalently, \( \mathcal{M}(p, s) \) is the set of all probabilistic assignments \( f \) such that \( g_f(p) = s \). Let \( K \) be the set of all total assignments that agree with \( \nu \) for all variables of the form \( X[a(\emptyset)]_p \). The probability \( [BN](\nu) = \sum_{\nu' \in \mathcal{K}} [BN](\nu') \) is analogous to the probability of any consistent assignment has probability 0. Since any non-consistent assignment has probability 0, \( [BN](\nu) = \sum_{\nu' \in \mathcal{K}} [BN](\nu') \), where \( \mathcal{K} \) is the set of all consistent assignments in \( K \).

From this, it follows that \( [p](s) = [BN](\nu) \) iff \( \sum_{f \in \mathcal{M}(p, s)} \text{prob}(f) = \sum_{\nu' \in \mathcal{K}} [BN](\nu') \). The latter follows trivially from Lemma B.6, which establishes a one-to-one mapping from \( \{ f \mid g_f(p) = s \} \) to \( K' \) that preserves probabilities. \( \blacksquare \)
B.4 Complexity of Inference

Here we prove our results about the complexity of inference for ProbLog programs.

B.4.1 Size of the encoding

Given a Bayesian Network $BN = \langle N, E, CPT \rangle$, the size of $BN$, denoted $|BN|$, is $|N| + |E| + \sum_{n \in N} |cpt(n)|$, where the size of a conditional probability table is just the number of rows in the table (i.e., the number of all assignments). The size of an atom $a(\tau)$ is $|a|$, whereas the size of a rule $h \leftarrow l_1, \ldots, l_n$ is $|h| + \sum_{1 \leq i \leq n} |l_i|$. Finally, the size of a program $p$ is $\sum_{r \in E} |r|$. The ground version of $p$, denoted $gv(p)$, is $\bigcup_{r \in E} ground(p, r)$, namely the relaxed grounding of all the rules in $p$. Note that $ground(p) \subseteq gv(p)$.

Let $p$ be a relaxed acyclic ProbLog program, $W$ be a witness for $p$, $p' = \alpha(\beta(p))$ be the transformed program, $g = gv(p')$ be its ground version, and $bn(p, W) = \langle N, E, CPT \rangle$ be the corresponding Bayesian Network derived by Algorithm 2. The number of nodes in $N$ is $O(|\text{rules}(p')| \cdot |g|)$. Indeed, there is a node $X[a(\tau)]$ for each ground atom in $g$. Moreover, for each rule $r \in p'$ and ground rule $r' \in g$, there are nodes $X[r, body(r'), head(r')]$ and $X[r, head(r')]$. Finally, the number of intermediate nodes generated by the tree procedure is twice the number of nodes of the form $X[r, body(r'), head(r')]$.

The number of edges in $E$ is $O(|\text{rules}(p')|^2 \cdot |g|)$. Indeed, there is an edge $X[r, a(\tau)] \rightarrow X[a(\tau)]$ for each rule $r$ and ground atom $a(\tau)$. Furthermore, there is an edge $X[h(\tau)] \rightarrow X[r, I, a(\tau)]$ for each rule $r$, ground rule $a(\tau) \leftarrow I$, and atom $b(\tau) \in I$. Finally, the number of edges introduced by the tree procedure is $O(|g|)$.

The size of $cpt$ is $O(|\text{rules}(p')| \cdot |g| \cdot \max(2, |\text{rules}(p)|)^{2+|\text{rules}(p')|})$. Indeed, we have a CPT for each node $n \in N$. The size of each CPT depends on (1) the number of parents for each node, and (2) the size of the domain associated to each variable. In the worst case, the size of the domain of each node is $\max(2, |\text{rules}(p)|)$ (since the size of the domain of a CPT-like predicate depends only on the rules — there are no CPT-like predicates defined by ground atoms). The number of parents for intermediate nodes is 2, and the size of each CPT is $O(\max(2, |\text{rules}(p)|)^3)$. The maximum number of parents for nodes of the form $X[r, a(\tau)]$ is 1, and the size of each CPT is $O(\max(2, |\text{rules}(p)|)^2)$. The maximum number of parents for nodes of the form $X[r, I, a(\tau)]$ is $|\text{rules}(p')|$, and the size of each CPT is $O(\max(2, |\text{rules}(p)|)^{2+|\text{rules}(p')|+1})$. Similarly, the maximum number of parents for nodes of the form $X[a(\tau)]$ is $1+|\text{rules}(p)|$, and the size of each CPT is $O(\max(2, |\text{rules}(p)|)^{2+|\text{rules}(p')|+1})$.

As a result, $bn(p, W) = O(|\text{rules}(p')|^2 \cdot |g| \cdot \max(2, |\text{rules}(p)|)^{2+|\text{rules}(p')|})$. Furthermore, since $\exists g \in O(|\text{prob}(p')|^{\mid\text{rules}(p')\mid})$, it follows that $bn(p, W) = O(|\text{rules}(p')|^2 \cdot |\text{prob}(p')|^{\mid\text{rules}(p')\mid} \cdot \max(2, |\text{rules}(p)|)^{2+|\text{rules}(p')|})$. Finally, since $\exists p, p' \in O(|\text{prob}(p)|)$ and $\exists r \in O(|\text{rules}(p)|)$, we can simplify the result as follows: $bn(p, W) = O(|\text{rules}(p')|^2 \cdot |\text{prob}(p')|^{\mid\text{rules}(p')\mid} \cdot \max(2, |\text{rules}(p)|)^{2+|\text{rules}(p')|})$. Hence, the size of $bn(p, W)$ is polynomial in $|\text{prob}(p)|$.

B.4.2 Complexity Proofs

We first define the inference problem $\text{Inf}$. Afterwards, we analyze its complexity.

Definition B.1. $\text{Inf}$ denotes the following decision problem:

Input: A database schema $\Sigma$, such that $\text{dom}$ is a finite domain, a ProbLog program $p$, a set of ground literals $E$, and a ground atom $a(\tau)$.

Output: The probability of $a(\tau)$ given evidence in $E$. \hfill \Box

The data complexity of $\text{Inf}(\Sigma, \text{dom}, p, E, a)$ for relaxed acyclic programs can be obtained by (1) fixing rules and varying only prob($p$) (and indirectly dom), and (2) requiring the program to be relaxed acyclic. The data complexity of the INF problem for relaxed-acyclic programs $p$ is the complexity of the following decision problem:

Definition B.2. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain, $R$ be a fixed set of ProbLog rules over $\Sigma$, $E$ be a set of ground literals, $a(\tau)$ be a ground atom, and $W$ be a CPT-schema. $\text{Inf}_{\Sigma, R,E,a(\tau),W}$ denotes the following problem:

Input: A set of probabilistic atoms $E'$ such that (1) atoms in $E$ and $a(\tau)$ refer only to constant values in $E'$ and $R$, and (2) $R \cup E'$ is a relaxed acyclic ProbLog program and $W$ is a safe CPT-schema and a witness for the acyclicity of $R \cup E'$.

Output: The probability of $a(\tau)$ given evidence in $E$. \hfill \Box

Theorem B.3. $\text{Inf}_{\Sigma, R,E,a(\tau),W}$ is in PTIME.

Proof. Let $(\Sigma, \text{dom})$ be a database schema such that $\text{dom}$ is a finite domain $R$ be a fixed set of ProbLog rules over $\Sigma$, $E$ be a set of ground literals, $a(\tau)$ be a ground atom, and $W$ be a CPT-schema. We consider only inputs $E'$ such that $E' \cup R$ is a relaxed acyclic ProbLog program and
W is a safe CPT-schema and a witness for the acyclicity of \( R \cup E' \). The size of \( E' \) is the sum of the sizes of all atoms, where the size of an atom is its cardinality.

Let \( e \) be the number of distinct constants occurring in \( E' \cup R \) and \( r = |R| \). Computing \( \text{Inf}(\Sigma, E', p, E, \alpha) \) can be done in the following steps:

1. Transform the original program \( p \) into the program \( p' \) (by applying the \( \alpha \) and \( \beta_W \) transformations).
2. Construct the ground version \( g \) of the program \( p' \).
3. Construct the Bayesian Network \( bn(p, W) \) from \( g \).
4. Perform the inference on \( bn(p, W) \).

The first step can be performed in linear time in the size of \( O(e+r) \). The second step can be performed in \( O(e^r) \) (because the grounding of \( p' \) can be done in \( O(e^{\text{rules}(p')}) \) and \( |\text{rules}(p')| \leq r \)). The third step can be performed in \( O(r^4 \cdot e^2 r \cdot \max(2, r)^{2+\epsilon}) \) (constructing \( N \) and \( E \) can be done in \( O(r^2 \cdot e^2 r) \), whereas the \( \text{cpt} \) can be constructed in \( O(r^4 \cdot e^2 \cdot \max(2, r)^{2+\epsilon}) \)). The fourth step can be performed in \( O(|bn(p)|) \) since \( bn(p) \) is a forest of poly-trees (see Proposition B.10). In particular, inference can be performed by (1) identifying the poly-tree that contains the atom \( a(\varpi) \) (in \( |bn(p)| \)), (2) identifying the subset \( E'' \subseteq E \) of all atoms in the poly-tree of \( a(\varpi) \) (again in \( |bn(p)| \)), and (3) performing inference using belief-propagation (in \( O(|bn(p)|) \) [105]). Therefore, the fourth step can be performed in \( O(r^2 \cdot e^2 \cdot \max(2, r)^{2+\epsilon}) \). As a result, the answer to \( \text{Inf} \) can be computed in \( O(r^2 \cdot e^2 \cdot \max(2, r)^{2+\epsilon}) \). Furthermore, since \( r \) is fixed and the number of constants in \( E' \) is \( |E'| \), there is a \( k \in \mathbb{N} \) such that \( e \in O(|E'|+k) \). From this, it follows that there is a \( k \in \mathbb{N} \) such that \( \text{Inf} \) can be computed in \( O(|E'|^k) \). Hence, the complexity of \( \text{Inf}_{\Sigma, \bar{R}, \bar{E}, \alpha(\varpi), \mu} \) (and the data complexity of \( \text{Inf} \)) is \( \text{PTIME} \).

\[ \text{Inf} \text{ for ProbLog is } \#P\text{-hard. This follows from the } \#P\text{-hardness of inference on arbitrary Bayesian networks (BNs) [105], which can be encoded in ProbLog. We now show that inference for ProbLog programs is } \#P\text{-hard even in terms of data complexity. We do this using a reduction from the } \#\text{SAT problem (counting the number of satisfying assignments of a propositional formula } \phi \text{), which is } \#P\text{-hard [126].} \]

**Proposition B.18.** \( \text{Inf} \) is \( \#P\)-hard in terms of data complexity for ProbLog programs.

**Proof.** We show this by reducing the \( \#\text{SAT} \) problem to inference for a ProbLog program \( p \) where (1) the formula can be encoded in \( \text{prob}(p) \), and (2) the rules are fixed. Let \( \phi \) be a propositional formula.

**First-order Signature.** Let \( \Sigma \) be the signature containing the following predicate symbols:

- \( e \) of arity 1, which is used to store the identifiers associated to all sub-expressions of \( \phi \),
- \( \text{state} \) of arity 1, which is used to store the propositions in the model,
- \( \text{sat} \) of arity 1, which is used to denote whether an expression is satisfiable or not,
- \( \text{conj} \) of arity 3 which is used to encode conjunctions,
- \( \text{disj} \) of arity 3 which is used to encode disjunctions,
- \( \text{neg} \) of arity 2 which is used to encode negations, and
- \( \text{prop} \) of arity 2 used to encode propositions.

**Rules.** We now define a fixed set of rules encoding the semantics of propositional logic.

\[
\begin{align*}
\text{sat}(x) & \leftarrow e(x), \text{prop}(x, y), \text{state}(y) \\
\text{sat}(x) & \leftarrow e(x), \text{neg}(x, y), \neg\text{sat}(y) \\
\text{sat}(x) & \leftarrow e(x), \text{conj}(x, y, z), \text{sat}(y), \text{sat}(z) \\
\text{sat}(x) & \leftarrow e(x), \text{disj}(x, y, z), \text{sat}(y) \\
\text{sat}(x) & \leftarrow e(x), \text{disj}(x, y, z), \text{sat}(z)
\end{align*}
\]

**Encoding a formula \( \phi \).** Given a formula \( \phi \), we define the following encoding using probabilistic ground atoms. We first associate to each sub-formula \( \psi \) of \( \phi \) a unique identifier \( id_\psi \). We also associate a unique identifier \( id_p \) to each propositional symbol \( p \). For each sub-formula \( \psi \wedge \gamma \), the ground atoms \( e(id_{\psi \wedge \gamma}) \) and \( \text{conj}(id_{\psi \wedge \gamma}, id_\psi, id_\gamma) \) are in \( E' \). For each sub-formula \( \psi \vee \gamma \), the ground atoms \( e(id_{\psi \vee \gamma}) \) and \( \text{disj}(id_{\psi \vee \gamma}, id_\psi, id_\gamma) \) are in \( E' \). For each sub-formula \( \neg \psi \), the ground atoms \( e(id_{\neg \psi}) \) and \( \text{neg}(id_{\neg \psi}, id_\psi) \) are in \( E' \). For each sub-formula \( \forall \psi \) such that \( \psi \) is a propositional symbol \( p \), the ground atoms \( e(id_p) \) and \( \text{prop}(id_p, id_\psi) \) are in \( E' \). Finally, for each propositional symbol \( \phi \), there is a propositional atom \( \forall x: \text{state}(id_\phi) \) in \( E' \).

**Reduction.** There are \( 2^{\text{sat}(\phi)} \) grounded instances of \( R \cup E' \), where \( n(\phi) \) is the number of distinct propositional symbols occurring in \( \phi \), each one with probability \( 1/2^{\text{sat}(\phi)} \). The grounded instances represent all possible assignments of \( \top \) and \( \bot \) to the proposition symbols in \( \phi \). It is easy to see that \( \text{sat}(id_\phi) \) can be derived in a grounded instance iff the instance represents a model for \( \phi \). Therefore, the probability \( \Pr[R \cup E'][\text{sat}(id_\phi)] \) is \( k/2^{\text{sat}(\phi)} \), where \( k \) is the number of models that satisfy \( \phi \).
Note that the encoding shown in the previous proof can be tweaked to work for acyclic ProbLog programs but only for propositional formulae without repetitions of propositional symbols. We remark that the \#SAT problem restricted to formulae without repetitions is no longer \#P-hard. Indeed, it can be solved in PTIME as follows: given a formula \( \phi \) without repetitions, we can construct a poly-tree boolean Bayesian Network \( BN \) encoding \( \phi \) in \( O(|\phi|) \), where the nodes associated to propositional symbols \( p \) have a uniform probability distribution (i.e., \( p \) is \( \top \) with probability \( 1/2 \) and \( \bot \) with probability \( 1/2 \)). Then, the probability associated to the root of \( \phi \) is going to be \( k/2^n(\phi) \), where \( k \) is the number of satisfying assignments. Since the inference on \( BN \) can be done in linear time in \( |BN| \) and \( |BN| \in O(|\phi|) \), then the whole problem is in PTIME.

### B.5 Expressiveness

Here we show that any BN that consists of a forest of poly-trees can be represented as a relaxed acyclic ProbLog program.

**Proposition B.19.** Any BN that is a forest of poly-trees can be represented as a relaxed acyclic ProbLog program.

**Proof.** Let \( bn \) be a BN that is a forest of poly-trees. We now construct the corresponding relaxed acyclic ProbLog program. In particular, for each random variable \( X \) in \( bn \), we show how to equivalently encode it using ProbLog rules. Without loss of generality, we assume there is a unique mapping \( id \) from random variables in \( bn \) to predicate symbols identifiers. With a slight abuse of notation, we use \( X \) to refer both to the random variable and to the corresponding symbol \( id(X) \).

**Boolean Random Variables without parents.** For each boolean random variable \( X \) such that (1) \( p(X) = \emptyset \), and (2) \( CPT(X) = \{ \top \mapsto v, \bot \mapsto (1 - v) \} \), we introduce a probabilistic atom \( \forall i::X \).

**Non-Boolean Random Variables without parents.** For each non-boolean random variable \( X \) with domain \( \{ q_1, \ldots, q_n \} \) such that (1) \( p(X) = \emptyset \), and (2) \( CPT(X) = \{ q_i \mapsto v_1, \ldots, q_n \mapsto v_n \} \), we introduce an annotated disjunction \( \forall i::X(q_i) \) where \( X \).

**Boolean Random Variables with parents.** Let \( X \) be a boolean random variable with parents \( p(X) = \{ X_1, \ldots, X_m \} \). CPT(\( X \)) be the function associated with \( X \), and \( X_1, \ldots, X_m \) be all possible assignments to the variables in \( p(X) \). We introduce \( m \) fresh predicate symbols \( sw_{X_1}, \ldots, sw_{X_m} \).

For each \( i \), we introduce the probabilistic atom \( c_i::sw_{X_i}, \) where \( \forall X_i \) \( CPT(X) = \forall i::CPT(X_i) = c_i \). Finally, for each \( i \), we also introduce the following rules:

\[
X \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_1), sw_{X_i}, sw_{X_i}]
\]

\[
X \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_{2i-1}), sw_{X_i}, sw_{X_i}]
\]

where \( \tilde{I}_1, \ldots, \tilde{I}_{2i-1} \) are all possible values in \( \{ \top, \bot \}^{i-1} \), \( sw_{l(1), \ldots, l_{i-1}} \) is the list of literals \( \forall l_{i} \) \( l_{i} \) \( l_{i} \subseteq \{ \top, \bot \} \), and given a symbol \( A, A[\top] = A, A[\bot] = \neg A, \) and \( A[v] = A(v) \) if \( v \notin \{ \top, \bot \} \).

**Non-Boolean Random Variables with parents.** Let \( X \) be a non-boolean random variable \( X \) with domain \( \{ q_1, \ldots, q_m \} \) and parents \( p(X) = \{ X_1, \ldots, X_m \} \). CPT(\( X \)) be the function associated with \( X \), and \( X_1, \ldots, X_m \) be all possible assignments to the variables in \( p(X) \). For each \( i \), we introduce the probabilistic atoms \( w_i^1::sw_{X_i}^{q_1}, \ldots, w_i^n::sw_{X_i}^{q_n} \), where for all \( 1 \leq i \leq n, w_i^j = c_i^j \cdot \left( 1 - \sum_{k<j} c_i^k \right) \) and \( CPT(X) = \forall i::CPT(X_i) = c_i \). Finally, for each \( i \), we also introduce the following rules:

\[
X[q_1] \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_1), sw_{X_i}^{q_1}, sw_{X_i}^{q_1}]
\]

\[
X[q_2] \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_1), -sw_{X_i}^{q_1}, sw_{X_i}^{q_2}, sw_{X_i}^{q_2}]
\]

\[
\vdots
\]

\[
X[q_m] \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_1), -sw_{X_i}^{q_1}, \ldots, -sw_{X_i}^{q_{m-1}}, sw_{X_i}^{q_m}, sw_{X_i}^{q_m}]
\]

\[
\vdots
\]

\[
X[q_1] \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_{2i-1}), sw_{X_i}^{q_1}, sw_{X_i}^{q_1}]
\]

\[
X[q_2] \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_{2i-1}), -sw_{X_i}^{q_1}, sw_{X_i}^{q_2}, sw_{X_i}^{q_2}]
\]

\[
\vdots
\]

\[
X[q_m] \leftarrow Y_1[\forall i::X(1), \ldots, Y_n[\forall i::X(n)], sw(\tilde{I}_{2i-1}), -sw_{X_i}^{q_1}, \ldots, -sw_{X_i}^{q_{m-1}}, sw_{X_i}^{q_m}]
\]

\[
\vdots
\]
where \( l_1, \ldots, l_{m-1} \) are all possible values in \( \{ \top, \bot \}^{m-1} \), \( \text{sw}(l_1, \ldots, l_{m}) \) is the list of literals \( su_{\pi l}(l_i) \), \( su_{\pi l, \pi r}(l_i) \), \( su_{\pi l, \pi r, \pi s}(l_i) \), and a given predicate symbol \( A \), \( A[l] = \top \), \( A[l] = \bot \), and \( A[v] = \bot \) if \( v \notin \{ \top, \bot \} \). Observe that the above rules are equivalent to having, for each \( \pi \), the annotated disjunction \( c^1 \wedge \ldots \wedge c^m \mapsto Y \). Finally, the program \( \text{sw}(l_1, \ldots, l_m) \) do not influence the derivation since there is a rule for any possible values for them. We need them only for the encoding. Therefore, also the encoding of random variables with parents directly encodes the corresponding CPTs.

Relaxed Acyclicity. Let \( p \) be the program produced by the above construction. It is easy to see that all predicates associated with non-boolean random variables are safe annotated disjunctions (as we just encoded their CPTs). The horizontal partitioning \( \pi_H \) contains, for each non-boolean random variable \( X \), only one set \( R \) that, in turns, contains a partition of the rules defining \( X \) based on the CPT row, whereas the vertical partitioning \( \pi_V \) is such that, for instance, \( \pi_V \{ X \mapsto Y \} = Y \). Finally, the function \( \mu(X) = \emptyset \) for boolean variables \( X \) and \( \mu(X) = \{ 1 \} \) for non-boolean variables. Observe that the schema \( \pi_H, \pi_V, \mu \) is safe. Therefore, the program \( \beta_W(p) \) is obtained by removing all constant values associated with the domains of non-boolean random variables.

Finally, the program \( \alpha(\beta_W(p)) \) collapses the rules for random variables with parents into a single rule (this follows from the use of \( \text{sw} \) in the rules’ bodies). The acyclicity of \( p' \) is equivalent to having \( \text{sw} \) and \( \text{sw} \) is cyclic in the rule of \( \text{sw}(l_1, \ldots, l_m) \) is equivalent to having, for each \( \pi \), the annotated disjunction \( c^1 \wedge \ldots \wedge c^m \mapsto Y \). Finally, the function \( \mu(X) = \emptyset \) for boolean variables \( X \) and \( \mu(X) = \{ 1 \} \) for non-boolean variables. Observe that the schema \( \pi_H, \pi_V, \mu \) is safe. Therefore, the program \( \beta_W(p) \) is obtained by removing all constant values associated with the domains of non-boolean random variables.

B.6 ANGERONA

Here we prove ANGERONA’s security and complexity.

B.6.1 Security Proof

Let \( C = \langle D, \Gamma \rangle \) be a system configuration and \( h \) be a C-history. The set \( \text{observations}(h, u) \) is \( \{ \phi \mid \exists h.t = ((u, \phi), T, T) \} \cup \{ \neg \phi \mid \exists h.t = ((u, \phi), T, \bot) \} \).

We first prove that ANGERONA’s result depends just on the queries in the history, and not on the actual database’s state.

**Proposition B.20.** Let \( C = \langle D, \Gamma \rangle \) be a system configuration, \( ATK \) be a C-ATKLOG model, and \( (u, q) \) be a C-query. ANGERONA\((C, s, h, ATK, (u, q)) = \text{ANGERONA}(C, s', h', ATK, (u, q)) \) whenever \( \text{observations}(h, u) = \text{observations}(h', u) \).

**Proof.** Let \( C = \langle D, \Gamma \rangle \) be a system configuration, \( ATK \) be an ATKLOG model, and \( (u, q) \) be a C-query. Furthermore, let \( s, s' \) be C-states and \( h, h' \) be C-histories such that \( \text{observations}(h, u) = \text{observations}(h', u) \). Assume, for contradiction’s sake, that \( \text{ANGERONA}(C, s, h, ATK, (u, q)) \neq \text{ANGERONA}(C, s', h', ATK, (u, q)) \). Without loss of generality, assume that \( \text{ANGERONA}(C, s, h, ATK, (u, q)) = \top \) and \( \text{ANGERONA}(C, s', h', ATK, (u, q)) = \bot \). This happens if either the result of \( pos \) or \( secure \) is different in the two cases. The results of \( pos \) and \( secure \), however, depend just on the attacker’s initial beliefs, the query \( q \), and the set of formulae in \( \text{observations}(h, u) \). From
this and \( \text{observations}(h,u) = \text{observations}(h',u) \), it follows that \( \text{ANGERONA}(C,s,h,\text{ATK},(u,q)) = \text{ANGERONA}(C,s',h',\text{ATK},(u,q)) \).

We now prove a key result for our security proof, namely that considering only the sentences in \( \text{observations}(h,u) \) is enough to determine secrecy-preservation.

**Proposition B.21.** Let \( C = (D,\Gamma) \) be a system configuration, \( \text{ATK} \) be a \( C \)-\text{ATKLog} model, \( f \) be the \( C \)-\text{PDP} obtained by parametrizing Algorithm 1 with \( C \) and \( \text{ATK} \), \( u \in U \) be a user, and \( r = (s,h) \) be a \((C,f)\)-run. The following fact holds: \( [r]_{\sim_u} = \{ db \in \Omega_D^r \mid \bigwedge_{\phi \in \text{observations}(h,u)} [\phi]^{\text{db}} = \top \} \).

**Proof.** \((\Rightarrow)\). Let \( C = (D,\Gamma) \) be a system configuration, \( \text{ATK} \) be an \text{ATKLog} model, \( f \) be the \( C \)-\text{PDP} obtained by parametrizing Algorithm 1 with \( C \) and \( \text{ATK} \), \( u \in U \) be a user, and \( r = (\langle db,U,P \rangle,h) \) be a \((C,f)\)-run. Furthermore, let \( r' = (\langle db',U',P' \rangle,h') \) be a run in \([r]_{\sim_u} \). From this, it follows that \( h'_{|u} = h_{|u} \). From this, it follows that all queries in \( \text{observations}(h,u) \) have the same result in \( db \) and \( db' \). Therefore, \( \langle db' \rangle \in \{ db \in \Omega_D^r \mid \bigwedge_{\phi \in \text{observations}(h,u)} [\phi]^{\text{db}} = \top \} \).

\((\Leftarrow)\). Let \( C = (D,\Gamma) \) be a system configuration, \( \text{ATK} \) be an \text{ATKLog} model, \( f \) be the \( C \)-\text{PDP} obtained by parametrizing Algorithm 1 with \( C \) and \( \text{ATK} \), \( u \in U \) be a user, and \( r = (\langle db,U,P \rangle,h) \) be a \((C,f)\)-run. Furthermore, let \( r' = (\langle db',U',P' \rangle,h') \) be a run in \([r]_{\sim_u} \). From this, it follows that \( h'_{|u} = h_{|u} \). From this, it follows that all queries in \( \text{observations}(h,u) \) have the same result in \( db \) and \( db' \). Therefore, \( \langle db' \rangle \in \{ db \in \Omega_D^r \mid \bigwedge_{\phi \in \text{observations}(h,u)} [\phi]^{\text{db}} = \top \} \).

To prove our claim that \( (\langle db',U,P \rangle,h_{|u}) \) is a run, we prove the stronger fact that for all \( 0 \leq i \leq |h| \), \( \langle db',U,P \rangle,h'_{|u} \) is a run. We prove this by induction on \( i \).

**Base case.** The base case \( (\langle db',U,P \rangle,h_{|u}) \) is trivial since \( db' \in \Omega_D^r \) and \( h_{|u} = \epsilon \), therefore \( (\langle db',U,P \rangle,h_{|u}) \) is a run.

**Induction Step.** Assume that \( (\langle db',U,P \rangle,h'_{|u}) \) is a run. We now show that \( (\langle db',U,P \rangle,h_{|u}) \) is a run as well. Let \((u',q),a,\text{res})\) be the last \( C \)-event in \( h' \). There are two cases:

1. \( u' \neq u \). From this, \( h_{|u}' = h_{|u} \). Therefore, \( (\langle db',U,P \rangle,h_{|u}') = (\langle db',U,P \rangle,h_{|u}) \) and our claim directly follows from the induction hypothesis.
2. \( u' = u \). From this, \( h_{|u}' = h_{|u} \). Assume, for contradiction’s sake, that \( (\langle db',U,P \rangle,h_{|u}') \) is not a run. From \( (\langle db',U,P \rangle,h_{|u}') \) is a run (which directly follows from the induction hypothesis), it follows that there are only three cases:

   1. \( f(\langle db',U,P \rangle,(u,q),h_{|u}') \neq a \). Since \( a \) is the security decision associated with the last event in \( h' \), it follows that \( a = f(\langle db',U,P \rangle,(u,q),h_{|u}') \). From this, it follows that \( f(\langle db',U,P \rangle,(u,q),h_{|u}') \neq f(\langle db',U,P \rangle,(u,q),h_{|u}') \). From \text{observations}' definition, it follows that \( \text{observations}(h_{|u}),u) = \text{observations}(h_{|u},u) \). From this and Proposition B.20, it follows that \( f(\langle db',U,P \rangle,(u,q),h_{|u}') = f(\langle db',U,P \rangle,(u,q),h_{|u}') \), leading to a contradiction.

   2. \( a = \bot \) but \( \text{res} \neq \bot \). This contradicts the fact that the history is derived from \( h' \), which comes from a proper run.

   3. \( a = \top \) but \( \text{res} \neq [q]^{\text{db}} \). From \( \text{res} \neq [q]^{\text{db}} \) (since \( \text{res} \) comes from the run \( r \)), it follows that \( [q]^a \neq [q]^{\text{db}} \). There are two cases:

      - \( [q]^a \neq \top \). From this and the definition of \text{observations}, it follows that \( \text{res} \in \text{observations}(h,u) \). Therefore, \( [q]^{\text{db}} = \top \) follows from \( db' \in \{ db' \in \Omega_D^r \mid \bigwedge_{\phi \in \text{observations}(h,u)} [\phi]^{\text{db}} \} \), leading to a contradiction.

      - \( [q]^{\text{db}} = \bot \). From this and the definition of \text{observations}, it follows that \( \neg q \in \text{observations}(h,u) \). From this and \( db' \in \{ db' \in \Omega_D^r \mid \bigwedge_{\phi \in \text{observations}(h,u)} [\phi]^{\text{db}} \} \), it follows that \( \neg [q]^{\text{db}} = \top \). From this, \( [q]^{\text{db}} = \bot \), leading to a contradiction.

This completes the proof of our claim.

Propositions B.22 and B.23 state the correctness of the secure and pax procedure.

**Proposition B.22.** Let \( C = (D,\Gamma) \) be a system configuration, \( \text{ATK} \) be a \( C \)-\text{ATKLog} model, \( f \) be the \( C \)-\text{PDP} obtained by parametrizing Algorithm 1 with \( C \) and \( \text{ATK} \), \( \text{ATK}' = \lambda u \in U.\|\text{ATK}(u)\|_D \) be the \((C,f)\)-attacker model associated to \( \text{ATK} \), \( r = (s,h) \) be a run in \text{runs}(C,f), and \( (u,\psi,l) \) be a secret in \text{r.S}. Then, secure(\text{C},\text{ATK},h,(u,\psi,l)) returns true iff \( \text{ATK}'(u,(s,h),(\psi,l))) < l \).

**Proof.** Let \( r = (s,h) \) be a run such that \( s \) is compatible with \( h \). The secure procedure returns \( \|\text{ATK}(u)\|_D(\|\psi\|\mid \bigwedge_{\phi \in \text{observations}(h,u)} [\phi]) < l \). From Proposition B.21, it follows that \( [r]_{\sim_u} = \{ db \in \Omega_D^r \mid \bigwedge_{\phi \in \text{observations}(h,u)} [\phi]^{\text{db}} = \top \} = \bigcap_{h \in \text{observations}(h,u)} [\phi]^{\text{db}} \). We can therefore rewrite secure’s result as \( \|\text{ATK}(u)\|_D(\|\psi\|\mid [s,h]_{\sim_u}) < l \). This is equivalent to \( \|\text{ATK}'(u,(s,h),(\psi,l))) < l \).
Appendix B. Proofs for Chapter 4

Proposition B.23. Let \( C = (D, \Gamma) \) be a system configuration, \( \text{ATK} \) be a \( C \)-\text{AtkLog} model, \( f \) be the \( C \)-PDP obtained by parametrizing Algorithm 1 with \( C \) and \text{ATK}, \( \text{ATK}' = \lambda u \in U. [\text{ATK}(u)]_D \) be the \((C, f)\)-attacker model associated to \( \text{ATK} \), \( r = (s, h) \) be a run in runs\((C, f)\), and \((u, q)\) be a query. Then, \( \text{poz}(C, \text{ATK}, h, (u, q)) \) returns true iff \( [\text{ATK}']((u, (s, h))(q)) > 0 \).

Proof. Let \( r = (s, h) \) be a run such that \( s \) is compatible with \( h \). The poz procedure returns \( [\text{ATK}](q) \sum_{\text{observations}(h, u)[(u, q)]} > 0 \). From Proposition B.21, it follows that \( [r']_{i,u} = \{ db \in \cap_{h \in \text{observations}(h, u)[(u, q)]} [\text{ATK}(u)]_D \} \). We can therefore rewrite poz’s result as \( [\text{ATK}']((q))([h, s]) < l. \) This is equivalent to \( [\text{ATK}']((u, (s, h))(q)) > 0 \).

We now prove that Angerona provides the desired security guarantees.

Theorem B.4. Let \( C = (D, \Gamma) \) be a system configuration, \( \text{ATK} \) be a \( C \)-\text{AtkLog} model, \( f \) be the \( C \)-PDP obtained by parametrizing Algorithm 1 with \( C \) and \text{ATK}, and \( \text{ATK}' = \lambda u \in U. [\text{ATK}(u)]_D \) be the \((C, f)\)-attacker model associated to \( \text{ATK} \). The PDP shown in Algorithm 1, parametrized with \( \text{ATK} \), provides data confidentiality with respect to \( C \) and \( \text{ATK}' \).

Proof. Let \( C = (D, \Gamma) \) be a system configuration, \( \text{ATK} \) be a \( C \)-\text{AtkLog} model, \( f \) be the \( C \)-PDP obtained by parametrizing Algorithm 1 with \( C \) and \text{ATK}, and \( \text{ATK}' = \lambda u \in U. [\text{ATK}(u)]_D \) be the \((C, f)\)-attacker model associated to \( \text{ATK} \). Furthermore, let \( f \) be the PDP shown in Algorithm 1. Assume, for contradiction’s sake, that \( f \) does not provide confidentiality with respect to \( C \) and \( \text{ATK}' \).

From this, it follows that there is a run \( r = ((db, U, P), h), \) a user \( u \in U \), a secret \((u, o, l) \in P \), and a \( 0 \leq i \leq |h| - 1 \) such that \( [\text{ATK}']((u, r^i))(o) < l \) and \( [\text{ATK}']((u, r^{i+1}))(o) \geq l \). As a result, the \( i \)-th query is the one that leaked information. There are two cases:

- The C-query is \((u, q)\), for some \( q \). There are three cases: - The result of the PDP is \( \top \) and the query \( q \) holds in the current state. From this, it follows that there is a database state (namely, the one in \( r \)) where the query \( q \) holds and the database is consistent with observations\((h, u)\). From this, it follows that \( [\text{ATK}']((u, r^i))(o) > 0 \). From this and Proposition B.23, it follows that \( \text{poz}(C, \text{ATK}, h^i, (u, q)) \) returns true. From this and the fact that the query has been authorized, it follows that \( \text{secure}(C, \text{ATK}, h^i, (u, o, l)) \) returned true. From this, the fact that \( r \) is actually a run (since \( q \) is satisfied in \( r.db \)), and Proposition B.22, it follows that \( [\text{ATK}']((u, r^{i+1}))(o) \leq l \), leading to a contradiction.

- The result of the PDP is \( \bot \) (namely the query is not authorized). From this and Proposition B.20, it follows that for any run \( r' \neq r \), the PDP result is the same. Therefore, \( [r^i]_{i,u} = [r^{i+1}]_{i,u} \). From this, \( [\text{ATK}']((u, r^i))(o) \leq l \), and \text{AtkLog} semantics, it follows that \( [\text{ATK}']((u, r^{i+1}))(o) \leq l \), leading to a contradiction.

- The query is \((u', q)\), where \( u' \neq u \) and \( q \) is a query. From this, it follows that \( [\text{ATK}']((u, r^i))(o) = [\text{ATK}']((u, r^{i+1}))(o) \) since \( u \)'s belief does not change in response to a query from another user (since \( r^i_{i,u} = r^{i+1}_{i,u} \)). From this and \( [\text{ATK}']((u, r^i))(o) \leq l \), and \( \text{AtkLog} \) semantics, it follows that both \( [\text{ATK}']((u, r^{i+1}))(o) \leq l \) and \( [\text{ATK}']((u, r^{i+1}))(o) \geq l \), leading to a contradiction. Since all cases ended in contradiction, this completes the proof of our claim.

B.6.2 Complexity Proof

First, we formalize literal queries, a fragment of relational calculus that can be composed with relaxed acyclic programs without modifying acyclicity. Afterwards, we prove that for acyclic AtkLog models and literal queries, Angerona has \( \text{PTime} \) data complexity.

A literal query is a quantifier-free relational calculus \((\Sigma, \text{dom})\)-formula either of the form \( R(\tau) \) or \( \neg R(\tau) \), where \( R \in \Sigma \) and \( \tau \in \text{dom}^{[0]} \). We say that a literal query \( R(\tau) \) (or \( \neg R(\tau) \)) is compatible with a relaxed acyclic program \( W \) (with witness \( W = (\pi_H, \pi_V, \mu) \)) iff \( \mu(R) \neq \emptyset \) implies \( \exists_{\mu(R) \in \text{dom}(R,p,\mu(R))} \). We now show that literal queries can be composed with relaxed ProbLog programs without introducing cycles.

Proposition B.24. Let \( D = (\Sigma, \text{dom}) \) be a database schema, \( p \) be a relaxed acyclic program, \( W \) be a witness for \( p \), \( R \) be a predicate symbol in \( \Sigma \) \( \in \text{dom}^{[0]} \) be a tuple, and \( \phi \) be a boolean \( D \)-query. If \( \phi \) is a literal query compatible with \( p \), then \( p \cup \text{PL}(\phi) \) is a relaxed acyclic ProbLog program as well.
Proof. Let \( D = (\Sigma, \Gamma) \) be a database schema, \( p \) be a relaxed acyclic program, and \( \phi \) be a boolean \( D \)-query. Furthermore, let \( \phi \) be a literal query. Assume, for contradiction’s sake, that \( p \cup PL(\phi) \) is not a relaxed acyclic program. This happens iff the program \( \alpha(\beta_\phi(p \cup PL(\phi))) \) is not an acyclic program. This happens iff the program \( \alpha(\beta_\phi(p \cup PL(\phi))) \) contains an unguarded cycle \( C \). The only interesting case is when \( C \) contains rules from \( PL(\phi) \). If \( \phi = R(\tau) \), then \( PL(\phi) \) contains a single rule \( fresh \leftarrow R(\tau) \), where \( fresh \) is a fresh predicate symbol. Similarly, if \( \phi = \neg R(\tau) \), then \( PL(\phi) \) contains two rules \( fresh_1 \leftarrow fresh_2 \) and \( fresh_2 \leftarrow \neg R(\tau) \), where \( fresh_1 \) and \( fresh_2 \) are fresh predicate symbols. From this, it follows that \( C \) is an undirected cycle (which is not a directed cycle). Observe that all rules in \( PL(\phi) \) are both strongly and weakly connected for \( T \). Hence, it follows that there is always an unguarded cycle \( C' \) that can be obtained from \( C \) by removing the rules in \( PL(\phi) \). This contradicts the fact that \( p \) is a relaxed acyclic program. 

Here, we show that for acyclic ATKLog models, ANGERONA has \( \text{Ptime} \) data complexity. Observe that we say that an ATKLog-model is relaxed acyclic if so are all the ProbLog programs in it.

Theorem B.5. Let \( C \) be a system configuration and \( f \) be the PDP shown in Algorithm 1. For any relaxed acyclic \( C \)-ATKLog model ATK, any action \((u,q)\) such that \( q \) is a literal query compatible with \( p \), and any run \( r \in \text{runs}(C,f) \) such that (1) all sentences in observations\((r,h,u)\), for any \( u \in r.U \), are literal queries compatible with \( p \), and (2) all secrets in \( r.S \) are literal queries compatible with \( p \), the algorithm shown in Algorithm 1 has \( \text{Ptime} \) data complexity.

Proof. Let \( B \) be the ProbLog program obtained from \( ATK \). The data complexity of \( f \) is its complexity when only the database \( r.db \) and \( \text{prob}(B) \) change. The algorithm in Algorithm 1 calls three times the secure procedure and twice the poz procedure. The set observations\((h,u)\) can be constructed in \( O(|h|) \). From this, it follows that the program \( p \) has size \( O(|\text{prob}(B)| + |\text{rules}(B)| + |h|) \). Furthermore, since all queries are literal queries, \( B \) is a relaxed acyclic ProbLog program, and Proposition B.24, it follows that \( p \) is a relaxed acyclic ProbLog program as well. From this and Theorem B.3, the inference can be performed in \( O(r^2 \cdot e^{\cdot \text{max}(2, r)^{2+k}}) \), where \( r \in O(|\text{rules}(B)| + (|r| + |\text{prob}(K)|)^{|\text{rules}(K)})) \) and \( e \in O(|\text{prob}(B)|) \). Since rules\((B), r, \) and \( K \) are fixed, then the data complexity of the secure procedure is \( O(|\text{prob}(B)|^k) \), for some \( k \in \mathbb{N} \). From this and the fact that Algorithm 1 three times the secure procedure and twice the poz procedure per secret in \( r.S \), it follows that the data complexity of Algorithm 1 is \( \text{Ptime} \).

B.6.3 Completeness Proof

We first introduce the notion of unconditionally secrecy-preserving query. Informally, a query \( \langle u', q \rangle \) is unconditionally secrecy-preserving given a run \( r \) and a secret \( \langle u, \psi, l \rangle \) iff disclosing the result of \( \langle u', q \rangle \) in any run \( r' \sim_u r \) does not violate the secret.

Definition B.3. Let \( C = (D, \Gamma) \) be a configuration, \( f \) be a \( C \)-PDP, and \( ATK \) be a \( (C,f) \)-attacker model. A query \( q \) is unconditionally secrecy-preserving for a \((C,f)\)-run \( r \), a user \( u \), a secret \( \langle u, \psi, l \rangle \), and \( ATK \) iff \( [ATK](u[r],\psi) \leq l \) implies that (a) if \( [ATK](u[r],q) > 0 \), then \( [ATK](u[r],q) \leq l \), and (b) if \( [ATK](u[r],\neg q) > 0 \), then \( [ATK](u[r],\neg q) \leq l \). 

We say that a PDP is complete if it authorizes all unconditionally secrecy-preserving queries. ANGERONA is complete. This directly follows from (1) Proposition B.21, (2) the use of exact inference procedures for ProbLog programs, and (3) the fact that ANGERONA directly checks whether queries are unconditionally secrecy-preserving or not.

Proposition B.25. Let \( C = (D, \Gamma) \) be a system configuration, \( ATK \) be a \( C \)-ATKLog model, \( f \) be the \( C \)-PDP obtained by parametrizing Algorithm 1 with \( C \) and \( ATK \), \( ATK' = \lambda u.\langle [ATK(u)]_D \rangle_D \) be the \( (C,f) \)-attacker model associated to \( ATK \), \( r = \langle \langle db, U, P \rangle, h \rangle \) be a run, and \( \langle u, q \rangle \) be a \( C \)-query. If \( q \) is unconditionally secrecy-preserving for \( u, r, ATK' \), and all secrets \( \langle u, \psi, l \rangle \in \text{secrets}(P, u) \), then \( f \) authorizes \( \langle u, q \rangle \).

Proof. Let \( C = (D, \Gamma) \) be a system configuration, \( ATK \) be a \( C \)-ATKLog model, \( f \) be the \( C \)-PDP obtained by parametrizing Algorithm 1 with \( C \) and \( ATK \), \( ATK' = \lambda u.\langle [ATK(u)]_D \rangle_D \) be the \( (C,f) \)-attacker model associated to \( ATK \), \( r = \langle \langle db, U, P \rangle, h \rangle \) be a run, and \( \langle u, q \rangle \) be a \( C \)-query. Assume, for contradiction’s sake, that our claim does not hold. Namely, there is query \( \langle u, q \rangle \) such that (1) the query is unconditionally secrecy-preserving \( u \), for all secrets \( \langle u, \psi, l \rangle \in \text{secrets}(P, u) \), \( r \), and \( ATK' \), and (2) ANGERONA (parametrized with \( ATK \)) does not authorize \( \langle u, q \rangle \). Since ANGERONA does not authorize \( \langle u, q \rangle \), it means that ANGERONA\((C,\langle db, U, P \rangle, h, ATK, \langle u, q \rangle) = \top \). From this, it follows that there is a secret \( \langle u, \psi, l \rangle \in \text{secrets}(P, u) \) such that:

- \( \text{secure}(C, ATK, h, \langle u, \psi, l \rangle) = \top \), and
- one of the two cases hold:
\[ \text{pox}(C, \text{ATK}, h, (u, q)) = \top \text{ and secure}(C, \text{ATK}, h', (u, \psi, l)) = \bot, \text{ where } h' = h \cdot ((u, q), \top, \top), \text{ or} \]

\[ \text{pox}(C, \text{ATK}, h, (u, \neg q)) = \top \text{ and secure}(C, \text{ATK}, (db, U, P), h'', (u, \psi, l)) = \bot, \text{ where } h'' = h \cdot ((u, q), \top, \bot). \]

Without loss of generality, we assume that \( \text{pox}(C, \text{ATK}, h, (u, q)) = \top \) and \( \text{secure}(C, \text{ATK}, h'(u, \psi, l)) = \top \) (the proof for the other case is identical). From \( \text{secure}(C, \text{ATK}, h, (u, \psi, l)) = \top \), Proposition B.22, and the fact that \( s \) is compatible with \( h \), it follows that \( \| \text{ATK}' \|(u, r)(\psi) > 0 \). From this and Proposition B.20, it follows that there is a system state \( s' = (db', U, P) \) such that (1) \( q \) holds in \( db' \), (2) \( s' \) is compatible with \( h' \), and (3) \( r' = (s', h') \) is a run indistinguishable from \( r \). From this, \( \text{secure}(C, \text{ATK}, h', (u, \psi, l)) = \top \), \( \text{secure}(C, \text{ATK}, (db, U, P), h'|_u, (u, \psi, l)) \), and Proposition B.22, it follows that \( \| \text{ATK}' \|(u, (s', h'|_u))(\psi) \geq l \). From this and the definition of \( h'|_u \), it follows that \( \| \text{ATK}' \|(u, r')(\psi | \{ db \in [r] | q = \top \}) \geq l \). From this, \( r \sim_u r' \), and \( \text{ATK}'(u, r) = \| \text{ATK}' \|(u, r') \) if \( r \sim_u r' \), it follows that \( \| \text{ATK}' \|(u, r)(\psi | \{ db \in [r] | q = \top \}) \geq l \). Therefore, we have that \( \| \text{ATK}' \|(u, r)(\psi) < l \), \( \| \text{ATK}' \|(u, r)(q) > 0 \), and \( \| \text{ATK}' \|(u, r)(\psi | \{ db \in [r] | q = \top \}) \geq l \). This contradicts the fact that \( q \) is an unconditionally secrecy-preserving query and completes the proof of our claim. \( \square \)
Appendix C

Proofs for Chapter 6

Here, we provide proofs of all the results in Chapter 6. For simplicity’s sake, we assume, without loss of generality, that all the relational calculus formulae do not use constant symbols inside predicates. For instance, instead of the formula $\exists x. R(x, 5, 10)$, we consider the equivalent formula $\exists x, y, z. R(x, y, z) \land y = 5 \land z = 10$. Furthermore, for readability, we write $secure_{P,u}(r, i \vdash_u \phi)$ instead of $secure_{P,R,P,u}(r, i \vdash_u \phi)$.

C.1 Soundness of the attacker model

Proposition C.1 states that our attacker model is sound with respect to the relational calculus semantics, i.e., every judgment $r, i \vdash_u \phi$ that holds in $\mathcal{ATK}_u$ is such that $\phi$ is satisfied in the $i$-th state of $r$. We first introduce the concept of derivation. Given a judgment $r, i \vdash_u \phi$, a derivation of $r, i \vdash_u \phi$ with respect to $\mathcal{ATK}_u$, or a derivation of $r, i \vdash_u \phi$ for short, is a proof tree, obtained by applying the rules defining $\mathcal{ATK}_u$, that ends in $r, i \vdash_u \phi$. With a slight abuse of notation, we use $r, i \vdash_u \phi$ to denote both the judgment and its derivation. The length of a derivation, denoted $|r, i \vdash_u \phi|$, is the number of rule applications in it. Note that a judgments $r, i \vdash_u \phi$ holds in $\mathcal{ATK}_u$ if there is a derivation for it.

Proposition C.1. Let $P$ be an extended configuration, $L$ be the $P$-LTS, $u$ be a user, $r \in \text{traces}(L)$ be an $L$-run, $\phi \in \mathcal{RC}_\text{bool}$ be a sentence, and $1 \leq i \leq |r|$. If $r, i \vdash_u \phi$ holds in $\mathcal{ATK}_u$, then $|\phi|^{db} = \top$, where $\text{last}(r^i) = \langle dU, s, T, V, c \rangle$.

Proof. Let $P$ be an extended configuration, $L$ be the $P$-LTS, $u$ be a user, $r \in \text{traces}(L)$ be an $L$-run, $\phi \in \mathcal{RC}_\text{bool}$ be a sentence, and $1 \leq i \leq |r|$. Furthermore, let $r, i \vdash_u \phi$ be a judgment that holds in $\mathcal{ATK}_u$, i.e., there is a derivation $d$ of $r, i \vdash_u \phi$ with respect to $\mathcal{ATK}_u$. We prove our claim by induction on the length of $d$.

Base Case: The base case is a derivation of length 1. There are a number of cases depending on the rule used to obtain $r, i \vdash_u \phi$.

1. SELECT Success - 1. Let $i$ be such that $r^i = r^{i-1} \langle u, \text{SELECT}, \phi \rangle$, where $s = \langle dU, s, T, V, c \rangle \in \Omega_M$ and $\text{last}(r^{i-1}) = \langle dU, s, T, V, c \rangle$. From the rules, it follows that $\text{res}(s) = \top$. From this and the LTS rules, it follows that $|\phi|^{db} = \top$.

2. INSERT Success - 2. The proof for this case is similar to that of SELECT Success - 1.

3. INSERT Success. Let $i$ be such that $r^i = r^{i-1} \langle u, \text{INSERT}, R, \bar{T} \rangle$, where $s = \langle dU, s, T, V, c \rangle \in \Omega_M$, $\text{last}(r^{i-1}) = \langle dB', s, T, V, c' \rangle$, and $\phi$ be $R(\bar{T})$. From the LTS rules, it follows that $\text{db} = \text{db}' [R \oplus \bar{T}]$. From $\oplus$’s definition, it follows that $\bar{T} \in \text{db}(R)$. Therefore, from the RC’s semantics, it follows that $[R(\bar{T})]^{db} = \top$. Since $\phi := R(\bar{T})$, it follows that $|\phi|^{db} = \top$.

4. INSERT Success - FD. Let $i$ be such that $r^i = r^{i-1} \langle u, \text{INSERT}, R, (\langle \bar{\tau}, \bar{\omega}, \bar{\gamma} \rangle) \rangle$, where $s = \langle dB, s, T, V, c \rangle \in \Omega_M$, $\text{last}(r^{i-1}) = \langle dB', s, T, V, c' \rangle$, and $\phi$ be $\neg \exists y. \exists z. R(\langle \bar{\tau}, \bar{\omega}, \bar{\gamma} \rangle \land \bar{\gamma} \neq \bar{\omega})$. We claim that $|\phi|^{db}$ holds. From this claim and the LTS semantics, it follows that there is no tuple $(\bar{\tau}', \bar{\omega}', \bar{\gamma}')$ in $\text{db}(R)$ such that $\bar{\tau}' = \bar{\tau}$ and $\bar{\omega}' \neq \bar{\omega}$. There are two cases:

(a) The INSERT command causes an integrity exception, i.e., $Ex(s) \neq \emptyset$. From this and the LTS semantics, it follows that $\text{db} = \text{db}'$. From this and $|\phi|^{db} = \top$, it follows that also $|\phi|^{db} = \top$.

(b) The INSERT command does not cause an integrity exception, i.e., $Ex(s) = \emptyset$. From this, $|\phi|^{db} = \top$, and $\text{db}(R) = \text{db}'(R) \cup \langle (\bar{\tau}, \bar{\omega}, \bar{\gamma}) \rangle$, it follows that there is no tuple $(\bar{\tau}', \bar{\omega}', \bar{\gamma}')$ in $\text{db}(R)$ such that $\bar{\tau}' = \bar{\tau}$ and $\bar{\omega}' \neq \bar{\omega}$. From this, it follows that also $|\phi|^{db}$ holds.

We now prove our claim that $|\phi|^{db}$ holds. Assume, for contradiction’s sake, that this is not the case. This means that there is a tuple $(\bar{\tau}', \bar{\omega}', \bar{\gamma}')$ in $\text{db}(R)$ such that $\bar{\tau}' = \bar{\tau}$ and $\bar{\omega}' \neq \bar{\omega}$. Let $\text{db}''$ be the state $\text{db}'(R \oplus \langle \bar{\tau}, \bar{\omega}, \bar{\gamma} \rangle)$. From this and $(\bar{\tau}', \bar{\omega}', \bar{\gamma}') \in \text{db}''(R)$ such that $\bar{\tau}' = \bar{\tau}$ and $\bar{\omega}' \neq \bar{\omega}$, it follows that there are two tuples $(\bar{\tau}, \bar{\omega}, \bar{\gamma})$ and $(\bar{\tau}, \bar{\omega}', \bar{\gamma}')$ in $\text{db}''(R)$ such that $\bar{\gamma}' \neq \bar{\omega}$. From this, it follows that both $\langle \bar{\tau}, \bar{\omega}, \bar{\gamma}, \bar{\omega}', \bar{\gamma}' \rangle$ and $\langle (R(\bar{\tau}, \bar{\omega}, \bar{\gamma}) \land R(\bar{\tau}, \bar{\omega}', \bar{\gamma}')) \rightarrow \bar{\gamma} = \bar{\gamma}' \rangle^{db} = \top$. This contradicts the fact that $\forall \bar{\tau}, \bar{\omega}, \bar{\gamma}, \bar{\omega}', \bar{\gamma}'. ((R(\bar{\tau}, \bar{\omega}, \bar{\gamma}) \land R(\bar{\tau}, \bar{\omega}', \bar{\gamma}'))) \rightarrow \bar{\gamma} = \bar{\gamma}'$ is not in $Ex(s)$. 

191
5. **INSERT Success - ID.** Let \( i \) be such that \( r^i = r^{i-1}.\langle u, \text{INSERT}, R, (\mathcal{V}, \mathcal{W}) \rangle \cdot s \), where \( s = \langle db, U, sec, T, V, c \rangle \in \Omega_M \), \( \text{last}(r^{i-1}) = \langle db', U, sec, T, V, c' \rangle \), and \( \phi \) be \( \not\exists \mathcal{W} S(\mathcal{V}, \mathcal{W}) \). We claim that \([\phi]^{db'}\) holds. From this claim and the LTS semantics, it follows that there is a tuple \((\mathcal{V}', \mathcal{W}')\) in \( db'(S) \) such that \( \mathcal{V}' = \mathcal{V} \). There are two cases:

(a) The **INSERT** command causes an integrity exception, i.e., \( Ex(s) \neq \emptyset \). From this and the LTS semantics, it follows that \( db = db' \). From this and \([\phi]^{db} = \top\), it follows that also \([\phi]^{db} = \top\)

(b) The **INSERT** command does not cause any integrity exception, i.e., \( Ex(s) = \emptyset \). From this, \([\phi]^{db'} = \top\) and \( db(S) = db'(S) \), it follows that there a tuple \((\mathcal{V}', \mathcal{W}')\) in \( db(S) \) such that \( \mathcal{V}' = \mathcal{V} \). From this, it follows that also \([\phi]^{db} = \top\).

We now prove our claim that \([\phi]^{db'}\) holds. Assume, for contradiction’s sake, that this is not the case. This means that there is no tuple \((\mathcal{V}, \mathcal{W})\) in \( db'(S) \) such that \( \mathcal{V} = \mathcal{V} \). From this and the LTS semantics, it follows that there a tuple \((\mathcal{V}, \mathcal{W})\) in \( db(S) \) such that \( \mathcal{V} = \mathcal{V} \). From this, it follows that also \([\phi]^{db} = \top\).

We now prove our claim that \([\phi]^{db'}\) holds. Assume, for contradiction’s sake, that this is not the case. This means that there is no tuple \((\mathcal{V}, \mathcal{W})\) in \( db'(S) \) such that \( \mathcal{V} = \mathcal{V} \). From this and the LTS semantics, it follows that there a tuple \((\mathcal{V}, \mathcal{W})\) in \( db(S) \) such that \( \mathcal{V} = \mathcal{V} \). From this, it follows that also \([\phi]^{db} = \top\).

We now prove our claim that \([\phi]^{db'}\) holds. Assume, for contradiction’s sake, that this is not the case. This means that there is no tuple \((\mathcal{V}, \mathcal{W})\) in \( db'(S) \) such that \( \mathcal{V} = \mathcal{V} \). From this and the LTS semantics, it follows that there a tuple \((\mathcal{V}, \mathcal{W})\) in \( db(S) \) such that \( \mathcal{V} = \mathcal{V} \). From this, it follows that also \([\phi]^{db} = \top\).
if $\tau = \tau'$, then $\tau'' = \tau''$.

Furthermore, from $db' \in \Omega_R$ , it follows that for all tuples ($\tau'$, $\tau''$, $\tau'''$) and ($\tau''$, $\tau'''$, $\tau''''$) in $db(R)$, it follows that $\forall \tau \neq \tau$, then $\forall \tau'' \neq \tau$. From this and $db[R \oplus (\tau', \tau', \tau'')] \forall \tau \neq \tau$, then $\forall \tau'' \neq \tau$. From these facts, it follows that $\forall \tau, \tau', \tau'' \in \Omega_R$, $(R(\tau, \tau', \tau'') \cap \tau(R(\tau', \tau'', \tau ))) \rightarrow \exists \tau' \in \Omega_R$, is in contradiction with the fact that the constraint $\forall \tau, \tau', \tau'' \in \Omega_R$, $(R(\tau, \tau', \tau'') \cap \tau(R(\tau', \tau'', \tau ))) \rightarrow \exists \tau' \in \Omega_R$.

11. INSERT ID Exception. Let $i$ be such that $r^* = r^{-1} \cup \{u, \text{INSERT}, R(\tau, \tau')\}$, where $s = (db', U, sec, T, V, c \in \Omega_M$), last$(r^{-1} \cup \{u, \text{INSERT}, R(\tau, \tau')\}) = \langle db', U, sec, T, V, c \rangle$, and $\phi$ be $\forall \tau, \tau', \tau'' \in \Omega_R$, $S(\tau, \tau') \rightarrow \tau \neq \tau$. We claim that $[\phi]^{db}$ holds.

From this claim and the LTS semantics, it follows that there is no tuple ($\tau, \tau'$) in $db'(S)$. From this and $db(S) = db'(S)$, it follows that there is no tuple ($\tau, \tau'$) in $db'(S)$. From this, it follows that also $[\phi]^{db}$ holds.

We now prove our claim that $[\phi]^{db}$. Assume, for contradiction’s sake, that this is not the case. This means that there is a tuple ($\tau, \tau'$) in $db'(S)$, for some $\tau'$. From this, it follows that for all tuples ($\tau, \tau'$) in $db'(R)$ such that $\tau \neq \tau$, there is a tuple ($\tau, \tau'$) in $db(R)$. From this, ($\tau, \tau'$) in $db'(S)$, $db[R \oplus (\tau, \tau')] \forall \tau \neq \tau$, and $db[R \oplus (\tau, \tau')] \forall \tau \neq \tau$, it follows that for all tuples ($\tau, \tau'$) in $db[R \oplus (\tau, \tau')]$, there is a tuple ($\tau, \tau'$) in $db[R \oplus (\tau, \tau')]$. From these facts, it follows that $\forall \tau, \tau', \tau'' \in \Omega_R$, $(\tau(R(\tau, \tau', \tau'')) \rightarrow \exists \tau' \in \Omega_R$, is in contradiction with the fact that the constraint $\forall \tau, \tau', \tau'' \in \Omega_R$, $(\tau(R(\tau, \tau', \tau'')) \rightarrow \exists \tau' \in \Omega_R$.

12. DELETE FD Exception. Let $i$ be such that $r^* = r^{-1} \cup \{u, \text{DELETE}, R(\tau, \tau')\}$, where $s = (db', U, sec, T, V, c \in \Omega_M$, last$(r^{-1} \cup \{u, \text{DELETE}, R(\tau, \tau')\}) = \langle db', U, sec, T, V, c \rangle$, and $\phi$ be $\forall \tau, \tau', \tau'' \in \Omega_R$, $S(\tau, \tau') \rightarrow \tau \neq \tau$. We claim that $[\phi]^{db}$ holds.

From this claim and the LTS semantics, it follows that there is a tuple ($\tau, \tau'$) in $db'(S)$ and all tuples ($\tau, \tau'$) in $db'(R)$ are such that $\tau = \tau$. From this, in $db'(S)$ and $db(S) = db'(S)$, it follows that ($\tau, \tau'$) in $db'(S)$. From this and all tuples ($\tau, \tau'$) in $db'(R)$ are such that $\tau = \tau$ and $db(R) = db'(R)$, it follows that all tuples ($\tau, \tau'$) in $db(R)$ are such that $\tau = \tau$. From this, in $db'(S)$ and the fact that all tuples ($\tau, \tau'$) in $db(R)$ are such that $\tau = \tau$, it follows that $[\phi]^{db}$ holds.

We now prove our claim that $[\phi]^{db}$. Assume, for contradiction’s sake, that this is not the case. There are two cases:

(a) all tuples ($\tau, \tau'$) in $db'(S)$ are such that $\tau \neq \tau$. From $db' \in \Omega_R$, it follows that for all tuples ($\tau, \tau'$) in $db'(R)$ such that $\tau \neq \tau$, there is a tuple ($\tau, \tau'$) in $db(R)$. From these facts, $db'[R \oplus (\tau, \tau')] \forall \tau \neq \tau$, and $db'[R \oplus (\tau, \tau')] \forall \tau \neq \tau$, it follows that for all tuples ($\tau, \tau'$) in $db[R \oplus (\tau, \tau')]$, there is a tuple ($\tau, \tau'$) in $db[R \oplus (\tau, \tau')]$. From this, it follows that $\forall \tau, \tau', \tau'' \in \Omega_R$, $(\tau(R(\tau, \tau', \tau'')) \rightarrow \exists \tau' \in \Omega_R$, is in contradiction with the fact that the constraint $\forall \tau, \tau', \tau'' \in \Omega_R$, $(\tau(R(\tau, \tau', \tau'')) \rightarrow \exists \tau' \in \Omega_R$.

(b) there is a tuple ($\tau, \tau'$) in $db'(R)$ such that $\tau \neq \tau'$. Furthermore, from $db' \in \Omega_R$, it follows that for all tuples ($\tau, \tau'$) in $db'(S)$ such that $\tau \neq \tau$, there is a tuple ($\tau, \tau'$) in $db'(R)$. From these facts, $db'[R \oplus (\tau, \tau')] \forall \tau \neq \tau$, and $db'[R \oplus (\tau, \tau')] \forall \tau \neq \tau$, it follows that for all tuples ($\tau, \tau'$) in $db'[R \oplus (\tau, \tau')]$, there is a tuple ($\tau, \tau'$) in $db'[R \oplus (\tau, \tau')]$. From this, it follows that $\forall \tau, \tau', \tau'' \in \Omega_R$, $(\tau(R(\tau, \tau', \tau'')) \rightarrow \exists \tau' \in \Omega_R$, is in contradiction with the fact that the constraint $\forall \tau, \tau', \tau'' \in \Omega_R$, $(\tau(R(\tau, \tau', \tau'')) \rightarrow \exists \tau' \in \Omega_R$.

13. Integrity Constraint. The proof of this case follows trivially from the fact that for any state $s = (db', U, sec, T, V, c \in \Omega_M and any $\gamma \in \Gamma$, $[\gamma]^{db} = \Gamma$ holds by definition.

14. Learn GRANT/REVOKE Backward. Let $i$ be such that $r^* = r^{-1} \cup \{u, \text{DELETE}, R(\tau, \tau')\}$, where $s = (db', U, sec, T, V, c \in \Omega_M$, last$(r^{-1} \cup \{u, \text{DELETE}, R(\tau, \tau')\}) = \langle db', U, sec', T, V, c \rangle$, and $t$ be a trigger whose WHEN condition is $\phi$ and whose action is either a GRANT or a REVOKE. From the rule’s definition, it follows $sec \neq sec'$. We now prove that $[\phi]^{db} = \Gamma$. Assume, for contradiction’s sake, that $[\phi]^{db}$. From this and the LTS rules for the triggers, it follows that the trigger $t$ is disabled. Therefore, according to the Trigger Disabled rule, $sec = sec'$. This case is similar to that of Trigger GRANT Disabled Backward.

15. Trigger GRANT Disabled Backward. Let $i$ be such that $r^* = r^{-1} \cup \{u, \text{DELETE}, R(\tau, \tau')\}$, where $s = (db', U, sec, T, V, c \in \Omega_M$, last$(r^{-1} \cup \{u, \text{DELETE}, R(\tau, \tau')\}) = \langle db', U, sec', T, V, c \rangle$, and $t$ be a trigger whose WHEN condition is $\psi$, and $\phi$ be $\psi$. Furthermore, let $q \in \Omega_R^{\omega}$ be the GRANT added by the trigger. From the rule’s definition, it follows $g \neq sec'$. We now prove that $[\phi]^{db} = \Gamma$. Assume, for contradiction’s sake, that $[\phi]^{db} = \Gamma$. This would imply that the trigger $t$ is enabled. Therefore, there are two cases:

(a) $t$’s execution is authorized. Therefore, $g \in sec'$, which contradicts $g \neq sec'$.

(b) $t$’s execution is not authorized. This contradicts secEx$(s) = \perp$.
last(r^{i-1}) = (db, U, sec', T, V, c'), and t be a trigger whose WHEN condition is φ and whose action is act. From the rule’s definition, it follows that t is enabled and that the evaluation of the WHEN condition is authorized. From this and the LTS’s rules, it follows that [φ]_{db} = T.

21. Trigger INSERT Exception. The proof for this case is similar to that of INSERT Exception.

22. Trigger DELETE Exception. The proof for this case is similar to that of DELETE Exception.

23. Trigger Rollback INSERT. Let i be such that r^{i} = r^{i-1-1}⟨u, INSERT, R, T⟩11.s1.t1.s2...t_{n} = \text{TRIGGER}_{D}, and φ be −R(T). Furthermore, let

\text{last(r^{i-1-1}) = (db', U', sec', T', V', c')} and s_{n} be (db, U, sec, T, V, c).

Assume, for contradiction’s sake, that [φ]_{db} = \bot. Therefore, \overline{t} \in \text{db}(R). From the LTS rules, it follows that db' = db. From this and \overline{t} \in \text{db}(R), it follows \overline{t} \notin \text{db'}(R). From r’s definition and the LTS rule INSERT Success - 2, it follows that \overline{t} \notin \text{db}(R), which leads to a contradiction.

24. Trigger Rollback DELETE. The proof for this case is similar to that of Trigger Rollback INSERT.

25. Learn from deny - actions. There are r, r', r'' ∈ traces(L), 1 ≤ i ≤ |r|, a ∈ \text{A}_{D,n}, s, s' ∈ \text{OM}, and φ such that:

r^{i} = r^{i-1}·a·s, r' = r''·a·s', r'' = r'' = r'' + 1 \geq r_{U} r'' = r'', secEx(s') \neq secEx(s), [φ]_{last(r^{i-1})} = \top, and [φ]_{last(r''')} = \bot. Then, [φ]_{last(r^{i-1})} = \top follows directly from the rule.

26. Learn from deny - triggers. The proof is similar to that of Learn from deny - actions. This completes the proof of the base step.

Induction Step: Assume that the claim holds for any derivation of r, j \vdash_{u} \psi such that [r, j \vdash_{u} \phi]. We now prove that the claim also holds for r, i \vdash_{u} \phi. There are a number of cases depending on the rule used to obtain r, i \vdash_{u} \phi.

1. View. The proof of this case follows trivially from the semantics of the relational calculus extended over views.

2. Propagate Forward SELECT. Let i be such that r^{i+1} = r^{i}·⟨u, SELECT, ψ⟩·s, where s = (db, U, sec', T, V, c) ∈ \text{OM} and last(r^{i}) = (db', U', sec', T', V', c'). From the rule’s definition, r, i \vdash_{u} \phi holds. From this, the induction hypothesis, and last(r^{i}) = (db', U', sec', T', V', c'), it follows that [φ]_{db'} = \top. From the LTS semantics, it follows that db = db'. From this and [φ]_{db'} = \top, it follows that [φ]_{db} = \top.

3. Propagate Forward CREATE. The proof for this case is similar to that of Propagate Forward SELECT.

4. Propagate Forward CREATE. The proof for this case is similar to that of Propagate Forward CREATE.

5. Propagate Backward SELECT. Let i be such that r^{i+1} = r^{i}·⟨u, SELECT, ψ⟩·s, where s = (db, U', sec', T', V', c') ∈ \text{OM} and last(r^{i}) = (db, U, sec, T, V, c). From the rule’s definition, r, i + 1 \vdash_{u} \phi holds. From this, the induction hypothesis, r^{i+1} = r^{i}·⟨u, SELECT, ψ⟩·s, and s = (db, U, sec, T, V, c), it follows that [φ]_{db'} = \top. From the LTS semantics, it follows that db = db'. From this and [φ]_{db'} = \top, it follows that [φ]_{db} = \top.

6. Propagate Backward CREATE. The proof is similar to that of Propagate Backward SELECT.

7. Propagate Backward CREATE VIEW. Let i be such that r^{i+1} = r^{i}·⟨u, CREATE, φ⟩·s', where s' = (db', U', sec', T', V', c') ∈ \text{OM} and last(r^{i}) = (db, U, sec, T, V, c). From the rule’s definition, r, i + 1 \vdash_{u} \phi holds. From this, the induction hypothesis, r^{i+1} = r^{i}·⟨u, CREATE, φ⟩·s', and s = (db, U, sec, T, V, c), it follows that [φ]_{db'} = \top. From the definition of replace, it follows that replace(φ', φ) and φ' are semantically equivalent. From this and [φ]_{db'} = \top, it follows that [φ]_{db} = \top.

8. Rollback Backward - 1. Let i be such that r^{i} = r^{i-1-1}·⟨u, op, R, T⟩11.s1.t1.s2...t_{n} = \text{TRIGGER}_{D}, and op is one of \{INSERT, DELETE\}. Furthermore, let s_{n} be (db', U', sec', T', V', c') ∈ \text{OM} and last(r^{i-1-1}) = (db, U, sec, T, V, c). From the rule’s definition, r, i \vdash_{u} \phi holds. From this, the induction hypothesis, and s_{n} = (db, U, sec, T, V, c) ∈ \text{OM}, it follows that [φ]_{db'} = \top. From the LTS semantics, it follows that db = db'. From this and [φ]_{db'} = \top, it follows that [φ]_{db} = \top.

9. Rollback Backward - 2. Let i be such that r^{i} = r^{i-1}·⟨u, op, R, T⟩·s, where s = (db', U', sec', T', V', c') ∈ \text{OM}, and last(r^{i-1}) = (db, U, sec, T, V, c), and op is one of \{INSERT, DELETE\}. From the rule’s definition, r, i \vdash_{u} \phi holds. From this, the induction hypothesis, r^{i} = r^{i-1}·⟨u, op, R, T⟩·s, and s = (db', U', sec', T', V', c'), it follows that [φ]_{db'} = \top. From the LTS semantics, it follows that db = db' (because a roll-back happened). From this and [φ]_{db'} = \top, it follows that [φ]_{db} = \top.

10. Rollback Backward - 1. Let i be such that r^{i} = r^{i-1-1}·⟨u, op, R, T⟩11.s1.t1.s2...t_{n} = \text{TRIGGER}_{D}, and op is one of \{INSERT, DELETE\}. Furthermore, let s_{n} be (db, U, sec, T, V, c) and last(r^{i-1-1}) = (db', U', sec', T', V', c'). From the rule’s definition, r, i \vdash_{u} \phi holds. From this, the induction hypothesis, r^{i} = r^{i-1-1}·⟨u, op, R, T⟩·s, and s = (db, U, sec, T, V, c), it follows that [φ]_{db'} = \top. From the LTS semantics, it follows that db = db' (because a roll-back happened). From this and [φ]_{db'} = \top, it follows that [φ]_{db} = \top.

11. Rollback Backward - 1. Let i be such that r^{i} = r^{i-1-1}·⟨u, op, R, T⟩11.s1.t1.s2...t_{n} = \text{TRIGGER}_{D}, and op is one of \{INSERT, DELETE\}. Furthermore, let s_{n} be (db, U, sec, T, V, c) and last(r^{i-1-1}) = (db', U', sec', T', V', c'). From the rule’s definition,
C.1. Soundness of the attacker model

12. Rollback Forward - 2. Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, op, R, T) \cdot s \), where \( op \in \{ \text{INSERT, DELETE} \} \). From the LTS semantics, it follows that \( db = db' \) (because a roll-back happened). From this and \([\phi]d^v = T\), it follows that \([\phi]db = T\).

13. Propagate Forward SUCCESS. Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, op, R, T) \cdot s \), where \( op \in \{ \text{INSERT, DELETE} \} \). From the rule's definition, \( r, i - 1 \vdash \phi \) holds. From this, the induction hypothesis, and \([\text{last}(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle \), it follows that \([\phi]d^v = T\). From the LTS semantics, it follows that \( db = db' \) (because a roll-back happened). From this and \([\phi]d^v = T\), it follows that \([\phi]db = T\).

14. Propagate Forward INSERT Success - 1. Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, op, R, T) \cdot s \), where \( op \in \{ \text{INSERT, DELETE} \} \). From the rule's definition, \( r, i - 1 \vdash \phi \) holds. From this, the induction hypothesis, and \([\text{last}(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle \), it follows that \([\phi]d^v = T\). From the LTS semantics, it follows that \( R \) does not occur in \( \phi \). From the LTS semantics, it follows that \( db(R') = db'(R') \) for all \( R' \neq R \). From this and the fact that \( R \) does not occur in \( \phi \), it follows that \([\phi]db = T\).

15. Propagate Forward DELETE Success - 1. The proof for this case is similar to that of Propagate Forward INSERT Success - 1.

16. Propagate Backward SUCCESS. The proof for this case is similar to that of Propagate Forward INSERT/DELETE Success.

17. Propagate Backward DELETE Success - 1. The proof for this case is similar to that of Propagate Forward INSERT/DELETE Success.

18. Propagate Forward DELETE Success - 1. The proof for this case is similar to that of Propagate Forward INSERT/DELETE Success - 1.

19. Reasoning. Let \( \Phi \) be a subset of \( \{ \phi \mid r, i \vdash \phi \} \) and \([\text{last}(r^i) = \langle db, U, sec, T, V, c \rangle \). From the induction hypothesis, it follows that \([\phi]d^v = T\) for any \( \phi \in \Phi \). From the rule's definition, it follows that \( \Phi \models \gamma \). From this and \([\phi]d^v = T\) for any \( \phi \in \Phi \), it follows that \([\gamma]d^v = T\).

20. Learn INSERT Backward - 3. Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, \text{INSERT}, R, T) \cdot s \), where \( s = \langle db', U', sec', T', V', c' \rangle \in \Omega_M \). From the rule's definition, \( r, i - 1 \vdash \phi \) and \( r, i - 1 \vdash \gamma \). From this and the induction hypothesis, it follows that \([\phi]d^v = T\) and \([\gamma]d^v = T\). Therefore, \([\phi]db = T\) and \([\gamma]db = T\). Hence, \( db \neq db' \) leading to a contradiction with \( db = db' \).

21. Learn DELETE Backward - 3. The proof is similar to that of Learn INSERT Backward - 3.

22. Propagate Forward Disabled Trigger. Let \( i \) be such that \( r^i = r^{i-1} \cdot t \cdot s \), where \( s = \langle db, U, sec, T, V, c \rangle \in \Omega_M \). From the rule's definition, it follows that \( r, i - 1 \vdash \phi \) holds. From this and the induction hypothesis, it follows that \([\phi]d^v = T\). From this, the fact that \( \psi \) is \( t \)'s WHEN condition, and the rule Trigger Disabled, it follows that \( db = db' \). From the rule's definition, it follows that \( r, i - 1 \vdash \phi \) holds. From this and the induction hypothesis, it follows that \([\phi]d^v = T\). From this, the rule's definition, it follows that \([\phi]db = T\).

23. Propagate Backward Disabled Trigger. The proof for this case is similar to that of Propagate Forward Disabled Trigger.

24. Learn INSERT Forward. Let \( i \) be such that \( r^i = r^{i-1} \cdot t \cdot s \), where \( s = \langle db, U, sec, T, V, c \rangle \in \Omega_M \). From the rule's definition, it follows that \( r, i - 1 \vdash \phi \) holds. From this and the induction hypothesis, it follows that \([\phi]d^v = T\). Furthermore, from the rule's definition, it follows that \([\phi]db = T\). From this, the fact that \( \psi \) is \( t \)'s WHEN condition, \([\psi]d^v = T\), and the rule Trigger DELETE - INSERT Success, it follows that \( db = db'[R \oplus T] \). From the definition of \( \oplus \), it follows that \([\psi]d^v = T\).
last(r⁻¹) = (db', U', sec', T', V', c'), and t ∈ TRIGGER D, and φ be ¬∃y, z. R(ʻy, z, ʻv) ∧ ʻv ≠ ʻv.
Furthermore, let ψ be t's condition where all free variables are replaced with the values in tlp(last(r⁻¹)) and (ʻu', INSERT, R, (ʻu, ʻv, ʻv)) be t's actual action. We claim that db(R) = db'(R) ∪ (ʻu, ʻv, ʻv)). Furthermore, we claim that [φ]db holds. From these claims, it follows that there is no tuple (ʻv, ʻw, ʻq) in db(R) such that ʻv = ʻv and ʻw ≠ ʻw. From this and db(R) = db'(R) ∪ (ʻu, ʻv, ʻv)), it follows that there is no tuple (ʻv, ʻw, ʻq) in db'(R) such that ʻv = ʻv and ʻw ≠ ʻw. From this, it follows that also [φ]db' holds.
We now prove our claim that db(R) = db'(R) ∪ (ʻu, ʻv, ʻv)). Assume, for contradiction's sake, that this is not the case. Since db is obtained from db', this would imply that the trigger t is disabled. Hence, this would imply that [φ]db' = ⊥. From the rule's definition, r, i - 1 ⊢ ψ.
From this, the induction's hypothesis, and last(r⁻¹) = (db', U, sec, T, V, c'), it follows that [ψ]db = T, which contradicts [ψ]db' = ⊥. We now prove our claim that [φ]db holds. Assume, for contradiction's sake, that this is not the case. This means that there is a tuple (ʻv, ʻw, ʻq) in db(R) such that ʻv = ʻv and ʻw ≠ ʻw. Note that, as we proved before, (ʻv, ʻw, ʻq) ∈ db(R). Therefore, there are two tuples (ʻv, ʻw, ʻq) and (ʻv, ʻw, ʻq) in db(R) such that ʻv ≠ ʻw. From this, it follows that [ψ]db = ⊥. This is in contradiction with the fact that the constraint ∀y, z, ʻv, ʻq. (R(ʻy, ʻv, ʻq) ∧ R(ʻy, ʻq, ʻv)) → ʻy = ʻy) holds in Γ. Indeed, since the constraint is in Γ, any state in Ω must satisfy it.
26. Learn INSERT - FD - 1. Let i be such that r⁻¹ = r⁻¹-t-s, where s = (db, U, sec, T, V, c) ∈ ΩM, last(r⁻¹) = (db, U, sec, T, V, c'), and t ∈ TRIGGER D, and φ be ¬∃y, z. R(ʻy, z, ʻv) ∧ ʻv ≠ ʻv. Furthermore, let ψ be t's condition where all free variables are replaced with the values in tlp(last(r⁻¹)) and (ʻu, INSERT, R, (ʻu, ʻv, ʻv)) be t's actual action. From the rule's definition, r, i - 1 ⊢ ψ. From this, the induction's hypothesis, and last(r⁻¹) = (db', U, sec, T, V, c'), it follows that [ψ]db = T. From this and the LTS semantics, it follows that the trigger t is enabled in last(r⁻¹).
We now prove our claim that [φ]db holds. Assume, for contradiction's sake, that this is not the case. This means that there is a tuple (ʻv, ʻw, ʻq) in db'(R) such that ʻv = ʻv and ʻw ≠ ʻw. Let db' be the state R[db ⊔ (ʻv, ʻw, ʻq)]. From db' = db'[R[db ⊔ (ʻv, ʻw, ʻq)]], and the fact that there is a tuple (ʻv, ʻw, ʻq) in db'(R) such that ʻv = ʻv and ʻw ≠ ʻw, it follows that there are two tuples (ʻv, ʻw, ʻq) and (ʻv, ʻw, ʻq) in db'(R) such that ʻv ≠ ʻw. From this, it follows that [ψ]db = ⊥. Which contradicts the fact that this is not the case. Since db is obtained from db', this would imply that the trigger t is disabled. Hence, this would imply that [ψ]db' = ⊥.
27. Learn INSERT - ID. The proof of this case is similar to that of Learn INSERT - FD. See also the proof of INSERT Success - ID.
28. Learn INSERT - ID - 1. The proof of this case is similar to that of Learn INSERT - FD - 1. See also the proof of INSERT Success - ID.
29. Learn INSERT Backward - 1. Let i be such that r⁻¹ = r⁻¹-t-s, where s = (db', U', sec', T', V', c') ∈ ΩM, last(r⁻¹) = (db, U, sec, T, V, c), t ∈ TRIGGER D, and φ be ¬R(ʻy, Z) ≈ (db', U, sec', T', V', c'), last(r⁻¹) = (db, U, sec, T, V, c), and last(r⁻¹) = (db, U, sec, T, V, c), it follows that [ψ]db = T and [ψ]db' = T. Therefore, [φ]db' = T. Hence, db ≠ db'. We now prove that [φ]db = T. Assume, for contradiction's sake, that [φ]db = ⊥. From this, the rule's definition, it follows that [ψ]db = T. Therefore, [ψ]db' = ⊥. Hence, db ≠ db'. We now prove that [φ]db = ⊥. Assume, for contradiction's sake, that [φ]db = ⊥. From this, the rule's definition, it follows that [ψ]db = T. Therefore, [ψ]db' = ⊥. Hence, db ≠ db'. We now prove that [φ]db = ⊥. Assume, for contradiction's sake, that [φ]db = ⊥. From this, the rule's definition, it follows that [ψ]db = T. Therefore, [ψ]db' = ⊥. Hence, db ≠ db'. We now prove that [φ]db = ⊥. Assume, for contradiction's sake, that [φ]db = ⊥. From this, the rule's definition, it follows that [ψ]db = T. Therefore, [ψ]db' = ⊥. Hence, db ≠ db'. We now prove that [φ]db = ⊥. Assume, for contradiction's sake, that [φ]db = ⊥. From this, the rule's definition, it follows that [ψ]db = T. Therefore, [ψ]db' = ⊥. Hence, db ≠ db'. We now prove that [φ]db = ⊥. Assume, for contradiction's sake, that [φ]db = ⊥. From this, the rule's definition, it follows that [ψ]db = T. Therefore, [ψ]db' = ⊥. Hence, db ≠ db'.
C.2. Indistinguishability is an equivalence relation

We now show that our indistinguishability definition is, indeed, an equivalence relation over traces.

**Proposition C.2.** Let \( P = (M, f) \) be an extended configuration, \( L \) be the P-LTS, and \( u \in U \) be a user. The indistinguishability relation \( \equiv_{P,u} \) is an equivalence relation over \( \text{traces}(L) \).

**Proof.** We now prove that \( \equiv_{P,u} \) is reflexive, symmetric, and transitive. This implies the fact that \( \equiv_{P,u} \) is an equivalence relation over \( \text{traces}(L) \). In the following, let \( P = (M, f) \) be an extended configuration, \( L \) be the P-LTS, and \( u \in U \) be a user. From the definition of data indistinguishability and Proposition 3.1, it follows that the data-indistinguishability relation \( \equiv_{P,u}^{\text{data}} \) is an equivalence relation over the set of all system states.

- **Reflexivity** Let \( r \in \text{traces}(L) \) be a run. It follows trivially that \( r|_u = r|_u \). From this, it follows that \( r|_u \) and \( r'|_u \) are consistent. It is easy to see that \( r \) is indistinguishable from \( r \). Indeed, the database states are the same in \( r \) and \( r \) and the data-indistinguishability relation is reflexive.

- **Symmetry** Let \( r, r' \in \text{traces}(L) \) be two runs such that \( r \equiv_{P,u} r' \). From this, it follows that \( r|_u \) and \( r'|_u \) are consistent. Note that the consistency definition is symmetric. Therefore, also \( r'|_u \) and \( r|_u \) are consistent. This and the symmetry of data indistinguishability, it follows that \( \equiv_{P,u} \) is symmetric.

- **Transitivity** Let \( r, r', r'' \in \text{traces}(L) \) be three runs such that \( r \equiv_{P,u} r' \) and \( r' \equiv_{P,u} r'' \). From this it follows that \( r|_u \) and \( r'|_u \) are consistent and \( r'|_u \) and \( r''|_u \) are consistent. It is easy to see that also \( r|_u \) and \( r''|_u \) are consistent as well. This and the transitivity of data indistinguishability, it follows that \( \equiv_{P,u} \) is transitive.

\( \square \)
C.3 Database Integrity Proofs

Here, we prove that the enforcement mechanism $f$ from Section 6.8 provides database integrity.

C.3.1 Extend function

Proposition C.3 states that the extend function produces only views that are equivalent to those given as input.

**Proposition C.3.** Let $M = (D, \Gamma)$ be a system configuration, $s = (db, U, sec, T, V)$ be an $M$-system state, and $V' \subseteq V$ be a set of views with owner’s privileges. For each view $v \in \text{extend}(M, s, V')$, there is a view $v' \in V'$ such that $v$ and $v'$ disclose the same data.

**Proof.** Assume, for contradiction’s sake, that there is a view $v \in \text{extend}(M, s, V')$ such that all the views in $V'$ disclose different data from $v$. This is impossible because $v$ has been obtained by a view $v' \in V'$ just by replacing the views with their definitions (and each view is semantically equivalent to its definition).

C.3.2 A sound under-approximation of query determinacy

Before proving that $	ext{apprDet}$, which is defined in Section 6.8.2, is a sound approximation of $	ext{determines}$, we extend $	ext{determines}$ from sentences to formulae.

Given a system’s configuration $M = (D, \Gamma)$, a formula $\phi$, a set of views $V$ with owner’s privileges, a set of tables $T$, and a well-formed assignment $\nu$ for $\phi$, we say that $V$ and $T$ determine $(\phi, \nu)$, denoted by $\text{determines}_M(T, V, \phi, \nu)$, iff $D, \Gamma \vdash Q \rightarrow \phi$ holds, where $Q$ is the set of queries containing all tables in $T$ and all views in $V$. In the following, given a view $(u, o, q, m)$, we denote by $\text{def}(u, o, q, m)$ its definition $q$. Furthermore, given a set of tables $T$, a set of views $V$, and a database state $db$, we denote by $[db]_{V, T}$ the set of all states $\{db' \in D^2 | \bigwedge_{t \in T} db(t) = db'(t) \land \bigwedge_{v \in V} \text{def}(v)^{db} = \text{def}(v)^{db'}\}$.

In Proposition C.4, we show that $\text{apprDet}$ is, indeed, a sound under-approximation of query determinacy.

**Proposition C.4.** Let $M = (D, \Gamma)$ be a system configuration, $s = (db, U, sec, T, V)$ be an $M$-system state, $T' \subseteq D$ be a set of tables, $V' \subseteq V$ be a set of views with owner's privileges, and $\phi$ be a formula. If $\text{apprDet}(T', V', \phi, s, M) = \top$, then for all well-formed assignments $\nu$ for $\phi$, $\text{determines}_M(T', V', \phi, \nu)$ holds.

**Proof.** Let $M = (D, \Gamma)$ be a system configuration, $s = (db, U, sec, T, V)$ be an $M$-system state, $T' \subseteq D$ be a set of tables, $V' \subseteq V$ be a set of views with owner’s privileges, and $\phi$ be a formula. We prove the lemma by structural induction over the formula $\phi$.

**Base Case:** There are a number of alternatives.

$\phi := R([\tau])$ Assume that $\text{apprDet}(T, V, R([\tau]), s, M) = \top$. There are two cases:

1. $R \in T'$. In this case, the set $T'$ trivially determines the formula $R$ for any well-formed assignment $\nu$. Therefore, $\text{determines}_M(T', V', R([\tau]), \nu)$ holds. Indeed, assume that this is not the case. Thus, there are three database states $db_1, db_2$, and $db_3$ such that $db_1, db_2 \in [db]_{V, T'}$, and $[db]_{V, T'}$. From this and the relational calculus semantics, it follows that $db_1(R) \neq db_2(R)$. From this, $R \in T'$, and $db_1, db_2 \in [db]_{V', T'}$, it follows that $db_1(R) = db_2(R)$ leading to a contradiction.

2. there is a view $v' \in \text{extend}(M, s, V')$ such that $\text{def}(v') = \{\tau | R([\tau])\}$. This means that there is a set of views $V_1, \ldots, V_n$ in $s$ such that $\text{def}(V_1) = \{\tau | R([\tau])\}$, $\text{def}(V_2) = \{\tau | V_1([\tau])\}$, ..., $\text{def}(V_n) = \{\tau | V_{n-1}([\tau])\}$, and $V_n \in V'$. Therefore, the set $V'$ trivially determines the formula $R([\tau])$ for any well-formed assignment $\nu$, and $V_n$ and $R$ are equivalent. Therefore, $\text{determines}_M(T', V', R([\tau]), \nu)$ holds.

$\phi := V([\tau])$ Assume that $\text{apprDet}(T, V, V([\tau]), s, M) = \top$. There are two cases:

1. There is a view $(V, o, q, \Omega) \in V'$. In this case, the set $V'$ trivially determines the formula $V([\tau])$ for any assignment $\nu$ that is well-formed for $\phi$. Therefore, $\text{determines}_M(T', V', V([\tau]), \nu)$ holds.

2. there is a view $v' \in \text{extend}(M, s, V')$ such that $\text{def}(v') = \{\tau | V([\tau])\}$. This means that there is a set of tables $V_1, \ldots, V_n$ in $s$ such that $\text{def}(V_1) = \{\tau | V([\tau])\}$, $\text{def}(V_2) = \{\tau | V_1([\tau])\}$, ..., $\text{def}(V_n) = \{\tau | V_{n-1}([\tau])\}$, and $V_n \in V'$. Therefore, the set $V'$ trivially determines the formula $V([\tau])$ for any well-formed assignment $\nu$, and $V_n$ and $V$ are equivalent. Therefore, $\text{determines}_M(T', V', V([\tau]), \nu)$ holds.

$\phi := x = v$ For any well-formed assignment $\nu$, the empty set trivially determines the formula $x = v$.

$\phi := \top$ The proof of this case is similar to that of $\phi := x = v$.

$\phi := \bot$ The proof of this case is similar to that of $\phi := x = v$. 


This concludes the proof of the base case.

**Induction Step:** Assume that the claim holds for all sub-formulae of \( \phi \). We now show that the claim holds also for \( \phi \). There are a number of cases:

1. \( \text{apprDet}(T', V', \psi, s, M) = T \) and \( \text{apprDet}(T', V, \psi, \gamma, s, M) = T \). From the induction hypothesis, it follows that both \( \text{determines}_M(T', V', \psi, \nu) \) and \( \text{determines}_M(T', V', \gamma, \nu) \) hold for all well-formed assignments \( \nu \). Therefore, also \( \text{determines}_M(T', V', \psi \land \gamma, \nu) \) holds for all well-formed assignments \( \nu \). Indeed, assume that this is not the case. Then, there are three database states \( db, db_1, \) and \( db_2 \) such that \( db_1, db_2 \in [\db_{T', V, T}] \) and \( [(\psi \land \gamma)\nu]^\theta_1 \neq [(\psi \land \gamma)\nu]^\theta_2 \). From this and the relational calculus semantics, there are two cases:
   
   (a) \( [\psi\nu]^\theta_1 \neq [\psi\nu]^\theta_2 \). From this, it follows that \( \text{determines}_M(T', V', \psi, \nu) \) does not hold.
   
   This contradicts the fact that \( \text{determines}_M(T', V', \psi, \nu) \) holds.
   
   (b) \( [\gamma\nu]^\theta_1 \neq [\gamma\nu]^\theta_2 \). The proof of this case is similar to the previous one.

2. there is a view \( v' \in \text{extend}(M, s, V') \) such that \( \text{def}(v') = \{ \tau \mid \psi \land \gamma \} \). From Proposition C.3, it follows that there is a view \( v'' \in V' \) that is equivalent to \( v' \), and, therefore, to \( \{ \tau \mid \psi \land \gamma \} \). Thus, \( \text{determines}_M(T', V', \psi \land \gamma, \nu) \) holds for all assignments \( \nu \) that are well-formed for \( \phi \).

\[ \phi := \psi \land \gamma \] This case is similar to \( \psi \land \gamma \).

\[ \phi := \neg \psi \] Assume that \( \text{apprDet}(T', V', \neg \psi, s, M) = T \). There are two cases:

1. \( \text{apprDet}(T', V', \psi, s, M) = T \). From the induction hypothesis, \( \text{determines}_M(T', V', \psi, \nu) \) holds. Therefore, also \( \text{determines}_M(T', V', \neg \psi, \nu) \) holds. Indeed, assume that this is not the case. This means that there are three database states \( db, db_1, \) and \( db_2 \) such that \( db_1, db_2 \in [\db_{V', T}] \) and \( [\neg \psi\nu]^\theta_1 \neq [\neg \psi\nu]^\theta_2 \). From this and the relational calculus semantics, it follows that \( [\psi\nu]^\theta_2 \neq [\psi\nu]^\theta_1 \). From this, it follows that \( \text{determines}_M(T', V', \psi, \nu) \) does not hold. This contradicts the fact that \( \text{determines}_M(T', V', \psi, \nu) \) holds.

2. there is a view \( v' \in \text{extend}(M, s, V') \) such that \( \text{def}(v') = \{ \tau \mid \exists x \psi \} \). From Proposition C.3, it follows that there is a view \( v'' \in V' \) that is equivalent to \( v' \), and, therefore, to \( \{ \tau \mid \exists x \psi \} \).

\[ \phi := \exists x \psi \] Assume that \( \text{apprDet}(T', V', \exists x \psi, s, M) = T \). There are two cases:

1. \( \text{apprDet}(T', V', \psi, s, M) = T \). From the induction hypothesis, \( \text{determines}_M(T', V', \psi, \nu) \) holds. Therefore, also \( \text{determines}_M(T', V', \exists x \psi, \nu) \) holds for all well-formed assignments \( \nu \). Note that any well-formed assignment for \( \psi \) is also a well-formed assignment for \( \exists x \psi \). Indeed, assume that this is not the case. This means that there are three database states \( db, db_1, \) and \( db_2 \) such that \( db_1, db_2 \in [\db_{V', T}] \) and \( [(\exists x \psi)\nu]^\theta_1 \neq [(\exists x \psi)\nu]^\theta_2 \). From this and the relational calculus semantics, it follows that there is a value \( v \in \text{dom} \) such that \( [\psi v[x \mapsto v]]^\theta_1 \neq [\psi v[x \mapsto v]]^\theta_2 \). Note that \( v[x \mapsto v] \) is a well-formed assignment for \( \psi \). Let’s call this assignment \( v' \). From this, it follows that \( [\psi v]^\theta_1 \neq [\psi v]^\theta_2 \). From this, it follows that \( \text{determines}_M(T', V', \psi, \nu) \) does not hold. This contradicts the fact that \( \text{determines}_M(T', V', \psi, \nu) \) holds for any well-formed assignment \( \nu \).

2. there is a view \( v' \in \text{extend}(M, s, V') \) such that \( \text{def}(v') = \{ \tau \mid \forall x \psi \} \). From Proposition C.3, it follows that there is a view \( v'' \in V' \) that is equivalent to \( v' \), and, therefore, to \( \{ \tau \mid \forall x \psi \} \). Thus, \( \text{determines}_M(T', V', \forall x \psi, \nu) \) holds for all well-formed assignments \( \nu \).

\[ \phi := \forall x \psi \] This case is similar to \( \exists x \psi \).

This concludes the proof of the induction step.

This completes the proof.

\[ \square \]

### C.3.3 \( \approx_{\text{appr}} \) is a sound approximation of \( \approx_{\text{auth}} \)

We now show that \( \approx_{\text{appr}} \) is a sound approximation of \( \approx_{\text{auth}} \). Namely, whenever \( s \approx_{\text{auth}} \) holds, then \( s \approx_{\text{appr}} \) act holds as well. A derivation of \( s \approx_{\text{appr}} \) act is a proof tree, obtained using the rules defining \( \approx_{\text{appr}} \), which ends in \( s \approx_{\text{appr}} \) act. The size of a derivation is the number of \( \approx_{\text{appr}} \) rules that are used to show that \( s \approx_{\text{appr}} \) act. In the following, we switch freely between statements of the form \( s \approx_{\text{appr}} \) act and their derivations. We denote the size of the derivation of \( s \approx_{\text{appr}} \) act as \( |s \approx_{\text{appr}} \) act|.

**Proposition C.5.** Let \( M = (D, \Gamma) \) be a system configuration, \( s \) be an \( M \)-state, \( c \) be an \( M \)-context, and \( act \in A_{\text{D}} \cup \text{TRIGG} \cup \text{RIGG} \) be an action or a trigger. If \( s \approx_{\text{auth}} \) act, then \( s \approx_{\text{auth}} \) act.

**Proof.** Let \( M = (D, \Gamma) \) be a system configuration, \( s \) be an \( M \)-state, \( c \) be an \( M \)-context, and \( act \in A_{\text{D}} \cup \text{TRIGG} \cup \text{RIGG} \) be a further derivation. Furthermore, we assume that there is a derivation of \( s \approx_{\text{auth}} \) act. We prove our claim by induction on the size of \( s \approx_{\text{auth}} \) act’s derivation.

**Base Case:** We now show that, for all \( s \) and \( act \) such that \( |s \approx_{\text{auth}} \) act| = 1, if \( s \approx_{\text{auth}} \) act, then \( s \approx_{\text{auth}} \) act. Observe that the base case is trivial. Indeed, for the rules `INSERT DELETE admin`,
CREATE VIEW admin, CREATE TRIGGER admin, SELECT, EXECUTE TRIGGER;

Induction Step: We now assume that, for all derivations of size less than \(|s \rightsquigarrow_{auth} act|\), it holds that if \(s' \rightsquigarrow_{auth} act'\), then \(s' \rightsquigarrow_{auth} act\). There are several cases:

1. Rule INSERT DELETE: Assume that \(s \rightsquigarrow_{auth} act\) holds and that \(act = \langle u, op', R, \tilde{g}\rangle\), where \(op'\) is one of \{INSERT, DELETE\}. From the rule’s definition, it follows that there is a grant \(g = \langle op, u, \langle op', R, u'\rangle \rangle\) in \(s, sec\) such that \(s \rightsquigarrow_{auth} g\). From this and the induction hypothesis, it follows that \(s \rightsquigarrow_{auth} g\). Therefore, \(s \rightsquigarrow_{auth} act\) holds (we can apply the INSERT DELETE rule in \(\rightsquigarrow_{auth}\)).

2. Rule CREATE VIEW: The proof is similar to the one for the INSERT DELETE rule.

3. Rule CREATE TRIGGER: The proof is similar to the one for the INSERT DELETE rule.

4. Rule EXECUTE TRIGGER-2: Assume that \(s \rightsquigarrow_{auth} act\) holds and that \(act = \langle i, v, e, R, t, s, A\rangle\) such that \([a(T,R) \rightarrow tpsl(s)]\). From the rule’s definition, it follows that both \(s \rightsquigarrow_{auth} action(st, ow, tpsl(s))\) and \(s \rightsquigarrow_{auth} action(st, invoker(s), tpsl(s))\). From this and the induction hypothesis, both \(s \rightsquigarrow_{auth} action(st, ow, tpsl(s))\) and \(s \rightsquigarrow_{auth} action(st, invoker(s), tpsl(s))\). From this and the EXECUTE TRIGGER-2 rule in \(\rightsquigarrow_{auth}\), it follows that also \(s \rightsquigarrow_{auth} act\) holds.

5. Rule EXECUTE TRIGGER-1: The proof is similar to the one for the EXECUTE TRIGGER-2 rule.

6. Rule GRANT-1: Assume that \(s \rightsquigarrow_{auth} act\) holds and that \(act = \langle op, u, p, u'\rangle\), where \(op \in \{\oplus, \ominus\}\). From the rule’s definition, it follows that there is a grant \(g = \langle \oplus, u', u'\rangle\) in \(s, sec\) such that \(s \rightsquigarrow_{auth} g\). From this and the induction hypothesis, it follows that \(s \rightsquigarrow_{auth} g\). From this and the GRANT-1 rule in \(\rightsquigarrow_{auth}\), it follows that \(s \rightsquigarrow_{auth} act\) holds.

7. Rule GRANT-2: Assume that \(s \rightsquigarrow_{auth} act\) holds and that \(act = \langle op, v, a, p, u, v\rangle\), where \(p = \langle SELECT, v\rangle\), \(v \in V\) \(\forall v\) \(\forall v\), \(op \in \{\oplus, \ominus\}\), and \(o = owner(v)\) such that \(o \neq admin\). Let \(T'\) be the set obtained through the \(\alpha\) function and \(V'\) be the set obtained through the \(\alpha\) function. From the rule’s definition, it follows that \(appr\) \(Def(T', V', \alpha(v)) = \top\). From this and Proposition C.4, it follows that \(determines\) \(T', V', \alpha(v)\). We now show that for any \(obj \in T' \cup V'\), \(hasAccess(s', \{obj\}, o, \oplus)\) holds. There are four cases:

   (a) \(o = admin\) and \(obj \in D\). Since \(obj \in T'\), it follows that there is a \(g = \langle \oplus, o, \{SELECT, obj\}, u'\rangle\) such that \(s \rightsquigarrow_{auth} g\). From this and the induction hypothesis, it follows that \(s \rightsquigarrow_{auth} g\). Therefore, \(hasAccess(s', \{obj\}, o, \oplus)\) holds.

   (b) \(o \neq admin\) and \(obj \in D\). Since \(obj \in T'\), it follows that there is a \(g = \langle \oplus, o, \{SELECT, obj\}, u'\rangle\) in \(sec\) such that \(s \rightsquigarrow_{auth} g\). From this and the induction hypothesis, it follows that \(s \rightsquigarrow_{auth} g\). Therefore, \(hasAccess(s', \{obj\}, o, \oplus)\) holds.

   (c) \(o = admin\) and \(obj \in V\). The proof of this case is similar to that of \(o = admin\) and \(obj \in D\).

   (d) \(o \neq admin\) and \(obj \in V\). The proof of this case is similar to that of \(o \neq admin\) and \(obj \in D\).

Note that from \(hasAccess(s', A, o, opp)\) and \(hasAccess(s', B, o, opp)\), it follows that \(hasAccess(s', A \cup B, o, opp)\). Thus, \(hasAccess(s, s' T' \cup V', o, \oplus)\) holds. From this, it follows that \(s \rightsquigarrow_{auth} act\) holds because we can apply the corresponding rule in \(\rightsquigarrow_{auth}\).

8. Rule GRANT-4: The proof is similar to the one for the GRANT-3 rule.

9. Rule REVOKE: Assume that \(s \rightsquigarrow_{auth} act\) holds and that \(act = \langle \ominus, u, p, u'\rangle\). From the rule’s definition, it follows that \(s' \rightsquigarrow_{auth} g\) for any \(g \in s', sec\), where \(s' = applyRev(s, \langle \ominus, u, p, u'\rangle)\). From the induction’s hypothesis, it follows that \(s' \rightsquigarrow_{auth} g\) for any \(g \in s', sec\). Therefore, we can apply the rule REVOKE of \(\rightsquigarrow_{auth}\) to derive \(s \rightsquigarrow_{auth} act\).

This completes our proof.

C.3.4 \(f\) provides Database Integrity

We are now ready to prove that \(f\) provides database integrity.

Lemma C.1. For any two states \(s = \langle db, U, sec, T, V, c\rangle\), \(s' = \langle db', U, sec, T, V, c'\rangle\) in \(\Omega_M\) and any action \(a \in \mathcal{A}_{D,H}\):

1. \(s \rightsquigarrow_{auth} a\) iff \(s' \rightsquigarrow_{auth} a\), and
2. \(s \rightsquigarrow_{auth} a\) iff \(s' \rightsquigarrow_{auth} a\).

Proof. It is easy to see that the only rules that depends on \(db, db', c, c'\) are EXECUTE TRIGGER - 1, EXECUTE TRIGGER - 2, and EXECUTE TRIGGER - 3. Since they are not used to evaluate whether \(s \rightsquigarrow_{auth} a\) and \((s \rightsquigarrow_{auth} a)\) hold for actions in \(\mathcal{A}_{D,H}\), the lemma follows trivially.

Lemma C.2. Let \(M\) be a system configuration. Then, for all \(M\)-states \(s = \langle db, U, sec, T, V, c\rangle \in \Omega_M\) such that \(trigger(s) = \epsilon\) and all actions \(act \in \mathcal{A}_{D,H}\), if \(fint(s, act) = \top\), then \(s \rightsquigarrow_{auth} act\).
Proof. We prove the theorem by contradiction. Assume, for contradiction’s sake, that the claim does not hold. Therefore, there is a state $s$ and an action $act$ such that $f_{out}(s, act) = \top$, $trigger(s) = \epsilon$, and $s \not\sim_{auth} act$. Thus, from $f_{out}(s, act) = \top$, $trigger(s) = \epsilon$, and $f_{out}$’s definition, it follows that $s \sim_{auth} act$.

From this and Proposition C.5, it follows that $s \sim_{auth} act$ leading to a contradiction.

Lemma C.3. Let $M$ be a system configuration. For all $M$-states $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and all triggers $t \in T$ such that $trigger(s) = t$, the following hold:

1. If $f_{out}(s, c) = \top$ and $\sigma^{db}_{s, c} = \perp$, then $s \not\sim_{auth} t$, where $c = \text{trigCond}(s) = \langle u, \text{SELECT}, \psi \rangle$.
2. If $f_{out}(s, a) = \top$, then $s \sim_{auth} act$, where $c = \text{trigCond}(s) = \langle u, \text{SELECT}, \psi \rangle$ and $a = \text{trigAct}(s)$.

Proof. We prove both claims by contradiction.

Assume, for contradiction’s sake, that the first claim does not hold. Therefore, there is a state $s$ and a trigger $t$ such that $f_{out}(s, c) = \top$ and $\sigma^{db}_{s, c} = \perp$ and $s \not\sim_{auth} t$. From $\sigma^{db}_{s, c} = \perp$, trigger$(s) = t$, and the rule $\text{EXECUTE TRIGGER - 3}$, it follows that $s \sim_{auth} t$ holds, which leads to a contradiction.

Assume, for contradiction’s sake, that the second claim does not hold. Therefore, there is a state $s$ and a trigger $t = \langle id, ow, c, R, \phi, st, m \rangle$ such that $f_{out}(s, a) = \top$ and $trigger(s) = t$, and $s \not\sim_{auth} t$. If $t$ is a trigger with owner’s privileges, from $f_{out}(s, a) = \top$ and $trigger(s) = t$, it follows that $s \sim_{auth} t$. From this and Proposition C.5, it follows that $s \sim_{auth} t$. If $t$ is a trigger with activator’s privileges, from $f_{out}(s, a) = \top$ and $trigger(s) = t$, it follows that $s \sim_{auth} t$. From this and Proposition C.5, it follows that $s \sim_{auth} t$.

We are now ready to prove our main result, namely that $f$ provides database integrity.

Theorem C.1. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $f$ be the PDP defined in Section 6.8, and $P = \langle M, f \rangle$ be an extended configuration. The PDP $f$ provides database integrity with respect to $P$.

Proof. We prove the lemma by contradiction. Assume, for contradiction’s sake, that $f$ does not satisfy the database integrity property. There are three cases:

1. There is a reachable state $s$ and an action $act \in A_{D, s}$ such that $trigger(s) = \epsilon$, $f(s, act) = \top$, and $s \not\sim_{auth} act$. From $f(s, act) = \top$, it follows that $f_{out}(s, act) = \top$. From this, $trigger(s) = \epsilon$, and Lemma C.2, it follows $s \not\sim_{auth} act$, which leads to a contradiction.

2. There is a reachable state $s$ and a trigger $t \in TRIGGER_D$ such that $trigger(s) = t$, $f(s, c) = \top$, $\sigma^{db}_{s, c} = \perp$, and $s \not\sim_{auth} t$, where $c = \langle u, \text{SELECT}, \psi \rangle$ is $t$’s condition. From $f(s, c) = \top$, it follows that $f_{out}(s, c) = \top$. From $f_{out}(s, c) = \top$, $\sigma^{db}_{s, c} = \perp$, trigger$(s) = t$, and Lemma C.3, it follows $s \not\sim_{auth} t$, which leads to a contradiction.

3. There is a reachable state $s$ and a trigger $t \in TRIGGER_D$ such that $trigger(s) = t$, $f(s, c) = \top$, $\sigma^{db}_{s, c} = \perp$, $f(s', a) = \top$, and $s \not\sim_{auth} t$, where $c = \langle u, \text{SELECT}, \psi \rangle$ is $t$’s condition, $a$ is $t$’s action, and $s'$ is the state obtained from $s$ by updating the context’s history. From $f(s', a) = \top$, it follows that $f_{out}(s', a) = \top$. Since $s$ and $s'$ are equivalent modulo the context’s history and $f_{out}$ does not depend on the context’s history, it follows that $f_{out}(s, a) = \top$. From $f_{out}(s, c) = \top$, $\sigma^{db}_{s, c} = \perp$, $f_{out}(s, a) = \top$, trigger$(s) = t$, and Lemma C.3, it follows $s \not\sim_{auth} t$, which leads to a contradiction.

This completes the proof.

We also prove that, by using $f$, any reachable state has a consistent policy. Observe that this is the underlying reason why $f$ prevents Attacks 6.2 and 6.3.

Lemma C.4. Let $P = \langle M, f \rangle$ be an extended configuration, where $M = \langle D, \Gamma \rangle$ is a system configuration and $f$ is the PDP defined in Section 6.8, and $L$ be the P-LTS. For each reachable state $s = \langle db, U, sec, T, V, c \rangle$, $s \sim_{auth} g$ for all $g \in sec$.

Proof. We claim that, for any run $r$, the state last$(r)$ is such that for all $p \in \text{last}(r), sec$, last$(r) \sim_{auth} p$. From this, the lemma follows trivially.

We now prove that for any run $r$, the state last$(r)$ is such that for all $p \in \text{last}(r), sec$, last$(r) \sim_{auth} p$. We do this by induction on the length of the run $r$.

Base Case: The base case consists of the runs containing only one initial state. Note that an initial state contains only grants issued by admin, together with views and triggers owned by admin. It is easy to see that for any permission $p = \langle op, u, pr, admin \rangle$ in a policy $sec$ in an initial state $s$, it holds that $s \sim_{auth} p$. There are two cases:
We now assume that for all runs \( (\text{respectively } \text{last}(r')) \)

We assume that the derivation has length 1. There are four cases depending on the rule

- \( \text{act} \) is \( \{ \text{op}, u, p, u' \} \). From \( \text{secEx(last(r'))} = \bot \), it follows that \( \text{last}(r') \sim \text{auth} g \) for all \( g \in \text{last}(r').sec \).
- \( \text{act} \) is \( \{ \text{op}, u, p, u' \} \). From \( \text{secEx(last(r'))} = \bot \), it follows that \( \text{last}(r') \sim \text{auth} g \) for all \( g \in \text{last}(r').sec \).
- \( \text{act} \) is a trigger and the \( \text{WHEN} \) condition is satisfied. In this case, \( \text{last}(r') \sim \text{auth} g \) for all \( g \in \text{last}(r').sec \).
- \( \text{act} \) is a trigger and the \( \text{WHEN} \) condition is satisfied. In this case, the proof is the same as the previous cases depending on the trigger's action.

Proposition C.6 proves that the rules in Figure 6.27 are a sound under-approximation of query containment.

Definition C.7. Let \( \Gamma = \langle D, \Gamma \rangle \) be a system configuration, and \( \phi(\overline{x}) \) and \( \psi(\overline{y}) \) be two formulas. If \( \phi \subseteq_M \psi \), according to the rules in Figure 6.27, then \( \forall d \in \Omega^\Gamma_D, ([\overline{x}] \mid \phi)^d \subseteq ([\overline{y}] \mid \psi)^d \), where \( \overline{x} \) (respectively \( \overline{y} \)) is the tuple defined by the variables in \( \text{free}(\phi) \) (respectively \( \text{free}(\psi) \)) ordered according to \( \leq_{\text{var}} \).

**Proof.** Observe that \( \phi \subseteq_M \psi \) iff there is a finite derivation that ends in \( \phi \subseteq_M \psi \) created using the rules in Figure 6.27. We prove our claim by induction on the derivation's length.

**Base Case** Assume that the derivation has length 1. There are four cases depending on the rule used to derive \( \phi \subseteq_M \psi \):

1. Rule **Conjunction.** From the rule's definition, it follows that \( \text{free}(\phi) = \text{free}(\phi \land \psi) \). Let \( d \in \Omega^\Gamma_D \) and \( t \in ([\overline{x}] \mid \phi \land \psi)^d \). From \( t \in ([\overline{x}] \mid \phi \land \psi)^d \) and the definition of non-boolean query, it
follows that \((\phi \land \psi)[\tau \to \tau]' = \top\). From this, it follows that \([\phi[\tau \to \tau]' = \top\). From this and the definition of non-boolean query, \(\tau \in \{[\phi]\}^d\). Therefore, \(\{[\tau \land \psi]\}^d \subseteq \{[\phi \land \psi]\}^d\).

2. Rule Disjunction. From the rule's definition, it follows that \(\text{free}(\phi) = \text{free}(\phi \lor \psi) = \tau\). Let \(d \in \Omega_\tau^\beta\) and \(\tau \in \{[\phi \land \psi]\}^d\). From \(\tau \in \{[\phi \land \psi]\}^d\) and the definition of non-boolean query, it follows that \([\phi \lor \psi][\tau \to \tau]' = \top\). From this, it follows that \([\phi \lor \psi][\tau \to \tau]' = \top\). From this and the definition of non-boolean query, \(\tau \in \{[\phi \lor \psi]\}^d\). Therefore, \(\{[\phi \lor \psi]\}^d \subseteq \{[\phi \land \psi]\}^d\).

3. Rule Identity. From the rule's definition, it follows that \(\text{free}(\phi) = \text{free}(\phi \lor \psi) = \tau\). Let \(d \in \Omega_\tau^\beta\) and \(\tau \in \{[\phi \land \psi]\}^d\). From \(\tau \in \{[\phi \land \psi]\}^d\) and the definition of non-boolean query, it follows that \([\phi \lor \psi][\tau \to \tau]' = \top\). From this and \(\text{free}(\phi) = \text{free}(\phi \lor \psi) = \tau\), it follows that \([\phi \lor \psi][\tau \to \tau]' = \top\). From this and the definition of non-boolean query, \(\tau \in \{[\phi \lor \psi]\}^d\). Therefore, \(\{[\phi \lor \psi]\}^d \subseteq \{[\phi \land \psi]\}^d\).

4. Rule Inclusion Dependency. From the rule's definition, it follows that \(\gamma := \forall \pi, \exists \pi. (R(\pi, \tau) \to \exists \pi. S(\pi, \tau))\) is in \(\Gamma\). Let \(d \in \Omega_\tau^\beta\) and \(\tau \in \{[\phi \land \psi]\}^d\). From \(\tau \in \{[\phi \land \psi]\}^d\) and the definition of non-boolean query, it follows that \([\exists \pi. R(\pi, \tau)]^d = \top\). From this and \(\text{free}(\phi) = \text{free}(\phi \lor \psi) = \tau\), it follows that \([\exists \pi. S(\pi, \tau)]^d \subseteq \{[\exists \pi. S(\pi, \tau)]^d\}^d\). This completes the proof for the base case.

**Induction Step** Assume now that the claim holds for all derivations of length less than that of \(\phi \subseteq_M \psi\). We now prove that it holds also for \(\phi \subseteq_M \psi\). There are two cases:

1. Rule Projection. In this case, \(\phi \subseteq_M \psi\) is of the form \(\exists x_i. \alpha \subseteq_M \exists y_j. \beta\) and it is obtained by applying the rule Projection to \(\alpha \subseteq_M \beta\). From the rule, it follows that \(\alpha \subseteq_M \beta\). Let \(1 \leq u \leq n\) and \(\tau\) (respectively \(\tau^\prime\) and \(\gamma\)) be the tuple obtained from \(\tau\) (respectively \(\tau^\prime\) and \(\gamma\)) by dropping the \(i\)-th value (respectively variable). We now prove that \(\{[\exists x_i. \alpha]\}^d \subseteq \{[\exists y_j. \beta]\}^d\). Assume, for contradiction's sake, that this is not the case, namely there is a tuple \(\tau^\prime\) such that \(\tau^\prime \in \{[\exists x_i. \alpha]\}^d\) but \(\tau^\prime \notin \{[\exists y_j. \beta]\}^d\). From \(\tau^\prime \in \{[\exists x_i. \alpha]\}^d\) and the relational calculus semantics, it follows that there is a tuple \(\tau_1\), obtained by adding a value to \(\tau^\prime\) in the \(i\)-th position, such that \(\tau_1 \in \{[\exists x_i. \alpha]\}^d\). From this, \(\alpha \subseteq_M \beta\) and the induction hypothesis, it follows that \(\tau_1 \in \{[\exists y_j. \beta]\}^d\). From this and the relational calculus semantics, it follows that \(\tau_1 \in \{[\exists y_j. \beta]\}^d\). This contradicts the fact that \(\tau^\prime \notin \{[\exists y_j. \beta]\}^d\).

2. Rule Transitivity. From the rule, it follows that both \(\phi \subseteq_M \gamma\) and \(\gamma \subseteq_M \psi\) hold. Let \(d \in \Omega_\tau^\beta\) and \(\tau \in \{[\phi \land \psi]\}^d\). From \(\phi \subseteq_M \gamma\), \(\tau \in \{[\phi]\}^d\), and the induction hypothesis, it follows that \(\tau \in \{[\gamma \land \psi]\}^d\). From this, \(\gamma \subseteq_M \psi\), and the induction hypothesis, it follows that \(\tau \in \{[\psi]\}^d\). Therefore, \(\phi \subseteq_M \psi\). This completes the proof.

C.4.2 Data security is a sound under-approximation of judgment’s security

Here we introduce data security, a weaker notion than judgment security. Afterwards, we show that it is a sound under-approximation of judgment’s security.

Let \(P = (M, f)\) be an extended configuration, \(L\) be the P-LTS, \(u \in U\) be a user, \(r \in \text{traces}(L)\) be an L-run, \(\phi \in \text{RC}_{\text{boot}}\) be a sentence, and \(1 \leq i \leq |r|\). Furthermore, let \(s\) be the \(i\)-th state of \(r\). The judgment \(r, i, u, \phi\) is data-secure for \(M\) and \(u\), denoted by \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\), if for all \(s', s'' \in \text{sysState}(s)\), \(\phi'^{s''} = \phi'^{s'}\). Where \(\text{sysState}(s)\) is the data-indistinguishability relation defined in Chapter 6 and \(\text{sysState}(s)\) is the set of \(s' \in \Pi M\) such that \(\text{sysState}(s)\).

Proposition C.7 states that \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\) holds for a given judgment \(r, i, u, \phi\), if \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\). The proof follows from our results in Chapter 3.

Proposition C.7. Let \(P = (M, f)\) be an extended configuration, \(L\) be the P-LTS, \(u \in U\) be a user, \(r \in \text{traces}(L)\) be an L-run, \(\phi \in \text{RC}_{\text{boot}}\) be a sentence, and \(1 \leq i \leq |r|\). Given a judgment \(r, i, u, \phi\), then \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\).

Proof. We prove the claim by contradiction. Let \(P = (M, f)\) be an extended configuration, \(L\) be the P-LTS, \(u \in U\) be a user, \(r \in \text{traces}(L)\) be an L-run, \(\phi \in \text{RC}_{\text{boot}}\) be a sentence, and \(1 \leq i \leq |r|\). Furthermore, let \(s = \langle db, U, sec, T, c, e \rangle\) be the \(i\)-th state of \(r\). Assume, for contradiction's sake, that \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\) holds and \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\) does not hold. We denote, for brevity's sake, the fact that \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\) does not hold as \(\neg \text{secure}^\text{data}_{P,u}(r, i, u, \phi)\). From \(\neg \text{secure}^\text{data}_{P,u}(r, i, u, \phi)\), it follows that there is a run \(r' \in \text{traces}(L)\), whose last state is \(s' = \langle db', U', sec', T', c', e \rangle\), such that \(r' \equiv_{P,u} r\) and \(\phi'^{s'} \neq \phi'^{s}\). From the \((P, u)\)-indistinguishability definition, it follows that \(\text{sysState}(\text{last}(r'))\) and \(\text{sysState}(\text{last}(r))\) are data indistinguishable according to \(M\) and \(u\), i.e., \(\text{sysState}(\text{last}(r')) \equiv_{\text{sysState}} \text{sysState}(\text{last}(r))\). From \(\text{secure}^\text{data}_{P,u}(r, i, u, \phi)\), it also follows that for all
C.4.3 Properties of $\phi^T_{s,u}$ and $\phi_{s,u}^\bot$

We now show that the rewritings $\phi^T_{s,u}$ and $\phi_{s,u}^\bot$ provide the desired properties. First, in Lemma C.5 we prove that the two rewritings satisfy the following invariants: “if $\phi^T_{s,u}$ holds in $s$ and also $\phi$ holds in $s$” and “if $\phi^T_{s,u}$ does not hold in $s$, then also $\phi$ does not hold in $s$”. Afterwards, in Lemma C.6 we show that both $\phi^T_{s,u}$ and $\phi_{s,u}^\bot$ are secure. Then, in Lemma C.8 we prove that $\phi^T_{s,u}$ and $\phi_{s,u}^\bot$ are equivalent to $\phi^T_{v,u}$ and $\phi_{v,u}^\bot$ for any two data indistinguishable M-state $s$ and $s'$. Finally, in Proposition C.8 we show that both $\phi^T_{s,u}$ and $\phi_{s,u}^\bot$ are domain-independent.

Lemma C.5. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ be a partial M-state, $u \in U$ be a user, and $\phi$ be a D-formula. For all assignments $\nu$ that are well-formed for $\phi$, the following conditions hold:

- If $[\phi^T_{s,u}]^{db}_\nu = \top$, then $[\nu \phi]^{db}_\nu = \top$, and $[\phi^T_{s,u}]^{db}_\nu = \bot$, then $[\nu \phi]^{db}_\nu = \bot$.

Proof. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, sec, T, V \rangle$ be a partial M-state, $u \in U$ be a user, and $\phi$ be a D-formula. Furthermore, let $\nu$ be an assignment that is well-formed for $\phi$. We prove our claim by induction on the structure of the formula $\phi$.

Base Case. There are four cases:

1. $\phi ::= x = y$. In this case, $\phi^T_{s,u} = \phi$. From this, it follows that $[(x = y)^T_{s,u}, \nu]^{db}_\nu = [(x = y)]^{db}_\nu$ and therefore, our claim follows trivially.

2. $\phi ::= \top$. The proof of this case is similar to that of $\phi ::= x = y$.

3. $\phi ::= \bot$. The proof of this case is similar to that of $\phi ::= x = y$.

4. $\phi ::= R(\tau)$. Let $\tau$ be the tuple $\nu(\tau)$. Note that since $\nu$ is well-formed for $\phi$, $\tau$ is well-defined. Assume that $[\nu \phi]^{db}_\nu = \top$. From this, $\phi_{s,u}^\bot = \nu(\tau)$. From $S \subseteq R^+_{s,u}$, it follows that there is an $s \in R^+_{s,u}$ such that $\phi \in \nu(\tau)$. Since $s \in R^+_{s,u}$, it follows that $S \subseteq M R$. From $S \subseteq M R$, $\tau \in \nu(\tau)$, and Proposition C.6, it follows that $I \in \nu(\tau)(R)$. From this, it follows that $[\nu \phi]^{db}_\nu = \top$. Assume that $[\phi^T_{s,u}]^{db}_\nu = \bot$. From this, and $\phi^T_{s,u} = \nu(\tau)$, it follows that there is an $s \in R^+_{s,u}$ such that $\phi \notin \nu(\tau)$. Since $s \in R^+_{s,u}$, it follows that $R \subseteq M S$. From $R \subseteq M S$, $\tau \notin \nu(\tau)$, and Proposition C.6, it follows that $I \notin \nu(\tau)(R)$. From this, it follows that $[\nu \phi]^{db}_\nu = \bot$. This completes the proof of the base case.

Induction Step. Assume that our claim holds for all sub-formulae of $\phi$. We now show that our claim holds also for $\phi$. There are a number of cases depending on $\phi$’s structure.

1. $\phi ::= \psi \land \gamma$. Assume that $[\phi^T_{s,u}]^{db}_\nu = \top$. From this and $\phi^T_{s,u} = \psi \land \gamma^T_{s,u}$, it follows that $[\psi^T_{s,u}, \nu]^{db}_\nu = \top$ and $[\gamma^T_{s,u}, \nu]^{db}_\nu = \top$. Since $\nu$ is well-formed for $\phi$, it is also well-formed for $\psi$ and $\gamma$ because $free(\psi) \subseteq free(\phi)$ and $free(\gamma) \subseteq free(\phi)$. From $[\psi^T_{s,u}, \nu]^{db}_\nu = \top$ and the induction hypothesis, it follows that $[\psi^db]^{db}_\nu = \top$. From $[\gamma^T_{s,u}, \nu]^{db}_\nu = \top$ and the induction hypothesis, it follows that $[\gamma^db]^{db}_\nu = \top$. From $[\nu \psi]^{db}_\nu = \top$ and $[\nu \gamma]^{db}_\nu = \top$, and the relational calculus semantics, it follows that $[\nu \psi \land \gamma]^{db}_\nu = \top$. Assume that $[\phi^T_{s,u}]^{db}_\nu = \bot$. From this and $\phi^T_{s,u} = \psi \land \gamma^T_{s,u}$, there are two cases:

(a) $[\psi^T_{s,u}, \nu]^{db}_\nu = \bot$. From $[\psi^T_{s,u}, \nu]^{db}_\nu = \bot$ and the induction hypothesis, it follows that $[\psi^db]^{db}_\nu = \bot$. From this and $\phi ::= \psi \land \gamma$, it follows that $[\phi^db]^{db}_\nu = \bot$.

(b) $[\gamma^T_{s,u}, \nu]^{db}_\nu = \bot$. From $[\gamma^T_{s,u}, \nu]^{db}_\nu = \bot$ and the induction hypothesis, it follows that $[\gamma^db]^{db}_\nu = \bot$. From this and $\phi ::= \psi \land \gamma$, it follows that $[\phi^db]^{db}_\nu = \bot$.

2. $\phi ::= \psi \lor \gamma$. The proof of this case is similar to that of $\phi ::= \psi \land \gamma$.

3. $\phi ::= \lnot \psi$. Assume that $[\phi^T_{s,u}]^{db}_\nu = \top$. From this and $\phi^T_{s,u} = \lnot \psi^T_{s,u}$, it follows that $[\psi^T_{s,u}, \nu]^{db}_\nu = \bot$. From this and the induction hypothesis, it follows that $[\psi^db]^{db}_\nu = \bot$. From this, $\phi ::= \psi$, and the relational calculus semantics, it follows that $[\phi^db]^{db}_\nu = \top$. Assume that $[\phi^T_{s,u}]^{db}_\nu = \bot$. From this and $\phi^T_{s,u} = \lnot \psi^T_{s,u}$, it follows that $[\psi^T_{s,u}, \nu]^{db}_\nu = \top$. From this and the induction hypothesis, it follows that $[\psi^db]^{db}_\nu = \bot$. From this and $\phi ::= \lnot \psi$, it follows that $[\phi^db]^{db}_\nu = \bot$.

4. $\phi ::= \exists x. \psi$. Assume that $[\phi^T_{s,u}]^{db}_\nu = \top$. From this and $\phi^T_{s,u} = \exists x. \psi^T_{s,u}$, it follows that there is a $v \in \text{dom}$ such that $[\psi^T_{s,u}, \nu(x \mapsto v)]^{db}_\nu = \top$. Note that since $v$ is well-formed for $\phi$, $\nu(x \mapsto v)$ is well-formed for $\psi$ because $\phi ::= \exists x. \psi$. From this, $[\psi^T_{s,u}, \nu(x \mapsto v)]^{db}_\nu = \top$, and the induction hypothesis, it follows that $[\nu \psi(x \mapsto v)]^{db}_\nu = \top$. From this and $\phi ::= \exists x. \psi$, it follows that $[\nu \psi]^{db}_\nu = \top$. Assume that $[\phi^T_{s,u}]^{db}_\nu = \bot$. From this and $\phi^T_{s,u} = \exists x. \psi^T_{s,u}$, it follows that for all $v \in \text{dom}$, $[\psi^T_{s,u}, \nu(x \mapsto v)]^{db}_\nu = \bot$. Note that since $v$ is well-formed for $\phi$, $\nu(x \mapsto v)$ is well-formed for $\psi$. From this, $[\psi^T_{s,u}, \nu(x \mapsto v)]^{db}_\nu = \bot$, and Proposition C.6, it follows that $I \notin \nu(\tau)(R)$.


because \( \phi := \exists x. \psi \). From this, \([\psi^t x \nu \vdash v]^{db} = \bot\), and the induction hypothesis, it follows that for all \( v \in \text{dom} \), \([\psi v(x \mapsto v)]^{db} = \bot\). From this and \( \phi := \exists x. \psi \), it follows that \([\psi]^{db} = \bot\).

5. \( \phi := \forall x. \psi \). The proof of this case is similar to that of \( \phi := \exists x. \psi \).

This completes the proof of the induction step.

This completes the proof of our claim.

In Lemma C.6, we prove that our rewritings are secure.

**Lemma C.6.** Let \( P = \langle M, f \rangle \) be an extended configuration, where \( M = \langle D, \Gamma \rangle \) is a system configuration and \( f \) is an \( M \)-PDP, \( r \in \text{traces}(L) \) be a run, \( \phi \) be a RC-formula, and \( 1 \leq i \leq r \). Furthermore, let \( s \) be the \( i \)-th state of \( r \). For all assignments \( \nu \) over \( \text{dom} \) that are well-formed for \( \phi \), \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \), \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^*_x x \nu) \), and \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^*_x x \nu) \) hold.

**Proof.** The security of \( r, i \vdash u \phi^T x \nu \) follows trivially from that of \( r, i \vdash u \phi^T x \nu \) and \( r, i \vdash u \phi^*_x x \nu \). Therefore, in the following we prove just that \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \) and \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^*_x x \nu) \) hold. Let \( M = \langle D, \Gamma \rangle \) be a system configuration, \( s = (db, U, sec, T, V) \) be a partial \( M \)-state, \( u \in U \) be a user, and \( \phi \) be a \( D \)-formula. Furthermore, let \( v \) be an assignment that is well-formed for \( \phi \). We prove our claim by induction on the structure of the formula \( \phi \).

**Base Case** There are four cases:

1. \( \phi := x = y \). The claim holds trivially. Indeed, \( \phi^T x \nu = \phi^*_x x \nu \) are always equivalent either to \( \top \) or to \( \bot \). Since for all \( s', s'' \in \text{sysState}(last(r')) \text{data}_{M,u} \) \( \text{sysState}(last(r')) \text{data}_{M,u} \) \( s' = [\top]^{db} \) \( [\top]^{db} \) and \( \bot \). From this and \( \phi := x = y \), it follows that both \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \) and \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^*_x x \nu) \) hold.

2. \( \phi := \top \). The proof of this case is similar to that of \( \phi := x = y \).

3. \( \phi := \bot \). The proof of this case is similar to that of \( \phi := x = y \).

4. \( \phi := R(\gamma) \). Assume, for contradiction’s sake, that \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \) does not hold. From this and \( \text{secure}^{\text{data}}_{P,u}'s \) definition, it follows that there are two \( M \)-system states \( s' = (db', U, sec, T, V) \) and \( s'' = (db'', U, sec, T, V) \in \text{sysState}(last(r')) \text{data}_{M,u} \) \( \text{sysState}(last(r')) \text{data}_{M,u} \) such that \( \phi^T x \nu \text{data}_{M,u} \) \( \phi^*_x x \nu \text{data}_{M,u} \) \( \phi^*_x x \nu \text{data}_{M,u} \). From this and \( \phi := R(\gamma) \), it follows that \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \) and \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^*_x x \nu) \) hold. Let this rules out the cases in which \( R^*_u = \emptyset \) for any \( v \in \{ \top, \bot \} \). We assume without loss of generality that \( \phi^T x \nu \text{data}_{M,u} = [\top]^{db} \) \( \phi^*_x x \nu \text{data}_{M,u} = [\bot]^{db} \). From this and \( \phi := R(\gamma) := \bigvee_{S \in R^*_u} S(\gamma) \), it follows that there is a predicate symbol \( S \) in the extended vocabulary such that \( \nu(\gamma) \in db'(S) \) \( \nu(\gamma) \notin db'(S) \) and \( \nu(\gamma) \notin db'(S) \).

The proof of \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \) is analogous. This completes the proof of the base case.

**Induction Step** Assume that our claim holds for all sub-formulae of \( \phi \). We now show that our claim holds also for \( \phi \). There are a number of cases depending on \( \phi \)’s structure.

1. \( \phi := \psi \land \gamma \). Assume, for contradiction’s sake, that \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \) does not hold. From this and \( \text{secure}^{\text{data}}_{P,u}'s \) definition, it follows that there are two \( M \)-system states \( s' = (db', U, sec, T, V) \) and \( s'' = (db'', U, sec, T, V) \in \text{sysState}(last(r')) \text{data}_{M,u} \) \( \text{sysState}(last(r')) \text{data}_{M,u} \) such that \( \phi^T x \nu \text{data}_{M,u} \) \( \phi^*_x x \nu \text{data}_{M,u} \) \( \phi^*_x x \nu \text{data}_{M,u} \). We assume, without loss of generality, that \( \phi^T x \nu \text{data}_{M,u} = [\top]^{db} \) \( \phi^*_x x \nu \text{data}_{M,u} = [\bot]^{db} \). From this and \( \phi := \psi \land \gamma \), it follows that \( \nu(\gamma) \in db'(S) \) \( \nu(\gamma) \notin db'(S) \) and \( \nu(\gamma) \notin db'(S) \).

The proof of \( \text{secure}^{\text{data}}_{P,u}(r, i \vdash u \phi^T x \nu) \) is analogous.

2. \( \phi := \psi \lor \gamma \). The proof of this case is similar to that of \( \phi := \psi \land \gamma \).

3With a slight abuse of notation, we denote by \( db(S) \) the materialization of \( S \) in \( db \).
3. $\phi := \neg \psi$. Assume, for contradiction’s sake, that $\text{secure}_\text{data}^\phi(r, i \mapsto u \phi^+ u \nu)$ does not hold. From this and $\text{secure}_\text{data}^\phi$’s definition, it follows that there are two $M$-system states $s' = \langle db', U, \text{sec}, T, V \rangle$ and $s'' = \langle db'', U, \text{sec}, T, V \rangle$ in $\text{sysState}(\text{last}(r'))^\text{data}^\phi_{M, u}$ such that $[\phi^+ u \nu]^{db'} \neq [\phi^+ u \nu]^{db''}$. We assume, without loss of generality, that $[\phi^+ u \nu]^{db'} = \top$ and $[\phi^+ u \nu]^{db''} = \bot$. From this and $\phi^+ u \nu = \psi^+ u \nu$, it follows that $[\psi^+ u \nu]^{db'} = \bot$ and $[\psi^+ u \nu]^{db''} = \top$. From this, it follows that $\text{secure}_\text{data}^\phi(r, i \mapsto u \psi^+ u \nu)$ does not hold. From the induction hypothesis and $\phi := \neg \psi$, it follows that $\text{secure}_\text{data}^\phi(r, i \mapsto u \psi^+ u \nu)$ holds leading to a contradiction. The proof of $\text{secure}_\text{data}^\phi(r, i \mapsto u \phi^+ u \nu)$ is analogous.

4. $\phi := \exists x. \psi$. Assume, for contradiction’s sake, that $\text{secure}_\text{data}^\phi(r, i \mapsto u \phi^+ u \nu)$ does not hold. From this and $\text{secure}_\text{data}^\phi$’s definition, it follows that there are two $M$-system states $s' = \langle db', U, \text{sec}, T, V \rangle$ and $s'' = \langle db'', U, \text{sec}, T, V \rangle$ in $\text{sysState}(\text{last}(r'))^\text{data}^\phi_{M, u}$ such that $[\phi^+ u \nu]^{db'} \neq [\phi^+ u \nu]^{db''}$. We assume, without loss of generality, that $[\phi^+ u \nu]^{db'} = \top$ and $[\phi^+ u \nu]^{db''} = \bot$. From this and $\phi^+ u \nu = \exists x. \psi^+ u \nu$, it follows that there is a $v' \in \text{dom}$ such that $[\psi^+ u \nu(x \mapsto v')]^{db'} = \top$ and there is no $v'' \in \text{dom}$ such that $[\psi^+ u \nu(x \mapsto v'')]^{db''} = \top$. Therefore, $[\psi^+ u \nu(x \mapsto v')]^{db'} = \top$ and $[\psi^+ u \nu(x \mapsto v'')]^{db''} = \bot$. Note that $\psi(x \mapsto v')$ is well-formed for $\psi^+ u \nu$. From this, it follows that $\text{secure}_\text{data}^\phi(r, i \mapsto u \psi^+ u \nu(x \mapsto v'))$ does not hold. However, from the fact that $\psi(x \mapsto v')$ is well-formed for $\psi^+ u \nu$ and the induction hypothesis, it follows that $\text{secure}_\text{data}^\phi(r, i \mapsto u \psi^+ u \nu[x \mapsto v'])$ holds leading to a contradiction. The proof of $\text{secure}_\text{data}^\phi(r, i \mapsto u \phi^+ u \nu)$ is analogous.

5. $\phi := \forall x. \psi$. The proof of this case is similar to that of $\phi := \exists x. \psi$.

This completes the proof of the induction step.

This completes the proof of our claim. 

Lemma C.7 states that the result of the bound function is the same for any two data indistinguishable states.

Lemma C.7. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, \text{sec}, T, V \rangle$ and $s' = \langle db', U', \text{sec}', T', V' \rangle$ be two partial $M$-states, $u \in U$ be a user, $v \in \{ \top, \bot \}$, and $\phi$ be a $D$-formula. If $s \equiv_{\text{data}}^\phi s'$, then $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v)$.

Proof. Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, \text{sec}, T, V \rangle$ and $s' = \langle db', U', \text{sec}', T', V' \rangle$ be two partial $M$-states, $u \in U$ be a user, $v \in \{ \top, \bot \}$, and $\phi$ be a $D$-formula. We prove our claim by induction on the structure of the formula $\phi$.

Base Case There are four cases:

1. $\phi := y = z$. The result of $\text{bound}(\phi, s, u, x, v)$ and $\text{bound}(\phi, s', u, x, v)$ does not depend on $s$.

Therefore, $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v)$.

2. $\phi := T$. $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v) = \bot$.

3. $\phi := \bot$. $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v) = \bot$.

4. $\phi := R(\overline{\phi})$. The result of $\text{bound}(\phi, s, u, x, v)$ and $\text{bound}(\phi, s', u, x, v)$ depend only on the sets $R_u^s$ and $R_u^{s'}$, which in turn depend on the content of the sets $R_u^v$, $R_u^{v'}$, $\text{AUTH}^s_u$, and $\text{AUTH}^{s'}_u$. Assume that $s \equiv_{\text{data}}^\phi s'$. From this, it follows that $R_u^v = R_u^{v'}$ (because $D$ is the same and $V = V'$) and $\text{AUTH}^s_u = \text{AUTH}^{s'}_u$ (because $sec = sec'$). From this, it follows that $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v)$.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all sub-formulae of $\phi$. We now show that our claim holds also for $\phi$. There are a number of cases depending on $\phi$’s structure.

1. $\phi := \psi \land \gamma$. Assume that $s \equiv_{\text{data}}^\phi s'$. From this and the induction hypothesis, it follows that $\text{bound}(\psi, s, u, x, v) = \text{bound}(\psi, s', u, x, v)$ and $\text{bound}(\gamma, s, u, x, v) = \text{bound}(\gamma, s', u, x, v)$. From this and $\text{bound}(\phi, s, u, x, v) := \text{bound}(\psi, s, u, x, v) \lor \text{bound}(\gamma, s, u, x, v)$, it follows that $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v)$.

2. $\phi := \psi \lor \gamma$. The proof of this case is similar to that of $\phi := \psi \land \gamma$.

3. $\phi := \neg \psi$. Assume that $s \equiv_{\text{data}}^\phi s'$. From this and the induction hypothesis, it follows that $\text{bound}(\neg \psi, s, u, x, v) = \text{bound}(\neg \psi, s', u, x, v)$. From this, $\neg \text{bound}(\neg \psi, s, u, x, v) = \text{bound}(\psi, s, u, x, v)$ and $\text{bound}(\neg \psi, s', u, x, v) = \text{bound}(\psi, s', u, x, v)$, it follows that $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v)$.

4. $\phi := \exists x. \psi$. Assume that $s \equiv_{\text{data}}^\phi s'$. There are two cases:

(a) $x = y$. In this case, the proof is trivial as $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s', u, x, v) = \bot$.

(b) $x \neq y$. In this case, $\text{bound}(\phi, s, u, x, v) = \text{bound}(\phi, s, u, y, v) \lor \text{bound}(\phi, s, u, y, v)$ and $\text{bound}(\phi, s', u, x, v) = \text{bound}(\phi, s', u, y, v) \lor \text{bound}(\phi, s', u, y, v)$. From $s \equiv_{\text{data}}^\phi s'$ and the induction hypothesis, it follows that $\text{bound}(\psi, s, u, x, v) = \text{bound}(\psi, s', u, x, v)$ and
bound(ψ, s, u, y, v) = bound(ψ, s', u, y, v). From this, bound(ψ, s, u, x, v) = bound(ψ, s', u, x, v) ∧ bound(ψ, s, u, y, v), and bound(ψ', s', u, x, v) = bound(ψ, s', u, x, v) ∧ bound(ψ, s', u, y, v), it follows that bound(ψ, s, u, x, v) = bound(ψ', s', u, x, v).

5. φ := ∀x. ψ. The proof of this case is similar to that of φ := ∃x. ψ.

This completes the proof of the induction step.

This completes the proof of our claim.

Lemma C.8 states that the formulae resulting from our rewriting are the same for any two data indistinguishable states.

Lemma C.8. Let M = (D, Γ) be a system configuration, s = (db, U, sec, T, V) and s' = (db', U', sec', T', V') be two partial M-states, u ∈ U be a user, and φ be a D-formula. If s ≡\text{data}_{M,u} s', then φ\text{\_s,u} = φ\text{\_s',u}, φ\text{\_s',u} = φ\text{\_s,u}, and φ\text{\_s',u} = φ\text{\_s,u}.

Proof. Let M = (D, Γ) be a system configuration, s = (db, U, sec, T, V) and s' = (db', U', sec', T', V') be two partial M-states, u ∈ U be a user, and φ be a D-formula. We prove our claim by induction on the structure of the formula φ.

Base Case There are four cases:
1. φ := x = y. The claim holds trivially. Indeed, φ\text{\_s,u} = φ\text{\_s',u} = φ.
2. φ := T. The proof of this case is similar to that of φ := x = y.
3. φ := ⊥. The proof of this case is similar to that of φ := x = y.
4. φ := R(τ). The formulae φ\text{\_s,u} and φ\text{\_s',u} depend only on the sets R\text{\_s,u} and R\text{\_s',u}, which in turn depends on R\text{\_s'} and R\text{\_s}, AUTH\text{\_s,u} and AUTH\text{\_s',u}. If s ≡\text{data}_{M,u} s', then R\text{\_s'} = R\text{\_s} (because D is the same and V = V') and AUTH\text{\_s,u} = AUTH\text{\_s',u} (because sec = sec'). Therefore, φ\text{\_s,u} = φ\text{\_s',u}.

The proof for φ\text{\_s',u} is analogous.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all formulae whose length is less than φ. We now show that our claim holds also for φ. There are a number of cases depending on φ’s structure.

1. φ := ψ ∧ γ. Assume that s ≡\text{data}_{M,u} s'. From this and the induction hypothesis, it follows that ψ\text{\_s,u} = ψ\text{\_s',u} and γ\text{\_s,u} = γ\text{\_s',u}. From this, φ := ψ ∧ γ, φ\text{\_s,u} := ψ\text{\_s,u} ∧ γ\text{\_s,u}, and φ\text{\_s,u} := ψ\text{\_s,u} ∧ γ\text{\_s,u}, it follows that φ\text{\_s,u} = φ\text{\_s',u}. The proof of φ\text{\_s,u} = φ\text{\_s',u} is analogous.
2. φ := ψ ∨ γ. The proof of this case is similar to that of φ := ψ ∧ γ.
3. φ := ¬ψ. Assume that s ≡\text{data}_{M,u} s'. From this and the induction hypothesis, it follows that ψ\text{\_s,u} = ψ\text{\_s',u} and ψ\text{\_s',u} = ψ\text{\_s,u}. From this, φ := ¬ψ, φ\text{\_s,u} := ¬ψ\text{\_s,u}, and φ\text{\_s,u} := ¬ψ\text{\_s,u}, it follows that φ\text{\_s,u} = φ\text{\_s',u}. The proof of φ\text{\_s,u} = φ\text{\_s',u} is analogous.
4. φ := ∃x. ψ. Assume that s ≡\text{data}_{M,u} s'. From this and the induction hypothesis, it follows that ψ\text{\_s,u} = ψ\text{\_s',u}. We remark that bound(ψ, s, u, x, T) = bound(ψ, s', u, x, T), as proved in Lemma C.7. There are two cases:
   a. bound(ψ, s, u, x, T) = T. From this, bound(ψ, s, u, x, T) = bound(ψ, s', u, x, T), ψ\text{\_s,u} = ψ\text{\_s',u}, φ := ∃x. ψ, φ\text{\_s,u} := ∃x. ψ\text{\_s,u}, and φ\text{\_s,u} := ∃x. ψ\text{\_s,u}, it follows that φ\text{\_s,u} = φ\text{\_s',u}.
   b. bound(ψ, s, u, x, T) = ⊥. From this, bound(ψ, s, u, x, T) = bound(ψ, s', u, x, T), and φ\text{\_s,u}'s definition, it follows that φ\text{\_s',u} = φ\text{\_s,u}.

The proof of φ\text{\_s,u} = φ\text{\_s',u} is analogous.

5. φ := ∀x. ψ. The proof of this case is similar to that of φ := ∃x. ψ.

This completes the proof of the induction step.

The equivalence φ\text{\_s,u} = φ\text{\_s',u} follows trivially from φ\text{\_s,u}'s definition and the fact that φ\text{\_s,u} = φ\text{\_s',u} and φ\text{\_s,u} = φ\text{\_s',u}.

Before proving the domain independence of φ\text{\_s,u} and φ\text{\_s',u}, we introduce some notation. The relation \text{gen}, introduced in [160], is the smallest relation defined by the rules in Figure C.1. Note that we extended \text{gen} by adding the rules \text{Equiv}, Const 1, and Const 2. A relational calculus formula φ is allowed if it satisfies the following conditions:

- for all x ∈ free(φ), \text{gen}(x, φ) holds,
- for every sub-formula ∃x. ψ in φ, \text{gen}(x, ψ) holds, and
- for every sub-formula ∀x. ψ in φ, \text{gen}(x, ¬ψ) holds.

As shown in [160], every allowed formula is domain independent. Note that the addition of the \text{Equiv}, Const 1, and Const 2 rules does not modify this result.

Lemma C.9 presents some useful equivalences that we use in our proof of domain independence.

Lemma C.9. Let M = (D, Γ) be a system configuration, s = (db, U, sec, T, V) be an M-system state, u ∈ U be a user, and v ∈ {⊤, ⊥}. For any formulae φ and ψ, the following equivalences hold:

- (¬φ)\text{\_s,u} ⇔ ¬φ\text{\_s,u},
Proof. Let $M = (D, \Gamma)$ be a system configuration, $s = \langle db,U,sec,T,V \rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and $\psi$ be two formulae.

- $(-\phi)^{v}_{s,u} \equiv -\phi^{v}_{s,u}$. This case follows trivially from the definition of the rewriting.
- $(\phi \wedge \psi)^{v}_{s,u} \equiv (\phi)^{v}_{s,u} \wedge (\psi)^{v}_{s,u}$. This case follows trivially from the definition of the rewriting.
- $(\phi \lor \psi)^{v}_{s,u} \equiv (\phi)^{v}_{s,u} \lor (\psi)^{v}_{s,u}$. This case follows trivially from the definition of the rewriting.
- $(\exists x. \phi)^{v}_{s,u} \equiv (\exists x. \phi)^{v}_{s,u}$. This case follows trivially from the definition of the rewriting.
- $(\forall x. \phi)^{v}_{s,u} \equiv (\forall x. \phi)^{v}_{s,u}$. This case follows trivially from the definition of the rewriting.

There are two cases:

1. $\text{bound}(\phi, s, u, x, v) = \top$. From this, it follows that $(\exists x. \phi)^{v}_{s,u} = (\exists x. \phi)^{v}_{s,u}$. From $(-\phi)^{v}_{s,u} \equiv -\phi^{v}_{s,u}$, it follows that $(\exists x. \phi)^{v}_{s,u} \equiv (\exists x. \phi)^{v}_{s,u}$. From the definition of $\text{bound}$, it follows that $\text{bound}(\neg \phi, s, u, x, v)$ holds. From this and $v = \neg v$, it follows that $\text{bound}(\neg \phi, s, u, x, v)$ holds. From this and $\text{bound}(\phi, s, u, x, v) = \top$, it follows that $\text{bound}(\neg \phi, s, u, x, v) = \top$. From this, it follows that $(\forall x. \neg \phi)^{v}_{s,u} = (\forall x. \neg \phi)^{v}_{s,u}$. From this and $(\phi)^{v}_{s,u} \equiv \neg \phi^{v}_{s,u}$, it follows that $(\forall x. \neg \phi)^{v}_{s,u} \equiv \forall x. \phi^{v}_{s,u}$. From this and standard RC equivalences, it follows that $(\forall x. \neg \phi)^{v}_{s,u} \equiv (\forall x. \neg \phi)^{v}_{s,u}$. From this and $\text{bound}(\phi, s, u, x, v)$, it follows that $(\forall x. \neg \phi)^{v}_{s,u} \equiv (\forall x. \neg \phi)^{v}_{s,u}$.

2. $\text{bound}(\phi, s, u, x, v) = \bot$. From this, it follows that $(\exists x. \phi)^{v}_{s,u} = \bot$. From $(-\phi)^{v}_{s,u} \equiv -\phi^{v}_{s,u}$ and $(\forall x. \phi)^{v}_{s,u}$ holds, it follows that $(\neg (\forall x. \phi)^{v}_{s,u} \equiv -\phi^{v}_{s,u}$. From the definition of $\text{bound}$, it follows that $\text{bound}(\neg \phi, s, u, x, v)$ holds. From this and $v = \neg v$, it follows that $\text{bound}(\neg \phi, s, u, x, v)$ holds. From this and $\text{bound}(\phi, s, u, x, v) = \bot$, it follows that $\text{bound}(\neg \phi, s, u, x, v) = \bot$. From this, it follows that $(\forall x. \neg \phi)^{v}_{s,u} = v$. From this and $(\forall x. \neg \phi)^{v}_{s,u} \equiv - (\forall x. \phi)^{v}_{s,u}$, it follows that $(\forall x. \neg \phi)^{v}_{s,u} \equiv -\phi^{v}_{s,u}$. From this and $\text{bound}(\phi, s, u, x, v)$, it follows that $(\forall x. \neg \phi)^{v}_{s,u} \equiv -\phi^{v}_{s,u}$.

The proof of this case is similar to that of $(\exists x. \phi)^{v}_{s,u} \equiv - (\forall x. \neg \phi)^{v}_{s,u}$.

This completes the proof.

Lemma C.10. Let $M = (D, \Gamma)$ be a system configuration, $s = \langle db,U,sec,T,V \rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and $\phi$ be a formula. Furthermore, let $x \in \text{free}(\phi) \cap \text{free}(\phi^{s,v})$. If $\text{gen}(x, \phi)$ holds, then $\text{gen}(x, \phi)^{s,v}$ holds.

Proof. Let $M = (D, \Gamma)$ be a system configuration, $s = \langle db,U,sec,T,V \rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and $\phi$ be a formula. Furthermore, let $x \in \text{free}(\phi) \cap \text{free}(\phi^{s,v})$. We prove our claim by induction on the length of $\phi$. In the following, the length of $\phi$ is the number of predicates, quantifiers, negations, conjunctions, and disjunctions in $\phi$.

Base Case There are four cases:

- $x \in \text{Gen}(x,R(\phi))$. Pred.
- $x \neq y \text{ and } \text{gen}(x,\phi)$ exists.
- $\text{gen}(x,\phi) \neq \text{gen}(x,\phi \lor \psi)$. Or.
- $\text{gen}(x,\phi) \neq \text{gen}(x,\phi \land \psi)$. And.

- $\text{gen}(x,\phi) \neq \text{gen}(x,\neg \phi)$. Neg.

- $\text{gen}(x,\phi) \neq \text{gen}(x,\phi)$. For all.

- $\phi^{s,u} \equiv \phi^{v}_{s,u}$. Equiv.

Figure C.1: Rules defining the gen relation.
C.4. Data Confidentiality Proofs

This completes the proof of the base case.

**Induction Step** Assume that our claim holds for all formulae whose length is less than \(\phi\)'s length.

1. \(\phi := x = y\). In this case, the claim holds trivially.
2. \(\phi := \top\). In this case, the claim holds trivially.
3. \(\phi := \bot\). In this case, the claim holds trivially.
4. \(\phi := R(x)\). Assume \(\text{gen}(x, \phi)\) holds. From this, it follows that \(x\) is one of the free variables in \(R\). Furthermore, from \(x \notin \text{free}(\phi_{x,u}^u)\), it follows that \(Rx_u \neq \emptyset\). There are two cases:
   (a) \(\phi_{x,u}^u\) is a conjunction of predicates \(S(R)\) such that \(\text{gen}(x, S(R))\) holds. From the rule And 1, it follows that \(\text{gen}(x, \phi_{x,u}^u)\) holds.
   (b) \(\phi_{x,u}^u\) is a disjunction of predicates \(S(R)\) such that \(\text{gen}(x, S(R))\) holds. From the rule Or, it follows that \(\text{gen}(x, \phi_{x,u}^u)\) holds.

This completes the proof of the base case.

Induction Step Assume that our claim holds for all formulae whose length is less than \(\phi\)'s length.

1. \(\phi := \psi \land \gamma\). Assume that \(\text{gen}(x, \phi)\) holds. From this and the rules And 1 and And 2, it follows that either \(\text{gen}(x, \psi)\) or \(\text{gen}(x, \gamma)\) hold. Assume, without loss of generality, that \(\text{gen}(x, \psi)\) holds. From this and the induction hypothesis, it follows that \(\text{gen}(x, \psi_{x,u}^u)\) holds. From this, \(\phi_{x,u}^u := \psi_{x,u} \land \gamma_{x,u}\), and the rule And 1, it follows that \(\text{gen}(x, \phi_{x,u}^u)\) holds.
2. \(\phi := \psi \lor \gamma\). Assume that \(\text{gen}(x, \phi)\) holds. From this and the rule Or, it follows that both \(\text{gen}(x, \psi)\) and \(\text{gen}(x, \gamma)\) hold. From this and the induction hypothesis, it follows that both \(\text{gen}(x, \psi_{x,u}^u)\) and \(\text{gen}(x, \gamma_{x,u}^u)\) hold. From this, \(\phi_{x,u}^u := \psi_{x,u} \lor \gamma_{x,u}\), and the rule Or, it follows that \(\text{gen}(x, \phi_{x,u}^u)\) holds.
3. \(\phi := \neg \psi\). Assume that \(\text{gen}(x, \phi)\) holds. From this and the rule Not, it follows that \(\text{gen}(x, \neg \psi)\). There are a number of cases depending on \(\psi\). In the following, we exploit standard relational calculus equivalences, see, for instance, [10], and the equivalences we proved in Lemma C.9.
   (a) \(\psi := (\alpha \land \beta)\). In this case, \(\text{push}(\neg \psi) = (\neg \alpha \lor \neg \beta)\). From this and \(\text{gen}(x, \psi)\), it follows that \(\text{gen}(x, (\neg \alpha \lor \neg \beta))\). From this and the Or rule, it follows that \(\text{gen}(x, \neg \alpha)\) and \(\text{gen}(x, \neg \beta)\). From this and the induction hypothesis, it follows that \(\text{gen}(x, (\neg \alpha)_{x,u}^u)\) and \(\text{gen}(x, (\neg \beta)_{x,u}^u)\). From this and the Or rule, it follows that \(\text{gen}(x, (\neg \alpha \lor \neg \beta)_{x,u}^u)\). From this, \((\neg \alpha \lor \neg \beta)_{x,u}^u \equiv (\neg \alpha \lor \neg \beta)_{x,u}^u\), and the Equiv rule, it follows that \(\text{gen}(x, (\neg \alpha \lor \neg \beta)_{x,u}^u)\).
   (b) \(\psi := (\alpha \lor \beta)\). The proof is similar to the \(\psi := (\neg \alpha \land \beta)\) case.
   (c) \(\psi := \exists y. \alpha\). In this case, \(\text{push}(\neg \psi) = \forall y. \neg \alpha\). From this and \(\text{gen}(x, \neg \psi)\), it follows \(\text{gen}(x, \forall y. \neg \alpha)\). From this and the induction hypothesis, it follows that \(\text{gen}(x, (\forall y. \neg \alpha)_{x,u}^u)\). From this, \((\forall y. \neg \alpha)_{x,u}^u \equiv (\forall y. \neg \alpha)_{x,u}^u\), and the Equiv rule, it follows that \(\text{gen}(x, (\forall y. \neg \alpha)_{x,u}^u)\). From this, \((\forall y. \neg \alpha)_{x,u}^u \equiv (\forall y. \neg \alpha)_{x,u}^u\), and the Equiv rule, it follows that \(\text{gen}(x, (\forall y. \neg \alpha)_{x,u}^u)\).
   (d) \(\psi := \forall y. \alpha\). The proof is similar to the \(\psi := (\neg \alpha \land \beta)\) case.
   (e) \(\psi := \neg \alpha\). In this case, \(\text{push}(\neg \psi) = \alpha\). From this and \(\text{gen}(x, \neg \psi)\), it follows \(\text{gen}(x, \alpha)\). From this and the induction hypothesis, it follows that \(\text{gen}(x, \alpha_{x,u}^u)\). From this, \(\neg \alpha_{x,u}^u \equiv \alpha_{x,u}^u\), and the Equiv rule, it follows that \(\text{gen}(x, (\neg \alpha)_{x,u}^u)\). From this, \((\neg \alpha)_{x,u}^u \equiv (\neg \alpha)_{x,u}^u\), and the Equiv rule, it follows that \(\text{gen}(x, (\neg \alpha)_{x,u}^u)\). From this and \(\psi := \neg \alpha\), it follows that \(\text{gen}(x, (\neg \alpha)_{x,u}^u)\).
   (f) \(\psi := x = y\). The proof for this case is trivial.
   (g) \(\psi := x \neq y\). The proof for this case is trivial.
4. \(\phi := \exists x. \psi\). Assume that \(\text{gen}(x, \phi)\) holds. From this and the rule Exists, it follows that \(\text{gen}(x, \psi)\) holds. From this and the induction hypothesis, it follows that \(\text{gen}(x, \psi_{x,u}^u)\) holds. From this, \(\phi_{x,u}^u := \exists x. \psi_{x,u}^u\), and the rule Exists, it follows that \(\text{gen}(x, \phi_{x,u}^u)\) holds.
5. \(\phi := \forall x. \psi\). The proof of this case is similar to that of \(\phi := \exists x. \psi\).

This completes the proof of the induction step.

Lemma C.11. Let \(M = (D, \Gamma)\) be a system configuration, \(s = (db, U, sec, T, V)\) be an M-system state, \(u \in U\) be a user, \(v \in \{T, \bot\}\), and \(\phi\) be a formula. For every sub-formula \(\exists x. \psi\) of \(\phi\), if \(\text{gen}(x, \psi)\) holds and \(x \in \text{free}(\psi) \cap \text{free}(\psi_{x,u}^u)\), then \(\text{gen}(x, \phi_{x,u}^u)\) holds.

Proof. Let \(M = (D, \Gamma)\) be a system configuration, \(s = (db, U, sec, T, V)\) be an M-system state, \(u \in U\) be a user, \(v \in \{T, \bot\}\), and \(\phi\) be a formula. We prove our claim by induction on the length of \(\phi\).
In the following, the size of $\phi$ is the number of predicates, quantifiers, negations, conjunctions, and disjunctions in $\phi$.

**Base Case** The claim is vacuously satisfied for the base cases as there is no sub-formula of the form $\exists x. \psi$.

**Induction Step** Assume that our claim holds for all formulae whose length is less than $\phi$. We now show that our claim holds also for $\phi$. There are a number of cases depending on $\phi$’s structure.

1. $\phi := \psi \land \gamma$. Let $\alpha$ be a sub-formula of $\phi$ of the form $\exists x. \beta$ such that $\text{gen}(x, \beta)$ holds and $x \in \text{free}(\beta) \cap \text{free}(\beta^s_u)$. The formula $\alpha$ is either a sub-formula of $\psi$ or a sub-formula of $\gamma$. From this and the induction hypothesis, it follows that $\text{gen}(x, \beta^s_u)$ holds.

2. $\phi := \psi \lor \gamma$. The proof of this case is similar to that of $\phi := \psi \land \gamma$.

3. $\phi := \lnot \psi$. Let $\alpha$ be a sub-formula of $\phi$ of the form $\exists x. \beta$ such that $\text{gen}(x, \beta)$ holds and $x \in \text{free}(\beta) \cap \text{free}(\beta^s_u)$. Since $\phi := \lnot \psi$, the formula $\alpha$ is also a sub-formula of $\psi$. From this and the induction hypothesis, it follows that $\text{gen}(x, \beta^s_u)$ holds.

4. $\phi := \exists x. \psi$. Let $\alpha$ be a sub-formula of $\phi$ of the form $\exists x. \beta$ such that $\text{gen}(x, \beta)$ holds and $x \in \text{free}(\beta) \cap \text{free}(\beta^s_u)$. There are two cases:
   - (a) $\alpha$ is a sub-formula of $\psi$. From this and the induction hypothesis, it follows that $\text{gen}(x, \beta^s_u)$ holds.
   - (b) $\alpha = \phi$. From $\text{gen}(x, \beta)$, $x \in \text{free}(\beta) \cap \text{free}(\beta^s_u)$, and Lemma C.10, it follows that $\text{gen}(x, \beta^s_u)$ holds.

5. $\phi := \forall x. \psi$. The proof of this case is similar to that of $\phi := \exists x. \psi$.

This completes the proof of the induction step.

This completes the proof of our claim.

**Lemma C.12.** Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, \text{sec}, T, V \rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and $\phi$ be a formula. For every sub-formula $\forall x. \psi$ of $\phi$, if $\text{gen}(x, \psi)$ holds and $x \in \text{free}(\psi) \cap \text{free}(\psi^s_u)$, then $\text{gen}(x, (\lnot \psi)^s_u)$ holds.

**Proof.** The proof is similar to that of Lemma C.11.

**Lemma C.13.** Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, \text{sec}, T, V \rangle$ be an M-system state, $u \in U$ be a user, $v \in \{\top, \bot\}$, and $\phi$ be a formula. Let $Q \in \{\exists, \forall\}$ be a quantifier and $\text{sub}_Q(\phi)$ be the set of sub-formulae of $\phi$ of the form $Q x. \psi$. There is a surjective function $f$ from $\text{sub}_Q(\phi)$ to $\text{sub}_Q(\phi^s_u)$ such that for any $Q x. \psi$ in $\text{sub}_Q(\phi)$, if $f(Q x. \psi)$ is defined, then $f(Q x. \psi)^s_u = Q x. \psi^s_u$.

**Proof.** The claim directly follows from the definition of $\phi^s_u$.

Finally, Proposition C.8 states that our rewriting is domain independent whenever the original formula and the views can be proved to be allowed (which implies that they are domain independent).

**Proposition C.8.** Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = \langle db, U, \text{sec}, T, V \rangle$ be an M-system state, $u \in U$ be a user, and $\phi$ be a formula. If $\phi$ is allowed and all views in $V$ are allowed, then $\phi^s_u$, $\phi^s_w$, and $\phi^s_i$ are domain independent.

**Proof.** From Lemmas C.10–C.13, it follows that if $\phi$ is allowed, then both $\phi^s_u$ and $\phi^s_w$ are allowed. Since every allowed formula is domain independent [160], it follows that both $\phi^s_u$ and $\phi^s_w$ are domain independent. Finally, the domain independence of $\phi^s_i$ follows easily from its definition and the domain independence of $\phi^s_u$ and $\phi^s_w$.

**C.4.4 The secure function is a sound under-approximation of data security**

We now prove the main result about our rewriting, namely that the secure function is, indeed, a sound, under-approximation of the notion of judgment’s security.

**Proposition C.9.** Let $P = \langle M, f \rangle$ be an extended configuration, $L$ be the $P$-LTS, $u \in U$ be a user, $r \in \text{traces}(L)$ be an $L$-run, $\phi \in \text{RC}_\text{tool}$ be a sentence, and $1 \leq i \leq |r|$. Furthermore, let $s$ be the $i$-th state in $r$. The following statements hold:

1. Given a judgment $r, i \vdash_u \phi$, if $\text{secure}(u, \phi, s) = \top$, then $\text{secure}^\text{data}(r, i, \vdash_u \phi)$ holds.
2. Given a judgment $r, i \vdash_u \phi$, if $\text{secure}(u, \phi, s) = \top$, then $\text{secure}(r, i, \vdash_u \phi)$ holds.

**Proof.** Note that the second statement follows trivially from Proposition C.7 and the first statement. Therefore, in the following we prove just that given a judgment $r, i \vdash_u \phi$, if $\text{secure}(u, \phi, s) = \top$, then $\text{secure}^\text{data}(r, i, \vdash_u \phi)$ holds.

Let $P = \langle M, f \rangle$ be an extended configuration, $L$ be the $P$-LTS, $u \in U$ be a user, $r \in \text{traces}(L)$ be an $L$-run, $\phi \in \text{RC}_\text{tool}$ be a sentence, and $1 \leq i \leq |r|$. Furthermore, let $s = \langle db, U, \text{sec}, T, V, c \rangle$ be the $i$-th state in $r$. Assume that $\text{secure}(u, \phi, s) = \top$. From this and secure’s definition, $[\phi^s_u]^\text{data} = \bot$. In the
following, with a slight abuse of notation we ignore the inline and ext functions in \( \phi_{\text{rw}}^{\text{data}} \)'s definition. This is without loss of generality since inline and ext do not modify \( \phi \)'s result. From this and \( \phi_{\text{rw}}^{\text{data}} \)'s definition, it follows that either \( [\phi_{\text{rw}}^{\text{data}}]_u = \top \) or \( [\phi_{\text{rw}}^{\text{data}}]_u = \bot \). Note that from Lemma C.6, it follows that
\[
\text{secure}_{P,u}^{\text{data}}(r, i \mapsto u \phi_{\text{rw}}^{\text{data}}) \text{ and secure}_{P,u}^{\text{data}}(r, i \mapsto u \phi_{\text{rw}}^{\text{data}}).
\]
Furthermore, let \( \Delta \) be the equivalence class
\[
\left[ \text{sysState}(s) \right]_{M,u}^{\text{data}}.
\]
There are two cases:

1. \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = \top\). From secure\(_{P,u}^{\text{data}}(r, i \mapsto u \phi_{\text{rw}}^{\top})\), it follows that for all \( s', s'' \in \Delta \), \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = [\phi_{\text{rw}}^{\top}]_u^{\text{data}}\). From this, \( s, s' \in \Delta \), and \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = \top\), it follows that \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = \top\) for all \( s' \in \Delta \).

From Lemma C.8, it follows that for all \( s', s'' \in \Delta \), \( \phi_{\text{rw}}^{\top} = \phi_{\text{rw}}^{\top} = \phi_{\text{rw}}^{\top} = \phi_{\text{rw}}^{\top} = \top\). From this and the fact that for all \( s' \in \Delta \), \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = \top\), it follows that for all \( s' \in \Delta \), \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = \top\). From this and and Lemma C.5, it follows that for all \( s' \in \Delta \), \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = \top\). From this, it follows that for all \( s', s'' \in \Delta \), \([\phi_{\text{rw}}^{\top}]_u^{\text{data}} = [\phi_{\text{rw}}^{\top}]_u^{\text{data}}\). From this, \( r \)'s definition, and secure\(_{P,u}^{\text{data}}\), it follows that secure\(_{P,u}^{\text{data}}(r, i \mapsto u \phi)\).

This completes the proof of our claim.

\[\square\]

Lemma C.14 states that the secure function produces the same result for indistinguishable states.

\textbf{Lemma C.14.} Let \( M \) be a system configuration, \( u \in U \) be a user, \( s, s' \in \Omega_M \) be two \( M \)-states such that \( \text{sysState}(s) \cong \text{sysState}(s') \) and \( \phi \) be a sentence. Then, secure\((u, \phi, s) = \top \) iff secure\((u, \phi, s') = \top \).

\textbf{Proof.} Let \( M \) be a system configuration, \( u \in U \) be a user, \( s, s' \in \Omega_M \) be two \( M \)-states such that \( \text{sysState}(s) \cong \text{sysState}(s') \) and \( \phi \) be a sentence. We now prove that secure\((u, \phi, s) = \top \) iff secure\((u, \phi, s') = \top \). Assume, for contradiction’s sake, that secure\((u, \phi, s) \neq \top \). From this, it follows that \( [\phi_{\text{rw}}^{\text{data}}]_u = [\phi_{\text{rw}}^{\text{data}}]_u \). From sysState\((s) \cong \text{sysState}(s') \) and C.8, it follows that \([\phi_{\text{rw}}^{\text{data}}]_u = [\phi_{\text{rw}}^{\text{data}}]_u = [\phi_{\text{rw}}^{\text{data}}]_u = [\phi_{\text{rw}}^{\text{data}}]_u = [\phi_{\text{rw}}^{\text{data}}]_u \). From this and this, it follows that for all \( s' \in \Delta \), \([\phi_{\text{rw}}^{\text{data}}]_u = \top \). From this, it follows that secure\(_{P,u}^{\text{data}}(r, i \mapsto u \phi_{\text{rw}}^{\text{data}})\), which has been proved in Lemma C.6.

This completes the proof of our claim.

\[\square\]

\textbf{C.4.5 Auxiliary results about } f_{\text{conf}}

Here we prove some auxiliary results about \( f_{\text{conf}} \).

\textbf{Lemma C.15.} Let \( M = (D, \Gamma) \) be a system configuration, \( u \in U \) be a user, \( s, s' \in \Omega_M \) be two \( M \)-states such that \( \text{sysState}(s) \cong \text{sysState}(s') \), \( \text{invoker}(s) = \text{invoker}(s') \), \( \text{tr}(s) = \text{tr}(s') \), and \( a \) be an action in \( A_{D,u} \). Then, \( f_{\text{conf}}(s, a) = f_{\text{conf}}(s', a) \).

\textbf{Proof.} Let \( M = (D, \Gamma) \) be a system configuration, \( u \in U \) be a user, \( s, s' \in \Omega_M \) be two \( M \)-states such that \( \text{sysState}(s) \cong \text{sysState}(s') \), \( \text{invoker}(s) = \text{invoker}(s') \), \( \text{tr}(s) = \text{tr}(s') \), and \( a \) be an action in \( A_{D,u} \). There are a number of cases depending on the action \( a \).

1. \( a = (u', \text{SELECT}, \phi) \). Assume, for contradiction’s sake, that \( f_{\text{conf}}(s, a) \neq f_{\text{conf}}(s', a) \). This happens iff secure\((u, \phi, s) \neq \top \) iff secure\((u, \phi, s') \neq \top \). This contradicts Lemma C.14 because sysState\((s) \cong \text{sysState}(s') \).

2. \( a = (u', \text{INSERT}, R, T) \). We claim that noLeak\((s, a, u) = \top \). Assume, for contradiction’s sake, that \( f_{\text{conf}}(s, a) \neq f_{\text{conf}}(s, a) \). This happens iff there is a formula \( \phi \), which has been derived using the getInfo, getInfD, or getInfD functions, such that secure\((u, \phi, s) \neq \top \). This contradicts Lemma C.14 because sysState\((s) \cong \text{sysState}(s') \).

We prove our claim that noLeak\((s, a, u) = \top \) for any two states \( s \) and \( s' \) such that sysState\((s) \cong \text{sysState}(s') \). Assume, for contradiction’s sake, that this is not the case. Without loss of generality we assume that noLeak\((s, a, u) = \top \). From noLeak\((s, a, u) = \top \) and \( D \), it follows that for all views \( V \) such that \( \langle V, \text{SELECT}, \phi \rangle \in \text{permissions}(s, u) \) and \( R \in \text{Det}(V, s, M) \), for all \( o \in \text{Det}(V, s, M) \), \( \langle V, \text{SELECT}, o \rangle \) is in permissions\((s, u) \). From sysState\((s) \cong \text{sysState}(s') \), it follows that sec = sec. From this, permissions\((s, u) = permissions(s', u) \). From noLeak\((s', a, u) = \top \), there are two views or tables \( V' \) and \( o' \) such that
This completes the proof. This contradicts noLeak(s, a, u) = ⊤.

3. \(a = \langle u', \text{DELETE}, R, T \rangle\). The proof of this case is similar to the a = \(\langle u', \text{INSERT}, R, T \rangle\) case.

4. \(a = \langle op, u', p, u' \rangle\), where \(op \in \{\oplus, \ominus\}\). Assume, for contradiction's sake, that \(f^\text{conf}(s, a) \neq f^\text{conf}(s', a)\). Note that this happens if \(p = \langle \text{SELECT}, \alpha \rangle\) for some \(\alpha\). Without loss of generality, we further assume that \(f^\text{conf}(s, a) = \top\) and \(f^\text{conf}(s', a) = \bot\). From \(f^\text{conf}(s, a) = \top\), it follows that \(\langle \oplus, \text{SELECT}, \alpha \rangle \in \text{permissions}(s, u)\). From sysState(s) \(\sqsubseteq^\text{data}_{M, u}\) sysState(s'), it follows permissions(s, u) = permissions(s', u). From this and \(\langle \oplus, \text{SELECT}, \alpha \rangle \in \text{permissions}(s, u)\), it follows that \(\langle \oplus, \text{SELECT}, \alpha \rangle \in \text{permissions}(s', u)\). This contradicts \(\langle \oplus, \text{SELECT}, \alpha \rangle \notin \text{permissions}(s, u)\). This completes the proof of our claim.

Lemma C.16. Let \(P\) be an extended configuration, \(L\) be the P-LTS, \(r \in \text{traces}(L)\) be a run, \(u\) be a user, \(\gamma\) be a sentence, and \(\Phi\) be a set of sentences such that \(\Phi \models_\text{fin} \gamma\). If, for all \(\phi \in \Phi\), secure\(_{P,u}(r, i \vdash u \phi)\) holds and \([\phi]_{\text{last}(r)}.db = \top\), then secure\(_{P,u}(r, i \vdash u \phi)\) holds and \([\phi]_{\text{last}(r)}.db = \top\).

Proof. Let \(P\) be an extended configuration, \(L\) be the P-LTS, \(r \in \text{traces}(L)\) be a run, \(u\) be a user, \(\gamma\) be a sentence, and \(\Phi\) be a set of sentences such that \(\Phi \models_\text{fin} \gamma\) such that for all \(\phi \in \Phi\), secure\(_{P,u}(r, i \vdash u \phi)\) holds and \([\phi]_{\text{last}(r)}.db = \top\). We now show that secure\(_{P,u}(r, i \vdash u \phi)\) holds and \([\gamma]_{\text{last}(r)}.db = \top\). From \(\Phi \models_\text{fin} \gamma\), \([\phi]_{\text{last}(r)}.db = \top\) for all \(\phi \in \Phi\), and \(\models_\text{fin}\)’s definition, it follows that \([\gamma]_{\text{last}(r)}.db = \top\). Assume, for contradiction’s sake, that secure\(_{P,u}(r, i \vdash u \phi)\) does not hold. From this and \([\gamma]_{\text{last}(r)}.db = \top\), it follows that there is a run \(r' \in \text{traces}(L)\) such that \(r' \equiv_{P,u} r'\) such that \([\gamma]_{\text{last}(r)}.db = \perp\). We claim that for all \(\phi \in \Phi\), \([\phi]_{\text{last}(r)}.db = \top\). From this and \(\Phi \models_\text{fin} \gamma\), it follows that \([\gamma]_{\text{last}(r)}.db = \top\), which contradicts \([\gamma]_{\text{last}(r)}.db = \perp\).

We now prove our claim that for all \(\phi \in \Phi\), \([\phi]_{\text{last}(r)}.db = \top\) for any trace \(r'\) such that \(r' \equiv_{P,u} r'\). From secure\(_{P,u}(r, i \vdash u \phi)\), it follows that \([\phi]_{\text{last}(r)}.db = [\phi]_{\text{last}(r)}.db\). From this and \([\phi]_{\text{last}(r)}.db = \top\), it follows that \([\phi]_{\text{last}(r)}.db = \top\).

Lemma C.17. Let \(M = \langle D, \Gamma \rangle\) be a system configuration, \(a\) be an action in \(\mathcal{A}_{D, M}\), and \(s, s' \in \Omega_M\) be two \(M\)-states such that sysState(s) \(\sqsubseteq_{M, u} \text{sysState}(s')\), invoker(s) = invoker(s'), and trigger(s) = trigger(s'). Then, \(f^\text{user}_{(a, s)}(s, a) = \top\) iff \(f^\text{conf}_{(a, s)}(s', a) = \top\).

Proof. Let \(s = \langle db, U, sec, T, V, c \rangle\) and \(s' = \langle db', U', sec', T', V', c' \rangle\) be two \(M\)-states. We assume that and \(s'\) satisfy sysState(s) \(\sqsubseteq_{M, u} \text{sysState}(s')\), invoker(s) = invoker(s'), and trigger(s) = trigger(s'). We first show that user(a, s) = user(a, s'). Since trigger(s) = trigger(s'), there are two cases:

- \(\text{trigger}(s) = \epsilon\). In this case, the result of user(a, s) depends just on \(a\). Therefore, user(a, s) = user(a, s').
- \(\text{trigger}(s) \neq \epsilon\). In this case, user(a, s) = invoker(s) and user(a, s') = invoker(s'). From invoker(s) = invoker(s'), it follows that user(a, s) = user(a, s').

Let \(u\) be the user user(a, s). From Lemma C.15, it follows that \(f^\text{conf}_{(a, s)}(s, a) = f^\text{conf}_{(a, s)}(s', a)\). This completes the proof.

C.4.6 Equivalence class preservation

We now introduce the concept of an action that preserves the equivalence class induced by an indistinguishability relation \(\cong\). We use this concept in our confidentiality proof (instantiated with the indistinguishability relation \(\cong_{P,u}\)). In the following, given an extended configuration \(P\), a run \(r\), and an indistinguishability relation \(\cong\), we denote by \(\text{traces}(L)\) the equivalence class of \(r\) defined by \(\cong\) over \(\text{traces}(L)\), where \(L\) is the P-LTS.

Definition C.1. Let \(P = \langle M, f \rangle\) be an extended configuration, where \(M = \langle D, \Gamma \rangle\) is a system configuration and \(f\) is an M-PDP, \(L\) be the P-LTS, \(r \in \text{traces}(L)\) be a run, and \(a\) be an action in \(\mathcal{A}_{D, M} \cup \mathcal{T}_{\text{TRIGGER}_D}\). We denote by extend\(_{r}(a),\) where \(r\) is a run and \(a\) is an action, the run \(r' \in \text{traces}(L)\), where \(s \in \Omega_M\) and \(r' = r \cdot a\cdot s\), obtained by executing the action \(a\) at the end of the run \(r\). If there is no such run, then extend\(_{r}(a)\) is undefined. We say that \(a\) preserves the equivalence
class for $r$, $P$, and $\equiv$ iff (1) $\text{extend}(r, a)$ is defined, and (2) there is a bijection $b$ between $\left[ r \right]_{P, \equiv}$ and $\left[ \text{extend}(r, a) \right]_{P, \equiv}$ such that for all $r' \in \left[ r \right]_{P, \equiv}$, $\text{extend}(r', a)$ is defined and $b(r') = \text{extend}(r', a)$.

Furthermore, given a formula $\phi$, we denote by $\text{tables}(\phi)$ the set of relation schemas used in $\phi$, where $\phi'$ is obtained from $\phi$ by replacing all views with their definitions until we reach a fixpoint.

**Lemma C.18.** Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ is a system configuration and $f$ is an $M$-DPD, L be the P-ILTS, $u$ be a user in $U$, $r$ be a run in traces($L$), $a \in A_{D, u}$ be an INSERT or DELETE action $(u, op, R, T)$, $\phi$ be a sentence, $i$ be such that $1 \leq i \leq |r|$, triggers(last($r'_{i}$)) = $\epsilon$, and $r^{i+1} = \text{extend}(r', a)$, and $\equiv$ be an indistinguishability relation. If (1) $a$ preserves the equivalence class for $r$, $P$, and $\equiv$, and (2) the execution of $a$ does not change any table in $\text{tables}(\phi)$ for any run $v \in \left[ r \right]_{P, \equiv}$, then $\text{secure}_{P, \equiv}(r, i+1, u, v, a, \phi)$ holds if $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$ holds.

**Proof.** Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ is a system configuration and $f$ is an $M$-DPD, L be the P-ILTS, $u$ be a user in U, $r$ be a run in traces($L$), $a \in A_{D, u}$ be an INSERT or DELETE action $(u, op, R, T)$, $\phi$ be a sentence, $i$ be such that $1 \leq i \leq |r|$, triggers(last($r'_{i}$)) = $\epsilon$, and $r^{i+1} = \text{extend}(r', a)$. Assume that (1) $a$ preserves the equivalence class for $r'$, $P$, and $\equiv$, and (2) the execution of $a$ does not change any table in $\text{tables}(\phi)$ for any run $v \in \left[ r \right]_{P, \equiv}$. Without loss of generality, assume that $a$ is an INSERT action. In the following, we denote the extend function by $e$. Furthermore, we also denote the fact that $\text{secure}_{P, \equiv}(r, i, u, a, \phi)$ does not hold as $\neg \text{secure}_{P, \equiv}(r, i, u, a, \phi)$.

From Definition C.1 and C.1 preserves the equivalence class for $r'$, $P$, and $\equiv$, it follows that $e(r', a)$ is defined for any $r' \in \left[ r \right]_{P, \equiv}$. Assume, for contradiction’s sake, that our claim does not hold. There are two cases:

1. $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$ holds and $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$ does not hold. From $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$, then it follows that for all $r' \in \left[ r \right]_{P, \equiv}$, $\left[ \phi \right]_{\text{last}(r')}_{a, \phi} = \left[ \phi \right]_{\text{last}(r')}_{a, \phi} \text{db}$ holds for any $r' \in \left[ r \right]_{P, \equiv}$.

2. $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$ and $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$ holds. From $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$, then it follows that for all $r' \in \left[ r \right]_{P, \equiv}$, $\left[ \phi \right]_{\text{last}(r')}_{a, \phi} = \left[ \phi \right]_{\text{last}(r')}_{a, \phi} \text{db}$ holds for any $r' \in \left[ r \right]_{P, \equiv}$.

We prove the claim that $\left[ \phi \right]_{\text{last}(r')}_{a, \phi} = \left[ \phi \right]_{\text{last}(r')}_{a, \phi} \text{db}$ holds for any $r' \in \left[ r \right]_{P, \equiv}$. Assume that this is not the case. This implies that the content of one of the relations that determines $\phi$ is different in last($r'$)$_{a, \phi}$ and last($e(r', a)$)$_{a, \phi}$. This is impossible. Indeed, if $a$’s execution has been successful, i.e., $\text{secEx}(\text{last}(e(r', a))) = \top$ and $\text{Ex}(\text{last}(e(r', a))) = \emptyset$, then $a$’s execution does not change any table in $\text{tables}(\phi)$, and the set of relations that determines $\phi$ is always a subset of $\text{tables}(\phi)$. This leads to a contradiction, and therefore, $\left[ \phi \right]_{\text{last}(r')}_{a, \phi} = \left[ \phi \right]_{\text{last}(r')}_{a, \phi} \text{db}$, and the claim holds trivially.

This completes the proof.

**Lemma C.19.** Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ is a system configuration and $f$ is an $M$-DPD, L be the P-ILTS, $u$ be a user in U, $r$ be a run in traces($L$), $a \in A_{D, u}$ be a SELECT or CREATE action, $\phi$ be a sentence, $\equiv$ be an indistinguishability relation, and $i$ be such that $1 \leq i \leq |r|$, triggers(last($r'_{i}$)) = $\epsilon$, and $r^{i+1} = \text{extend}(r', a)$. If $a$ preserves the equivalence class for $r'$, $P$, and $\equiv$, then $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$ holds if $\text{secure}_{P, \equiv}(r, i+1, u, a, \phi)$ holds.
Proof. The proof of this statement is similar to that of Lemma C.18. □

Lemma C.20. Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ is a system configuration and $f$ is an $M$-PDP, $L$ be the $P$-LTS, $u$ be a user in $U$, $r$ be a run in traces($L$), $a \in A_{D_u}$ be a GRANT or REVOKE action, $\phi$ be a sentence, $\Xi$ be an indistinguishability relation, and $i$ be such that $1 \leq i \leq |r|$, triggers(last($r'$)) = $\epsilon$, and $r'^{i+1} = \text{extend}(r', a)$. If $a$ preserves the equivalence class for $r'$, $P$, and $\Xi$, then secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ holds if secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ holds.

Proof. The proof of this statement is similar to that of Lemma C.18. □

Lemma C.21. Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ is a system configuration and $f$ is an $M$-PDP, $L$ be the $P$-LTS, $u$ be a user in $U$, $r$ be a run in traces($L$), $a \in A_{D_u}$ be a GRANT or REVOKE action, and $\phi$ be a sentence, $\Xi$ be an indistinguishability relation, and $i$ be such that $1 \leq i \leq |r|$, triggers(last($r'$)) = $\epsilon$, and $r'^{i+1} = \text{extend}(r', a)$. If (1) $a$ preserves the equivalence class for $r'$, $P$, and $\Xi$, (2) if $a$’s action is either an INSERT or DELETE, then $t$’s execution does not change any table in tables($\phi$) for any run $v \in [r']_{P,\phi}$, and (3) secExt(last(extend($v, a$))) = $\bot$ and Ext(last(extend($v, a$))) = $\emptyset$ for any run $v \in [r']_{P,\phi}$, then secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ holds if secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ holds.

Proof. Let $P = (M, f)$ be an extended configuration, where $M = (D, \Gamma)$ is a system configuration and $f$ is an $M$-PDP, $L$ be the $P$-LTS, $u$ be a user in $U$, $r$ be a run in traces($L$), $a \in A_{D_u}$ be a GRANT or REVOKE action, and $\phi$ be a sentence, $\Xi$ be an indistinguishability relation, and $i$ be such that $1 \leq i \leq |r|$, triggers(last($r'$)) = $\epsilon$, and $r'^{i+1} = \text{extend}(r', a)$. Assume also that (1) $a$ preserves the equivalence class for $r'$, $P$, and $\Xi$, and (2) secExt(last(extend($r', a$))) = $\bot$ and Ext(last(extend($r', a$))) = $\emptyset$. In the following, we denote the extend function by $e$. Furthermore, we also denote the fact that secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ does not hold as secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$. From Definition C.1 and the fact that $a$ preserves the equivalence class for $r'$, $P$, and $\Xi$, it follows that $e(r', a)$ is defined for any $r' \in [r']_{P,\phi}$. Assume, for contradiction’s sake, that our claim does not hold. There are two cases:

- secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ holds and secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ does not hold. From secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$, it follows that $[\phi]_{\text{last}(r')}_{db}$ = $[\phi]_{\text{last}(r')_{db}}$ for any $r' \in [r']_{P,\phi}$. We claim that $[\phi]_{\text{last}(r')}_{db}$ = $[\phi]_{\text{last}(r')_{db}}$ holds for any $r' \in [r']_{P,\phi}$. From secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$, it follows that there is a $r'' \in [r']_{P,\phi}$ such that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r'')_{db}}$. Let $b$ be a command such that $[\phi]_{\text{last}(r''}_{db}}$ holds for any $r' \in [r']_{P,\phi}$, and the fact that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r''_{db})}$, it follows that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r'')_{db}}$. From this, $[\phi]_{\text{last}(r'')_{db}} = [\phi]_{\text{last}(r'')_{db}}$ holds for any $r' \in [r']_{P,\phi}$, and the fact that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r''_{db})}$, it follows that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r'')_{db}}$. This contradicts the fact that $[\phi]_{\text{last}(r'')_{db}} = [\phi]_{\text{last}(r'')_{db}}$ for any $r' \in [r']_{P,\phi}$.

Now we prove that $[\phi]_{\text{last}(r'')_{db}} = [\phi]_{\text{last}(r'')_{db}}$ holds for any $r' \in [r']_{P,\phi}$. Assume, for contradiction’s sake, that there is a run $r' \in [r']_{P,\phi}$ such that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r'')_{db}}$. Since executing the trigger does not throw security or integrity exceptions in any run $r' \in [r']_{P,\phi}$, there are three cases:

- the trigger $a$ is not enabled in $e(r', a)$. From this and the LTS semantics, it follows that last($r'$) = last($e(r', a)$). From this, it therefore follows that $[\phi]_{\text{last}(r')_{db}} = [\phi]_{\text{last}(e(r', a))_{db}}$. This contradicts our assumption.

- the trigger $a$ is enabled in $e(r', a)$ and its action is a GRANT or a REVOKE command. From this and the LTS semantics, it therefore follows that last($r'$) = last($e(r', a)$). From this, it thus follows that $[\phi]_{\text{last}(r')_{db}} = [\phi]_{\text{last}(e(r', a))_{db}}$. This contradicts our assumption.

- the trigger $a$ is enabled in $e(r', a)$ and its action is an INSERT or DELETE command. Thus, from $[\phi]_{\text{last}(r')_{db}} = [\phi]_{\text{last}(e(r', a))_{db}}$, it follows that the content of one of the relations that determines $\phi$ is different in last($r'$) and last($e(r', a)$). This contradicts the fact that $a$’s execution does not change the tables in tables($\phi$) for any run $r' \in [r']_{P,\phi}$.

- secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ holds and secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$ does not hold. We have already shown that $[\phi]_{\text{last}(r')_{db}} = [\phi]_{\text{last}(e(r', a))_{db}}$ holds for any $r' \in [r']_{P,\phi}$. From secure$_{P,\phi}(r, i + 1 \mapsto u, \phi)$, it follows that there is $r'' \in [r']_{P,\phi}$ such that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r'')_{db}}$. Let $b$ be a command such that $[\phi]_{\text{last}(r'')_{db}}$ holds for any $r' \in [r']_{P,\phi}$, it follows that $[\phi]_{\text{last}(r'')_{db}} \neq [\phi]_{\text{last}(r'')_{db}}$. From this, $[\phi]_{\text{last}(r'')_{db}} = [\phi]_{\text{last}(r'')_{db}}$ holds for any $r' \in [r']_{P,\phi}$, it follows that $[\phi]_{\text{last}(r'')_{db}} = [\phi]_{\text{last}(r'')_{db}}$. This completes the proof. □
C.4. Data Confidentiality Proofs

C.4.7 Auxiliary results about getInfoV and getInfoS

Proposition C.10 states that the getInfoS and getInfoV functions correctly capture the information an attacker may learn by an integrity exception (or its absence).

Proposition C.10. Let \( P = (M, f) \) be an extended configuration, where \( M = (D, \Gamma) \) is a system configuration and \( f \) is an \( M\text{-PDP} \). Let be the P-LTS, \( a \in A_{D,u} \) be an INSERT or DELETE action, and \( r \) be a run such that (1) extend\((r,a)\) is defined and (2) seeEx(\(\text{last}(\text{extend}(r,a))\))) = \(\perp\). For any constraint \( \gamma \) in \( \text{Dep}(\Gamma,a) \), the following statements hold:

- \([\text{getInfoS}(\gamma,a)]^{\text{last}(r,db)} = \top \iff \gamma \not\in \text{Ex}(\text{last}(\text{extend}(r,a)))\),
- \([\text{getInfoV}(\gamma,a)]^{\text{last}(r,db)} = \top \iff \gamma \in \text{Ex}(\text{last}(\text{extend}(r,a)))\).

Proof. Let \( P = (M, f) \) be an extended configuration, where \( M = (D, \Gamma) \) is a system configuration and \( f \) is an \( M\text{-PDP} \). Let be the P-LTS, \( a \in A_{D,u} \) be an INSERT or DELETE action, and \( r \) be a run such that (1) extend\((r,a)\) is defined and (2) seeEx(\(\text{last}(\text{extend}(r,a))\))) = \(\perp\). Furthermore, let \( \gamma \) be a constraint in \( \text{Dep}(\Gamma,a) \). We first note that \(\text{getInfoS}(\gamma,a) = \lnot\text{getInfoV}(\gamma,a)\). From this, it follows trivially that proving one of the two claims is enough. We thus prove that \([\text{getInfoS}(\gamma,a)]^{\text{last}(r,db)} = \top \iff \gamma \not\in \text{Ex}(\text{last}(\text{extend}(r,a)))\). There are two cases:

1. \( a = (u, \text{INSERT}, R, T) \). There are two cases depending on \( \gamma \):
   - (a) \( \gamma \) is of the form \( \forall \exists !, \exists !, \exists !, \exists !, (\exists R (\exists !, \exists !, \exists !)) \Rightarrow \exists ! = \exists ! \). Let \( T \) be \((\exists !, \exists !, \exists !), db \) be the state \(\text{last}(r,db)\), and \( db^\prime \) be the state \(db[R \uplus T] \).
     - (\(\Rightarrow\)). Assume that \([\text{getInfoS}(\gamma,a)]^{\text{last}(r,db)} = \top \). From this and \(\text{getInfoS}(\gamma,a)\)’s definition, it follows that for all tuples \((\exists !, \exists !, \exists !) \in db(R)\), then \(\exists ! = \exists !\). From a’s definition and the LTS semantics, it follows that \(db[R] = db[R] \cup (\exists !, \exists !, \exists !)\). From this and the fact that for all tuples \((\exists !, \exists !, \exists !) \in db(R)\), then \(\exists ! = \exists !\), it follows that \(db[R] \cup (\exists !, \exists !, \exists !) \in db(R)\), then \(\exists ! = \exists !\). Furthermore, since \(db \in \Omega^\text{LTS}_R\), it follows that for all tuples \((\exists !, \exists !, \exists !), (\exists !, \exists !, \exists !), (\exists !, \exists !, \exists !) \in db(R)\), if \(\exists ! = \exists !\) and \(\exists ! = \exists !\), then \(\exists ! = \exists !\). Therefore, it follows that for all tuples \((\exists !, \exists !, \exists !), (\exists !, \exists !, \exists !) \in db(R)\), if \(\exists ! = \exists !\), then \(\exists ! = \exists !\). Therefore, \([\gamma]^{db} = \top \). From this and the LTS semantics, it follows that \([\gamma]^{db} = \top \).
     - (\(\Leftarrow\)). Assume that \(\exists ! = \exists !\). From this and the LTS semantics, it follows that \(\exists ! = \exists !\). Therefore, for any two tuples \((\exists !, \exists !, \exists !) \in db(R)\), if \(\exists ! = \exists !\), then \(\exists ! = \exists !\). Assume, for contradiction’s sake, that \([\text{getInfoS}(\gamma,a)]^{db} = \perp\). This means that there is a tuple \((\exists !, \exists !, \exists !) \in db(R)\) such that \(\exists ! = \exists !\). From \(db[R] = db[R] \cup (\exists !, \exists !, \exists !)\), it follows that for all tuples \((\exists !, \exists !, \exists !) \in db(R)\), if \(\exists ! = \exists !\), then \(\exists ! = \exists !\). Therefore, \([\gamma]^{db} = \top \). This is in contradiction with \([\gamma]^{db} = \top \).

(b) \( \gamma \) is of the form \( \forall \exists !, \exists !, R (\exists !, \exists !) \Rightarrow \exists !, S(\exists !, \exists !) \). Let \( T \) be \((\exists !, \exists !, \exists !), db \) be the state \(\text{last}(r,db)\), and \( db^\prime \) be the state \(db[R \uplus T] \).
   - (\(\Rightarrow\)). Assume that \([\text{getInfoS}(\gamma,a)]^{db} = \top \). From this and \(\text{getInfoS}(\gamma,a)\)’s definition, it follows that there is a tuple \((\exists !, \exists !) \in db(S)\). From a’s definition and the LTS semantics, it follows that \(db[S] = db[S]\). From this, it follows that there is a tuple \((\exists !, \exists !) \in db(S)\). Furthermore, since \(db \in \Omega^\text{LTS}_R\), it follows that for all tuples \((\exists !, \exists !) \in db(R)\), if \(\exists ! = \exists !\), then \(\exists ! = \exists !\). Assume, for contradiction’s sake, that \([\text{getInfoS}(\gamma,a)]^{db} = \perp\). This means that for any tuple \((\exists !, \exists !) \in db(S)\), \(\exists ! = \exists !\). From \(db[S] = db[S]\), it follows that for any tuple \((\exists !, \exists !) \in db(S)\), \(\exists ! = \exists !\). From \(db[R] = db[R] \cup (\exists !, \exists !)\), it follows that for all tuples \((\exists !, \exists !) \in db(R)\) such that \(\exists ! = \exists !\), there is no tuple \((\exists !, \exists !) \in db(S)\). From this and the relational calculus semantics, it follows that \([\gamma]^{db} = \perp\). This is in contradiction with \([\gamma]^{db} = \top\).

2. \( a = (u, \text{DELETE}, R, T) \). In this case, \( \gamma \) is of the form \( \forall \exists !, \exists !, S(\exists !, \exists !) \Rightarrow \exists !, R(\exists !, \exists !) \). Let \( T \) be \((\exists !, \exists !, \exists !), db \) be the state \(\text{last}(r,db)\), and \( db^\prime \) be the state \(db[R \uplus T] \).
   - (\(\Rightarrow\)). Assume that \([\text{getInfoS}(\gamma,a)]^{db} = \top \). From this and \(\text{getInfoS}(\gamma,a)\)’s definition, it follows that either there is no tuple \((\exists !, \exists !) \in db(S)\) or there is a tuple \((\exists !, \exists !) \in db(R)\) such that \(\exists ! = \exists !\). There are two cases:
     - (a) there is no tuple \((\exists !, \exists !) \in db(S)\). From this, a’s definition, and the LTS semantics, it follows that there is no tuple \((\exists !, \exists !) \in db(S)\). From \(db \in \Omega^\text{LTS}_R\), it follows that for all tuples \((\exists !, \exists !) \in db(S)\) such that \(\exists ! = \exists !\), there is a tuple \((\exists !, \exists !) \in db(R)\). From this, \(db[R] = db[R] \setminus \{(\exists !, \exists !)\}, db(S) = db(S)\), and there is no tuple \((\exists !, \exists !) \in db(R)\), it follows that for all tuples
We prove our three claims by contradiction.

(b) there is a tuple \((\tau, \mu')\) in \(db(R)\) such that \(\mu' \neq \mu\). From this, \(a\)'s definition, and the LTS semantics, it follows that there is a tuple \((\tau, \mu)\) in \(db'(R)\) such that \(\mu \neq \mu'\). From \(db \in \Omega^B\), it follows that for all tuples \((\tau, \mu')\) in \(db(S)\) such that \(\mu' \neq \mu\), there is a tuple \((\tau, \mu'')\) in \(db(R)\). From this, \(db'(R) = db(R) \setminus \{(\tau, \mu')\}\), \(db'(S) = db(S)\), and there is a tuple \((\tau, \mu')\) in \(db'(R)\) such that \(\mu' \neq \mu\). Therefore, \(\gamma[db'] = \top\). From this and the LTS semantics, it follows that \(\gamma[db] \neq \top\). This is in contradiction with \(\gamma[db] = \top\).

This completes the proof.

Proposition C.11 extends Proposition C.10 to triggers.

Proposition C.11. Let \(P = (M, f)\) be an extended configuration, where \(M = (D, \Gamma)\) is a system configuration and \(f\) is an \(M\)-PDP. \(L\) be the L-PDS, \(t \in T \hat{R}IGGER_{\beta}\) is a trigger whose action is an INSERT or DELETE command, and \(r\) be a run such that (1) \(extend(r, t)\) is defined, (2) the trigger \(t\) is enabled, and (3) \(secEx(extend(r, t)) = \top\). Furthermore, we denote by \(a\) the actual action performed by the trigger \(t\). For any constraint \(\gamma\) in \(Dep(\Gamma, a)\), the following conditions hold:

- \([getInfoS(\gamma, a)]_{last(\cdot) db} = \top\) iff \(\gamma \notin Ex(extend(r, t))\),
- \([getInfoV(\gamma, a)]_{last(\cdot) db} = \top\) iff \(\gamma \in Ex(extend(r, t))\).

Proof. The proof of this statement is almost identical to that of Proposition C.10.

C.4.8 Auxiliary results about \(f\)

Lemma C.22 states that the PDP \(f\) returns the same result in any two data-indistinguishable states.

Lemma C.22. Let \(M = (D, \Gamma)\) be a system configuration, \(s, s' \in \Omega_M\) be two \(M\)-states such that \(sysState(s) \approx_{data} sysState(s')\), \(tuple(s) = tuple(s')\), \(invoker(s) = invoker(s')\), and \(trigger(s) = trigger(s')\), and \(f\) be the PDP as above. The following conditions hold:

1. If \(trigger(s) = \epsilon\), then \(f(s, \alpha) = \top\) iff \(f(s', \alpha) = \top\) for any action \(\alpha\) in \(A_{D, U}\).
2. If \(trigger(s) \in T \hat{R}IGGER_{\beta}\), then \(f(s, trigCond(s)) = \top\) iff \(f(s', trigCond(s)) = \top\).
3. If \(trigger(s) \in T \hat{R}IGGER_{\beta}\), \(trigCond(s) = (\langle a, SELECT, \psi \rangle, \sigma)\), and \(\psi'^{a db} = \psi'^{a db} = \top\), then \(f(s, trigAct(s)) = \top\) iff \(f(s', trigAct(s)) = \top\).

Proof. We prove our three claims by contradiction.

1. Assume, for contradiction's sake, that there are two states \(s\) and \(s'\) and an action \(\alpha\) such that \(trigger(s) = \epsilon, sysState(s) \approx_{data} sysState(s')\), \(tuple(s) = tuple(s')\), \(invoker(s) = invoker(s')\), and \(trigger(s) = trigger(s')\), and \(f\) be the PDP as above. From \(f\)'s definition, \(f(s, \alpha) = \top\), \(f(s', \alpha) = \top\), and Lemma C.17, it follows that \(f_{int}(s, \alpha) = \top\), \(f_{int}(s', \alpha) = \top\), and \(f_{user}(s, \alpha) = f_{user}(s', \alpha) = \top\). From \(f_{int}(s, \alpha) = \top\), it follows that \(\psi' \not\approx_{auth} a\). From \(f_{int}(s', \alpha) = \top\), it follows \(\psi' \not\approx_{auth} a\). From this, \(\psi' \not\approx_{auth} a\), which contradicts \(\psi' \not\approx_{auth} a\). This completes the proof for the first claim.

2. Assume, for contradiction's sake, that there are two states \(s\) and \(s'\) such that \(trigger(s) = \epsilon, sysState(s) \approx_{data} sysState(s')\), \(tuple(s) = tuple(s')\), \(invoker(s) = invoker(s')\), and \(trigger(s) = trigger(s')\), and \(f\) be the PDP as above. From \(f\)'s definition, \(f(s, \alpha) = \top\), \(f(s', \alpha) = \top\), and \(f_{user}(s, \alpha) = f_{user}(s', \alpha) = \top\), and \(f_{conf}(s, \alpha) = f_{conf}(s', \alpha) = \top\). From \(f_{int}\)'s definition, \(trigger(s) = \epsilon\), and \(a = \top\), \(f_{int}(s, \alpha) = \top\), which contradicts \(f_{int}(s', \alpha) = \top\). This completes the proof for the second claim.

3. Assume, for contradiction's sake, that there are two states \(s\) and \(s'\) such that \(trigger(s) = \epsilon, sysState(s) \approx_{data} sysState(s')\), \(\psi'^{a db} = \psi'^{a db} = \top\), \(f(s, \alpha) = \top\), \(f(s', \alpha) = \top\), and \(f_{int}(s, \alpha) = f_{int}(s', \alpha) = \top\), and \(f_{user}(s, \alpha) = f_{user}(s', \alpha) = \top\), and \(f_{conf}(s, \alpha) = f_{conf}(s', \alpha) = \top\). From \(f_{int}\)'s definition, \(f(s, \alpha) = \top\), \(f(s', \alpha) = \top\), and \(f_{int}(s, \alpha) = f_{int}(s', \alpha) = \top\), which contradicts \(f_{int}(s, \alpha) = \top\). This completes the proof for the third claim.
f_{act}(s,a) = \top, and f_{act}(s',a) = \bot. From this, it follows that s' \not\approx_{\text{with}} t. From f_{act}(s,a) = \top, it follows s \approx_{\text{with}} t. There are two cases depending on t’s security mode:
(a) mode(t) = A. From this and \( s \approx_{\text{with}} t \), it follows that \( s' \approx_{\text{with}} a \) and \( s' \approx_{\text{with}} a' \), where \( a' = \text{action}(\text{statement}(t), \text{owner}(t), \text{tuple}(s)) \) is the trigger’s action associated with the trigger’s owner. Note that s and s’ are data indistinguishable. From this, \( a, a' \in A_{D,U} \) and Lemma C.1, it follows that \( s' \approx_{\text{with}} a \) and \( s' \approx_{\text{with}} a' \). From \( s' \approx_{\text{with}} a \), \( s' \approx_{\text{with}} a' \), \( [v]s'.\text{db} = \top \), and the rule \texttt{EXECUTE TRIGGER-2}, it follows that \( s' \approx_{\text{with}} t \), which contradicts \( s' \not\approx_{\text{with}} t \).
(b) mode(t) = O. From this and \( s \approx_{\text{with}} t \), it follows that \( s \approx_{\text{with}} a \). Note that s and s’ are data indistinguishable. From this, \( a, a' \in A_{D,U} \), and Lemma C.1, it follows that \( s' \approx_{\text{with}} a \). From this, \( [v]s'.\text{db} = \top \), and the rule \texttt{EXECUTE TRIGGER-1}, it follows that \( s' \approx_{\text{with}} t \), which contradicts \( s' \not\approx_{\text{with}} t \).

This completes the proof for the third claim.

This completes the proof.

\[ \square \]

C.4.9 \( f \) preserves the equivalence class

In Lemmas C.23 and C.24, we prove that, whenever we use \( f \) as PDP, actions and triggers preserve the equivalence class.

Lemma C.23. Let \( u \) be a user in \( U \), \( P = (M, f) \) be an extended configuration, where \( M = (D, \Gamma) \) is a system configuration and \( f \) is the PDP from Section 6.8, and \( L \) be the P-LTS. For any run \( r \in \text{traces} \) and any action \( a \in A_{D,U} \), if \( \text{extend}(r,a) \) is defined, then \( a \) preserves the equivalence class for \( r, P, \text{and } \equiv_{P,u} \).

Proof. Let \( u \) be a user in \( U \), \( P = (M, f) \) be an extended configuration, where \( M = (D, \Gamma) \) is a system configuration and \( f \) is the PDP from Section 6.8, and \( L \) be the P-LTS. In the following, we use \( e \) to refer to the extend function. We prove our claim by contradiction. Assume, for contradiction’s sake, that there is a run \( r \in \text{traces} \) and an action \( a \in A_{D,U} \) such that \( e(r,a) \) is defined and \( a \) does not preserve the equivalence class for \( r, P, \text{and } \equiv_{P,u} \). According to the LTS semantics, the fact that \( e(r,a) \) is defined implies that \( \text{triggers}(last(r)) = \epsilon \). Therefore, \( \text{triggers}(last(r')) = \epsilon \) holds as well for any \( r' \in [r]_{P,u} \) (because \( r \) and \( r' \) are indistinguishable and, therefore, their projections are consistent), and, thus, \( e(r',a) \) is defined as well for any \( r' \in [r]_{P,u} \). There are a number of cases depending on \( a \):

1. \( a = \langle u, \text{SELECT}, q \rangle \). There are two cases:
   (a) \( \text{select(last(e(r,a)))} = \bot \). From the LTS rules and \( \text{select(last(e(r,a)))} = \bot \), it follows that \( f(last(r),a) = \top \). From this and Lemma C.22, it follows that \( f(last(s'),a) = \top \) for any \( r' \in [r]_{P,u} \). From this and the LTS rules, it follows \( \text{select(last(e(r',a)))} = \bot \) for any \( r' \in [r]_{P,u} \). From \( f(last(r'),a) = \top \) for any \( r' \in [r]_{P,u} \), it follows \( f_{\text{conf}}(last(r'),a) = \top \) for any \( r' \in [r]_{P,u} \). Note that \( \text{user(last(r'))} = u \) for any \( r' \in [r]_{P,u} \) because \( \text{trigger(last(r'))} = \epsilon \) and \( u \in A_{D,U} \). From this, \( f_{\text{conf}}(last(r'),a) = \top \) for any \( r' \in [r]_{P,u} \), and \( f_{\text{conf}} \)’s definition, it follows that \( \text{secure}(a,q,\text{last(r'))} = \top \) for any \( r' \in [r]_{P,u} \). From this and Proposition C.9, it follows that \( [q]^{\text{last(r'))} \text{db} = [q]^{\text{last(r'))} \text{db} \) for all \( r' \in [r]_{P,u} \). Furthermore, it follows trivially from the LTS rule \texttt{SELECT Success}, that the state after \( a \)’s execution is data indistinguishable from \( \text{last}(r) \). It is also easy to see that \( e(r',a) \) is well-defined for any \( r' \in [r]_{P,u} \). From the considerations above and \( r' \in [r]_{P,u} \), it follows trivially that \( e(r',a) \in [e(r,a)]_{P,u} \). The bijection \( b \) is trivially \( b(r') = e(r',a) \). This leads to a contradiction.
   (b) \( \text{select(last(e(r,a)))} = \top \). From the LTS rules and \( \text{select(last(e(r,a)))} = \top \), it follows that \( f(last(r),a) = \bot \). From this and Lemma C.22, it follows that \( f(last(r'),a) = \bot \) for any \( r' \in [r]_{P,u} \). From this and the LTS rules, it follows \( \text{select(last(e(r,a)))} = \top \) for any \( r' \in [r]_{P,u} \). The data indistinguishability between \( \text{last}(e(r',a)) \) and \( \text{last}(e(r,a)) \) follows trivially from the data indistinguishability between \( \text{last}(r') \) and \( \text{last}(r) \). For any run \( r' \in [r]_{P,u} \), there is exactly one run \( e(r',a) \). From the considerations above, it follows trivially that \( e(r',a) \in [e(r,a)]_{P,u} \). The bijection \( b \) is trivially \( b(r') = e(r',a) \). This leads to a contradiction.

Both cases lead to a contradiction. This completes the proof for \( a = \langle u, \text{SELECT}, q \rangle \).

2. \( a = \langle u, \text{INSERT}, R, T \rangle \). In the following, we denote by \( gl \) the function \text{getInfo}, by \( gS \) the function \text{getInfoS}, and by \( gV \) the function \text{getInfoV}. There are three cases:
   (a) \( \text{select(last(e(r,a)))} = \bot \) and \( \text{select(last(e(r,a)))} = \emptyset \). From the LTS rules and \( \text{select(last(e(r,a)))} = \bot \), it follows that \( f(last(r),a) = \top \). From this and Lemma C.22, it follows that \( f(last(r'),a) = \top \) for any \( r' \in [r]_{P,u} \). From this and the LTS rules, it follows that \( \text{select(last(e(r',a)))} = \bot \) for any \( r' \in [r]_{P,u} \). From \( f(last(r'),a) = \top \), it follows
that $f_{\text{user}}(\text{last}(r), a) = \top$ because $\text{user}(\text{last}(r), a) = u$ since $\text{trigger}(\text{last}(r), a) = \epsilon$ and $a \in A_{DB}$. From this and $f_{\text{user}}$’s definition, it follows that $\text{secure}(u, gS(\gamma, a), \text{act}) = \top$ for any integrity constraint $\gamma$ in $\text{Dep}(\Gamma, a)$. From $\text{exec}(\text{last}(e(r, a))) = \emptyset$ and Proposition C.10, it follows that $[gS(\gamma, a)]^\text{last}(r'), db = \top$. From this, $\text{secure}(u, gS(\gamma, a), \text{act})$, and Proposition C.9, it follows that $[gS(\gamma, a)]^\text{last}(r'), db = \top$ for any $r' \in [r]_{P,u}$. From this and Proposition C.10, it follows that $\text{exec}(\text{last}(e(r', a))) = \emptyset$ for any $r' \in [r]_{P,u}$. We claim that, for any $r' \in [r]_{P,u}$, $\text{last}(e(r, a))$ and $\text{last}(e(r', a))$ are data indistinguishable. From this and the above considerations, it follows trivially that $e(r', a) \in [e(r, a)]_{P,u}$. The bijection $b$ is trivially $b(r') = e(r', a)$. This leads to a contradiction.

We now prove our claim that for any $r' \in [r]_{P,u}$, $\text{last}(e(r, a))$ and $\text{last}(e(r', a))$ are data indistinguishable. We prove the claim by contradiction. Let $s_2 = \langle db_2, U_2, sec_2, T_2, V_2 \rangle$ be $\text{sysState}(\text{last}(e(r, a)))$, $s'_2 = \langle db_2', U_2', sec_2', T_2', V_2' \rangle$ be $\text{sysState}(\text{last}(e(r', a)))$, $s_1 = \langle db_1, U_1, sec_1, T_1, V_1 \rangle$ be $\text{sysState}(\text{last}(r))$, and $s'_1 = \langle db_1', U_1', sec_1', T_1', V_1' \rangle$ be $\text{sysState}(\text{last}(r'))$. In the following, we denote the permissions function by $p$. Furthermore, note that $s_1$ and $s'_1$ are data-indistinguishable because $r' \in [r]_{P,u}$. There are a number of cases:

i. $U_2 \neq U_2'$. Since $a$ is an INSERT operation, it follows that $U_1 = U_2$ and $U_2' = U_2'$. Furthermore, from $s_1 \models_{\text{data}, a} s'_1$, it follows that $U_1 = U_1'$. Therefore, $U_2 = U_2'$ leading to a contradiction.

ii. $sec_2 \neq sec_2'$. The proof is similar to the case $U_2 \neq U_2'$.

iii. $T_2 \neq T_2'$. The proof is similar to the case $U_2 \neq U_2'$.

iv. $V_2 \neq V_2'$. The proof is similar to the case $U_2 \neq U_2'$.

v. there is a table $R'$ for which $\langle a, \text{SELECT}, R \rangle \in p(s_2, u)$ and $db_2(R') \neq db_2'(R')$. Note that $p(s_2, u) = p(s_1, u)$. There are two cases:

- $R \equiv R'$. From $s_1 \models_{\text{data}, a} s'_1$ and $\langle a, \text{SELECT}, R \rangle \in p(s_2, u)$, it follows that $db_2(R') = db_2'(R')$. From this and the fact that $a$ has been executed successfully both in $e(r, a)$ and $e(r', a)$, it follows that $db_2(R') = db_2'(R') \cup \{T\}$ and $db_2'(R') = db_2'(R') = db_2'(R') \cup \{T\}$. From this and $db_1(R') = db_1'(R')$, it follows that $db_2(R') = db_2'(R')$ leading to a contradiction.

- $R \neq R'$. From $s_1 \models_{\text{data}, a} s'_1$ and $\langle a, \text{SELECT}, R \rangle \in p(s_2, u)$, it follows that $db_2(R') = db_2'(R')$. From this and the fact that $a$ does not modify $R'$, it follows that $db_2(R') = db_2'(R')$ and $db_2'(R') = db_2'(R')$. From this and $db_1(R') = db_1'(R')$, it follows that $db_2(R') = db_2'(R')$ leading to a contradiction.

vi. there is a view $v$ for which $\langle a, \text{SELECT}, v \rangle \in p(s_2, u)$ and $db_2(v) \neq db_2'(v)$. Note that $p(s_2, u) = p(s_1, u)$. Since $a$ has been successfully executed in both states, we know that $\text{leak}(s_1, a, u)$ holds. There are two cases:

- $R \not\in \text{Det}(v, s, M)$. Then, $v(s_1) = v(s_2)$ and $v(s'_1) = v(s'_2)$ (because $R$’s content does not determine $v$’s materialization). From $s_1 \models_{\text{data}, a} s'_1$, it follows that $v(db_2(v)) = v(db_2'(v))$ leading to a contradiction.

- $R \in \text{Det}(v, s, M)$ and for all $o \in \text{tDet}(v, s, M) < a, \text{SELECT}, o \in p(s_1, u)$. From this and $s_1 \models_{\text{data}, a} s'_1$, it follows that, for all $o \in \text{tDet}(v, s, M)$, $o(s_1) = o(s'_1)$. If $o(\neq R)$, $o(s_1) = o(s'_1) = o(s_2) = o(s'_2)$. From $\langle a, \text{SELECT}, R \rangle \in p(s_2, u)$ and $s_1 \models_{\text{data}, a} s'_1$, it follows that $db_1(R) = db_1'(R)$. From this and the fact that $a$ has been executed successfully both in $e(r, a)$ and $e(r', a)$, it follows that $db_2(R) = db_2'(R) \cup \{T\}$ and $db_2'(R) = db_2'(R) \cup \{T\}$. From this and $db_1(R) = db_1'(R)$, it follows that $db_2(R) = db_2'(R)$. From this and for all $o \in \text{tDet}(v, s, M)$ such that $o(\neq R)$, $o(s_2) = o(s'_2)$; it follows that for all $o \in \text{tDet}(v, s, M)$, $o(s_2) = o(s'_2)$. Since the content of all tables determining $v$ is the same in $s_2$ and $s'_2$, it follows that $db_2(v) = db_2'(v)$ leading to a contradiction.

All the cases lead to a contradiction. (b) $\text{secExec}(\text{last}(e(r, a))) = \bot$ and $\text{exec}(\text{last}(e(r, a))) = \emptyset$. From the LTS rules and $\text{secExec}(\text{last}(e(r, a))) = \bot$ and $\text{exec}(\text{last}(e(r, a))) = \emptyset$. From this and Lemma C.22, it follows that $f_{\text{user}}(\text{last}(r), a) = \top$ for any $r' \in [r]_{P,u}$. From this and the LTS rules, it follows that $\text{exec}(\text{last}(e(r', a))) = \bot$ for any $r' \in [r]_{P,u}$. Assume that the exception has been caused by the constraint $\gamma$, i.e., $\gamma \in \text{exec}(\text{last}(e(r, a)))$. From this and Proposition C.10, it follows that $gV(\gamma, a)$ holds in $\text{last}(r), db$. From $f_{\text{user}}(\text{last}(r), a) = \top$ and $f_{\text{user}}$’s definition, it follows that $f_{\text{user}}(\text{last}(r), a) = \top$ because $\text{user}(\text{last}(r), a) = u$ since $\text{trigger}(\text{last}(r)) = \epsilon$ and $a \in A_{DB}$. From this and $f_{\text{user}}$’s definition, it follows that $\text{secure}(u, gV(\gamma, a), \text{act})$ holds. From this, Proposition C.9, and $[gV(\gamma, a)]^\text{last}(r'), db = \top$, it follows that also $[gV(\gamma, \text{act})]^\text{last}(r'), db = \top$ for any $r' \in [r]_{P,u}$. From this and Proposition C.10, it follows that $\gamma \in \text{exec}(\text{last}(e(r', a)))$ for any $r' \in [r]_{P,u}$. The data indistinguishability between $\text{last}(e(r, a))$ and $\text{last}(e(r', a))$ follows trivially from the data indistinguishability between $\text{last}(r)$ and $\text{last}(r')$ for any $r' \in [r]_{P,u}$. Therefore, for any run $r' \in [r]_{P,u}$, there is exactly one run $e(r', a)$. From
the considerations above, it follows trivially that \( e(r', a) \in [e(r, a)]_{P,u} \). The bijection \( b \) is trivially \( b(r') = e(r', a) \). This leads to a contradiction.

(c) \( \text{secEx}([\text{last}(e(r, a))]) = T \). From the LTS rules and \( \text{secEx}([\text{last}(e(r, a))]) = T \), it follows that \( f([\text{last}(r), a]) = \bot \). From this and Lemma C.22, it follows that \( f([\text{last}(r'), a]) = \bot \) for any \( r' \in [r]_{P,u} \). From this and the LTS rules, it follows \( \text{secEx}([\text{last}(e(r', a))]) = T \) for any \( r' \in [r]_{P,u} \). The data indistinguishability between \( e(r', a) \) and \( e(r', a) \) follows trivially from that between \( \text{last}(r) \) and \( \text{last}(r') \) for any \( r' \in [r]_{P,u} \). Therefore, for any run \( r' \in [r]_{P,u} \), there is exactly one run \( e(r', a) \). From the considerations above, it follows trivially that \( e(r', a) \in [e(r, a)]_{P,u} \). The bijection \( b \) is trivially \( b(r') = e(r', a) \). This leads to a contradiction.

All cases lead to a contradiction. This completes the proof for \( a = (u, \text{INSERT}, R, T) \).

3. \( a = (u, \text{DELETE}, R, T) \). The proof is similar to that for \( a = (u, \text{INSERT}, R, T) \).

4. \( a = (\text{⊕}, u', p, u) \). There are two cases:
   (a) \( \text{secEx}([\text{last}(e(r, a))]) = \bot \). We assume that \( p = \langle \text{SELECT}, O \rangle \) for some \( O \in D \cup V \). If this is not the case, the proof is trivial. Furthermore, we also assume that \( u' \neq u \); otherwise the proof is again, trivial since the new permission does not influence \( u \)’s permissions. From the LTS rules and \( \text{secEx}([\text{last}(e(r, a))]) = \bot \), it follows that \( f([\text{last}(r), a]) = \top \). From this and Lemma C.22, it follows that \( f([\text{last}(r'), a]) = \top \) for any \( r' \in [r]_{P,u} \). From this and the LTS rules, it follows \( \text{secEx}([\text{last}(e(r', a))]) = \bot \) for any \( r' \in [r]_{P,u} \). From secEx([last(e(r', a))]) = ⊥, we have \( f_{\text{new}} \)’s definition, it follows that \( \text{last}(r').sec = \text{last}(e(r', a)).sec \). Therefore, since \( \text{last}(r) \) and \( \text{last}(r') \) are data indistinguishable, for any \( r' \in [r]_{P,u} \), then also \( e(r, a) \) and \( e(r', a) \) are data indistinguishable. Therefore, for any run \( r' \in [r]_{P,u} \), there is exactly one run \( e(r', a) \). From the considerations above, it follows trivially that \( e(r', a) \in [e(r, a)]_{P,u} \). The bijection \( b \) is trivially \( b(r') = e(r', a) \). This leads to a contradiction.

(b) \( \text{secEx}([\text{last}(e(r, a))]) = \top \). From the LTS rules and \( \text{secEx}([\text{last}(e(r, a))]) = \top \), it follows that \( f([\text{last}(r), a]) = \bot \). From this and Lemma C.22, it follows that \( f([\text{last}(r'), a]) = \bot \) for any \( r' \in [r]_{P,u} \). From this and the LTS rules, it follows \( \text{secEx}([\text{last}(e(r', a))]) = \bot \) for any \( r' \in [r]_{P,u} \). The data indistinguishability between \( e(r', a) \) and \( e(r', a) \) follows trivially from the data indistinguishability between \( \text{last}(r') \) and \( \text{last}(r) \). Therefore, for any run \( r' \in [r]_{P,u} \), there is exactly one run \( e(r', a) \). From the considerations above, it follows trivially that \( e(r', a) \in [e(r, a)]_{P,u} \). The bijection \( b \) is trivially \( b(r') = e(r', a) \). This leads to a contradiction.

Both cases lead to a contradiction. This completes the proof for \( a = (\text{⊕}, u', p, u) \).

5. \( a = (\text{⊕}, u', p, u) \). The proof is similar to that for \( a = (\text{⊕}, u', p, u) \).

6. \( a = (\text{⊕}, u', p, u) \). The proof is similar to that for \( a = (u, \text{SELECT}, q) \). The only difference is in proving that for any \( r' \in [r]_{P,u} \), \( \text{last}(e(r, a)) \) and \( \text{last}(e(r', a)) \) are data indistinguishable. Assume, for contradiction’s sake, that this is not the case. Let \( s_2 = (db_2, U_2, sec_2, T_2, V_2) \) be \( \text{sysState}(\text{last}(e(r, a))) \) and \( s'_2 = (db'_2, U'_2, sec'_2, T'_2, V'_2) \) be \( \text{sysState}(\text{last}(e(r', a))) \). Furthermore, let \( s_1 = (db_1, U_1, sec_1, T_1, V_1) \) be \( \text{sysState}(\text{last}(r)) \) and \( s'_1 = (db'_1, U'_1, sec'_1, T'_1, V'_1) \) be \( \text{sysState}(\text{last}(r')) \). In the following, we denote the \text{permissions} function by \( p \). Furthermore, note that \( s_1 \) and \( s'_1 \) are data-indistinguishable because \( r' \in [r]_{P,u} \). There are a number of cases:
   (a) \( U_2 \neq U'_2 \). Since \( a \) is an \text{REVOKE} operation, it follows that \( U_1 = U_2 \) and \( U'_1 = U'_2 \). Furthermore, from \( s_1 \triangleq_{M,u} s'_1 \), it follows that \( U_1 = U'_1 \). Therefore, \( U_2 = U'_2 \) leading to a contradiction.
   (b) \( sec_2 \neq sec'_2 \). From \( s_1 \triangleq_{M,u} s'_1 \), it follows that \( sec_1 = sec'_1 \). From \( a \)’s definition and the LTS rules, it follows that \( sec_1 = \text{revolve} (sec_1, u', p, u) \) and \( sec_2 = \text{revolve} (sec'_1, u', p, u) \). From this and \( sec_1 = sec'_1 \), it follows that \( sec_2 = sec'_2 \) leading to a contradiction.
   (c) \( T_2 \neq T'_2 \). The proof is similar to the case \( U_2 \neq U'_2 \).
   (d) \( V_2 \neq V'_2 \). The proof is similar to the case \( U_2 \neq U'_2 \).
   (e) There is a table \( R \) for which \( (\text{⊕}, \text{SELECT}, R) \in p(s_2, u) \) and \( db_2(R) \neq db'_2(R) \). Since \( a \) is an \text{REVOKE} operation, it follows that \( db_1 = db_2 \) and \( db'_1 = db'_2 \). Furthermore, from \( s_1 \triangleq_{M,u} s'_1 \), it follows that \( db_1(R) = db'_1(R) \). From this, \( db_1 = db_2 \), and \( db'_1 = db'_2 \), it follows that \( db_2(R) = db'_2(R) \) leading to a contradiction.
   (f) There is a view \( v \) for which \( (\text{⊕}, \text{SELECT}, v) \in p(s_2, u) \) and \( db_2(v) \neq db'_2(v) \). Since \( a \) is an \text{REVOKE} operation, it follows that \( db_1 = db_2 \) and \( db'_1 = db'_2 \). Furthermore, from \( s_1 \triangleq_{M,u} s'_1 \), it follows that \( db_1(v) = db'_1(v) \). From this, \( db_1 = db_2 \), and \( db'_1 = db'_2 \), it follows that \( db_2(v) = db'_2(v) \) leading to a contradiction.

All the cases lead to a contradiction.

7. \( a = (u, \text{CREATE}, o) \). The proof is similar to that for \( a = (\text{⊕}, u', p, u) \).

8. \( a = (u, \text{ADD_USER}, u') \). The proof is similar to that for \( a = (\text{⊕}, u', p, u) \).

This completes the proof.
Lemma C.24. Let \( u \) be a user in \( \cal U \), \( P = \langle M, f \rangle \) be an extended configuration, where \( M = \langle D, \Gamma \rangle \) is a system configuration and \( f \) is the PDP from Section 6.8, and \( L \) be the \( \cal P \)-LTS. For any run \( r \in \text{traces}(L) \) such that \( \text{invoker}(\text{last}(r)) = u \) and any trigger \( t \in \text{TRIGGER}_D \), if \( \text{extend}(r, t) \) is defined, then \( t \) preserves the equivalence class for \( r, M, \) and \( \equiv_{P,u} \).

Proof. Let \( u \) be a user in \( \cal U \), \( P = \langle M, f \rangle \) be an extended configuration, where \( M = \langle D, \Gamma \rangle \) is a system configuration and \( f \) is the PDP from Section 6.8, and \( L \) be the \( \cal P \)-LTS. For the following, we use \( e \) to refer to the \text{extend} function. The proof in cases where the trigger \( t \) is not enabled is similar to the proof of the \text{SELECT} case of Lemma C.23. In the following, we therefore assume that the trigger \( t \) is enabled. We also assume that its \text{WHEN} condition is secure. We handle the case of triggers with an insecure \text{WHEN} condition separately. We prove our claim by contradiction. Assume, for contradiction’s sake, that there is a run \( r \in \text{traces}(L) \) such that \( \text{invoker}(\text{last}(r)) = u \) and a trigger \( t \) such that \( e(r, t) \) is defined and \( t \) does not preserve the equivalence class for \( r, P, \) and \( \equiv_{P,u} \). Since \( \text{invoker}(\text{last}(r)) = u \) and \( e(r, t) \) is defined, then \( e(r', t) \) is defined as well for any \( r' \in [r]_{P,u} \) (indeed, from \( \text{invoker}(\text{last}(r)) = u \), it follows that the last action in \( r \) is either an action issued by \( u \) or a trigger invoked by \( u \). From this, the fact that \( e(r, t) \) is defined, and the fact that \( r \) and \( r' \) are indistinguishable, it follows that \( \text{trigger}(\text{last}(r)) = \text{trigger}(\text{last}(r')) = t \). Let \( a \) be \( t \)'s action and \( w = (a, '\text{SELECT}, q) \) be the \text{SELECT} command associated with \( t \)'s \text{WHEN} condition. Let \( s \) be the state \( \text{last}(r, t) \). There are a number of cases depending on \( t \)'s action \( a \):

1. \( a = (u', '\text{INSERT}, R, \overline{t}) \).
   - (a) \( \text{seeEx}(\text{last}(e(r, a))) = \bot \) and \( \text{Ex}(\text{last}(e(r, a))) = \emptyset \). The proof of this case is similar to that of the corresponding case in Lemma C.23.
   - (b) \( \text{seeEx}(\text{last}(e(r, a))) = \bot \) and \( \text{Ex}(\text{last}(e(r, a))) \neq \emptyset \). The only difference between the proof of this case with respect to the corresponding case in Lemma C.23 is that we have to establish again the data indistinguishability between \( \text{last}(e(r, t)) \) and \( \text{last}(e(r', t)) \). Indeed, for triggers the roll-back state is, in general, different from the one immediately before the trigger’s execution, i.e., it may be that \( \text{sysState}(\text{last}(e(r, t))) \neq \text{sysState}(\text{last}(r)) \). We now prove that \( \text{last}(e(r, t)) \) and \( \text{last}(e(r', t)) \) are data indistinguishable. From the LTS semantics, it follows that \( r = p \cdot s_0 \cdot \langle \text{invoker}(\text{last}(r)), op, R, \overline{t}, \overline{\overline{t}} \rangle \cdot s_1 \cdot t_1 \cdot \ldots \cdot s_{n-1} \cdot t_n \cdot s_n \), where \( p \in \text{traces}(L) \) and \( t_1, \ldots, t_n \in \text{TRIGGER}_P \). Similarly, \( r' = p' \cdot s_0' \cdot \langle \text{invoker}(\text{last}(r)), op, R', \overline{t}, \overline{\overline{t}} \rangle \cdot s_1' \cdot t_1' \cdot \ldots \cdot s_{n-1}' \cdot t_n' \cdot s_n' \), where \( p' \in \text{traces}(L) \), \( p \equiv_{P,u} p' \) and all states \( s_i \) and \( s_i' \) are data indistinguishable. Then, the roll-back states are, respectively, \( s_0 \) and \( s_0' \), which are data indistinguishable. From the LTS rules, \( \text{last}(e(r, a)) = s_0 \) and \( \text{last}(e(r', a)) = s_0' \). Therefore, the data indistinguishability between \( \text{last}(e(r, a)) \) and \( \text{last}(e(r', a)) \) follows trivially for any \( r' \in [r]_{P,u} \).
   - (c) \( \text{seeEx}(e(r, a)) = \top \). The proof is similar to the previous case.

2. \( a = (u', '\text{DELETE}, R, \overline{t}) \).
   - The proof is similar to that for \( a = (u', '\text{INSERT}, R, \overline{t}) \).

3. \( a = (\oplus, u', p, u') \).
   - (a) \( \text{seeEx}(\text{last}(e(r, a))) = \bot \). In this case, the proof is similar to the corresponding case in Lemma C.23.
   - (b) \( \text{seeEx}(\text{last}(e(r, a))) = \top \). The proof is similar to the \text{seeEx}(\text{last}(e(r, a))) = \top case of \( a = (u', '\text{INSERT}, R, \overline{t}) \).

Both cases lead to a contradiction. This completes the proof for \( a = (\oplus, u', p, u') \).

4. \( a = (\oplus', u'', p, u') \).
   - The proof is similar to that for \( a = (\oplus, u'', p, u') \).

5. \( a = (\oplus, u', p, u') \).
   - The proof is similar to that for \( a = (u', '\text{INSERT}, R, \overline{t}) \).

This completes the proof.

C.4.10 \( f \) provides Data Confidentiality

In Theorem C.2, we finally prove the main result of this section, namely that \( f \) provides data confidentiality.

Theorem C.2. Let \( M \) be a system configuration, \( f \) be the PDP from Section 6.8, and \( P = \langle M, f \rangle \) be an extended configuration. For any user \( u \in \cal U \), the PDP \( f \) provides data confidentiality with respect to \( P, u, \text{ATK}_u, \) and \( \equiv_{P,u} \).
Proof. Let \( u \) be a user in \( U \), \( P = \{ M, f \} \) be an extended configuration, where \( M = (D, \Gamma) \) is a system configuration and \( f \) in the PDP from Section 6.8, and \( L \) be the P-LTS. Furthermore, let \( r \) be a run in \( \text{traces}(L) \), \( i \) be an integer such that \( 1 \leq i \leq |r| \), and \( \phi \) be a sentence such that \( r, i \vdash_u \phi \) holds. We claim that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds. The theorem follows trivially from the claim.

We now show that for all \( r \in \text{traces}(L) \), all \( i \) such that \( 1 \leq i \leq |r| \), and all sentences \( \phi \) such that \( r, i \vdash_u \phi \) holds, then \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds. In the following, we denote by \( c \) the function \( \text{extend} \).

**Base Case:** Assume that \( r, i \vdash_u \phi \) holds. There are a number of cases depending on the way \( r \) is used to obtain \( i \vdash_u \phi \).

1. **SELECT Success - 1.** Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, \text{SELECT}, \phi) \cdot s \), where \( s = (db, U, \text{sec}, T, V, c) \in \Omega_M \) and \( \text{last}(r^{i-1}) = s' \), and \( s' = (db, U, \text{sec}, T, V, c') \). From the rules, it follows that \( f(s', (u, \text{SELECT}, \phi)) = \top \). From this and \( f \)’s definition, it follows that \( f_{\text{conf}}(s', (u, \text{SELECT}, \phi)) = \top \) and \( f_{\text{conf}}(s', (u, \text{SELECT}, \phi)) = \top \). Hence, \( \text{secure}(u, \phi, s') = \top \). From this, Lemma C.14, and \( \text{sysState}(s') = \text{sysState}(s) \), it follows \( \text{secure}(u, \phi, s) = \top \). From this, Proposition C.9, and \( \text{last}(r^i) = s \), it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.

2. **SELECT Success - 2.** The proof for this case is similar to that of **SELECT Success - 1.**

3. **INSERT Success.** Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, \text{INSERT}, R, T) \cdot s \), where \( s = (db, U, \text{sec}, T, V, c) \in \Omega_M \) and \( \text{last}(r^{i-1}) = (db', U, \text{sec}, T, V, c') \), and \( \phi = R(T) \). Then, \( \text{secure}_{P,u}(r, i \vdash_u R(T)) \) holds. Indeed, in all runs \( r^i (P, u) \)-indistinguishable from \( r^i \) the last action is \( (u, \text{INSERT}, R, T) \).

Furthermore, the action has been executed successfully. Therefore, according to the LTS rules, \( T \in \text{last}(r^i \cdot \text{db}) \) for all runs \( r^i \in [\text{db}]_{P,u} \). From this and the relational calculus semantics, it follows that \( |R(T)|_{\text{last}(r^i \cdot \text{db})} = |r| \) for all runs \( r^i \in [\text{db}]_{P,u} \). Hence, \( \text{secure}_{P,u}(r, i \vdash_u R(T)) \) holds.

4. **INSERT Success - FD.** Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, \text{INSERT}, R, P, \mu, \eta) \cdot s \), where \( s = (db, U, \text{sec}, T, V, c) \in \Omega_M \) and \( \text{last}(r^{i-1}) = (db', U, \text{sec}, T, V, c') \), and \( \phi = (\neg \exists \gamma, \exists \tau, R(P, \mu, \eta) \land \eta \neq \mu) \). From the rule's definition, it follows that \( \text{secure}(s, \phi, s') = \top \). From this and the LTS rules, it follows that \( f(s', (u, \text{INSERT}, R, P, \mu, \eta)) = \top \). From this and \( f \)’s definition, it follows that \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = \top \) holds because \( \phi \) is equivalent to \( \text{getInfo}(\gamma, \tau) \) for some \( \gamma \in \text{Dep}(\Gamma) \) where \( \Gamma = (u, \text{INSERT}, R, P, \mu, \eta) \). From this and Proposition C.9, it follows that \( \text{secure}_{P,u}(r, i-1 \vdash_u \phi) \) holds. From this and Proposition C.7, it follows that also \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.

We now prove our claim that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds. Let \( s' \) be the state \( \text{last}(r^{i-1}) \). Note that, for brevity’s sake, in the following we omit the \( \text{sysState} \) function where needed. For instance, with a slight abuse of notation, we write \( [s']_{M,u} \) instead of \( \text{[sysState}(s')]_{M,u} \).

There are two cases:

(a) The \( \text{INSERT} \) command has caused an integrity constraint violation, i.e., \( \text{Ex}(s) \neq \emptyset \). From \( \text{secure}(u, \phi, s') = \top \) and Proposition C.9, it follows that \( \text{secure}_{P,u}(r, i-1 \vdash_u \phi) \) holds.

(b) The \( \text{INSERT} \) command has not caused exceptions, i.e., \( \text{Ex}(s) = \emptyset \). From \( \text{secure}(u, \phi, s') = \top \) and Proposition C.9, it follows that \( \text{secure}_{P,u}(r, i-1 \vdash_u \phi) \) holds. From this, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.

5. **INSERT Success - ID.** The proof of this case is similar to that for the **INSERT Success - FD.**

6. **DELETE Success.** The proof for this case is similar to that of **INSERT Success.**

7. **DELETE Success - ID.** The proof of this case is similar to that for the **INSERT Success - ID.**

8. **INSERT Exception.** Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, \text{INSERT}, R, T) \cdot s \), where \( s = (db, U, \text{sec}, T, V, c) \in \Omega_M \) and \( \text{last}(r^{i-1}) = (db', U, \text{sec}, T, V, c') \), and \( \phi = \neg R(T) \). From the rule’s definition, it follows that \( \text{secure}(s, \phi, \text{last}(r^{i-1})) = \top \). From this and the LTS rules, it follows that \( f(s', (u, \text{INSERT}, R, T)) = \top \). From this and \( f \)’s definition, it follows that \( f_{\text{conf}}(s', (u, \text{INSERT}, R, T)) = \top \), because \( \text{user}(s', (u, \text{INSERT}, R, T)) = u \). From this and \( f_{\text{conf}} \)’s definition, it follows that \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = \top \).
holds because $\phi = getInfO(u, INSERT, R, T)$. From this and Proposition 9.9, it follows that $secure_p_u(r, i - 1 \vdash u, \phi)$ holds. From the LTS semantics, it follows that $sysState(s) \models_{data} sysState(last(r^{i-1}))$. From this, $secure(u, \phi, last(r^{i-1})) = \top$, and Lemma C.14, it follows that $secure(u, \phi, last(r^{i})) = \top$. From this and Proposition C.9, it follows that $secure_p_u(r, i \vdash u, \phi)$ holds.

9. DELETE Exception. The proof for this case is similar to that of INSERT Exception.

10. INSERT FD Exception. Let $i$ be such that $r^i = r^{i-1} \cdot \langle u, INSERT, R, (\pi, \pi, \pi) \rangle$, where $s = \langle \delta b, U, sec', T, V, c \rangle \in \Omega_M$ and last$^{\langle i \rangle} = \langle \delta b', U, sec, T, V, c \rangle$, and $\phi \in \mathcal{F}$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(s', \langle u, INSERT, R, (\pi, \pi, \pi) \rangle) = \top$. From this and $f$'s definition, it follows that $f_{conf}'(s', \langle u, INSERT, R, (\pi, \pi, \pi) \rangle) = \top$, because $user(s', \langle u, INSERT, R, (\pi, \pi, \pi) \rangle) = u$. From this and $f_{conf}'$'s definition, it follows that $secure(u, \phi, last(r^{i-1})) = \top$ because $\phi = getInfO(\gamma, (u, INSERT, R, (\pi, \pi, \pi)))$ for some constraint $\gamma \in Dep(\Gamma, \langle u, INSERT, R, (\pi, \pi, \pi) \rangle))$. From this and Proposition C.9, it follows that $secure_p_u(r, i - 1 \vdash u, \phi)$ holds. From the LTS semantics, it follows that $sysState(s) \models_{data} sysState(last(r^{i-1}))$. From this, Lemma C.14, and $secure(u, \phi, last(r^{i-1})) = \top$, it follows that $secure(u, \phi, last(r^{i})) = \top$. From this and Proposition C.9, it follows that $secure_p_u(r, i \vdash u, \phi)$ holds.

11. INSERT ID Exception. The proof for this case is similar to that of INSERT FD Exception.

12. DELETE FD Exception. The proof for this case is similar to that of INSERT FD Exception.

13. Integrity Constraint. The proof of this case follows trivially from the fact that for any state $s = \langle \delta b, U, sec', T, V, c \rangle \in \Omega_M$ and any $\gamma \in \Gamma$, $\gamma = \top$ by definition.

14. Learn GRANT/REVOKE Backward. Let $i$ be such that $r^i = r^{i-1} \cdot \mathcal{F}$, where $s = \langle \delta b, U, sec, T, V, c \rangle \in \Omega_M$, last$^{\langle i \rangle} = \langle \delta b, U, sec', T, V, c \rangle$, and $t$ be a trigger whose WHEN condition is $\psi$, and whose action is either a GRANT or a REVOKE. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$, where $u'$ is either the trigger's owner or the trigger's invoker depending on the security mode. From this and $f$'s definition, it follows $f_{conf}'(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$, because $user(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = u$ because $t$'s invoker is $u$ according to the rules. From this and $f_{conf}'$'s definition, it follows $secure(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that $secure_p_u(r, i \vdash u, \phi)$ holds.

15. Trigger GRANT Disabled Backward. Let $i$ be such that $r^i = r^{i-1} \cdot \mathcal{F}$, where $s = \langle \delta b, U, sec, T, V, c \rangle \in \Omega_M$, last$^{\langle i \rangle} = \langle \delta b, U, sec', T, V, c \rangle$, and $t$ be a trigger whose WHEN condition is $\psi$, and whose action is an INSERT statement $\langle u', INSERT, R, (\pi, \pi, \pi) \rangle$. Furthermore, let $\phi$ be $\exists \pi. \exists R. (\pi, \pi, \pi) \land \theta \neq \pi$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$, where $u'$ is either the trigger's owner or the trigger's invoker depending on the security mode. From this and $f$'s definition, it follows $f_{conf}'(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$, because $user(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = u$ because $t$'s invoker is $u$ according to the rules. From this and $f_{conf}'$'s definition, it follows that secure$^{\langle i \rangle}(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that secure$^{\langle i \rangle}(r, i - 1 \vdash u, \phi)$ holds.

16. Trigger REVOKE Disabled Backward. The proof for this case is similar to that of Trigger GRANT Disabled Backward.

17. Trigger INSERT FD Exception. Let $i$ be such that $r^i = r^{i-1} \cdot \mathcal{F}$, where $s = \langle \delta b, U, sec, T, V, c \rangle \in \Omega_M$, last$^{\langle i \rangle} = \langle \delta b, U, sec', T, V, c \rangle$, and $t$ be a trigger whose WHEN condition is $\psi$, and whose action is an INSERT statement $\langle u', INSERT, R, (\pi, \pi, \pi) \rangle$. From the rule's definition, it follows that $secEx(s) = \bot$. From this and the LTS rules, it follows that $f(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$. From this and $f$'s definition, it follows $f_{conf}'(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$, because $user(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = u$ since $t$'s invoker is $u$ according to the rules. From this and $f_{conf}'$'s definition, it follows that secure$^{\langle i \rangle}(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that secure$^{\langle i \rangle}(r, i \vdash u, \phi)$ holds.

18. Trigger INSERT ID Exception. The proof for this case is similar to that of Trigger INSERT ID Exception.

19. Trigger DELETE ID Exception. The proof for this case is similar to that of Trigger DELETE ID Exception.

20. Trigger Exception. Let $i$ be such that $r^i = r^{i-1} \cdot \mathcal{F}$, where $s = \langle \delta b, U, sec, T, V, c \rangle \in \Omega_M$, last$^{\langle i \rangle} = \langle \delta b, U, sec', T, V, c \rangle$, and $t$ be a trigger whose WHEN condition is $\phi$ and whose action is $act$. From the rule's definition, it follows that $f(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$, where $u'$ is either the trigger's owner or the trigger's invoker depending on the security mode. From this and $f$'s definition, it follows $f_{conf}'(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = \top$, because $user(last(r^{i-1}), \langle u', SELECT, \phi \rangle) = u$ since $t$'s invoker is $u$ according to the rules. From this and $f_{conf}'$'s definition, it follows that secure$^{\langle i \rangle}(u, \phi, last(r^{i-1})) = \top$. From this and Proposition C.9, it follows that secure$^{\langle i \rangle}(r, i \vdash u, \phi)$ holds.

21. Trigger INSERT Exception. The proof for this case is similar to that of INSERT Exception.

22. Trigger DELETE Exception. The proof for this case is similar to that of DELETE Exception.
23. **Trigger Rollback INSERT**. Let \( i \) be such that \( r^i = r^{i-1}\cdot(u, \text{INSERT}, R, T)\cdot s_1\cdot s_2\cdot \ldots \cdot s_n \), where \( s_1, s_2, \ldots , s_n \in \Omega_M \) and \( t_1, \ldots , t_n \in \text{TRIGGER}_D \), and \( \phi \) be \( \neg R(T) \). Furthermore, let \( \text{last}(r^{i-1}) = (db', U', sec', T', V', c') \) and \( s_n \) be \( (db, U, sec, T, V, c) \). From the rule's definition, it follows that \( \text{secEx}(s_1) = \bot \). From this, it follows that \( f(\text{last}(r^{i-1}), (u, \text{INSERT}, R, T)) = T \). From this and \( f's \) definition, it follows \( \text{f}_\text{conf}^{\text{last}(r^{i-1})}, (u, \text{INSERT}, R, T) = u \). From this and \( \text{f}_\text{conf}^{\text{last}(r^{i-1})} \)'s definition, it follows \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = T \) because \( \phi \) is \( \text{getInfo}, (u, \text{INSERT}, R, T) \). From the LTS semantics, it follows that \( \text{last}(r^{i-1}) = \text{sysState}(\text{last}(r^{i-1})) = \text{sysState}(s_n) \). From this, Lemma C.14, and \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = T \), it follows that \( \text{secure}(u, \phi, s_n) = T \). From this and Proposition C.9, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.

24. **Trigger Rollback DELETE**. The proof for this case is similar to that of Trigger Rollback INSERT.

25. **Learn from deny -- actions**. We prove this case by contradiction. Assume that we can derive \( r \vdash_{i - 1} u \vdash_u \phi \) using the rule Learn from deny -- actions. There are \( r^i, r^{i'} \in \text{traces}(L) \), \( 1 < i \leq |r|, a \in \mathcal{A}_{D,u}, s, s' \in \Omega_M \), and \( \phi \) such that: \( r^i = r^{i-1}\cdot a \cdot s, r^{i'} = r^{i'-1}\cdot a' \cdot s', r^{i-1} \equiv_{P,u} r^{i'}, \text{secEx}(s') \neq \text{secEx}(s), [\phi]\text{last}(r^{i-1})_{\text{db}} = \top, \) and \( [\phi]\text{last}(r^{i'})_{\text{db}} = \bot \). From this, \( r^{i-1} \equiv_{P,u} r^{i'}, \text{extend}(r^{i-1}, a) \) and \( \text{extend}(r^{i'}, a) \) are well-defined. From this, Lemma C.23, and \( a \in \mathcal{A}_{D,u}, a \) preserves the equivalence class with respect to \( r^{i-1}, P, \) and \( \equiv_{P,u} \). From this, the rule's definition, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.

26. **Learn from deny -- triggers**. The proof is similar to that of the case Learn from deny -- actions. However, we use Lemma C.24 instead of Lemma C.23.

This completes the proof of the step.

**Induction Step:** Assume that the claim hold for any derivation of \( r, j \vdash_u \psi \) such that \( |r, j \vdash_u \psi| < |r, i \vdash_u \phi| \). We now prove that the claim also holds for \( r, i \vdash_u \phi \). There are a number of cases depending on the rule used to obtain \( r, i \vdash_u \phi \).

1. **View.** The proof of this case follows trivially from the semantics of the relational calculus extended over views.

2. **Propagate Forward SELECT.** Let \( i \) be such that \( r^{i+1} = r^i \cdot (u, \text{SELECT}, \psi) \cdot s \), where \( s = (db, U, sec, T, V, c) \in \Omega_M \) and \( \text{last}(r^i) = (db', U', sec', T', V', c') \). From the rule, it follows that \( r, i \vdash_u \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds. From Lemma C.23, the action \( (u, \text{SELECT}, \psi) \) preserves the equivalence class with respect to \( r^{i'}, P, \) and \( \equiv_{P,u} \). From this, Lemma C.19, and \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds, it follows that \( \text{secure}_{P,u}(r, i + 1 \vdash_u \phi) \) holds.

3. **Propagate Forward GRANT/REVOKE.** Let \( i \) be such that \( r^{i+1} = r^i \cdot (op, u', p, u) \cdot s \), where \( s = (db, U, sec, T, V, c) \in \Omega_M \) and \( \text{last}(r^i) = (db', U', sec', T', V', c') \). From the rule, it follows that \( r, i \vdash_u \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds. From Lemma C.23, the action \( (op, u', p, u) \) preserves the equivalence class with respect to \( r^{i'}, P, \) and \( \equiv_{P,u} \). From this, Lemma C.20, and \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds, it follows that \( \text{secure}_{P,u}(r, i + 1 \vdash_u \phi) \) holds.

4. **Propagate Forward CREATE.** The proof for this case is similar to that of Propagate Forward SELECT.

5. **Propagate Backward SELECT.** Let \( i \) be such that \( r^{i-1} = r^i \cdot (u, \text{SELECT}, \psi) \cdot s \), where \( s = (db', U', sec', T', V', c') \in \Omega_M \) and \( \text{last}(r^i) = (db, U, sec, T, V, c) \). From the rule, it follows that \( r, i - 1 \vdash_u \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i - 1 \vdash_u \phi) \) holds. From Lemma C.23, the action \( (u, \text{SELECT}, \psi) \) preserves the equivalence class with respect to \( r^{i'}, P, \) and \( \equiv_{P,u} \). From this, Lemma C.19, and \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.

6. **Propagate Backward GRANT/REVOKE.** Let \( i \) be such that \( r^{i-1} = r^i \cdot (op, u', p, u) \cdot s \), where \( s = (db', U', sec', T', V', c') \in \Omega_M \) and \( \text{last}(r^i) = (db, U, sec, T, V, c) \). From the rule, it follows that \( r, i + 1 \vdash_u \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i + 1 \vdash_u \phi) \) holds. From Lemma C.23, the action \( (op, u', p, u) \) preserves the equivalence class with respect to \( r^{i'}, P, \) and \( \equiv_{P,u} \). From this, Lemma C.20, and \( \text{secure}_{P,u}(r, i + 1 \vdash_u \phi) \) holds, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.

7. **Propagate Backward CREATE TRIGGER.** The proof for this case is similar to that of the Propagate Backward SELECT case.

8. **Propagate Backward CREATE VIEW.** Note that the formulae \( \psi \) and \( \text{replace}(\psi, o) \) are semantically equivalent. This is the only difference between the proof for this case and the one for the Propagate Backward SELECT case.

9. **Rollback Backward - 1.** Let \( i \) be such that \( r^{i-1} = r^{i-1} \cdot (u, \text{INSERT}, R, T)\cdot s_1\cdot s_2\cdot \ldots \cdot s_n \), where \( s_1, s_2, \ldots , s_n \in \Omega_M \), and \( t_1, \ldots , t_n \in \text{TRIGGER}_D \), and \( op \) is one of \{ INSERT, DELETE \}. Furthermore, let \( s_n \) be \( (db', U', sec', T', V', c') \) and \( \text{last}(r^{i-1}) = (db, U, sec, T, V, c) \). From the rule's definition, it follows that \( r, i \vdash_u \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i \vdash_u \phi) \) holds.
From Lemma C.24, the trigger \( t_j \) preserves the equivalence class with respect to \( r^{i-n-1+j}, P \), and \( \Xi_{P,u} \) for any \( 1 \leq j \leq n \). Therefore, for any \( v \in [r^{i-1}]_{P,u} \), the run \( e(v, t_1) \) contains the rollback. Therefore, for any \( v \in [r^{i-1}]_{P,u} \), the state \( last(e(v, t_1)) \) is the state just before the action \( (u, op, R, T) \). Let \( A \) be the set of system states associated with the rollback states. It is easy to see that \( A \) is the same as \( \{ \text{sysState}(last(t')) | t' \in [r^{i-n-1}]_{P,u} \} \). From secure_{P,u}(r, i - n \downarrow u), it follows that \( \phi \) has the same result over all states in \( A \). From this and \( A = \{ \text{sysState}(last(t')) | t' \in [r^{i-n-1}]_{P,u} \} \), it follows that \( \phi \) has the same result over all states in \( \{ \text{sysState}(last(t')) | t' \in [r^{i-n-1}]_{P,u} \} \). From this, it follows that secure_{P,u}(r, i - n \downarrow u) holds.

10. Rollback Backward - 2. Let \( i \) be such that \( r^i = r^{i-1}.(u, op, R, T), s \), where \( s = \{ db', U', sec', T', V', c' \} \in \Omega_M \), \( last(r^{i-1}) = \{ db, U, sec, T, V, c \} \), and \( op \) is one of \{INSERT, DELETE\}. From the rule’s definition, \( r, i \vdash u \phi \) holds. From this and the induction hypothesis, it follows that secure_{P,u}(r, i - 1 \vdash u \phi) holds. From Lemma C.24, the trigger \( t_j \) preserves the equivalence class with respect to \( r^{i-n-1+j}, P \), and \( \Xi_{P,u} \) for any \( 1 \leq j \leq n \). Independently on the cause of the rollback (either a security exception or an integrity constraint violation), we claim that the set \( A \) of rollback system states is \( \{ \text{sysState}(last(t')) | t' \in [r^{i-n-1}]_{P,u} \} \). From secure_{P,u}(r, i - n \downarrow u \phi), the result of \( \phi \) is the same for all states in \( A \). From this and \( A = \{ \text{sysState}(last(t')) | t' \in [r^{i-n-1}]_{P,u} \} \), it follows that also secure_{P,u}(r, i \vdash u \phi) holds.

We now prove our claim. It is trivial to see (from the LTS semantics) that the set of rollback states is a subset of \{last(v) | v \in [r^{i-1}]_{P,u} \}. Assume, for contradiction’s sake, that there is a state in \{last(v) | v \in [r^{i-1}]_{P,u} \} that is not a rollback state for the runs in \([r^i]_{P,u} \). This is impossible since all triggers \( t_1, \ldots, t_p \) preserve the equivalence class.

12. Rollback Forward - 2. Let \( i \) be such that \( r^i = r^{i-1}.(u, op, R, T), s \), where \( op \in \{INSERT, DELETE\} \), \( s = \{ db, U, sec, T, V, c \} \in \Omega_M \), and \( last(r^{i-1}) = \{ db', U', sec', T', V', c' \} \). From the rule’s definition, \( r, i - 1 \vdash u \phi \) holds. From this and the induction hypothesis, it follows that secure_{P,u}(r, i - 1 \vdash u \phi) holds. From Lemma C.23, the action \( (u, op, R, T) \) preserves the equivalence class with respect to \( r^{i-1}, P \), and \( \Xi_{P,u} \). From this, Lemma C.18, the fact that the action does not modify the database state, and secure_{P,u}(r, i \vdash u \phi), it follows secure_{P,u}(r, i - 1 \vdash u \phi).

13. Propagate Forward INSERT/DELETE Success. Let \( i \) be such that \( r^i = r^{i-1}.(u, op, R, T), s \), where \( op \in \{INSERT, DELETE\} \), \( s = \{ db, U, sec, T, V, c \} \in \Omega_M \), and \( last(r^{i-1}) = \{ db', U', sec', T', V', c' \} \). From the rule’s definition, \( r, i - 1 \vdash u \phi \) holds. From this and the induction hypothesis, it follows that secure_{P,u}(r, i - 1 \vdash u \phi) holds. From Lemma C.23, the action \( (u, op, R, T) \) preserves the equivalence class with respect to \( r^{i-1}, P \), and \( \Xi_{P,u} \). From this, Lemma C.18, and secure_{P,u}(r, i \vdash u \phi), it follows that secure_{P,u}(r, i - 1 \vdash u \phi).

14. Propagate Forward INSERT Success - 1. Let \( i \) be such that \( r^i = r^{i-1}.(u, op, R, T), s \), where \( op \in \{INSERT, DELETE\} \), \( s = \{ db, U, sec, T, V, c \} \in \Omega_M \), and \( last(r^{i-1}) = \{ db', U', sec', T', V', c' \} \). From the rule’s definition, \( r, i - 1 \vdash u \phi \) holds. From this and the induction hypothesis, it follows that secure_{P,u}(r, i - 1 \vdash u \phi) holds. From Lemma C.23, the action \( (u, op, R, T) \) preserves the equivalence class with respect to \( r^{i-1}, P \), and \( \Xi_{P,u} \). We claim that the execution of \( (u, INSERT, R, T) \) does not alter the content of the tables \( \text{tables}(\phi) \) for any \( v \in [r^{i-1}]_{P,u} \). From this, Lemma C.18, and secure_{P,u}(r, i \vdash u \phi), it follows that secure_{P,u}(r, i - 1 \vdash u \phi).

15. Propagate Forward DELETE Success - 1. The proof for this case is similar to that of Propagate Forward INSERT Success - 1.

16. Propagate Backward INSERT/DELETE Success. Let \( i \) be such that \( r^i = r^{i-1}.(u, op, R, T), s \), where \( op \in \{INSERT, DELETE\} \), \( s = \{ db, U, sec, T, V, c \} \in \Omega_M \), and \( last(r^{i-1}) = \{ db', U', sec', T', V', c' \} \). From the rule’s definition, \( r, i \vdash u \phi \) holds. From this and the induction hypothesis, it follows that secure_{P,u}(r, i \vdash u \phi) holds. From Lemma C.23, the action \( (u, op, R, T) \) preserves the equivalence class with respect to \( r^{i-1}, P \), and \( \Xi_{P,u} \). From this, Lemma C.18, secure_{P,u}(r, i \vdash u \phi) holds.

We now prove our claim that the execution of \( (u, INSERT, R, T) \) does not alter the content of the tables \( \text{tables}(\phi) \). From the rule’s definition, it follows that \( r, i - 1 \vdash u \phi \) holds. From this and Proposition C.1, it follows that \( [R(T)]^\text{last}(r^{i-1}) \vdash \top \). From \( r, i - 1 \vdash u \phi \), and the induction hypothesis, it follows that secure_{P,u}(r, i - 1 \vdash u \phi) holds. From this and \([R(T)]^\text{last}(r^{i-1}) \vdash \top \), it follows that \( [R(T)]^\text{last}(v) \vdash \top \) for any \( v \in [r^{i-1}]_{P,u} \). From this and the relational calculus semantics, it follows that the execution of \( (u, op, R, T) \) does not alter the content of the tables \( \text{tables}(\phi) \) for any \( v \in [r^{i-1}]_{P,u} \).
class with respect to \( r^{i-1} \), \( P \), and \( \Xi_{P,u} \). From \( \text{rename}(r^{i-1}, \phi, r^i) \), it follows that the execution of \( (u, \text{op}, R, T) \) does not alter the content of the tables in \( \text{tables}(\phi) \) for any \( v \in [r^{i-1}]_{P,u} \). From this, Lemma C.18, and \( \text{secure}_{P,u}(r, i-1 \mapsto \phi) \), it follows that \( \text{secure}_{P,u}(r, i \mapsto \phi) \) holds.

17. Propagate Backward \text{INSERT} Success - 1. Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, \text{op}, R, T) \cdot s \), where \( \text{op} \) is one of \{\text{INSERT}, DELETE\}, \( s = (db, U, sec, T, V, c) \in \Omega_M \) and \( \text{last}(r^{i-1}) = (db', U', sec', T', V', c') \). From the rule’s definition, \( r, i \mapsto \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i \mapsto \phi) \) holds. From Lemma C.23, the action \( (u, \text{op}, R, T) \) preserves the equivalence class with respect to \( r^{i-1} \), \( P \), and \( \Xi_{P,u} \). We claim that the execution of \( (u, \text{INSERT}, R, T) \) does not alter the content of the tables in \( \text{tables}(\phi) \) for any \( v \in [r^{i-1}]_{P,u} \) (the proof of this claim is in the proof of the Propagate Forward \text{INSERT} Success - 1 case). From this, Lemma C.18, and \( \text{secure}_{P,u}(r, i \mapsto \phi) \), it follows that \( \text{secure}_{P,u}(r, i-1 \mapsto \phi) \) holds.

18. Propagate Backward \text{DELETE} Success - 1. The proof for this case is similar to that of Propagate Forward \text{DELETE} Success - 1.

19. Reasoning. Let \( \Delta \) be a subset of \( \{ \delta \mid r, i \mapsto \delta \} \) and \( \text{last}(r^i) = (db, U, sec, T, V, c) \). From the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i \mapsto \phi) \) holds for any \( \delta \) in \( \Delta \). Note that, given any \( \delta \) in \( \Delta \), from \( r, i \mapsto u \) and Proposition C.1, it follows that \( \delta \) holds in \( \text{last}(r^i) \). From this, \( \text{secure}_{P,u}(r, i \mapsto \phi) \) holds.

20. Learn \text{INSERT} Backward - 3. Let \( i \) be such that \( r^i = r^{i-1} \cdot (u, \text{INSERT}, R, T) \cdot s \), where \( s = (db', U', sec', T', V', c') \in \Omega_M \) and \( \text{last}(r^{i-1}) = (db, U, sec, T, V, c) \). From the rule’s definition, \( \text{Ex}(s) = \perp \). From this and the LTS rules, it follows that \( f(\text{last}(r^{i-1})), (u, \text{INSERT}, R, T) \) is \( \top \). From this and \( \phi \)'s definition, it follows that \( \text{Ex}_u(\text{last}(r^{i-1})), (u, \text{INSERT}, R, T) \) is \( \top \). From this and \( \text{Ex}_u \)'s definition, it follows that \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = \top \) because \( u, \phi \). From this and \( \text{Ex}_u \)'s definition, it follows that \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = \top \) because \( \phi \) is \( \text{getInfo}(u, \text{INSERT}, R, T) \). From this and Proposition C.9, it follows that \( \text{secure}_{P,u}(r, i \mapsto \phi) \) holds.

21. Learn \text{DELETE} Backward - 3. The proof is similar to that of Learn \text{INSERT} Backward - 3.

22. Propagate Forward Disabled Trigger. Let \( i \) be such that \( r^i = r^{i-1} \cdot t \cdot s \), where \( s = (db, U, sec, T, V, c) \in \Omega_M \), \( \text{last}(r^{i-1}) = (db, U, sec, T, V, c) \), and \( t \) be a trigger. Furthermore, let \( \psi \) be \( t \)'s condition where all free variables are replaced with \( \text{tpl}(\text{last}(r^{i-1})) \). From the rule, it follows that \( r, i \mapsto u \) and \( \Xi_{P,u} \). From this and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i-1 \mapsto \phi) \) holds. Furthermore, from Lemma C.24, it follows that \( t \) preserves the equivalence class with respect to \( r^{i-1} \), \( P \), and \( \Xi_{P,u} \). If the trigger’s action is an \text{INSERT} or a \text{DELETE} operation, we claim that the operation does not change the content of any table in \( \text{tables}(\phi) \). From this, the fact that \( t \) preserves the equivalence class with respect to \( r^{i-1} \), \( P \), and \( \Xi_{P,u} \), we now prove our claim. Assume that \( t \)'s action is an \text{INSERT} or a \text{DELETE} operation. From the rule, it follows that \( r, i-1 \mapsto u \). From this and Proposition C.1, \( [\psi]_{\text{last}(r^{i-1})} = \top \). From \( r, i-1 \mapsto u \) and \( \Xi_{P,u} \), the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i-1 \mapsto \phi) \) holds. From this and \( [\psi]_{\text{last}(r^{i-1})} = \top \), it follows that \( [\psi]_{\text{db}} = \top \) for any \( v \in [r^{i-1}]_{P,u} \). Therefore, the trigger \( t \) is disabled in any \( v \in [r^{i-1}]_{P,u} \). From this and the LTS semantics, it follows that \( t \)'s execution does not change the content of any table in \( \text{tables}(\phi) \). From this, the fact that \( t \) preserves the equivalence class (see Lemma C.24), it immediately follows that executing the trigger \( t \) in \( v \) does not generate security or integrity exceptions. From this and the fact that \( t \) preserves the equivalence class (see Lemma C.24), it immediately follows that executing the trigger \( t \) in \( v \) does not generate security and integrity exceptions for any \( v \in [r^{i-1}]_{P,u} \).

23. Propagate Forward Disabled Trigger. The proof for this case is similar to that of Propagate Forward Disabled Trigger.

24. Learn \text{INSERT} Forward. Let \( i \) be such that \( r^i = r^{i-1} \cdot s \cdot t \cdot s \), where \( s = (db, U, sec, T, V, c) \in \Omega_M \), \( \text{last}(r^{i-1}) = (db, U, sec, T, V, c) \), and \( t \) be a trigger, and \( \phi \) be \( R(T) \). Furthermore, let \( \psi \) be \( t \)'s condition where all free variables are replaced with \( \text{tpl}(\text{last}(r^{i-1})) \). From the rule’s definition, it follows that \( t \)'s action is \( (u, \text{INSERT}, R, T) \) and that \( r, i \mapsto u \). From Proposition C.1 and \( r, i \mapsto u \) \( \psi \) holds, it follows that \( [\psi]_{\text{db}} = \top \). From this, \( \text{Ex}(s) = \top \) and \( \text{Ex}(s) = \emptyset \), it follows that \( t \)'s action has been executed successfully. From this, it follows that \( t \in \text{db}(R) \). From \( r, i \mapsto u \) \( \psi \) and the induction hypothesis, it follows that \( \text{secure}_{P,u}(r, i \mapsto \phi) \). From this and \( [\psi]_{\text{db}} = \top \), it follows that \( [\psi]_{\text{db}} = \top \) for any \( v \in [r^{i-1}]_{P,u} \). From this, it follows that \( t \) preserves the equivalence class with respect to \( r^{i-1} \), \( P \), and \( \Xi_{P,u} \). From this, \( \text{Ex}(s) = \emptyset \), \( \text{Ex}(s) = \emptyset \), and the fact that \( t \) preserves \( \Xi_{P,u} \), it follows that \( t \)'s action is executed successfully in any run \( v \in [r^{i-1}]_{P,u} \). From this, it follows that \( db''(R) \), where \( db'' = t \) \( \text{Ex}(v, t) \cdot db \), for any \( v \in [r^{i-1}]_{P,u} \). Therefore, \( \text{secure}_{P,u}(r, i \mapsto \phi) \).
25. **Learn \texttt{INSERT} - FD.** Let \( i \) be such that \( r^i = r^{i-1}.t.s \), where \( s = \langle \text{db}, \text{U}, \text{sec}, T, V, c \rangle \in \Omega_M \), \( \text{last}(r^{i-1}) = \langle \text{db}, \text{U}', \text{sec}', T', V', c' \rangle \), and \( t \in \text{TRIGGER}_R \), and \( \phi \) be the \( i \)’s actual condition. Let \( \psi' \) be the \( i \)’s current condition where all free variables are replaced with the values in \( \text{tp}(\text{last}(r^{i-1})) \) and \( \langle u', \text{INSERT}, R, (\tau, \pi, \eta) \rangle \) be \( t \)’s actual action. From the rule, it follows that \( r, i-1 \vdash \psi' \). From this and Proposition C.1, it follows that \[ [\psi']^\text{last}(r^{i-1}).\text{db} = \top. \] From this, \( \text{Ex}(s) = \emptyset \), and \( \text{secEx}(s) = \lambda \), it follows that \( f(s', \langle u', \text{INSERT}, R, \tau \rangle) = \top \), where \( s' \) is the state just after the execution of the \texttt{SELECT} statement associated with \( t \)’s \texttt{WHEN} clause. From this and \( f \)’s definition, it follows that \( f_{\text{conf}}(s', \langle u', \text{INSERT}, R, \tau \rangle) = \top \) because \( \text{user}(s', \langle u', \text{INSERT}, R, \tau \rangle) = u \) since \( u \) is \( t \)’s invoker. From this and \( f_{\text{conf}} \)’s definition, it follows that \( \text{secure}(u, \phi, s') = \top \). From this, \( \text{sysState}(s') = \text{sysState}((\text{last}(r^{i-1}))) \), and Lemma C.14, it follows \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = \top \). From this and Proposition C.9, it follows \( \text{secure}_{\text{data}}(r, i-1.u \phi) \). We claim that \( \text{secure}_{\text{data}}(r, i-1.u \phi) \) holds. From this and Proposition C.7, it follows that also \( \text{secure}_{\text{data}}(r, i-1.u \phi) \) holds. We now prove our claim that \( \text{secure}_{\text{data}}(r, i-1.u \phi) \) holds. Let \( s' \) be the state just after the execution of the \texttt{SELECT} statement associated with \( t \)’s \texttt{WHEN} clause and \( s'' \) be the state \( \text{last}(r^{i-1}) \). Furthermore, for brevity’s sake, in the following we omit the \texttt{sysState} function where needed. For instance, with a slight abuse of notation, we write \([s]^M_{i,u} \) instead of \([\text{sysState}(s')]^M_{i,u} \). From \( \text{secure}(u, \phi, s') = \top \), \( s'' \equiv_{\text{data}} s' \), Lemma C.14, and Proposition C.9, it follows that \( \text{secure}_{\text{data}}(r, i-1.u \phi) \) holds. From this, it follows that \([\phi]^v = [\phi']^v \) for any \( v \in [s']^M_{i,u} \). Furthermore, from Proposition C.10 and \( \text{Ex}(s) = \emptyset \), it follows that \( \phi \) holds in \( s'' \). Let \( \mathcal{A}_{u, \text{R}, \tau} \) be the set \( \{ (\text{db}'(R \oplus T), U, \text{sec}, T, V) \in \Omega_M \mid (\text{db}', U, \text{sec}, T, V) \in [s'']^M_{i,u} \} \). It is easy to see that \([s'']^M_{i,u} \subseteq \mathcal{A}_{u, \text{R}, \tau} \). We now show that \( \phi \) holds for any \( z \in \mathcal{A}_{u, \text{R}, \tau} \). Let \( z \in [s^M_{i,u}]^M_{i,u} \). From \([\phi]^v = [\phi']^v \) for any \( v \in [s''^M_{i,u}]^M_{i,u} \) and the fact that \( \phi \) holds in \( s'' \), it follows that \([\phi']^z = \top \). Therefore, for any \( (k_1, k_2, k_3) \in R_{z}(z_1) \) such that \( [k_1] = [\mathcal{I}], [k_2] = [\mathcal{I}], \) and \( [k_3] = [\mathcal{I}] \), if \( k_1 = \mathcal{I} \), then \( k_2 = \mathcal{I} \). Then, for any \( (k_1, k_2, k_3) \in R_{z}(z_1) \cup \{(\mathcal{I}, \mathcal{I}, \mathcal{I})\} \) such that \( [k_1] = [\mathcal{I}], [k_2] = [\mathcal{I}], \) and \( [k_3] = [\mathcal{I}] \), if \( k_1 = \mathcal{I} \), then \( k_2 = \mathcal{I} \). Therefore, \( \phi \) holds also in \( z \in (R \oplus T) \in \text{sysState}(s''), \text{R}, \text{τ} \). Hence, \([\phi]^z = \top \) for any \( z \in \mathcal{A}_{u, \text{R}, \tau} \). From this and \([ s]^M_{i,u} \subseteq \mathcal{A}_{u, \text{R}, \tau} \), it follows that \([\phi]^s = \top \) for any \( z \in [s]^M_{i,u} \). From this, it follows that \( \text{secure}_{\text{data}}(r, i-1.u \phi) \) holds.

26. **Learn \texttt{INSERT} - FD - 1.** The proof of this case is similar to that of **Learn \texttt{INSERT} - FD.**

27. **Learn \texttt{INSERT} - ID.** The proof of this case is similar to that of **Learn \texttt{INSERT} - FD.** See also the proof of **INSERT Success - ID.**

28. **Learn \texttt{INSERT} - ID - 1.** The proof of this case is similar to that of **Learn \texttt{INSERT} - ID.**

29. **Learn \texttt{INSERT} Backward - 1.** Let \( i \) be such that \( r^i = r^{i-1}.t.s \), where \( s = \langle \text{db}', \text{U}, \text{sec}', T', V', c' \rangle \in \Omega_M \), \( \text{last}(r^{i-1}) = \langle \text{db}, \text{U}, \text{sec}, T, V, c \rangle \), and \( t \in \text{TRIGGER}_R \), and \( \phi \) be \( i \)’s actual \texttt{WHEN} condition, where all free variables are replaced with the values in \( \text{tp}(\text{last}(r^{i-1})) \). From the \( \phi \)’s definition, it follows that \( \text{secEx}(s) = \top. \) From this, the LTS semantics, and \( \text{secEx}(s) = \top. \) it follows that \( f(\text{last}(r^{i-1}), \langle u', \text{SELECT}, \phi \rangle) = \top. \) From this and \( f \)’s definition, it follows \( f_{\text{conf}}(\text{last}(r^{i-1}), \langle u', \text{SELECT}, \phi \rangle) = \top \) because \( \text{user}(\text{last}(r^{i-1}), \langle u', \text{SELECT}, \phi \rangle) = u \) since \( u \) is \( t \)’s invoker. From this and \( f_{\text{conf}} \)’s definition, it follows that \( \text{secure}(u, \phi, \text{last}(r^{i-1})) = \top \). From this and Proposition C.9, it follows \( \text{secure}_{\text{data}}(r, i-1.u \phi) \). We now prove our claim that \( [\phi]^\text{db} = \top \). Assume, for contradiction’s sake, that this is not the case. From this and the LTS rules, it follows that \( [\phi] = [\phi]^\text{db} \). From the \( \phi \)’s definition, it follows that there is a \( \psi \) such that \( r, i-1.u \psi \) and \( r, i-1.u \psi \). From this, Proposition C.1, \( \phi = \langle \text{db}', \text{U}, \text{sec}', T', V', c' \rangle \), and \( \text{last}(r^{i-1}) = \langle \text{db}, \text{U}, \text{sec}, T, V, c \rangle \), it follows that \([\phi] = \top \) and \([\phi]^\text{db} = \top \). Therefore, \([\phi] = \top \) and \([\phi]^\text{db} = \top \). Hence, \( \phi \neq [\phi]^\text{db} \), which contradicts \( \phi = [\phi]^\text{db} \).

31. **Learn \texttt{DELETE} Forward.** The proof of this case is similar to that of **Learn \texttt{INSERT} Forward.**

32. **Learn \texttt{DELETE} - ID.** The proof of this case is similar to that of **Learn \texttt{INSERT} - FD.** See also the proof of **DELETE Success - ID.**

33. **Learn \texttt{DELETE} - ID - 1.** The proof of this case is similar to that of **Learn \texttt{DELETE} - ID.**

34. **Learn \texttt{DELETE} Backward - 1.** The proof is similar to that of **Learn \texttt{INSERT} Backward - 1.**
35. **Learn DELETE Backward - 2.** The proof is similar to that of **Learn INSERT Backward - 2.**

36. **Propagate Forward Trigger Action.** Let i be such that r' = r_t-s, where t is a trigger, s = (db, U, sec, T, V, c) ∈ Ω_M and last(r_t-s) = (db', U', sec', T', V', c'). From the rule's definition, r, i − 1 ⊨_u φ holds. From this and the induction hypothesis, it follows that secure_P_u(r, i − 1 ⊨_u φ) holds. From Lemma C.24, the trigger t preserves the equivalence class with respect to r_t-s, P, and ≡_P_u. We claim that (1) the execution of t does not alter the content of the tables in tables(φ), and (2) executing the trigger in v does not throw integrity or security exceptions for any v ∈ [r_t-s]P_u. From this, Lemma C.21, and secure_P_u(r, i − 1 ⊨_u φ), it follows secure_P_u(r, i ⊨_u φ).

We now prove our claim that the execution of t does not alter the content of the tables in tables(φ). If the trigger is not enabled, the claim is trivial. In the following, we assume the trigger is enabled. There are four cases:

- t’s action is an INSERT statement. This case amount to claiming that the INSERT statement (u', INSERT, R, T) does not alter the content of the tables in tables(φ) in case revise(r_t-s, φ, r') = ⊤. We proved the claim in the **Propagate Forward INSERT/DELETE Success case.**
- t’s action is an DELETE statement. The proof is similar to that of the INSERT case.
- t’s action is an REVOKE statement. The proof is similar to that of the GRANT case.

We now show that executing the trigger t in any run v ∈ [r_t-s]P_u does not generate security or integrity exceptions. From the rule, it follows that executing the trigger t in r_t-s does not generate security and integrity exceptions. From this and the fact that t preserves the equivalence class (see Lemma C.24), it immediately follows that executing the trigger t in v does not generate security and integrity exceptions for any v ∈ [r_t-s]P_u. From this, Lemma C.21, and secure_P_u(r, i − 1 ⊨_u φ), it follows secure_P_u(r, i ⊨_u φ).

37. **Propagate Backward Trigger Action.** The proof of this case is similar to **Propagate Backward Trigger Action.**

38. **Propagate Forward INSERT Trigger Action.** Let i be such that r' = r_t-s, where t is a trigger, s = (db, U, sec, T, V, c) ∈ Ω_M and last(r_t-s) = (db', U', sec', T', V', c'). From the rule’s definition, r, i − 1 ⊨_u φ holds. From this and the induction hypothesis, it follows that secure_P_u(r, i − 1 ⊨_u φ) holds. From Lemma C.24, the trigger t preserves the equivalence class with respect to r_t-s, P, and ≡_P_u. We claim that (1) the execution of t does not alter the content of the tables in tables(φ), and (2) executing the trigger in v does not throw integrity or security exceptions for all v ∈ [r_t-s]P_u. From this, Lemma C.21, and secure_P_u(r, i − 1 ⊨_u φ), it follows secure_P_u(r, i ⊨_u φ).

We now prove our claim that the execution of t does not alter the content of the tables in tables(φ). If the trigger is not enabled, the claim is trivial. In the following, we assume the trigger is enabled. Then, t’s action is an INSERT statement. This case amount to claiming that the INSERT statement (u', INSERT, R, T) does not alter the content of the tables in tables(φ) in case r, i − 1 ⊨_u R(T) holds. We proved the claim in the **Propagate Forward INSERT Success - 1 case.** We now show that executing the trigger t in any run v ∈ [r_t-s]P_u does not generate security or integrity exceptions. From the rule, it follows that executing the trigger t in r_t-s does not generate security and integrity exceptions. From this and the fact that t preserves the equivalence class (see Lemma C.24), it immediately follows that executing the trigger t in v does not generate security and integrity exceptions for all v ∈ [r_t-s]P_u. From this, Lemma C.21, and secure_P_u(r, i − 1 ⊨_u φ), it follows secure_P_u(r, i ⊨_u φ).

39. **Propagate Forward DELETE Trigger Action.** The proof of this case is similar to that of **Propagate Forward INSERT Trigger Action.**

40. **Propagate Backward INSERT Trigger Action.** The proof of this case is similar to that of **Propagate Forward INSERT Trigger Action.**

41. **Propagate Forward DELETE Trigger Action.** The proof of this case is similar to that of **Propagate Forward INSERT Trigger Action.**

42. **Trigger FD INSERT Disabled Backward.** Let i be such that r' = r_t-s, where s = (db', U', sec', T', V', c') ∈ Ω_M, t ∈ TRIGGER_R_D, last(r_t-s) = (db, U, sec, T, V, c), and φ be t’s actual WHEN condition obtained by replacing all free variables with the values in tp(last(r_t-s)). Furthermore, let act = (u', INSERT, R, T, ⊥, ⊥) be t’s actual action and α be ≡_P_u.R(T, ⊥, ⊥) ∧ ⊥ ≠ ⊥. From the rule’s definition, it follows that secureEx(s) = ⊤. From this, it follows that f(last(r_t-s), (u', SELECT, φ)) = ⊤. From this and f’s definition, it follows f'_{conf}(last(r_t-s), (u', SELECT, φ)) = ⊤. From this and f'_{conf}’s definition, it follows that secure(u, φ, last(r_t-s)) = ⊤. From this, it follows that secure(u, φ, last(r_t-s)) = ⊤. From this and Proposition C.9, it follows secure_P_u(r, i − 1 ⊨_u φ).

43. **Trigger ID INSERT Disabled Backward.** The proof of this case is similar to that of **Trigger FD INSERT Disabled Backward.**

44. **Trigger ID DELETE Disabled Backward.** The proof of this case is similar to that of **Trigger FD INSERT Disabled Backward.**
This completes the proof of the induction step.

This completes the proof.

C.5 Complexity Proofs

Here we prove the complexity of our enforcement mechanism.

C.5.1 Complexity of f_int

Theorem C.3 states that f_int runs in constant time in terms of data complexity.

**Theorem C.3.** The data complexity of f_int is O(1).

**Proof.** Let \( M = \langle D, \Gamma \rangle \) be some fixed system configuration, \( a \in A_{D,U} \) be some fixed action, \( u \in U \) be some fixed user, \( U \subseteq U \) be some fixed set of users, \( sec \in \Omega_{U,D}^M \) be some fixed policy, \( T \) be some fixed set of triggers over \( D \) whose owners are in \( U \), \( V \) be some fixed set of views over \( D \) whose owners are in \( U \), and \( c \) be some fixed context. Furthermore, let \( db \in \Omega_{D}^M \) be a database state such that \( \langle db, U, sec, T, V, c \rangle \in \Omega_{M} \). We denote by \( s \) the state \( \langle db, U, sec, T, V, c \rangle \). By analyzing \( f_{\text{int}} \)'s definition in Section 6.8.2, it is immediate to see that the complexity of \( f_{\text{int}} \) directly depends on the complexity of computing \( \sim_{\text{app}} \). Observe that for all actions \( a \in A_{D,U} \), we can compute \( s \sim_{\text{app}} a \) in constant time in terms of data complexity. Observe also that \( f_{\text{int}} \) only uses checks statements of the form \( s \sim_{\text{app}} a \), where \( a \in A_{D,U} \). As a result, the overall data complexity is \( O(1) \).

**Lemma C.25.** The data complexity of the \( \text{apprDet} \) is \( O(1) \) as well.

**Lemma C.25.** The data complexity of \( \text{apprDet} \) is \( O(1) \).

**Proof.** Let \( M = \langle D, \Gamma \rangle \) be a system configuration, \( T \subseteq \mathcal{T} \) be a set of tables, \( V \subseteq \mathcal{V} \mathcal{E} \mathcal{W}^\text{worm} \) be a set of views over \( D \), \( \phi \) be a formula over \( D \), and \( s \) be an \( M \)-state. An algorithm that computes \( \text{apprDet}(T,V,\phi,s,M) \) is as follows:

1. Compute the set \( \text{extend}(M,s,V) \).
2. Compute the set \( S \) of all sub-formulae of \( \phi \), i.e., \( S = \text{subF}(\phi) \). Note that \( \phi \in \text{subF}(\phi) \).
3. Sort by length the set of sub-formulae in such a way that the shortest formula is the first one.
4. Let \( S' := \emptyset \).
5. For each sub-formula \( \psi \) in the sequence:
   a. Check whether there is a view \( v \in \text{extend}(M,s,V) \) such that \( \psi \) is \( v \)'s definition. If this is the case, let \( S' = S' \cup \text{subF}(\psi) \).
   b. Perform a case distinction on \( \psi \):
      i. If \( \psi := R(\pi) \) and \( R \in \mathcal{T} \), \( S' = S' \cup \text{subF}(\psi) \).
      ii. If \( \psi := V(\pi) \) and \( \{V, u, q, O \} \in \mathcal{V} \), \( S' = S' \cup \text{subF}(\psi) \).
      iii. If \( \psi := \alpha \land \beta \) and \( \alpha, \beta \in S' \), then \( S' = S' \cup \text{subF}(\psi) \).
      iv. If \( \psi := \alpha \lor \beta \) and \( \alpha, \beta \in S' \), then \( S' = S' \cup \text{subF}(\psi) \).
      v. If \( \psi := \neg \alpha \) and \( \alpha \in S' \), then \( S' = S' \cup \text{subF}(\psi) \).
      vi. If \( \psi := \exists x. \alpha \) and \( \alpha \in S' \), then \( S' = S' \cup \text{subF}(\psi) \).
      vii. If \( \psi := \forall x. \alpha \) and \( \alpha \in S' \), then \( S' = S' \cup \text{subF}(\psi) \).
6. \( \text{apprDet}(T,V,\phi,s,M) = \top \) iff \( S = S' \).

Observe that none of the above steps involve the database state \( db \). Therefore, \( \text{apprDet} \) can be executed in constant time in terms of data complexity.

C.5.2 Complexity of f_{conf}'

In this section, we prove that data complexity of \( f_{\text{conf}}' \) is \( AC^0 \). Note that the complexity class \( AC^0 \) identifies those problems that can be solved using constant-depth, polynomial-size boolean circuits with AND, OR, and NOT gates with unbounded fan-in [10]. Note also that, in the following, with \( AC^0 \) we usually refer to uniform-\( AC^0 \) [10]. Given a database schema \( D \) and a database state \( db \in \Omega_{D}^M \), the size of \( db \), denoted also as \( |db| \), is \( |db| = \sum_{R \subseteq D} |\Sigma_{R \subseteq D} |\Sigma_{\text{extVoc}(D,V)}|^{|R|} \) where the size \(|R|\) of a tuple \( R \) is just its cardinality. Similarly, the size of the schema \( D \), denoted \( |D| \), is \( |D| = \sum_{R \subseteq D} |\Sigma_{R \subseteq D}| \). Finally, given a set of views \( V \) over \( D \), the size of the extended vocabulary \( \text{extVoc}(D,V) \), denoted \( |\text{extVoc}(D,V)| \), is \( \sum_{R \subseteq D} \sum_{O \subseteq \Sigma^{|O|}} \frac{|O|!}{(|O| - |O|)!} \). Note that, given a view \( V \), we denote by \( |V| \) its cardinality.
Furthermore, given a RC-formula $\phi$, the size of $\phi$, denoted as $|\phi|$, is defined as follows:

$$
|\phi| = \begin{cases} 
1 + |\bar{\gamma}| & \text{if } \phi := R(\bar{\gamma}) \\
1 & \text{if } \phi := T \\
1 & \text{if } \phi := \bot \\
3 & \text{if } \phi := x = y \\
1 + |\psi| + |\gamma| & \text{if } \phi := \psi O \gamma \text{ and } O \in \{\lor, \land\} \\
1 + |\psi| & \text{if } \phi := \neg\psi \\
2 + |\psi| & \text{if } \phi := Q x. \psi \text{ and } Q \in \{\exists, \forall\}
\end{cases}
$$

Lemma C.27 shows that the rewritten formula $\phi^*_{s,u}$, for some $v \in \{T, \bot\}$, is linear in the size of the original formula $\phi$.

**Lemma C.26.** Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = (db, U, sec, T, V)$ be a partial $M$-state, $u \in U$ be a user, and $\phi$ be a $D$-formula. For all formulae $\phi$ and all $v \in \{T, \bot\}$, $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$.

**Proof.** Let $M = \langle D, \Gamma \rangle$ be a system configuration, $s = (db, U, sec, T, V)$ be a partial $M$-state, and $u \in U$ be a user. Let $\phi$ be an arbitrary formula over $D \cup V$ and $v$ be an arbitrary value in $\{T, \bot\}$. We now prove that $|\phi^*_{s,u}| \leq m|\phi|$ by induction over the structure of the formula $\phi$.

**Base Case** There are four cases:

1. $\phi := x = y$. In this case, $\phi^*_{s,u} = \phi$. From this, $|\phi^*_{s,u}| = |\phi|$. From this, it follows trivially that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$.

2. $\phi := T$. The proof of this case is similar to that of $\phi := x = y$.

3. $\phi := \bot$. The proof of this case is similar to that of $\phi := x = y$.

4. $\phi := R(\bar{\gamma})$. Without loss of generality, we assume that $v = T$. From this, it follows that $\phi^*_{s,u} = \bigvee_{S \in R^{-1}_{s,u}} S(\bar{\gamma})$. From this, it follows that $|\phi^*_{s,u}| = (|R^{-1}_{s,u}| - 1) + \sum_{S \in R^{-1}_{s,u}} |S(\bar{\gamma})|$. From this and $|S(\bar{\gamma})| = 1 + |\bar{\gamma}|$, it follows that $|\phi^*_{s,u}| = (|R^{-1}_{s,u}| - 1) + \sum_{S \in R^{-1}_{s,u}} (1 + |\bar{\gamma}|)$. From this, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$. From this and $|\phi^*_{s,u}| = |\phi| + (|R^{-1}_{s,u}| - 1)$, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$. From this, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$.

We now prove our claim that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$. The set $R^{-1}_{s,u}$ is a subset of $extVocabulary(D, V)$ by construction. The set $extVocabulary(D, V)$ contains any possible projection of tables in $D$ and views in $V$. It is easy to check that the cardinality of $extVocabulary(D, V)$ is, indeed, $(|extVoc(D, V)| + 1)$.

This completes the proof of the base case.

**Induction Step** Assume that our claim holds for all sub-formulae of $\phi$. We now show that our claim holds also for $\psi$. There are a number of cases depending on $\phi$'s structure.

1. $\phi := \psi \land \gamma$. From this, it follows that $\phi^*_{s,u} := \psi^*_{s,u} \land \gamma^*_{s,u}$. From this, it follows that $|\phi^*_{s,u}| = 1 + |\psi^*_{s,u}| + |\gamma^*_{s,u}|$. From the induction hypothesis, it follows that $|\psi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\psi|$ and $|\gamma^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\gamma|$. From this and $|\psi^*_{s,u}| = 1 + |\psi^*_{s,u}| + |\gamma^*_{s,u}|$, it follows that $|\phi^*_{s,u}| \leq 1 + (|extVoc(D, V)| + 1)|\psi| + (|extVoc(D, V)| + 1)|\gamma|$. From this and $|\phi^*_{s,u}| \geq 0$, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\psi| + (|extVoc(D, V)| + 1)|\gamma|$. From this, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$. From this and $|\phi^*_{s,u}| = 1 + |\psi^*_{s,u}| + |\gamma^*_{s,u}|$, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$.

2. $\phi := \neg\psi$. From this, it follows that $\phi^*_{s,u} := \neg\psi^*_{s,u}$. From this, it follows that $|\phi^*_{s,u}| = 1 + |\psi^*_{s,u}|$. From the induction hypothesis, it follows that $|\psi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\psi|$. From this and $|\phi^*_{s,u}| = 1 + |\psi^*_{s,u}|$, it follows that $|\phi^*_{s,u}| \leq 1 + (|extVoc(D, V)| + 1)|\psi|$. From this and $|\phi^*_{s,u}| \geq 0$, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\psi|$. From this, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$. From this and $|\phi^*_{s,u}| = 1 + |\psi^*_{s,u}|$, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$.

3. $\phi := \forall x. \psi$. If $\phi^*_{s,u}$ is $\neg\psi$, then the claim holds trivially since $|\phi^*_{s,u}| = 1$. In the following, we assume that $\phi^*_{s,u} := \exists x. \psi^*_{s,u}$. From this, it follows that $|\phi^*_{s,u}| = 2 + |\psi^*_{s,u}|$. From the induction hypothesis, it follows that $|\psi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\psi|$. From this and $|\phi^*_{s,u}| = 2 + |\psi^*_{s,u}|$, it follows that $|\phi^*_{s,u}| \leq 2 + (|extVoc(D, V)| + 1)|\psi|$. From this and $|\phi^*_{s,u}| \geq 0$, it follows that $|\phi^*_{s,u}| \leq 2 + (|extVoc(D, V)| + 1)|\psi|$. From this, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)(2 + |\psi|)$. From this and $|\phi^*_{s,u}| = 2 + |\psi|$, it follows that $|\phi^*_{s,u}| \leq (|extVoc(D, V)| + 1)|\phi|$.
This completes the proof of the induction step.

This completes the proof of our claim. \qed

Lemma C.27 states that our rewriting has size that is linear in the original formula's size.

**Lemma C.27.** Let \( M = \langle D, \Gamma \rangle \) be a system configuration, \( s = \langle \text{db}, U, \text{sec}, T, V \rangle \) be a partial M-state, \( u \in U \) be a user, and \( \phi \) be a D-formula. For all sentences \( \phi \) and all \( v \in \{ \top, \bot \} \), \( |\phi^+_u| \leq |\text{extVoc}(D, V)| + 1 \cdot |\phi| \) and \( |\neg \phi^+_u \land \phi^-_u| \leq 2(\text{|extVoc}(D, V)| + 1) \cdot |\phi| \).

**Proof.** Let \( M = \langle D, \Gamma \rangle \) be a system configuration, \( s = \langle \text{db}, U, \text{sec}, T, V \rangle \) be a partial M-state, \( u \in U \) be a user, and \( \phi \) be a D-formula. Furthermore, let \( \phi \) be a sentence and \( v \) be a value in \( \{ \top, \bot \} \). The fact that \( |\phi^+_u| \leq (|\text{extVoc}(D, V)| + 1) \cdot |\phi| \) follows trivially from Lemma C.26. Let \( \psi \) be the formula \( \neg \phi^+_u \land \phi^-_u \). The size of \( \psi \) is \( 2 + |\phi^+_u| + |\phi^-_u| \). From this and Lemma C.26, it follows that \( |\psi| \leq 2(\text{|extVoc}(D, V)| + 1) \cdot |\phi| \). From this, it follows that \( |\psi| \leq 2(\text{|extVoc}(D, V)| + 1) \cdot |\phi| \). This completes the proof. \qed

In the following, we study the data complexity of our PDP. Note that, given a PDP \( f \), the data complexity of \( f \) is the data complexity of the following decision problem:

**Definition C.2.** Let \( M = \langle D, \Gamma \rangle \) be some fixed system configuration, \( a \in \mathcal{A}_{D}, \gamma \) be some fixed action, \( u \in U \) be some fixed user, \( U \subseteq \mathcal{U} \) be some fixed set of users, \( sec \in \mathcal{O}_{D}^\text{sec} \) be some fixed policy, \( T \) be some fixed set of triggers over \( D \) whose owners are in \( U \), \( V \) be some fixed set of views over \( D \) whose owners are in \( U \), and \( c \) be some fixed context.

**Input:** A database state \( \text{db} \) such that \( \langle \text{db}, U, \text{sec}, T, V, c \rangle \in \Omega_M \).

**Question:** Is \( f(\langle \text{db}, U, \text{sec}, T, V, c \rangle, a) = \top \)?

We define in a similar way the data complexity of the secure procedure, which we analyze in Lemma C.28.

**Lemma C.28.** The data complexity of the secure procedure is \( AC^0 \).

**Proof.** Let \( M = \langle D, \Gamma \rangle \) be some fixed system configuration, \( \phi \) be some fixed sentence, \( u \in U \) be some fixed user, \( U \subseteq \mathcal{U} \) be some fixed set of users, \( sec \in \mathcal{O}_{D}^\text{sec} \) be some fixed policy, \( T \) be some fixed set of triggers over \( D \) whose owners are in \( U \), \( V \) be some fixed set of views over \( D \) whose owners are in \( U \), and \( c \) be some fixed context. Furthermore, let \( \text{db} \in \Omega_D \) be a database state such that \( \langle \text{db}, U, \text{sec}, T, V, c \rangle \in \Omega_M \). We denote by \( s \) the state \( \langle \text{db}, U, \text{sec}, T, V, c \rangle \). We can check whether \( \text{secure}(u, \phi, \langle \text{db}, U, \text{sec}, T, V, c \rangle) = \top \) as follows:

1. Compute the formula \( \phi^+_{u,\text{db}} \).
2. Compute \( [\phi^+_{u,\text{db}}]^\phi \).
3. \( \text{secure}(u, \phi, \langle \text{db}, U, \text{sec}, T, V, c \rangle) = \top \) iff \( [\phi^+_{u,\text{db}}]^\phi = \bot \).

We claim that the first step can be done in constant time in terms of data complexity. It is well-known that the data complexity of query execution is \( AC^0 \) [10]. From this, it follows that the data complexity of secure is also \( AC^0 \).

We now prove our claim that computing the formula \( \phi^+_{u,\text{db}} \) can be done in constant time in terms of data complexity. The extended vocabulary \( \text{extVocabulary}(D, V) \) does not depend on the database state. From this and the definition of \( R^*_c \), where \( R \) is a predicate symbol and \( \emptyset \) and \( \emptyset \) are the sets \( R^*_c \) (and the time needed to compute it) depends just on the database schema \( D \) and the set of views \( V \). The set \( AUTH_{s,\text{db}}^u \) and the time needed to compute it depend just on the size of the policy \( sec \). Furthermore, the time needed to compute \( AUTH_{s,\text{db}}^u \) depends just on the size of the policy \( sec \) and of the extended vocabulary. Therefore, for any predicate \( R \), the set \( R^*_c \) can be computed in constant time in terms of database size.

The computation of the formula \( \phi' \), obtained by replacing sub-formulae of the form \( x \in \mathcal{R}(\mathcal{X}, \mathcal{Y}) \) with the corresponding predicates in the extended vocabulary, can be done in linear time in terms of \( |\phi| \) and in constant time in terms of \( |\text{db}| \). Note that the size of the resulting formula is linear in \( |\phi| \). It is easy to see that also computing \( \phi^+_{u,\text{db}} \) and \( \phi^-_{u,\text{db}} \) can be done in linear time in terms of \( |\phi| \) and in constant time in terms of \( |\text{db}| \). As shown in Lemma C.27, the size of the resulting formula is linear in \( |\phi| \). Finally, we can replace the predicates in the extended vocabulary with the corresponding sub-formulae again in linear time in terms of \( |\phi| \). Note that, again, the size of the resulting formula is linear in \( |\phi| \). Therefore, the overall rewriting process can be done in linear time in the size of \( \phi \) and in constant time in the size of \( \text{db} \).

Lemma C.29 states that the data complexity of the \( f_{\text{conf},1,\emptyset}^u \) function is \( AC^0 \).

**Lemma C.29.** The data complexity of \( f_{\text{conf},1,\emptyset}^u \) is \( AC^0 \).
Proof. Let $M = (D, \Gamma)$ be some fixed system configuration, $a \in A_{D,U}$ be some fixed INSERT or DELETE action, $u \in U$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $\text{sec} \in \Omega_D^{\text{trig}, \text{sec}}$ be some fixed policy, $T$ be some fixed set of triggers over $D$ whose owners are in $U$, $V$ be some fixed set of views over $D$ whose owners are in $U$, and $c$ be some fixed context. Furthermore, let $db \in \Omega_D^\text{trig}$ be a database state such that $(db, U, sec, T, V, c) \in \Omega_M$. We can check whether $f_{\text{conf}, \text{t}}^u ((db, U, sec, T, V, c), a) = \top$ as follows:

1. If trigger$(s) = \epsilon$ and $a \notin A_{D,u}$, return $\top$.
2. If trigger$(s) \neq \epsilon$ and invoker$(s) \neq u$, return $\top$.
3. Compute the result of noLeak$(s, a, u)$. If noLeak$(s, a, u) = \bot$, then return $\bot$.
4. Compute the set $\text{Dep} (\Gamma, a)$.
5. Compute $\text{secure} (u, \text{getInfo} (a), s)$. If its result is $\bot$, return $\bot$.
6. For each $\gamma \in \text{Dep} (\Gamma, a)$, compute $\text{secure} (u, \text{getInfo} (a, \gamma), s)$. If its result is $\bot$, return $\bot$.
7. For each $\gamma \in \text{Dep} (\Gamma, a)$, compute $\text{secure} (u, \text{getInfo} (a, \gamma), s)$. If its result is $\bot$, return $\bot$.
8. Return $\top$.

The data complexity of the steps 1 and 2 is $O(1)$. We claim that also the data complexity of the third step is $O(1)$. The complexity of the fourth step is $O(|\Gamma|)$ and therefore its data complexity is again $O(1)$. From the definition of $\text{getInfo}$, the resulting formula is constant in the size of the database. Furthermore, also constructing the formula can be done in constant time in the size of the database. From this and Lemma C.28, it follows that the data complexity of the fifth step is $AC^0$. For a similar reason, the data complexity of the sixth and seventh steps is also $AC^0$. Therefore, the overall data complexity of the $f_{\text{conf}, \text{t}}^u$ procedure is $AC^0$.

We now prove our claim that the data complexity of the $\text{noLeak}$ procedure is $O(1)$. An algorithm implementing the $\text{noLeak}$ procedure is as follows: for each view $v \in V$, for each grant $g \in \text{sec}$, if $g = (\text{op}, u, (\text{SELECT}, v), u')$, then (1) compute the set $t\text{Det}(v, s, M)$. (2) if $R \in t\text{Det}(v, s, M)$, for each $\alpha \in t\text{Det}(v, s, M)$, check whether $(\text{op}, u, (\text{SELECT}, \alpha), u''') \in \text{sec}$. The size of the set $t\text{Det}(v, s, M)$ is at most $|D|$ and its computation depends just on $\text{sec}$. From this, it follows that the complexity of the step 1. (b) is $O(1)$. From Lemma C.25 and the definition of $t\text{Det}$, the data complexity of computing $t\text{Det}(v, s, M)$ is $O(1)$, and the rest of the computation does not depend on $db$. The overall data complexity is, therefore, $O(1)$.

Lemma C.30 states that the data complexity of the $f_{\text{conf}, \text{t}}^u$ function is $O(1)$.

Lemma C.30. The data complexity of $f_{\text{conf}, \text{t}}^u$ is $O(1)$.

Proof. Let $M = (D, \Gamma)$ be some fixed system configuration, $a \in A_{D,U}$ be some fixed GRANT action, $u \in U$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $\text{sec} \in \Omega_D^{\text{trig}, \text{sec}}$ be some fixed policy, $T$ be some fixed set of triggers over $D$ whose owners are in $U$, $V$ be some fixed set of views over $D$ whose owners are in $U$, and $c$ be some fixed context. Furthermore, let $db \in \Omega_D^\text{trig}$ be a database state such that $(db, U, sec, T, V, c) \in \Omega_M$. We can check whether $f_{\text{conf}, \text{t}}^u ((db, U, sec, T, V, c), \langle \text{op}, u'', p, u'' \rangle)) = \top$ as follows:

1. If trigger$(s) = \epsilon$ and $a \notin A_{D,u}$, return $\top$.
2. If trigger$(s) \neq \epsilon$ and invoker$(s) \neq u$, return $\top$.
3. If $p$ is not a SELECT privilege, return $\top$.
4. If $u'' \neq u$, return $\top$.
5. For each $g \in \text{sec}$, if $g = (\text{op}, u, p, u')$, return $\top$.
6. Return $\bot$.

The complexity of the fifth step is $O(|\text{sec}|)$, whereas the complexity of the other steps is $O(1)$. Therefore, the overall complexity of the $f_{\text{conf}, \text{t}}^u$ procedure is $O(|\text{sec}|)$. From this, it follows that the data complexity of the $f_{\text{conf}, \text{t}}^u$ procedure is $O(1)$.

Lemma C.31 states that the data complexity of the $f_{\text{conf}, \text{t}}^u$ function is $AC^0$.

Lemma C.31. The data complexity of $f_{\text{conf}, \text{t}}^u$ is $AC^0$.

Proof. Let $M = (D, \Gamma)$ be some fixed system configuration, $a \in A_{D,U}$ be some fixed SELECT action $(u', \text{SELECT}, \phi), u \in U$ be some fixed user, $U \subseteq \mathcal{U}$ be some fixed set of users, $\text{sec} \in \Omega_D^{\text{trig}, \text{sec}}$ be some fixed policy, $T$ be some fixed set of triggers over $D$ whose owners are in $U$, $V$ be some fixed set of views over $D$ whose owners are in $U$, and $c$ be some fixed context. Furthermore, let $db \in \Omega_D^\text{trig}$ be a database state such that $(db, U, sec, T, V, c) \in \Omega_M$. We can check whether $f_{\text{conf}, \text{t}}^u ((db, U, sec, T, V, c), a) = \top$ as follows:

1. If trigger$(s) = \epsilon$ and $a \notin A_{D,u}$, return $\top$.
2. If trigger$(s) \neq \epsilon$ and invoker$(s) \neq u$, return $\top$.
3. Compute $\text{secure} (u, \phi, s)$ and return its result.
The complexity of the first and second steps is $O(1)$. From Lemma C.28, it follows that the data complexity of the third step is $AC^0$. From this, it follows that the data complexity of $f_{\text{conf}}^u$ procedure is $AC^0$.

Finally, Theorem C.4 states that the data complexity of $f_{\text{conf}}^u$ is $AC^0$.

**Theorem C.4.** The data complexity of $f_{\text{conf}}^u$ is $AC^0$.

**Proof.** Let $M = \langle D, \Gamma \rangle$ be some fixed system configuration, $a \in A_{D,U}$ be some fixed action, $u \in U$ be some fixed user, $U \subseteq U$ be some fixed set of users, $sec \in \Omega_{U,D}$ be some fixed policy, $T$ be some fixed set of triggers over $D$ whose owners are in $U$, $V$ be some fixed set of views over $D$ whose owners are in $U$, and $c$ be some fixed context. The data complexity of $f_{\text{conf}}^u$ is the maximum of the data complexities of $f_{\text{conf},1}^u$, $f_{\text{conf},2}^u$, and $f_{\text{conf},3}^u$. From Lemmas C.29–C.31, it follows that: (a) the data complexity of $f_{\text{conf},1}^u$ is $AC^0$, (b) the data complexity of $f_{\text{conf},2}^u$ is $AC^0$, and (c) the data complexity of $f_{\text{conf},3}^u$ is $O(1)$. From this, it follows that the data complexity of $f_{\text{conf}}^u$ is $max(AC^0, O(1))$. Hence, the data complexity of $f_{\text{conf}}^u$ is $AC^0$.

**C.5.3 Complexity of the overall algorithm**

Here we prove the data complexity of the PDP $f$.

**Theorem C.5.** The data complexity of $f$ is $AC^0$.

**Proof.** From $f$’s definition, it follows that $f$’s data complexity is the maximum complexity between $f_{\text{conf}}^u$’s complexity and $f_{\text{int}}^u$’s complexity. From this, Theorem C.3, and Theorem C.4, it follows that the data complexity of $f$ is $AC^0$. 


Appendix D

Proofs for Chapter 7

We remark that in the following we consider only sequences of queries, *WhileSQL* programs, and configurations that are well-defined, i.e., that do not get stuck according to the *WhileSQL*’s semantics. Observe that this requires, for instance, that all views are used only after their declaration.

D.1 From Database Access Control to Information-flow Control

Here, we prove the main results from Section 7.5, namely Theorems 7.1 and 7.2. In this section, we refer to terminology, notation, and auxiliary functions taken from Chapters 5 and 6.

D.1.1 Weak indistinguishability

We now introduce a new notion of indistinguishability among database runs. This notion plays a key role in our proofs. Observe that this indistinguishability notion is strictly weaker than the one taken in Chapter 6.

**Definition D.1.** Let $M$ be a system configuration, $f$ be an $M$-PDP, $L$ be the $(M, f)$-LTS, and $u$ be a user. We say that two $M$-system states $s = \langle db, U, \text{sec}, T, V \rangle$ and $s' = \langle db', U', \text{sec}', T', V' \rangle$ are configuration indistinguishable, written $s \cong_{\text{conf}} s'$, iff $U = U'$, $\text{sec} = \text{sec}'$, $T = T'$, and $V = V'$. Moreover, two runs $r$ and $r'$ in $\text{traces}(L)$ are $(M, f, u)$-weakly indistinguishable, written $r \cong_{\text{w}} (M, f) r'$, iff

1. $r_u$ and $r'_u$ are consistent.
2. If $r_u^{i+1} = r_u^{i+1} \cdot s' \cdot a \cdot s$ and $a \neq s$, then $\text{sysState}(\text{last}(r_u^{i+1}))$ and $\text{sysState}(\text{last}(r_u^{i+1}))$ are $(M, u)$-data indistinguishable.
3. If $r_u^{i+1} = r_u^{i+1} \cdot s' \cdot a \cdot s$ and $a \neq s$, then $\text{sysState}(\text{last}(r))$ and $\text{sysState}(\text{last}(r'))$ are $(M, u)$-data indistinguishable.
4. For all $i$ such that $1 \leq i \leq |r_u| - 2$, if $r_u^{i+1} = r_u^{i+1} \cdot s' \cdot a \cdot s$ and $a \neq s$, then $\text{sysState}(\text{last}(r_u^{i+1}))$ and $\text{sysState}(\text{last}(r_u^{i+1}))$ are $(M, u)$-data indistinguishable.
5. For all $i$ such that $1 \leq i \leq |r_u| - 1$, if $r_u^{i+1} = r_u^{i+1} \cdot s' \cdot a \cdot s$, and $s \in \Omega_M$, then $\text{sysState}(\text{last}(r_u^{i+1}))$ and $\text{sysState}(\text{last}(r_u^{i+1}))$ are configuration indistinguishable.
6. $\text{sysState}(\text{last}(r))$ and $\text{sysState}(\text{last}(r'))$ are configuration indistinguishable.

We denote by $[r]_{\cong_{\text{w}}}^{W}$ the equivalence class defined by $r$ with respect to $\cong_{\text{w}}^{W}$.

D.1.2 Auxiliary notation

Here, we introduce some notation and machinery for (1) moving between *WhileSQL* runs and database runs from Chapter 5, and (2) moving between a set of initial states (at a certain point in the run) and the final states obtained by executing a given sequence of queries.

**Extension.** Let $M = (D, \Gamma)$ be a system configuration, $f$ be an $M$-PDP, $L$ be the $(M, f)$-LTS, and $r$ be a run in $\text{traces}(L)$. Furthermore, let $(u, q)$ be a query. We denote by $\text{extend}(\langle u, q \rangle)$ the run obtained by extending $r$ with $(u, q)$ (if such a run exists). Furthermore, we denote by $\text{extend}^\ast(r, \langle u, q \rangle)$ the run obtained by extending $r$ with the action $(u, q)$ and all triggers executed in response to $(u, q)$. Similarly, given a trigger $t$, we denote by $\text{extend}(r, t)$ the run obtained by extending $r$ with $t$ (if such a run exists). We refer the reader to Appendix C for a formal definition of extend. For conciseness, in the following we use $e$ and $e^\ast$ instead of extend and extend$^\ast$.

**Notation for database access control.** Let $M = (D, \Gamma)$ be a system configuration, $f$ be an $M$-PDP, and $L$ be the $(M, f)$-LTS. The trigger-free projection of a run $r$, denoted $\mu(r)$, is defined as follows:

$$
\mu(e) = e, \mu(s) = s \text{ where } s \in \Omega_M,
$$

$$
\mu(r') = \begin{cases}
\mu(r') \cdot a \cdot s & r' \in \text{traces}(L) \land s \in \Omega_M \land a \in A_{D, L} \land r = r' \cdot a \cdot s \land \neg \text{ID}(a) \\
\mu(r') \cdot a \cdot s & r' \in \text{traces}(L) \land s, s_1, \ldots, s_n \in \Omega_M \land t_1, \ldots, t_n \in \text{TRIGGER} \land a \in A_{D, L} \land r = r' \cdot a \cdot s_1 \cdot t_1 \cdot s_2 \ldots \cdot s_n \cdot t_n \cdot s \land \text{ID}(a)
\end{cases}
$$

233
where $ID(a)$ returns $\top$ iff $a$ is an INSERT or DELETE action.

**Notation for triggers.** Given a run $r \in \text{traces}(L)$, we denote by $\text{triggers}(r,i)$ the sequence of triggers executed in $r$ in response to the $i$-th query.

**Notation for Reduction 7.1.** We denote by $\text{code}(\text{ctx},q,u)$ (respectively $\mathcal{S}(\text{ctx},q,u)$ and $\text{mem}(\text{ctx},q,u)$) the programs $c_1 \ldots c_n$ (respectively the scheduler $\mathcal{S}$ and the memories $m_1 \ldots m_n$) constructed according to Reduction 7.1.

Let $M = (D, \Gamma)$ be a system configuration, $\text{idEnf}$ be an enforcement mechanism for WHILESQL programs that is sound and stable (as required by Theorem 7.1), $u$ be a user, and $\text{dbEnf}$ be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = (M, \text{dbEnf})$ be an extended configuration, $L$ be the $P$-LTS, and $(s, \text{ctx})$ be the last state of $r$, i.e. $(s, \text{ctx}) = \text{last}(r)$. In the following, we use $f$ to refer to $\text{dbEnf}$. With a slight abuse of notation, given a run $r \in \text{traces}(L)$ and an action $(u,q)$, we write $[r,(u,q)]^{\text{idEnf}}$ to denote the run generated by the WHILESQL programs produced by our reduction and starting from the state and scheduler resulting from Reduction 7.1. Observe that each program produced by our reduction is processed by the WHILESQL's semantics in two steps, namely one application of the $\text{M-EVAL-STEP}$ rule and one application of the $\text{M-EVAL-END}$ rule. Abusing notation, in the following we ignore the steps associated with the $\text{M-EVAL-END}$ rule. Therefore, the $i$-th statement executed in the run $[r,(u,q)]^{\text{idEnf}}$ corresponds to the $i$-th command executed in $\mu(e(r,(u,q)))$ and $[\mu(e(r,(u,q)))]$.

**Notation for epochs.** Let $ep$ be a user-based epoch (as defined by the predicates given in Section 7.5) in $[r,(u,q)]^{\text{idEnf}}$ and $i$ be a natural number such that $1 \leq i \leq |ep|$. We denote by $\text{start}(ep)$ the first state in the epoch and by $\text{end}(ep)$ the last state in the epoch. Since our reduction does not use the memory, with a slight abuse of notation, we ignore memories. For instance, we assume that $\text{start}(ep)$ and $\text{end}(ep)$ consist just of database states. Similarly, we write $\text{idEnf}(C,S,u,s)$ instead of $\text{idEnf}(C,M,S,u,s)$. Furthermore, we denote by $\text{code}(ep)$ the initial WHILESQL program associated with the epoch $ep$, and $\mathcal{S}(ep)$ the initial scheduler in $ep$. We also denote by $\text{trace}(ep)$ the trace associated with $ep$ and by $\text{trace}(ep,i)$ the trace obtained after executing $i$ queries in $ep$. Finally, we denote by $PK_u(ep,i)$ the knowledge $PK_u(\text{start}(ep), \text{code}(ep), \mathcal{S}(ep), \text{trace}(ep,i))$.

Given a set of runs $R$ and a sequence of queries $\vec{q} = \langle q_1, q_2 \rangle \ldots \langle q_n, q_n \rangle$, we denote by $\langle \vec{q}, \vec{r} \rangle$ the set of states obtained by repeatedly extending all runs in $R$ with the commands in $\vec{q}$. Furthermore, we denote by $\{ \text{dbEnf}(R) \}$ the set $\{ \text{dbEnf}(r) \mid r \in R \}$.

### D.1.3 Proofs about weak indistinguishability

We now prove that the indistinguishability notion from Chapter 6 implies weak indistinguishability (or, equivalently, that the equivalence class defined by the indistinguishability notion is included in the one defined by weak indistinguishability).

**Proposition D.1.** Let $M$ be a system configuration, $f$ be an $M$-PDP, $L$ be the $(M,f)$-LTS, and $u$ be a user. Furthermore, let $r, r'$ be two runs in $\text{traces}(L)$. The following facts hold:

1. If $r \equiv_{w,M,f} r'$, then $r \equiv_{u,M,f} r'$.

2. $[r]^{w}_{u,M,f} \subseteq \{ r' \in \text{traces}(L) \mid r \equiv_{u,M,f} r' \}$.

**Proof.** We first show that the second claim follows from the first one. Assume, for contradiction’s sake, that $[r]^{w}_{u,M,f} \not\subseteq \{ r' \in \text{traces}(L) \mid r \equiv_{u,M,f} r' \}$. Therefore, there is a run $r'' \in [r]^{w}_{u,M,f}$ such that $r'' \not\in \{ r' \in \text{traces}(L) \mid r \equiv_{u,M,f} r' \}$. From this, it follows that $r \not\equiv_{u,M,f} r'$ and $r \not\equiv_{w,M,f} r'$, which contradicts the first claim.

We now prove that if $r \equiv_{w,M,f} r'$, then $r \equiv_{u,M,f} r'$. Let $M$ be a system configuration, $f$ be an $M$-PDP, $L$ be the $(M,f)$-LTS, and $u$ be a user. Furthermore, let $r, r'$ be two runs in $\text{traces}(L)$ such that $r \equiv_{w,M,f} r'$. From this, it follows:

(a) $r|_u$ and $r'|_u$ are consistent, and

(b) $\text{sysState}(\text{last}(r))$ and $\text{sysState}(\text{last}(r'))$ are $(M,u)$-data indistinguishable, and

(c) for all $i$ such that $1 \leq i \leq |r|_u - 1$, if $r|_u^{i+1} = r|_u^{i+1}$, $a \neq \ast$, and $s \in \Omega_M$, then $\text{sysState}(\text{last}(r|_u^{i}))$ and $\text{sysState}(\text{last}(r'|_u^{i}))$ are $(M,u)$-data indistinguishable.

From (b), it follows that $\text{sysState}(\text{last}(r))$ and $\text{sysState}(\text{last}(r'))$ are configuration indistinguishable (since data indistinguishability implies configuration indistinguishability). From (c), it follows that (1) if $r|_u^{i} = s\cdot a\cdot s$ and $a \neq \ast$, then $\text{sysState}(\text{last}(r|_u^{i}))$ and $\text{sysState}(\text{last}(r'|_u^{i}))$ are $(M,u)$-data indistinguishable, (2) if $r|_u^{iência} = r|_u^{iência}$, $\ast \cdot s\cdot a\cdot s$, then $\text{sysState}(\text{last}(r))$ and $\text{sysState}(\text{last}(r'))$ are $(M,u)$-data indistinguishable, and (3) for all $i$ such that $1 \leq i \leq |r|_u - 2$, if $r|_u^{i+2} = r|_u^{i+2}$, $s\cdot a\cdot s$ and $a \neq \ast$, then $\text{sysState}(\text{last}(r|_u^{i+1}))$ and $\text{sysState}(\text{last}(r'|_u^{i+1}))$ are $(M,u)$-data indistinguishable, and (4) for all $i$ such that $1 \leq i \leq |r|_u - 1$, if $r|_u^{iência} = r|_u^{iência}$, $a \neq \ast$, and $s \in \Omega_M$, then $\text{sysState}(\text{last}(r|_u^{i}))$ and $\text{sysState}(\text{last}(r'|_u^{i}))$ are data indistinguishable, which implies that $\text{sysState}(\text{last}(r|_u^{i}))$ and $\text{sysState}(\text{last}(r'|_u^{i}))$ are configuration indistinguishable. Therefore, it follows that $r \equiv_{w,M,f} r'$. □
Using Proposition D.1, we now show that judgment security with respect to weak indistinguishability implies judgment security with respect to Chapter 6's indistinguishability. The same applies to data confidentiality.

**Proposition D.2.** Let $M$ be a system configuration, $f$ be an M-PDP, $P = (M, f)$, $L$ be the P-LTS, $A$ be an attacker model, and $u$ be a user. The following facts hold:

1. $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$ implies $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$.

2. If $f$ provides data confidentiality with respect to $P$, $u$, $A$, and $\mathbb{W}_u$, then $f$ provides data confidentiality with respect to $P$, $u$, $A$, and $\mathbb{W}_u$.

**Proof.** Let $M$ be a system configuration, $f$ be an M-PDP, $P = (M, f)$, $L$ be the P-LTS, $A$ be an attacker model, and $u$ be a user. We first prove that $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$ implies $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$. Assume, for contradiction's sake, that this is not the case. From $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$, it follows that for all $r' \in \text{traces}(L)$ such that $r' \not\preceq_{P, \mathbb{W}_u} r'$, it holds that $[\phi]^{db} = [\phi]^{av}$, where $\text{last}(r') = (db, U, S, T, V, c)$ and $\text{last}(r') = (db', U', S', T', V', c')$. From the fact that $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$ does not hold, it follows that there is a run $r' \in \text{traces}(L)$ such that $r' \not\preceq_{P, \mathbb{W}_u} r'$ and $[\phi]^{db} \neq [\phi]^{av}$, where $\text{last}(r') = (db, U, S, T, V, c)$ and $\text{last}(r') = (db', U', S', T', V', c')$. From this and Proposition D.1, it follows that there is a run $r' \in \text{traces}(L)$ such that $r' \not\preceq_{P, \mathbb{W}_u} r'$ and $[\phi]^{db} \neq [\phi]^{av}$, where $\text{last}(r') = (db, U, S, T, V, c)$ and $\text{last}(r') = (db', U', S', T', V', c')$. This contradicts $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$.

We now prove the second claim. Assume that $f$ provides data confidentiality with respect to $P$, $u$, $A$, and $\mathbb{W}_u$. From this, it follows that $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$ holds for all judgments $r, i \vdash_u \phi$ that hold in $A$. From this and the first claim, it follows that $\text{secure}_{P, \mathbb{W}_u}^M(r, i \vdash_u \phi)$ for all judgments $r, i \vdash_u \phi$ that hold in $A$. From this, it follows that $f$ provides data confidentiality with respect to $P$, $u$, $A$, and $\mathbb{W}_u$.

**D.1.4 Correctness of Reduction 7.1**

Here we prove some facts about Reduction 7.1 and how it relates to the database runs.

**Proposition D.3.** Let $M = (D, \Gamma)$ be a system configuration, $j\text{Enf}$ be an enforcement mechanism for WHILESQL programs, $u$ be a user, and $db\text{Enf}$ be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = (M, db\text{Enf})$ be the extended configuration and $L$ be the P-LTS. Let $r$ be a run in traces$(L)$ and $(u, q)$ be a database command such that $e(r, (u, q))$ is defined. The run $r_1 = [r, (u, q)]^{av}$ produced by executing code$(ctx, q, u)$ starting from $(\text{mem}(ctx, q, u), \text{init}(s, ctx))$ with scheduler $S$(ctx, q, u), where ctx is the context in the last state of $r$ and $s$ is the last state in $r$, satisfies the following conditions:

1. For $1 \leq i \leq |u, q| - 1$, the $i$-th query executed in $e(r, (u, q))$ is $\langle u, q_1 \rangle$ iff the $i$-th statement executed in $r_1$ is $\langle u, x \leftarrow q_1 \rangle$ if $\text{secEx}(\text{last}(\mu(e(r, (u, q))^{i+1})) = \bot$ and $\text{skip}$ otherwise.

2. The $|\mu(e(r, (u, q)))| - 1$-th query executed in $e(r, (u, q))$ is $\langle u, q_1 \rangle$ iff the $|\mu(e(r, (u, q)))| - 1$-th statement executed in $r_1$ is $\langle u, x \leftarrow q_1 \rangle$.

3. For all $1 \leq i \leq |\mu(e(r, (u, q)))|$, (1) $\mu(e(r, (u, q))^{i}) = \langle s, ctx \rangle$ if the $i$-th configuration in $r_1$ is $\langle C, M, \langle s, ctx, S \rangle, S \rangle$, and (2) inTrigger$(ctx) = \text{inTrigger}(ctx') = \bot$.

4. For all $1 \leq i \leq |\mu(e(r, (u, q)))| - 1$, if the $i$-th query $(u, q_1)$ in $\mu(e(r, (u, q)))$ is authorized and no triggers associated with $q_1$ throw integrity or security exceptions (i.e., $\text{secEx}(\text{last}(\mu(e(r, (u, q))^{i+1})) = \bot$ and $\text{Ex}(\text{last}(\mu(e(r, (u, q))^{i+1})) = \bot$), then the result of $(u_1, q_1)$ in $\mu(e(r, (u, q)))$ is $\text{K}$ if the $i$-th step in $r_1$ is associated with the label $(u', q, k, \tau)$, where $u' = \text{Usr}(u_1, x \leftarrow q_1)$.

5. For all $1 \leq i \leq |\mu(e(r, (u, q)))| - 1$, if the $i$-th query $(u, q_1)$ in $\mu(e(r, (u, q)))$ is authorized (i.e., $\text{secEx}(\text{last}(\mu(e(r, (u, q))^{i+1})) = \bot$) and the query itself has caused an exception (i.e., $\text{Ex}(\text{last}(\mu(e(r, (u, q))^{i+1})) \neq \bot$ and triggers$(e(r, (u, q)), i) = e$), then $\text{Ex}(\text{last}(\mu(e(r, (u, q))^{i+1})) = \text{K}$ if the $i$-th step in $r_1$ is associated with the label $(db(u, q), e, \text{IntEx}(K), \tau)$.

6. For all $1 \leq i \leq |\mu(e(r, (u, q)))| - 1$, if the $i$-th query $(u, q_1)$ in $\mu(e(r, (u, q)))$ is authorized (i.e., $\text{secEx}(\text{last}(\mu(e(r, (u, q))^{i+1})) = \bot$ and the trigger $t$ executed in response to $q$ has caused an exception (i.e., $\text{Ex}(\text{last}(\mu(e(r, (u, q))^{i+1})) \neq \bot$ and triggers$(e(r, (u, q)), i) = t$), then $\text{Ex}(\text{last}(\mu(e(r, (u, q))^{i+1})) = \text{K}$ if the $i$-th step in $r_1$ is associated with the label $(db(u, q), t, \text{IntEx}(K), \tau)$.

7. For all $1 \leq i \leq |\mu(e(r, (u, q)))| - 1$, the label $(u, q, m, \tau, \langle \text{public}, t, q, \tau \rangle)$ is associated with the $i$-th step in $r_1$ iff the trigger $t$ is successfully executed in response to the $i$-th query in $\mu(e(r, (u, q)))$, its condition is enabled, and it modifies the security policy (and it has been the case for $|\tau|+1$-th trigger to do so in the $i$-th step).

**Proof.** Let $M = (D, \Gamma)$ be a system configuration, $j\text{Enf}$ be an enforcement mechanism for WHILESQL programs, $u$ be a user, and $db\text{Enf}$ be the database access control mechanism constructed using
Reduction 7.1. Furthermore, let $P = \{M, dbEnf\}$ be the extended configuration and $L$ be the P-LTS. Let $r$ be a run in $\text{LTS}(L)$ and $(u, q)$ be a database command such that $e(r, (u, q))$ is defined. Finally, let $r_1 = [r, (u, q)]^{(r)}_k$ be the run produced by executing $\text{code}(ctx, q, u)$ starting from $(\text{mem}(ctx, q, u), \text{init}(s, ctx))$ with scheduler $S(ctx, q, u)$, where $(s, ctx)$ is the database state in the last state of $r$.

**Proof of (1).** The first claim directly follows from the construction of Reduction 7.1. The reduction associates a WHILESQL program $c_i = (u_i, s_i)$ to each query $(u, q)$ in $\mu(r)$ such that $s_i$ is $x \leftarrow q_i$ if $q_i$ is authorized (i.e., $\text{secEx}(\text{last}(\mu(r)^{i+1})) = \bot$) and $s_i = \text{skip}$ otherwise. Furthermore, the order in which the WHILESQL programs are executed is directly dictated by the run $e(r, (u, q))$. In particular, the $i$-th step in $r_1$ is associated to the program $c_i$, i.e., to the $i$-th query in $e(r, (u, q))$ (and all the associated triggers). Let $i$ be a value such that $1 \leq i < |\mu(e(r, (u, q)))| - 1$. The $i$-th statement executed in $r_1$ is exactly the program $c_i$. From this, it follows that the $i$-th query executed in $\mu(e(r, (u, q)))$ is $(u_i, q_i)$ iff the $i$-th statement executed in $r_1$ is $(u_i, x \leftarrow q_i)$ if $\text{secEx}(\text{last}(\mu(e(r, (u, q)))^{i+1})) = \bot$ and skip otherwise (by construction). This completes the proof of the first claim.

**Proof of (2).** The proofs of the second claim also directly follows from the construction of Reduction 7.1. From $e(r, (u, q))$ being defined, it follows that $\text{inTrigger}(ctx) = \bot$. From this, the last query in $L'$ is not part of the history in ctx. Therefore, the program $c_i |\mu| (u, x \leftarrow q_i)$ together with $|\mu| = \mu(e(r, (u, q)) \{1\} - 1)$ gives the second claim.

**Proof of (3).** We now prove, by induction on $i$, that for all $1 \leq i \leq |\mu(e(r, (u, q)))|$, $\text{last}(\mu(r)^i, (u, q))^{(i)}) = (s, ctx)$ iff the $i$-th configuration in $r_1$ is $\{C, M, (s, ctx'), S\}$, $\text{inTrigger}(ctx) = \bot$, and $\text{inTrigger}(ctx') = \bot$. The proof of the base case $i = 1$ is trivial. Indeed, the reduction directly extracts the initial system state from $s$ and ctx. From this, the initial configuration in $r_1$ is $\{c code(ctx, q, u), M, s, S(ctx, q, u)\}$ and $s = \text{last}(\mu(e(r, (u, q)))^{(i)})$ by construction. For the induction’s step, assume that $\text{last}(\mu(e(r, (u, q)))^{(j)}) = (s, ctx)$ and the $j$-th configuration in $r_1$ is $\{C', M', (s', ctx'), S'\}$ and $\text{inTrigger}(ctx') = \bot$ for any $j < i$. We now prove that $\text{last}(\mu(e(r, (u, q)))^{(i)}) = (s, ctx)$. There are two cases:

1. Assume that $c_{i-1} = (u_{i-1}, \text{skip})$. From this, it directly follows that the $i$-th configuration in $r_1$ is $\{C, M, (s, ctx'), S\}$. From this and the E-Skip and M-Eval-Step rules, it follows that $(s, ctx')$ and the $i$-th configuration are the same as those in the $(i - 1)$-th configuration in $r_1$. Furthermore, from the first two claims, it follows that the $(i - 1)$-th query (or one of the associated triggers) in $\mu(e(r, (u, q)))^{(i-1)}$ has been authorized. Therefore, from the database operational semantics, it follows that $(1)$ $s$ is the same as the $(i - 1)$-th database state in $e(r, (u, q))$, and $(2)$ $\text{inTrigger}(ctx) = \bot$. From these facts, we can derive that $\text{last}(\mu(e(r, (u, q)))^{i}) = (s, ctx)$ and the $i$-th configuration in $r_1$ is $\{C, M, (s, ctx', S\}$ and $\text{inTrigger}(ctx) = \text{inTrigger}(ctx') = \bot$ (by applying the induction hypothesis to the $(i - 1)$-th states).

2. Assume that $c_{i-1} = (u_{i-1}, x \leftarrow q_{i-1})$. From this and the WHILESQL’s semantics, it follows that the $(s, ctx')$ is the database state obtained by executing the query $(u_{i-1}, q_{i-1})$ (together with all associated triggers) on the state $(s_1, ctx_1)$ in the $(i-1)$-th configuration in $r_1$. Similarly, the $i$-th configuration $(s_2, ctx_2)$ is obtained by executing the query $(u_{i-1}, q_{i-1})$ (and all associated triggers) on the $(i-1)$-th configuration $(s_2, ctx_2)$ in $e(r, (u, q))$. Furthermore, from the first two claims, we know that the query and the associated triggers have all been authorized by $dbEnf$. Finally, since $c_{i-1} \neq (u_{i-1}, \text{skip})$, we know that the execution of $x \leftarrow q_{i-1}$ (and of the associated triggers) does not throw security exceptions according to the WHILESQL semantics. From the induction hypothesis, it follows that $s_1 = s_2$ and $\text{inTrigger}(ctx_1) = \text{inTrigger}(ctx_2) = \bot$. From this and the determinism of the database operational semantics, it follows that $s = s'$ and $\text{inTrigger}(ctx) = \text{inTrigger}(ctx') = \bot$.

This completes the proof of the third claim.

**Proof of (4).** We now prove our fourth claim. Namely that for all $1 \leq i < |\mu(e(r, (u, q)))| - 1$, if $\text{secEx}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = \bot$, and $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = \emptyset$, then the result of the $i$-th query $(u, q)$ in $\mu(e(r, (u, q)))$ is $k$ if the $i$-th step in $r_1$ is associated with the label $(u', q, k)$, where $u' = \text{Usr}(u_i, x \leftarrow q_i)$. Let $i$ be a value such that $1 \leq i < |\mu(e(r, (u, q)))| - 1$, and $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = \emptyset$. From the third claim, it follows that the $i$-th database state in $\mu(e(r, (u, q)))^{(i)}$ is $(s', ctx')$ and the database state in the $i$-th configuration of $r_1$ is $(s', ctx')$. From this and WHILESQL’s semantics, it follows that the result of the query $q_i$ is $k$ and $[q_i](s', ctx') = (u', k, r, e)$. From this, $(u', q, k)$ is one of the labels associated with the $i$-th step in $r_1$.

**Proof of (5).** We now prove that for all $1 \leq i < |\mu(e(r, (u, q)))| - 1$, if $\text{secEx}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = \bot$, and $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) \neq \emptyset$, then $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = \emptyset$, and $\text{triggers}(r, i) = e$ then $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = K$ if the $i$-th step in $r_1$ is associated with the label $(db(u_i), q_i, (\text{IntEx}, K), e)$. Let $i$ be a value such that $1 \leq i < |\mu(e(r, (u, q)))| - 1$, and $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = \bot$, and $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) \neq \emptyset$, and $\text{triggers}(r, i) = e$, and $K \neq \emptyset$. From the third claim, it follows that the $i$-th database state in $\mu(e(r, (u, q)))^{(i)}$ is $(s', ctx')$ and the database state in the $i$-th configuration of $r_1$ is $(s', ctx')$. From this, $\text{Ex}(\text{last}(\mu(e(r, (u, q)))^{(i+1)}) = K$, triggers(r, i) = e, and WHILESQL’s semantics, it follows that
\[ p((s', ctz'), u_i) = (s'', t, (\text{IntEx}, K, \epsilon)), \text{ where } (s', ctz') \text{ is the } i\text{-th database state in } r_1. \] From this, it follows that \( (db(u), q_i, (\text{IntEx}, K, \epsilon)) \) is the label associated with the \( i\text{-th step in } r_1. \)

**Proof of (6).** We prove that for all \( 1 \leq i < |\mu(e(r, \langle u, q_i \rangle))| - 1, \) if \( E_3(\text{last}(\mu(e(r, \langle u, q_i \rangle)))) = \bot, \) and \( (s', ctz') \) is the \( i\text{-th step in } r_1 \) associated with the label \( (db(u), q_i, (\text{IntEx}, K, \tau)) \). Let \( i \) be a value such that \( 1 \leq i < |\mu(e(r, \langle u, q_i \rangle))| - 1, \) \( E_3(\text{last}(\mu(e(r, \langle u, q_i \rangle)))) = \bot, \) \( \text{triggers}(r, i) = \top \) and \( K \neq \emptyset. \) From the third claim, it follows that the \( i\text{-th database state in } \mu(e(r, \langle u, q_i \rangle)) \) is \( (s', ctz') \) and the database state in the \( i\text{-th configuration of } r_1 \) is \( (s', ctz'). \) From this, \( E_3(\text{last}(\mu(e(r, \langle u, q_i \rangle)))) = K, \) \( \text{triggers}(r, i) = \top \) and \( \text{WHILESQL's semantics}, \) it follows that \( p((s', ctz'), u_i) = (s', \epsilon), (t, B, \text{IntEx}, K, \tau) \), where \( \langle s', ctz' \rangle \) is the \( i\text{-th database state in } r_1. \) From this, it follows that \( (db(u), q_i, (t, B, \text{IntEx}, K, \tau)) \) is the label associated with the \( i\text{-th step in } r_1. \)

**Proof of (7).** Finally, we prove our claim that for all \( 1 \leq i < |\mu(e(r, \langle u, q_i \rangle))| - 1, \) the label \( \langle u, q_i, \mu(r', i) \rangle \) is associated with the \( i\text{-th step in } r_1 \) if the trigger \( t \) is successfully executed in response to the \( i\text{-th query in } \mu(e(r, \langle u, q_i \rangle)), \) its condition is enabled, it has modified the security policy, and \( t \) was the \( |r| + 1 \) trigger executed in response to the \( i\text{-th query to modify the policy.} \) From the third claim, it follows that the \( i\text{-th database state in } \mu(e(r, \langle u, q_i \rangle)) \) is \( (s', ctz') \) and the database state in the \( i\text{-th configuration of } r_1 \) is \( (s', ctz'). \) From this and \( \text{WHILESQL's semantics, see the definition of } p(( \cdot, \cdot )); \) in Figures 7.8–7.9, the label \( \langle u, q_i, \mu(r', i) \rangle \) is associated with the \( i\text{-th step in } r_1 \) if the trigger \( t \) is successfully executed, its condition is enabled, \( t \) actually modifies the policy, and all and only the triggers in the labels in \( \tau \) have been executed before \( t. \)

**Proposition D.4.** Let \( M = \langle D, \Gamma \rangle \) be a system configuration, \( \text{ifEnf} \) be an enforcement mechanism for \( \text{WHILESQL programs}, u \) be a user, and \( \text{dbEnf} \) be the database control mechanism constructed using Reduction 7.1. Furthermore, let \( P = \langle M, \text{dbEnf} \rangle \) be the extended configuration and let \( \text{P-LTS}. \) Let \( r \) be a run in \( \text{traces}(L), t \) be a trigger such that \( e(r, t) \) is defined, and \( \langle u, q_i \rangle \) be a command. Finally, let \( r' \) be the \( r \) run obtained by extending \( e(r, t) \) with all scheduled triggers until either all triggers are executed or an exception is thrown. The \( i\text{-th query in } r_1 \) if the trigger \( t \) is successfully executed, its condition is enabled, \( t \) actually modifies the policy, and all and only the triggers in the labels in \( \tau \) have been executed before \( t. \)

1. For \( 1 \leq i < |\mu(r')| \), the \( i\text{-th query executed in } \mu(r') \) is \( \langle u, q_i \rangle \) iff the \( i\text{-th statement executed in } r_1 \) is \( \langle u, x \leftarrow q_i \rangle \) and \( \text{skip} \) otherwise.
2. For all \( 1 \leq i < |\mu(r')| \), last(\( \mu(r') \)) = \( (s, ctz) \) iff the \( i\text{-th configuration in } r_1 \) is \( \langle C, M, (s, ctz'), S \rangle \) and inTrigger(\( ctz \)) = inTrigger(\( ctz' \)).
3. For all \( 1 \leq i < |\mu(r')| \), if the \( i\text{-th query } \langle u, q_i \rangle \) in \( \mu(r') \) is authorized and no triggers associated with \( q_i \) throw integrity or security exceptions (i.e., \( \text{secEnf}(\text{last}(\mu(r'))^{i+1}) = \bot \) and \( \text{Ex}(\text{last}(\mu(r'))^{i+1}) = 0 \)), then the result of \( \langle u, q_i \rangle \) in \( \mu(r') \) is \( k \) iff the \( i\text{-th step in } r_1 \) is associated with the label \( \langle u', q_i, k \rangle \), where \( u' = \text{usr}(u, x \leftarrow q_i) \).
4. For all \( 1 \leq i < |\mu(r')| \), if the \( i\text{-th query } \langle u, q_i \rangle \) in \( \mu(r') \) is authorized (i.e., \( \text{secEnf}(\text{last}(\mu(r'))^{i+1}) = \bot \) and the query itself has caused an exception (i.e., \( \text{Ex}(\text{last}(\mu(r'))^{i+1}) = 0 \)) and triggers(\( r', i \) = \( \epsilon \)), then \( \text{Ex}(\text{last}(\mu(r'))^{i+1}) = K \) iff the \( i\text{-th step in } r_1 \) is associated with the label \( \langle db(u), q_i, (\text{IntEx}, K, \epsilon) \rangle \).
5. For all \( 1 \leq i < |\mu(r')| \), if the \( i\text{-th query } \langle u, q_i \rangle \) in \( \mu(r') \) is authorized (i.e., \( \text{secEnf}(\text{last}(\mu(r'))^{i+1}) = \bot \) and the trigger \( t \) executed in response to \( q \) has caused an exception (i.e., \( \text{Ex}(\text{last}(\mu(r'))^{i+1}) = 0 \)) and triggers(\( r', i \) = \( \top \) \)), then \( \text{Ex}(\text{last}(\mu(r'))^{i+1}) = K \) iff the \( i\text{-th step in } r_1 \) is associated with the label \( \langle db(u), q_i, (t, B, \text{IntEx}, K, \tau) \rangle \).

**Proof.** The proof is similar to that of Proposition D.3. The only insight is that Reduction 7.1 does not add the query \( (u, q) \) to the list of queries \( L \) (this follows from the fact that \( e(r, t) \) is defined and therefore inTrigger(\( ctz \)) = \( \top \)).

**D.1.5 Equivalence-class preservation**

Here, we prove that under proper conditions the enforcement mechanism constructed following Reduction 7.1 preserves the equivalence classes (as formalized in Chapter C) for weak indistinguishability.

**Proposition D.5.** Let \( M = \langle D, \Gamma \rangle \) be a system configuration, \( \text{ifEnf} \) be an enforcement mechanism for \( \text{WHILESQL programs}, u \) be a user, and \( \text{dbEnf} \) be the database access control mechanism constructed using Reduction 7.1. Furthermore, let \( P = \langle M, \text{dbEnf} \rangle \) be the extended configuration and \( L \) be the
P-LTS. Let \( r, r' \) be two runs in traces(\( L \)) and \( (u, q) \) be a database command such that \( r \succeq_{W_u} r' \) hold and \( c(r, (u, q)) \) and \( c(r', (u, q)) \) are defined. The runs \( r_1 = [r, (u, q)]^e \) and \( r_2 = [r', (u, q)]^e \) produced by executing code(\( ctx, q, u \)) starting from \( \langle \text{mem}(ctx', q, u), \text{init}(s, ctx') \rangle \) with scheduler \( S(\text{ctx}', q, u) \) and code(\( ctx', q, u \)) starting from \( \langle \text{mem}(ctx, q, u), \text{init}(s, ctx) \rangle \) with scheduler \( S(\text{ctx}, q, u) \), where \( (s, ctx) = \text{last}(r) \) and \( (s', ctx') = \text{last}(r') \), satisfy the following conditions:

1. \( r_1 \) and \( r_2 \) terminate,
2. \( f(r_1, ctz) \) and \( f(r_2, ctz') \) are \( u \)-epoch-equivalent, where \( f(r, ctz) \) is the run obtained from \( r \) by replacing the observations produced by the execution of the last statement with \( e \) if \( e \) in \( \text{Trigger}(ctz) = \perp \)
3. the last statement executed in both runs is the same.

Proof. Let \( M = \langle D, \Gamma \rangle \) be a system configuration, \( \text{ifEnf} \) be an enforcement mechanism for \( \text{WHILESQL} \) programs, \( u \) be a user, and \( \text{dbEnf} \) be the database access control mechanism constructed using Reduction 7.1. Furthermore, let \( P = \langle M, \text{dbEnf} \rangle \) be the extended configuration and \( L \) be the P-LTS. Let \( r, r' \) be two runs in traces(\( L \)) such that \( r \succeq_{P_u} r' \) hold, \( (u, q) \) be a database command, and \( M \) be a sequence of memories. Finally, let \( r_1 \) and \( r_2 \) be the runs \( r_1 = [r, (u, q)]_{E^1} \) and \( r_2 = [r', (u, q)]_{E^1} \) produced by executing code(\( ctx, q, u \)) starting from \( \langle \text{mem}(ctx, q, u), \text{init}(s, ctx) \rangle \) with scheduler \( S(\text{ctx}, q, u) \) and code(\( ctx', q, u \)) starting from \( \langle \text{mem}(ctx', q, u), \text{init}(s', ctx') \rangle \) with scheduler \( S(\text{ctx}', q, u) \), respectively, where \( (s, ctx) = \text{last}(r) \) and \( (s', ctx') = \text{last}(r') \). The reduction constructs programs without \( \text{while} \) statements, and therefore \( r_1 \) and \( r_2 \) trivially terminate. Similarly, the construction ensures that the last statement executed both in \( r_1 \) and \( r_2 \) is the same. Indeed, the last statement in both runs is \( (u, x = q) \), which corresponds to the query \( (u, q) \).

We now prove that \( f(r_1, ctz) \) and \( f(r_2, ctz') \) are \( u \)-epoch-equivalent. Given a run, we denote by \( \text{idz}(e, r) \), where \( e \) is an epoch in \( \langle E^{user} \rangle(\Gamma) \), the position of the epoch \( e \) in the run, e.g., \( \text{idz}(e, r) = 3 \) if \( e \) is the third epoch in \( r \). Furthermore, we denote by \( \text{idz}(e, r, u) \) the position of the epoch \( e \) in \( r \) when restricting ourselves to \( u \)-epochs, namely \( \text{idz}(e, r, u) = k \) if \( e \) is the \( k \)-th epoch in \( r \) where all commands are executed by \( u \). Let \( \nu \) be the mapping from \( \langle E^{user} \rangle(f(r_1, ctz)) \) to \( \langle E^{user} \rangle(f(r_2, ctz')) \) defined as follows: \( \nu(e) = e' \), where \( \text{idz}(e, f(r_1, ctz)) = \text{idz}(e', f(r_2, ctz')) \). We claim that \( \nu \) is a bijection from \( \langle E^{user}, S^{user} \rangle(\Gamma)(f(r_1, ctz)) \) to \( \langle E^{user}, S^{user} \rangle(\Gamma)(f(r_2, ctz')) \). Note that \( \nu \) is order preserving by construction. To show that \( f(r_1, ctz) \) and \( f(r_2, ctz') \) are \( u \)-epoch-equivalent we just have to show that for all \( e_1 \in \langle E^{user}, S^{user} \rangle(\Gamma)(f(r_1, ctz)) \), the initial configurations in \( e_1 \) and \( \nu(e_1) \) are indistinguishable for \( u \) and the traces in \( e_1 \) and \( \nu(e_1) \) are the same.

We now prove that for all \( e_1 \in \langle E^{user}, S^{user} \rangle(\Gamma)(f(r_1, ctz)) \), the traces in \( e_1 \) and \( \nu(e_1) \) are the same. The equivalence of the traces in \( e_1 \) and \( \nu(e_1) \) directly follows from Proposition D.3 and \( r \succeq_{P_u} r' \). Assume, for contradiction’s sake, that this is not the case. Namely, there is an epoch \( e_1 \) such that the traces in \( e_1 \) and \( \nu(e_1) \) are different. This may happen for two cases:

1. The commands executed in \( e_1 \) and \( \nu(e_1) \) are different. From the first two points of Proposition D.3, this happens if either the queries executed in the portions of \( r \) and \( r' \) corresponding to \( e_1 \) and \( \nu(e_1) \) are different or the security decisions for a common query are different in the portions of \( r \) and \( r' \) corresponding to \( e_1 \) and \( \nu(e_1) \). Both cases, however, contradict \( r \succeq_{P_u} r' \).
2. The commands executed in \( e_1 \) and \( \nu(e_1) \) are the same but they produce different labels. There are two cases:
   a. The different labels are caused by a query that is not an \text{INSERT} or \text{DELETE}. From this and the fourth and fifth points in Proposition D.3, this implies that the queries behaved differently in \( r \) and \( r' \). This directly contradicts \( r \succeq_{P_u} r' \).
   b. The different labels are caused by an \text{INSERT} or \text{DELETE} query. If the different label is directly associated with the \text{INSERT} or \text{DELETE} query, this implies that the queries behaved differently in \( r \) and \( r' \). This directly contradicts \( r \succeq_{P_u} r' \). If the different label is caused by one of the triggers, this implies that the policy has been modified in a different way in \( r \) and \( r' \). This again contradicts \( r \succeq_{P_u} r' \) (since \( r \succeq_{P_u} r' \) implies that the configuration is the same in the \text{u}-projections of \( r \) and \( r' \)).

Note that the case where Proposition D.3 and \( r \succeq_{P_u} r' \) cannot be used to determine that the labels are the same in \( e_1 \) and \( \nu(e_1) \) is the last statement \( (u, q) \). In the above proof, this case does not occur since we reason about \( f(r_1, ctz) \) and \( f(r_2, ctz') \) (which ignore the result of the last statement) instead of \( r_1 \) and \( r_2 \).

Finally, we prove, by induction on \( \text{idz}(e_1, r, u) \), that the initial configurations in \( e_1 \) and \( \nu(e_1) \) are indistinguishable for \( u \). For the base case, let \( \text{idz}(e_1, r, u) = 1 \). Therefore, \( \text{idz}(\nu(e_1), r, u) = 1 \). Let \( \langle M_1, (s_1, ctx_1) \rangle \) be the global state at the beginning of \( e_1 \) and \( \langle M_2, (s_2, ctx_2) \rangle \) be the global state at the beginning of \( \nu(e_1) \). From Proposition D.3, it follows that the corresponding database states in \( r \) and \( r' \) are, respectively, \( (s_1, ctx_1') \) and \( (s_2, ctx_2') \). From this and \( r \succeq_{P_u} r' \), it follows that \( s_1 \approx_s s_2 \). Furthermore, since \( e_1 \) and \( \nu(e_1) \) are the first epochs for the user \( u \), the memories associated with the user \( u \) are all set to the (same) initial memory and from \( r \succeq_{P_u} r' \) and Reduction 7.1 it follows that \( M_1|_u = M_2|_u \). Therefore, \( M_1 \approx_s M_2 \). Thus, the initial global states in \( e_1 \) and \( \nu(e_1) \) are
Proof. Let \( a \) for \( e \) have to prove that the memories are \( D.1 \). From Database Access Control to Information-flow Control (and trivially that have been executed before \( e \) and \( \nu(e) \) directly follows, instead, from the fact that the labels in all previous epochs (the only ones that may modify memories visible for \( u \)) are the same in \( f(r_1, ctx) \) and \( f(r_2, ctx') \). Therefore, the results of the queries is the same and the memories are \( u \)-indistinguishable.

We now prove our claim that \( \nu \) is a bijection from \( [E^\text{user}_u, S^\text{user}_u](f(r_1, ctx)) \) to \( [E^\text{user}_u, S^\text{user}_u](f(r_2, ctx')) \). The fact that \( [E^\text{user}_u, S^\text{user}_u](f(r_1, ctx)) = [E^\text{user}_u, S^\text{user}_u](f(r_2, ctx')) \) directly follows from the construction of Reduction 7.1 and \( r \cong W_{u} r' \). Indeed, from \( r \cong W_{u} r' \), it follows that \( r|_u \) and \( r' \mid u \) are consistent. Therefore, the commands issued by \( u \) are the same in \( r \) and \( r' \), they are executed in the same order, and they are interleaved with commands from other users in the same way. From this and the fact that Reduction 7.1 constructs programs that exactly mimic the database runs by associating a single statement to a single query (see Proposition D.3), it follows that each sequence of queries issued by \( u \) in, say, \( r \) is associated to exactly one epoch in \( r_1 \) (the same holds for \( r_2 \) and \( r_2 \)). Furthermore, \( \nu(e) \neq \nu(e') \) whenever \( e \neq e' \) for any two \( e, e' \in [E^\text{user}_u, S^\text{user}_u](f(r_1, ctx)) \) by construction. Therefore, \( \nu \) is a bijection from \( [E^\text{user}_u, S^\text{user}_u](f(r_1, ctx)) \) to \( [E^\text{user}_u, S^\text{user}_u](f(r_2, ctx')) \). □

**Proposition D.6.** Let \( M = (D, \Gamma) \) be a system configuration, \( ifEnf \) be an enforcement mechanism for WHILESQL programs, \( u \) be a user, and \( dbEnf \) be the database access control mechanism constructed using Reduction 7.1. Furthermore, let \( P = \langle M, dbEnf \rangle \) be the extended configuration and \( L \) be the \( P \)-LTS. For any run \( r \in traces(L) \) and any action \( a \in \mathcal{A}_{D,u} \), if the following conditions hold:

1. \( ifEnf \) is sound for the progress-sensitive variant of Definition 7.2 given \( E_{\text{user}}^u \) and \( S_{\text{user}}^u \).
2. \( ifEnf \) is \( u \)-stable, and
3. \( extend(r, a) \) is defined,

\( a \) preserves the equivalence class for \( r \), \( \langle M, dbEnf \rangle \), and \( \cong W_{u} \).

Proof. Let \( M = (D, \Gamma) \) be a system configuration, \( ifEnf \) be an enforcement mechanism for WHILESQL programs that is (1) sound, and (2) stable (as required by the proposition), \( u \) be a user, and \( dbEnf \) be the database access control mechanism constructed using Reduction 7.1.

Furthermore, let \( P = \langle M, dbEnf \rangle \) be the extended configuration, \( L \) be the \( P \)-LTS, and \( \langle s, ctx \rangle \) be the last state of \( r \), i.e., \( \langle s, ctx \rangle = last(r) \). In the following, we use \( e \) to refer to \( extend \) and \( f \) to refer to \( dbEnf \). With a slight abuse of notation, given a run \( r \in traces(L) \) and an action \( u, q \), we write \( [r, (u, q)]^{W_{u}} \) to denote the run generated by the WHILESQL programs generated according to our reduction and starting from the state and scheduler generated by the reduction algorithm. Note that the \( i \)-th statement executed in the run \( [r, (u, q)]^{W_{u}} \) corresponds to the \( i \)-th command executed in \( \mu(e, (u, q)) \) and \( [r, (u, q)]^{W_{u}} = \mu(e, (u, q)) \).

We prove our claim by contradiction. Assume, by contradiction’s sake, that there is a run \( r \in traces(L) \) and an action \( a \in \mathcal{A}_{D,u} \) such that \( e, a \) is defined and \( a \) does not preserve the equivalence class for \( r \), \( \langle M, dbEnf \rangle \), \( \cong W_{u} \), \( M, dbEnf \), and \( u \). Since \( e, a \) is defined, \( \text{triggers}(last(r)) = e \). From this, it follows that \( \text{triggers}(last(r')) = e \) also for any \( r' \in [r]^{W_{u}} \). Finally, let \( \epsilon' \) and \( \epsilon'' \) be a run such that \( r \cong W_{u} r' \). From this and Proposition D.5, it follows that the runs \( r_1 = [r, (u, q)]^{W_{u}} \) and \( r_2 = [r', (u, q)]^{W_{u}} \) produced by \( code(ctx, q, u) \) starting from \( \langle mem(ctx, q, u), init(s, ctx) \rangle \) with scheduler \( S(ctx, q, u) \) and \( code(ctx', q, u) \) starting from \( \langle mem(ctx', q, u), init(s', ctx') \rangle \) with scheduler \( S(ctx', q, u) \), where \( (s, ctx) = last(r) \) and \( (s', ctx') = last(r') \), satisfy the following conditions: (a) \( r_1 \) and \( r_2 \) terminate, (b) \( f(r_1, ctx) \) and \( f(r_2, ctx') \) are \( u \)-equivalent, and (c) the last statement executed in both runs is the same. From this and the stability of \( ifEnf \), it follows that \( ifEnf(code(ctx, q, u), S(ctx, q, u), u, (s, \epsilon)) = ifEnf(code(ctx', q, u), S(ctx', q, u), u, (s', \epsilon)) \). We claim that \( f(s, ctx', (u, q)) = f(s, ctx, (u, q)) \), where \( (s', ctx') = last(r') \). There are two cases:

1. \( f(s, ctx, (u, q)) = \bot \). Let \( e^{W_{u}} \) be the last epoch in the run \( [r, (u, q)]^{W_{u}} \). Furthermore, let \( init \) (respectively \( init' \) ) be the index associated with the beginning of \( e^{W_{u}} \) in \( e, (u, q) \) (respectively \( e^{W_{u}} \) in \( e', (u, q) \)). Note that \( init \) (and \( init' \)) are always well-defined (they correspond to the \( i \)-th query in the run \( r \), where \( \mathcal{I} \) is the number of statements executed in \( [r, (u, q)]^{W_{u}} \) up to the beginning of \( e^{W_{u}} \)). Finally, let \( idx \) be the index of the last query \( (u, c) \) inside the epoch \( e^{W_{u}} \) and \( \mathcal{T}(e^{W_{u}}) \) be all the queries executed
in $ep^{ifc}$ up to the $i$-th query (included). Note that $idx$ is also the index of the last query $(u, c)$ inside $ep^{ifc}$ (this directly follows from our reduction).

We remark that $code(ep^{ifc})$ is the same as $code(ep^{ifc'})$ (it follows from the reduction’s definition). Furthermore, the order in which the statements in $code(ep^{ifc})$ and $code(ep^{ifc'})$ are executed (this again directly follows from Reduction 7.1). In the rest of the proofs, we ignore the memories in WHILESQL runs as they are never read for programs generated by our reduction. Hence, with a slight abuse of notation, we use the same notation for global states and runtime states.

We now show that executing $(u, q)$ produces the same results (and exceptions) both in $(s, ctx)$ and $(s', ctx')$. From this and $r \cong^W_{Pa} r'$, it directly follows that $e(r, (u, q)) \cong^W_{Pa} e(r', (u, q))$ (leading to a contradiction with $(u, q)$ not preserving the equivalence class).

From $f$’s construction, $f(s, ctx, (u, q)) = \top$, and if $Enf$ soundness, it follows that $PK_u(ep^{ifc}, idx) = [start(ep^{ifc})]_u$. $PK_u(ep^{ifc}, idx - 1) = [start(ep^{ifc})]_u$, $PK_u(ep^{ifc'}, idx) = [start(ep^{ifc'})]_u$, and $PK_u(ep^{ifc'}, idx - 1) = [start(ep^{ifc'})]_u$. Furthermore, from our reduction, the notion of user-based epochs, and $r \cong^W_{Pa} r'$, then $start(ep^{ifc})$ and $start(ep^{ifc'})$ are data-indistinguishable for $u$. From this, $[start(ep^{ifc})]_{u_0} = [start(ep^{ifc'})]_{u_0}$ (since data indistinguishability is an equivalence relation). We refer to $[start(ep^{ifc})]_{u_0}$ and $[start(ep^{ifc'})]_{u_0}$ as $C$ and $C'$ in the following. From $PK_u(ep^{ifc}, idx) = PK_u(ep^{ifc'}, idx - 1) = C$, it follows that for all runs starting from $s \in C$, executing all queries in $ep^{ifc}$ up to the $idx$-th query produces the same observations (and possible integrity exceptions) for all database states $s$, $\forall s \in C$.

From this, $[db([r]^{r}_{Pa})]_u = [db([r']^{r'}_{Pa})]_u$ (since $db([r]^{r}_{Pa}) \subseteq o(C, \bar{\tau}^{idx-1}(ep^{ifc}))$). From this, $[db([\bar{r}]^{\bar{r}}_{Pa})]_u \subseteq C'$, and the fact that all (and only) the queries in $\tau^{idx-1}$ have been executed in $r$, it follows that for all database states $(s, ctx) \in [db([r]^{r}_{Pa})]_u$ the result of $\varphi$ is the same (since $db([r]^{r}_{Pa}) \subseteq o(C, \bar{\tau}^{idx-1}(ep^{ifc}))$). From this, from $C = C'$, it follows that the result of $\varphi$ is the same in $(s, ctx)$ and $(s', ctx')$.

If the $idx$-th command in $ep^{ifc}$ is a SELECT command $(u, SELECT \varphi)$, then the only observation is $(db(u), q)$, $w$ for $u$ is the SELECT query result. From this, it follows that (1) for all database states $s \in o(C, \bar{\tau}^{idx-1}(ep^{ifc}))$, the result of $\varphi$ is the same, and (2) for all database states $s' \in o(C', \bar{\tau}^{idx-1}(ep^{ifc'}))$, the result of $\varphi$ is the same. From this, $[db([\bar{r}]^{\bar{r}}_{Pa})]_u \subseteq C'$, and the fact that all (and only) the queries in $\tau^{idx-1}$ have been executed in $r$, it follows that for all database states $(s, ctx) \in [db([r]^{r}_{Pa})]_u$ the result of $\varphi$ is the same (since $db([r]^{r}_{Pa}) \subseteq o(C', \bar{\tau}^{idx-1}(ep^{ifc})))$. From this and $C = C'$, it follows that the result of $\varphi$ is the same in $(s, ctx)$ and $(s', ctx')$.

If the $idx$-th command in $ep^{ifc}$ is an INSERT or DELETE command, then the query produces an observation $(db(u), q, r, \tau)$, where $r$ is either $\top$ or an exception and $\tau$ is a (possibly empty) sequence of public observations associated with $GRANT$ and $REVOKE$ commands. For simplicity, we restrict ourselves to integrity exceptions caused by INSERT and DELETE commands (for exceptions related with $GRANT$ and $REVOKE$ commands see the next case). From $PK_u(ep^{ifc}, idx) = PK_u(ep^{ifc'}, idx - 1) = C$, it follows that for all states $s \in o(C, \bar{\tau}^{idx-1}(ep^{ifc}))$, the INSERT or DELETE command (and the triggers executed in response to it) produces the same results and exceptions (and public observations $\tau$). Similarly, from $PK_u(ep^{ifc}, idx) = PK_u(ep^{ifc'}, idx - 1) = C'$, it follows that for all states $s \in o(C', \bar{\tau}^{idx-1}(ep^{ifc'}))$, the INSERT or DELETE command (and the triggers executed in response to it) produces the same results and exceptions (and public observations $\tau$). From this, $C = C'$, $[db([\bar{r}]^{\bar{r}}_{Pa})]_u \subseteq C'$, $[db([\bar{r}]^{\bar{r}}_{Pa})]_u \subseteq C'$, $(s, ctx) \in [db([r]^{r}_{Pa})]_u$, $(s', ctx') \in [db([r']^{r'}_{Pa})]_u$, $[db([\bar{r}]^{\bar{r}}_{Pa})]_u \subseteq o(C', \bar{\tau}^{idx-1}(ep^{ifc'}))$, and $[db([\bar{r}]^{\bar{r}}_{Pa})]_u \subseteq o(C', \bar{\tau}^{idx-1}(ep^{ifc'}))$. It follows that the result (and the exceptions) produced by the query is the same in $r$ and $r'$ (and the same holds for $\tau$).

If the $idx$-th command in $ep^{ifc}$ is an INSERT command modifying the database configuration (i.e., $GRANT$, $REVOKE$, $ADD_USER$, or $CREATE$ command), then the only observation is $(public, q, r, \epsilon)$, where $r$ is either the positive query result or an exception. From $r \cong^W_{Pa} r'$, it follows that the database configuration is the same in $(s, ctx)$ and $(s', ctx')$. From this, the operational semantics of configuration commands from Chapter 6, and the definition of $[\cdot](\cdot)$, it follows that the result of the query is the same in $(s, ctx)$ and $(s', ctx')$ (i.e., either $(u, q)$ is successfully executed in both cases or its execution throws an exception both in $(s, ctx)$ and $(s', ctx')$).

In all cases, we have shown that the result of the query (as well as possible integrity exceptions and additional observations) is the same for $(s, ctx)$ and $(s', ctx')$. We claim that the last states in $(r, (u, q))$ and $(r', (u, q))$ are configuration indistinguishable. From this and $r \cong^W_{Pa} r'$, it follows that $e(r, (u, q)) \cong^W_{Pa} e(r', (u, q))$. From this, it follows that for any $r' \in [r]^{r'}_{Pa}$, $e(r', (u, q)) \in [e(r, (u, q))]^{r}_{Pa}$. From this, it follows that $(u, q)$ preserves the equivalence class for $r$,
\langle M, dbEnf \rangle$, and $\equiv^{W}_{(M, dbEnf), u}$.

We now prove our claim that the last states in $e(r, (u, q))$ and $e(r', (u, q))$ are configuration indistinguishable. Note that from $r \equiv^{W}_{P_{u}} r'$, the last states in $r$ and $r'$ are configuration indistinguishable. For SELECT, INSERT, and DELETE queries, the claim follows trivially since these statements do not modify the database configuration and the last states in $r$ and $r'$ are configuration indistinguishable. For configuration commands, the claim directly follows from (1) the last states in $r$ and $r'$ are configuration indistinguishable, and (2) the commands would either both succeed or both fail in $e(r, (u, q))$ and $e(r', (u, q))$ (as shown above). If the commands fail, the configuration indistinguishability directly follows from the one of the last states in $r$ and $r'$. If both commands succeed, then the configurations are modified in the same way and the last states in $e(r, (u, q))$ and $e(r', (u, q))$ are configuration indistinguishable.

This completes the proof of our claim.

We now prove our claim that $f(s', ctx', (u, q)) = f(s, ctx, (u, q))$, where $(s', ctx') = last(r')$. Assume, for contradiction's sake, that this is not the case. From the stability of $ifEnf$, it follows that $ifEnf(code(ctx', q, u), S(ctx', q, u), \sigma, (s', e)) = ifEnf(code(ctx', q, u), S(ctx', q, u), u, (s', e))$. From this, $f(s', ctx', (u, q)) \neq f(s, ctx, (u, q))$, and $dbEnf$'s definition, it follows that (1) $ifEnf(code(ctx', q, u), S(ctx', q, u), u, (s', e)) = \top$, and (2) $ifEnf(code(ctx', q, u), S(ctx', q, u), u, (s', e)) = T$. There are three cases:

- The run $[r, (u, q)]^{ic}$ terminates and the run $[r', (u, q)]^{ic}$ does not terminate. Since $r \equiv^{W}_{P_{u}} r'$, the last states in $r$ and $r'$ are configuration indistinguishable. From this and $extend(r', (u, q))$ is defined, it follows that $extend(r', (u, q))$ is defined. From this, $[r', (u, q)]^{ic}$ terminates without getting stuck, leading to a contradiction.

- The run $[r', (u, q)]^{ic}$ terminates and the run $[r, (u, q)]^{ic}$ does not terminate. This contradicts the fact that $extend(r, (u, q))$ is defined.

- Both runs terminate without getting stuck and the last observation in $[r, (u, q)]^{ic}$ and $[r', (u, q)]^{ic}$ are different (one results in a security exception and the other does not). Let $ep^{ic}$ (respectively $ep^{ic'}$) be the last epoch in the run $[r, (u, q)]^{ic}$ (respectively $[r', (u, q)]^{ic'}$). Observe that the last observations belong to the last epochs $ep^{ic}$ and $ep^{ic'}$ respectively. From $r \equiv^{W}_{P_{u}} r'$, it follows that the states at the beginning of the epochs $ep^{ic}$ and $ep^{ic'}$ are data indistinguishable. From this and $ifEnf(code(ctx, q, u), S(ctx, q, u), u, (s, e)) = \top$, the observations produced in $ep^{ic}$ and $ep^{ic'}$ must be the same, leading to a contradiction.

This completes the proof of our claim.

\[\]

**Proposition D.7.** Let $M = (D, \Gamma)$ be a system configuration, $ifEnf$ be an enforcement mechanism for WHILE SQL programs, $u$ be a user, and $dbEnf$ be the database access control mechanism constructed using Reduction 7.1. Furthermore, let $P = (M, dbEnf)$ be the extended configuration and $L$ be the $P$-LTS. For any run $r \in traces(L)$ and any trigger $t$, if the following conditions hold:

1. $ifEnf$ is sound for the progress-sensitive variant of Definition 7.2 given $E^{\text{ww}}_{\text{uw}}$ and $S_{\text{uw}}$,
2. $ifEnf$ is u-stable, and
3. $extend(r, t)$ is defined,

$t$ preserves the equivalence class for $r, (M, dbEnf)$, and $\equiv^{W}_{(M, dbEnf), u}$.

**Proof.** Let $r \in traces(L)$ be a run and $t$ be a trigger such that $e(r, t)$ is defined. We claim that Reduction 7.1 always authorizes any command $(u, q)$ if $inTrigger(ctx) = \top$. This combined with the database semantics (which ensures that only a trigger’s condition and action can be executed while executing the trigger $t$), implies that the execution of the trigger $t$ never throws security exceptions. Therefore, proving that $t$ preserves the equivalence class for $r, (M, dbEnf)$, and $\equiv^{W}_{(M, dbEnf), u}$ amounts to proving (1) if $t$ throws an integrity exception in $e(r, t)$ then it throws an integrity exception also in $e(r', t)$ for any $r' \equiv^{W}_{P_{u}} r$, and (2) $last(e(r, t))$ and $last(e(r', t))$ are configuration equivalent for any $r' \equiv^{W}_{P_{u}} r$.

We prove that if $t$ throws an integrity exception in $e(r, t)$ then it throws an integrity exception also in $e(r', t)$ for any $r' \equiv^{W}_{P_{u}} r$. Assume, for contradiction’s sake, that this is not the case. This implies that the labels produced by executing the statement $a$ in the program computed by Reduction 7.1 differ for $r$ and $r'$. This implies that executing $a$ (and the associated triggers) leaks information. This, therefore, contradicts the fact that $a$ is authorized (see the proof of Proposition D.6).

We now prove that $last(e(r, t))$ and $last(e(r', t))$ are configuration equivalent for any $r' \equiv^{W}_{P_{u}} r$. From $r' \equiv^{W}_{P_{u}} r$, it follows that $last(r)$ and $last(r')$ are configuration equivalent. If $t$'s action is an
INSERT or DELETE command, the configuration equivalence of last(e(r,t)) and last(e(r',t)) directly follows from the configuration equivalence of last(r) and last(r'). Instead, assume that t's action is a GRANT or REVOKE command influencing the user u. Furthermore, assume for contradiction's sake, that last(e(r,t)) and last(e(r',t)) are not configuration equivalent. There are four cases:

1. t is enabled in both e(r,t) and e(r',t). From this and the configuration equivalence of last(r) and last(r'), it follows that last(e(r,t)) and last(e(r',t)) are configuration equivalent as well, leading to a contradiction.
2. t is enabled in e(r,t) but it is not enabled in e(r',t). There are two cases:
   (a) auth(last(r), u) = auth(last(e(r,t)), u). From this, it follows that the database configuration is the same in last(r) and last(e(r,t)). Furthermore, from the fact that t is not enabled in e(r',t) it follows that the database configuration is the same in last(r') and last(e(r',t)). From this and the configuration equivalence of last(r) and last(r'), it directly follows that last(e(r,t)) and last(e(r',t)) are configuration equivalent, leading to a contradiction.
   (b) auth(last(r), u) ≠ auth(last(e(r,t)), u). From this and the fact that we are executing the trigger, it follows that the query α that fired the trigger has been authorized. From this and auth(last(r), u) ≠ auth(last(e(r,t)), u), it follows that (public, t, q, e), where q is the GRANT or REVOKE command associated with t's action, belongs to the labels produced by executing the statement associated with u in the program defined by Reduction 7.1. From this and the fact that u has been authorized, it follows that (public, t, q, e) must be associated to the execution of α for any indistinguishable run. Therefore, (public, t, q, e) should be derived also from the run in e(r',t) (since it was indistinguishable from r when we executed α). Therefore, t must be enabled also in e(r',t), leading to a contradiction.
3. t is not enabled in e(r,t) but it is enabled in e(r',t). The proof is similar to the previous case.
4. t is not enabled in both e(r,t) and e(r',t). From this and the configuration equivalence of last(r) and last(r'), it follows that last(e(r,t)) and last(e(r',t)) are configuration equivalent as well since the database configuration in last(r) and last(e(r,t)) is the same (and the same hold for r' and e(r',t)), leading to a contradiction.

Finally, we prove our claim that Reduction 7.1 authorizes any command (u, q) if inTrigger(ctx). This directly follows from the fact that the program constructed by Reduction 7.1 contain only secure commands (since the insecure ones are replaced with skip statements).

D.1.6 Proof of the main result

We are now ready to prove the main result in this section. Namely that given a sound and stable enforcement mechanism for WHILESQL programs, our construction can be used to construct a secure database access control mechanism.

Theorem D.1. Let M be a system configuration, j|Enf be an enforcement mechanism for WHILESQL programs, u be a user, and db|Enf be the database access control mechanism constructed using Reduction 7.1. If for all u ∈ UID, (1) j|Enf is sound for the progress-sensitive variant of Definition 7.2 given E^user and S^user, and (2) j|Enf is u-stable, then db|Enf provides data confidentiality with respect to (M, db|Enf), u, ATK_u, and S^user(M, db|Enf) for any user u ∈ UID, where ATK_u is the attacker model from Chapter 6.

Proof. Let M = (D, Γ) be a system configuration, j|Enf be an enforcement mechanism for WHILESQL programs, u ∈ UID be a user, and db|Enf be the database access control mechanism constructed using Reduction 7.1. Furthermore, we denote by f be mechanism db|Enf, by P the extended configuration ⟨M, db|Enf⟩, by A the attacker model ATK_u, and by L the P-LTS. Let r be a run in traces(L), i be an integer such that 1 ≤ i ≤ |r|, and φ be a sentence such that r, i ⊩ u φ holds in A. We claim that also secure^W_{P_u} (r, i ⊩ u φ) holds. The theorem follows trivially from the claim and Proposition D.2.

We now show that for all r ∈ traces(L), all i such that 1 ≤ i ≤ |r|, and all sentences φ such that r, i ⊩ u φ holds, then also secure^W_{P_u} (r, i ⊩ u φ) holds. We prove our claim by induction on the length of the derivation r, i ⊩ u φ. In the following, we denote by e the function extend.

Base Case: Assume that |r, i ⊩ u φ| = 1. There are a number of cases depending on the rule used to obtain r, i ⊩ u φ.

1. SELECT Success - 1. Let i be such that r' = r^i-1·⟨u, SELECT, φ⟩·(s, c), where s = ⟨db, U, sec, T, V⟩ and last(r'^-1) = ⟨s', c'⟩, where s' = ⟨db, U, sec, T, V⟩. We claim that the result of φ is the same for all runs r' ∈ [r]^W_{P_u}. From this, secure^W_{P_u} (r, i ⊩ u φ) holds.

We now prove our claim that the result of φ is the same for all runs r' ∈ [r]^W_{P_u}. Let r' be a run in [r]^W_{P_u}. From r' ⊆ P_u r', it follows that r'|u and r'|u are consistent. From this, it follows that (1) the last action in r'|u is (u, SELECT, φ), and (2) res(last(r'|u)) = res(last(r'|u)). Therefore, φ^db' = φ^db, where db' is the database state in last(r') and db is the database state in res(last(r')). This completes the proof of our claim.
2. SELECT Success - 2. The proof for this case is similar to that of SELECT Success - 1.
3. INSERT Success. Let $i$ be such that $r^i = r^{i-1} \cdot (u, \text{INSERT}, R, T) \cdot s$, where $s = (db, U, sec, T, V, c) \in \Omega_M$ and last($r^{i-1}$) = $(db', U, sec, T, V, c')$, and $\phi$ be $R(\overline{t})$. Then, $\text{secure}_{W^u}(r, i \vdash_u R(\overline{t}))$ holds. Indeed, in all runs $r' \subseteq W^u$, $r^i$ the last action is $(u, \text{INSERT}, R, T)$. Furthermore, the action has been executed successfully. Therefore, according to the LTS rules, $\overline{t} \in \text{last}(r^i) \cdot db(R)$ for all runs $r' \in [r^i]_{P,u}$. From this and the relational calculus semantics, it follows that $[R(\overline{t})\text{last}(r^i) \cdot db] = \top$ for all runs $r' \in [r^i]_{P,u}$. Hence, $\text{secure}_{W^u}(r, i \vdash_u R(\overline{t}))$ holds.

4. INSERT Success - FD. Let $i$ be such that $r^i = r^{i-1} \cdot (u, \text{INSERT}, R, (\tau, \varpi, \eta)) \cdot s$, where $s = (db, U, sec, T, V, c) \in \Omega_M$ and last($r^{i-1}$) = $(db', U, sec, T, V, c')$, and $\phi$ be $-\exists \varpi, \eta. R(\tau, \varpi, \eta) \land \eta \neq \varpi$. From the rule’s definition, it follows that the INSERT command has not violated the integrity constraint $\gamma$. From this, it follows that for all runs $r' \in [r^i]_{P,u}$ the INSERT command has not violated $\gamma$. From this and $\phi$ is the weakest precondition for not violating $\gamma$ (see Proposition C.10), it follows that $\phi$ holds in all runs $r' \in [r^i]_{P,u}$. From this, $\text{secure}_{W^u}(r, i \vdash_u \phi)$ holds.

5. INSERT Success - ID. The proof of this case is similar to that for the INSERT Success - FD.

6. DELETE Success. The proof for this case is similar to that of INSERT Success.

7. DELETE Success - ID. The proof of this case is similar to that for the INSERT Success - FD.

8. INSERT Exception. Let $i$ be such that $r^i = r^{i-1} \cdot (u, \text{INSERT}, R, T) \cdot s$, where $s = (db, U, sec, T, V, c) \in \Omega_M$ and last($r^{i-1}$) = $(db', U, sec, T, V, c')$, and $\phi$ be $-R(\overline{t})$. From the rule’s definition, it follows that the INSERT command has violated the integrity constraint $\gamma$. From this and the LTS semantics, it follows that (1) $R(\overline{t})$ does not hold before executing the command (otherwise the command would not have thrown an exception), and (2) $R(t)$ does not hold after executing the command (since the system rolled-back to the state before the INSERT). Furthermore, for all runs $r' \in [r^i]_{P,u}$, the INSERT command has violated $\gamma$. For the same reason, it follows that (1) $R(\overline{t})$ does not hold before executing the command in $r'$. Otherwise, the command would not have thrown an exception, and (2) $R(t)$ does not hold after executing the command in $r'$ (since the system rolled-back to the state before the INSERT). Therefore, $\text{secure}_{W^u}(r, i \vdash_u \phi)$ holds.

We now prove that $\text{secure}_{W^u}(r, i \vdash_u \phi)$ holds as well. Let $r'$ be a run in $[r^{i-1}]_{P,u}$. From $r^{i-1} \subseteq W^u$, the fact that the command has been authorized in $r^i$, the stability of $\text{ifEnf}$, and Proposition D.5, it follows that executing the INSERT command is authorized also in $r^i$. From this and the soundness of $\text{ifEnf}$, it follows that the observations produced by executing the INSERT command in $r'$ are the same as those produced by executing the command in $r^{i-1}$. From this, it follows that the INSERT command would not violate the integrity constraint $\gamma$ in $r'$. From this and $\phi$ is the weakest precondition for violating $\gamma$ (see Proposition C.10), it follows that $[\phi]_{\text{last}(r^i)} \cdot db = [\phi]_{\text{last}(r^{i-1})} \cdot db$. From this, $\text{secure}_{W^u}(r, i \vdash_u \phi)$ holds.

9. DELETE Exception. The proof for this case is similar to that of INSERT Exception.

10. DELETE FD Exception. Let $i$ be such that $r^i = r^{i-1} \cdot (u, \text{INSERT}, R, (\tau, \varpi, \eta)) \cdot s$, where $s = (db, U, sec, T, V, c) \in \Omega_M$ and last($r^{i-1}$) = $(db', U, sec, T, V, c')$, and $\phi$ be $-\exists \varpi, \eta. R(\tau, \varpi, \eta) \land \eta \neq \varpi$. From the rule’s definition, it follows that the INSERT command is authorized and that it has violated an integrity constraint, which we call $\gamma$. From this, it follows that for all runs $r' \in [r^i]_{P,u}$ the last INSERT command has violated $\gamma$. From this and $\phi$ is the weakest precondition for violating $\gamma$ (see Proposition C.10), it follows that $\phi$ holds in all runs $r' \in [r^i]_{P,u}$. From this, $\text{secure}_{W^u}(r, i \vdash_u \phi)$ holds.

We now prove that $\text{secure}_{W^u}(r, i \vdash_u \phi)$ holds as well. Let $r'$ be a run in $[r^{i-1}]_{P,u}$. From $r^{i-1} \subseteq W^u$, the fact that the INSERT command has been authorized in $r^i$, the stability of $\text{ifEnf}$, and Proposition D.5, it follows that executing the INSERT command is authorized also in $r^i$. From this and the soundness of $\text{ifEnf}$, it follows that the observations produced by executing the INSERT command in $r'$ are the same as those produced by executing the command in $r^{i-1}$. From this and $\phi$ it follows that the INSERT command would violate the integrity constraint $\gamma$ also in $r'$. From this and $\phi$ is the weakest precondition for violating $\gamma$ (see Proposition C.10),
it follows that $\lbrack \phi \rbrack^{r'.db} = \lbrack \phi \rbrack^{r.db}$. From this, $\text{secure}_{\phi}^{\mathcal{W}}(r, i - 1 \vdash u, \phi)$ holds.

11. INSERT ID Exception. The proof for this case is similar to that of INSERT FD Exception.

12. DELETE FD Exception. The proof for this case is similar to that of INSERT FD Exception.

13. Integrity Constraint. The proof of this case follows trivially from the fact that for any state $s = (db, U, sec, T, V, c) \in \Omega_M$ and any $\gamma \in \Gamma$, $\lbrack \gamma \rbrack^{db} = \gamma$ by definition.

14. Learn GRANT/REVOKE Backward. Let $i$ be such that $r^i = r^{i-1}.t.s$, where $s = (db, U, sec, T, V, c) \in \Omega_M, last(r^{i-1}) = (db, U, sec', T, V, c')$, and $t$ be a trigger whose WHEN condition is $\phi$ and whose action is either a GRANT or a REVOKE. From the rule’s definition, the trigger has been scheduled. From this, it follows that there has been a previous INSERT or DELETE command that has been authorized and executed such that this trigger has been executed in response to the command. Let $k$ be the index of this command and $r' \in \lbrack r^k \rbrack_{\mathcal{W}}^{\mathcal{P}_u}$ (note that $r'$ is the run obtained from $r$ by dropping the last the operations associated with the command and the subsequent triggers).

From $r' \models W^{r', k}$, the fact that the operation has been authorized in $r^{k+1}$, ifEnf stability, and Proposition D.5, it follows that the k-th command (and all his triggers) is authorized also in $r'$. From this and the soundness of ifEnf, it follows that the observations produced by executing the command (and the subsequent triggers) are the same for all $r' \in \lbrack r^k \rbrack_{\mathcal{W}}^{\mathcal{P}_u}$. From this, the trigger in $r'$ effectively modifies the database policy, and the fact that policy changes are public events, it follows that the trigger $t$ modifies the policy in response to the command for any (suitable extension of a) run $r' \in \lbrack r^k \rbrack_{\mathcal{W}}^{\mathcal{P}_u}$ (otherwise the corresponding label would not be part of the observations associated with the k-th command). From this, $\text{secure}_{\phi}^{\mathcal{W}}(r, i - 1 \vdash u, \phi)$.

15. Trigger GRANT Disabled Backward. The proof is similar to that of Learn GRANT/REVOKE Backward.

16. Trigger REVOKE Disabled Backward. The proof for this case is similar to that of Trigger GRANT Disabled Backward.

17. Trigger INSERT FD Exception. Let $i$ be such that $r^i = r^{i-1}.t.s$, where $s = (db, U, sec, T, V, c) \in \Omega_M, last(r^{i-1}) = (db, U, sec', T, V, c')$, and $t$ be a trigger whose WHEN condition is $\phi$ and whose action act is a INSERT statement $(u', \mathcal{W}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{C})$. Furthermore, let $\phi \equiv \exists y, \mathcal{F}, \mathcal{G}, \mathcal{T}, \mathcal{R}(\mathcal{F}, \mathcal{G}, \mathcal{T}, \mathcal{R}) \land \mathcal{F} \neq \mathcal{G}$ (and let $\gamma$ be the constraint associated with $\phi$). From the rule definition, the trigger has been scheduled. From this, it follows that there has been a previous INSERT or DELETE command that has been authorized and executed such that this trigger has been executed in response to the command. Let $k$ be the index of this command and $r' \in \lbrack r^k \rbrack_{\mathcal{W}}^{\mathcal{P}_u}$, the fact that the k-th command has been authorized in $r^{k+1}$, ifEnf stability, and Proposition D.5, it follows that the k-th command is authorized also in $r'$. From this and the soundness of ifEnf, it follows that the observations produced by executing the command (and the subsequent triggers) are the same for all $r' \in \lbrack r^k \rbrack_{\mathcal{W}}^{\mathcal{P}_u}$ (otherwise the corresponding label would not be part of the observations associated with the k-th command). From this, $\text{secure}_{\phi}^{\mathcal{W}}(r, i - 1 \vdash u, \phi)$.

18. Trigger INSERT ID Exception. The proof for this case is similar to that of Trigger INSERT ID Exception.

19. Trigger DELETE ID Exception. The proof for this case is similar to that of Trigger DELETE ID Exception.

20. Trigger Exception. Let $i$ be such that $r^i = r^{i-1}.t.s$, where $s = (db, U, sec, T, V, c) \in \Omega_M, last(r^{i-1}) = (db, U, sec', T, V, c')$, and $t$ be a trigger whose WHEN condition is $\phi$ and whose action act is $\mathcal{A}$. From the rule definition, the trigger has been scheduled. From this, it follows that there has been a previous INSERT or DELETE command that has been authorized and executed such that this trigger has been executed in response to the command. Let $k$ be the index of this command and $r' \in \lbrack r^k \rbrack_{\mathcal{W}}^{\mathcal{P}_u}$, the fact that the k-th command has been authorized in $r^{k+1}$, Proposition D.5, and the stability of ifEnf, it follows that the k-th command is authorized also in $r'$. From this and the soundness of ifEnf, it follows that the observations produced by executing the command (and the associated triggers) in $r'$ are the same as those produced by executing the command in $r^k$. From this and $r'$, it follows that the trigger would violate the integrity constraint $\gamma$ also in any suitable extension of $r'$. From this and $\phi$ is the weakest precondition for violating $\gamma$ (see Proposition C.10), it follows that $\phi$ holds immediately before executing $t$ in the extension of $r'$. From this, $\text{secure}_{\phi}^{\mathcal{W}}(r, i - 1 \vdash u, \phi)$.

21. Trigger Rollback INSERT. Let $i$ be such that $r^i = r^{i-n-1}.(u, \mathcal{W}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{C})$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, and $t_1, \ldots, t_n \in \mathcal{T}_D$, and $\phi = \mathcal{A}(\mathcal{T})$. Furthermore, let $\text{last}(r^{i-n-1}) = (db, U, sec', T, V, c')$ and $s_n$ be $(db, U, sec, T, V, c)$. From the rule’s definition, it follows that one of the triggers fired by the INSERT command has thrown an exception. From
this and the LTS semantics, it follows that (1) $R(\overline{T})$ does not hold before executing the command (otherwise the command would not have had effect), and (2) $R(\overline{t})$ does not hold after executing the command (since the system rolled-back to the state before the INSERT). From this, it follows that for all runs $r' \in [r]^\infty_{\Omega}$ the INSERT command has violated $\gamma$. Therefore, $\secure{T}(r, i \vdash u \phi)$.

22. Trigger Rollback DELETE. The proof for this case is similar to that of Trigger Rollback INSERT.

23. Learn from deny – actions. We prove this case by contradiction. Assume that we can derive $r, i - 1 \vdash u \phi$ using the rule Learn from deny – actions. There are $r, r', r'' \in \traces(L)$, $1 \leq i \leq |r|$, $a \in A_{\Omega}$, $s, s' \in \Omega_{\Omega}$, and $\psi$ such that: $r' = r^{-1} \cdot a \cdot s$, $r'' = r^{-1} \cdot a \cdot s'' \neq r^{-1} \cdot a \cdot s$, $secEx(s') \neq secEx(s)$, $[\phi]_{a\cdot s} = T$, and $[\phi]_{a\cdot s''} = F$. From $r' = r^{-1} \cdot a \cdot s$ and $r'' = r^{-1} \cdot a \cdot s''$, it follows that $extend(r', a)$ and $extend(r'', a)$ are well-defined. From this, if $\text{Eqf}$'s soundness and stability, Proposition D.6, and $a \in A_{\Omega}$, $a$ preserves the equivalence class for $r^{-1}$, $\{M, db\text{Enf}\}$, and $\con{T}_{P, u}$. From $r^{-1} \con{T}_{P, u} r''$ and Proposition D.1, it follows that $r^{-1} \con{T}_{P, u} r''$. From this, $r^{-1} \con{T}_{P, u} r''$, $extend(r', a) = r'$, and $extend(r'', a) = r''$, it follows that $r^{-1} \con{T}_{P, u} r'$. This, however, contradicts $secEx(s') \neq secEx(s)$.

24. Learn from deny – triggers. The proof is similar to that of the case Learn from deny – actions. However, we use Proposition D.7 instead of Proposition D.6. This completes the proof of the step.

**Induction Step:** Assume that the claim hold for any derivation of $r, j \vdash u \psi$ such that $|r, j| \vdash u \phi$. We now prove that the claim also holds for $r, i \vdash u \phi$. There are a number of cases depending on the rule used to obtain $r, i \vdash u \phi$.

1. View. The proof of this case follows trivially from the semantics of the relational calculus extended over views.

2. Propagate Forward SELECT. Let $i$ be such that $r^{i+1} = r^i \cdot \langle u, SELECT, \psi \rangle \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_{\Omega}$ and $last(r') = \langle db', U', sec', T', V', c' \rangle$. From the rule, it follows that $r, i \vdash u \phi$ holds. From this and the induction hypothesis, it follows that $\secure{T}(r, i \vdash u \phi)$ holds. From Proposition D.6, the action $\langle u, SELECT, \psi \rangle$ preserves the equivalence class with respect to $r^i$, $P$, and $\con{T}_{P, u}$. From this, Lemma C.19, and $\secure{T}(r, i \vdash u \phi)$, it follows that also $\secure{T}(r, i \vdash u \phi)$ holds.

3. Propagate Forward GRANT/REVOKE. Let $i$ be such that $r^{i+1} = r^i \cdot \langle op, u', p, u \rangle \cdot s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_{\Omega}$ and $last(r') = \langle db', U', sec', T', V', c' \rangle$. From the rule, it follows that $r, i \vdash u \phi$ holds. From this and the induction hypothesis, it follows that $\secure{T}(r, i \vdash u \phi)$ holds. From Proposition D.6, the action $\langle op, u', p, u \rangle$ preserves the equivalence class with respect to $r^i$, $P$, and $\con{T}_{P, u}$. From this, Lemma C.20, and $\secure{T}(r, i \vdash u \phi)$, it follows that also $\secure{T}(r, i \vdash u \phi)$ holds.

4. Propagate Forward CREATE. The proof for this case is similar to that of Propagate Forward SELECT.

5. Propagate Backward SELECT. Let $i$ be such that $r^{i+1} = r^i \cdot \langle u, SELECT, \psi \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_{\Omega}$ and $last(r') = \langle db, U, sec, T, V, c \rangle$. From the rule, it follows that $r, i \vdash u \phi$ holds. From this and the induction hypothesis, it follows that $\secure{T}(r, i \vdash u \phi)$ holds. From Proposition D.6, the action $\langle u, SELECT, \psi \rangle$ preserves the equivalence class with respect to $r^i$, $P$, and $\con{T}_{P, u}$. From this, Lemma C.19, and $\secure{T}(r, i \vdash u \phi)$, it follows that also $\secure{T}(r, i \vdash u \phi)$ holds.

6. Propagate Backward GRANT/REVOKE. Let $i$ be such that $r^{i+1} = r^i \cdot \langle op, u', p, u \rangle \cdot s$, where $s = \langle db', U', sec', T', V', c' \rangle \in \Omega_{\Omega}$ and $last(r') = \langle db, U, sec, T, V, c \rangle$. From the rule, it follows that $r, i \vdash u \phi$ holds. From this and the induction hypothesis, it follows that $\secure{T}(r, i \vdash u \phi)$ holds. From Proposition D.6, the action $\langle op, u', p, u \rangle$ preserves the equivalence class with respect to $r^i$, $P$, and $\con{T}_{P, u}$. From this, Lemma C.20, and $\secure{T}(r, i \vdash u \phi)$, it follows that also $\secure{T}(r, i \vdash u \phi)$ holds.

7. Propagate Backward CREATE TRIGGER. The proof for this case is similar to that of Propagate Backward SELECT.

8. Propagate Backward CREATE VIEW. Note that the formulae $\psi$ and replace($\psi, a$) are semantically equivalent. This is the only difference between the proof for this case and the one for the Propagate Backward SELECT case.

9. Rollback Backward - I. Let $i$ be such that $r^i = r^{i-n-1} \cdot \langle u, \text{op}, R, T, J \rangle \cdot s_1 \cdot s_2 \cdot \ldots \cdot s_n$, where $s_1$, $s_2$, $\ldots$, $s_n \in \Omega_{\Omega}$, $t_1, \ldots, t_n \in T \text{RZGG}\text{R}_P$, and $\text{op}$ is one of $\{\text{INSERT}, \text{DELETE}\}$. Furthermore, let $s_n$ be $\langle db', U', sec', T', V', c' \rangle$ and $last(r^{i-n-1}) = \langle db, U, sec, T, V, c \rangle$. From the rule’s definition, $r, i \vdash u \phi$ holds. From this and the induction hypothesis, it follows that $\secure{T}(r, i \vdash u \phi)$.
From Proposition D.7, the trigger $t_j$ preserves the equivalence class with respect to $r^{n-1-j}P$, and $\equiv^W_{P,u}$ for any $1 \leq j \leq n$. Therefore, for any $v \in [r^{n-1}]^W_{P,u}$, the run $e(v,t_u)$ contains the roll-back. Therefore, for any $v \in [r^{n-1}]^W_{P,u}$, the state last($e(v,t_u)$) is the state just before the action $(u, op, R,T)$. Let $A$ be the set of system states associated with the roll-back states. It is easy to see that $A$ is the same as $\{\text{sysState}(\text{last}(t')) | t' \in [r^{n-1}]^W_{P,u} \}$. Therefore, for any $\phi \in [r^{n-1}]^W_{P,u}$, the state last($\phi$, $t_u$) is the state just before the action $(u, op, R,T)$. Let $A$ be the set of system states associated with the roll-back states. It is easy to see that $A$ is the same as $\{\text{sysState}(\text{last}(t')) | t' \in [r^{n-1}]^W_{P,u} \}$. From $\text{sec}_{W,P,u}(r,i \vdash \phi)$, it follows that $\phi$ has the same result over all states in $A$. From this and $A = \{\text{sysState}(\text{last}(t')) | t' \in [r^{n-1}]^W_{P,u} \}$, it follows that $\phi$ has the same result over all states in $\{\text{sysState}(\text{last}(t')) | t' \in [r^{n-1}]^W_{P,u} \}$. From this, it follows that $\text{sec}_{W,P,u}(r,i - n - 1 \vdash \phi)$ holds.

10. **Rollback Backward - 2.** Let $i$ be such that $r' = r^{n-1}.(u, op, R,T).s$, where $s = (db', U', sec', T', V', c) \in \Omega_M$, last($r^{n-1}$) = $\langle db, U, sec, T, V, c \rangle$. From the rule's definition, $r,i \vdash \phi$ holds. From this and the induction hypothesis, it follows that $\text{sec}_{W,P,u}(r,i \vdash \phi)$ holds. From Proposition D.6, the action $(u, op, R,T)$ preserves the equivalence class with respect to $r^{n-1}$, $P$, and $\equiv^W_{P,u}$. From this, Lemma C.18, the fact that the action does not modify the database state, and $\text{sec}_{W,P,u}(r,i \vdash \phi)$, it follows that $\text{sec}_{W,P,u}(r,i - 1 \vdash \phi)$ holds.

11. **Rollback Forward - 1.** Let $i$ be such that $r' = r^{n-1}.(u, op, R,T).s_1.t_1.s_2.t_2.\ldots.t_n.s_n$, where $s_1, s_2, \ldots, s_n \in \Omega_M$, $t_1, t_2, \ldots, t_n \in \text{TRIGGER}_P$, and $op$ is one of $\{\text{INSERT, DELETE} \}$. Furthermore, let $s_n = (db, U, sec, T, V, c)$ and last($r^{n-1}$) = $(db', U', sec', T', V', c)$. From the rule's definition, $r,i - n - 1 \vdash \phi$ holds. From this and the induction hypothesis, it follows that $\text{sec}_{W,P,u}(r,i \vdash \phi)$. From Proposition D.7, the trigger $t_j$ preserves the equivalence class with respect to $r^{n-1}$, $P$, and $\equiv^W_{P,u}$, for any $1 \leq j \leq n$. Independently on the cause of the roll-back (either a security exception or an integrity constraint violation), we claim that the set $A$ of rollback system states is $\{\text{sysState}(\text{last}(t')) | t' \in [r^{n-1}]^W_{P,u} \}$. From $\text{sec}_{W,P,u}(r,i \vdash \phi)$, it follows that $\text{sec}_{W,P,u}(r,i - 1 \vdash \phi)$ holds.

We now prove our claim. It is trivial to see (from the LTS's semantics) that the set of rollback's states is a subset of $\{\text{sysState}(\text{last}(v)) | v \in [r^{n-1}]^W_{P,u} \}$. For the other direction, assume, for contradiction's sake, that there is a state in $\{\text{sysState}(\text{last}(v)) | v \in [r^{n-1}]^W_{P,u} \}$ that is not a rollback state for the runs in $[r]_{P,u}$. This is impossible since all triggers $t_1, \ldots, t_n$ preserve the equivalence class.

12. **Rollback Forward - 2.** Let $i$ be such that $r' = r^{n-1}.(u, op, R,T).s$, where $op \in \{\text{INSERT, DELETE} \}$, $s = (db, U, sec, T, V, c) \in \Omega_M$, and last($r^{n-1}$) = $(db', U', sec', T', V', c)$. From the rule's definition, $r,i - 1 \vdash \phi$ holds. From this and the induction hypothesis, it follows that $\text{sec}_{W,P,u}(r,i \vdash \phi)$. From Proposition D.6, the action $(u, op, R,T)$ preserves the equivalence class with respect to $r^{n-1}$, $P$, and $\equiv^W_{P,u}$. From this, Lemma C.18, the fact that the action does not modify the database state, and $\text{sec}_{W,P,u}(r,i - 1 \vdash \phi)$, it follows that also $\text{sec}_{W,P,u}(r,i \vdash \phi)$ holds.

13. **Propagate Forward INSERT/DELETE Success.** Let $i$ be such that $r' = r^{n-1}.(u, op, R,T).s$, where $op \in \{\text{INSERT, DELETE} \}$, $s = (db, U, sec, T, V, c) \in \Omega_M$, and last($r^{n-1}$) = $(db', U', sec', T', V', c)$. From the rule's definition, $r,i - 1 \vdash \phi$ holds. From this and the induction hypothesis, it follows that $\text{sec}_{W,P,u}(r,i \vdash \phi)$. From Proposition D.6, the action $(u, op, R,T)$ preserves the equivalence class with respect to $r^{n-1}$, $P$, and $\equiv^W_{P,u}$. From this, Lemma C.18, and $\text{sec}_{W,P,u}(r,i - 1 \vdash \phi)$, it follows that $\text{sec}_{W,P,u}(r,i \vdash \phi)$ holds.

14. **Propagate Forward INSERT Success - 1.** Let $i$ be such that $r' = r^{n-1}.(u, op, R,T).s$, where $op$ is one of $\{\text{INSERT, DELETE} \}$, $s = (db, U, sec, T, V, c) \in \Omega_M$, and last($r^{n-1}$) = $(db', U', sec', T', V', c)$. From the rule's definition, $r,i - 1 \vdash \phi$ holds. From this and the induction hypothesis, it follows that $\text{sec}_{W,P,u}(r,i - 1 \vdash \phi)$ holds. From Proposition D.6, the action $(u, op, R,T)$ preserves the equivalence class with respect to $r^{n-1}$, $P$, and $\equiv^W_{P,u}$. We claim that the execution of $(u, \text{INSERT}, R,T)$ does not alter the content of the tables in $\text{tables}(\phi)$. From this, $\text{sec}_{W,P,u}(r,i - 1 \vdash \phi)$, and Lemma C.18, it follows that $\text{sec}_{W,P,u}(r,i \vdash \phi)$ holds.

We now prove our claim that the execution of $(u, \text{INSERT}, R,T)$ does not alter the content of the tables in $\text{tables}(\phi)$. From the rule's definition, it follows that $r,i - 1 \vdash \phi$, $R(T)$ holds. From this and the soundness of $\vdash$, it follows that $[R(T)]^\text{last}(r^{n-1}) \vdash \phi$, $R(T)$ holds. From this, Lemma C.18, and $\text{sec}_{W,P,u}(r,i \vdash \phi)$, it follows that $\text{sec}_{W,P,u}(r,i \vdash \phi)$ holds.
15. Propagate Forward DELETE Success - 1. The proof for this case is similar to that of Propagate Forward INSERT Success - 1.

16. Propagate Backward INSERT/DELETE Success. Let $i$ be such that $r^i = r^{i-1} \langle \Omega, \alpha, \beta, \gamma \rangle$, where $\alpha$ is one of \{INSERT, DELETE\}. Let $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$. From the rule's definition, $r, i \vdash_\alpha \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\phi}^W (r, i \vdash_\alpha \phi)$ holds. From Proposition D.6, the action $\langle \Omega, \alpha, \beta, \gamma \rangle$ preserves the equivalence class with respect to $r^{i-1}$, $P$, and $\exists W \in \Omega_M$. From Proposition C.18, it follows that the execution of $\langle \Omega, \alpha, \beta, \gamma \rangle$ does not alter the content of the tables $\phi$ for any $v \in \Omega_M$.

17. Propagate Backward INSERT Success - 1. Let $i$ be such that $r^i = r^{i-1} \langle \Omega, \alpha, \beta, \gamma \rangle$, where $\alpha$ is one of \{INSERT, DELETE\}. Let $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db', U', sec', T', V', c' \rangle$. From the rule's definition, $r, i \vdash_\alpha \phi$ holds. From this and the induction hypothesis, it follows that $secure_{\phi}^W (r, i \vdash_\alpha \phi)$ holds. From Proposition D.6, the action $\langle \Omega, \alpha, \beta, \gamma \rangle$ preserves the equivalence class with respect to $r^{i-1}$, $P$, and $\exists W \in \Omega_M$. We claim that the execution of $\langle \Omega, \alpha, \beta, \gamma \rangle$ does not alter the content of the tables $\phi$ for any $v \in \Omega_M$.

18. Propagate Backward DELETE Success - 1. The proof for this case is similar to that of Propagate Forward DELETE Success - 1.

19. Reasoning. Let $\Delta$ be a subset of $\{ \delta \mid r, i \vdash_\alpha \delta \}$ and $last(r^i) = \langle db, U, sec, T, V, c \rangle$. From the induction hypothesis, it follows that $secure_{\phi}^W (r, i \vdash_\alpha \delta)$ holds for any $\delta \in \Delta$. Note that, given any $\delta \in \Delta$, from $r, i \vdash_\alpha \delta$ and the soundness of $\vdash_\alpha$ (see Proposition C.1), it follows that $\delta$ holds in $last(r^i)$. From this, $secure_{\phi}^W (r, i \vdash_\alpha \delta)$ holds for any $\delta \in \Delta$, $\Delta = \vdash_\alpha \phi$, and Lemma C.16, it follows that $secure_{\phi}^W (r, i \vdash_\alpha \phi)$ holds.

20. Learn INSERT Backward - 3. Let $i$ be such that $r^i = r^{i-1} \langle \Omega, INSERT, R, \gamma \rangle$, where $\gamma = \langle db', U', sec', T', V', c' \rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$ and $\phi \models \neg (R(\gamma))$. Furthermore, we denote by $a$ the INSERT action. From the rule, it follows that $r, i - 1 \vdash_\psi \psi$ and $r, i \vdash_\psi \neg \psi$ hold. From $r, i - 1 \vdash_\psi \psi$, $r, i \vdash_\psi \neg \psi$, and $\neg \psi$'s soundness, it follows that $\neg \psi$ holds in $last(r^i)$. From this, $Secure_{\phi}^W (r, i \vdash_\alpha \phi)$ holds. From this and the induction hypothesis, it follows that $Secure_{\phi}^W (r, i - 1 \vdash_\alpha \psi)$ and $Secure_{\phi}^W (r, i \vdash_\alpha \phi)$ hold.

21. Learn DELETE Backward - 3. The proof for this case is similar to that of Learn INSERT Backward - 3.

22. Propagate Forward Disabled Trigger. Let $i$ be such that $r^i = r^{i-1} \cdot t.s$, where $s = \langle db, U, sec, T, V, c \rangle \in \Omega_M$ and $last(r^{i-1}) = \langle db, U, sec, T, V, c \rangle$, and $t$ be a trigger. Furthermore, let $\psi$ be $t$'s condition where all free variables are replaced with $t(p(last(r^{i-1})))$. From the rule, it follows that $r, i - 1 \vdash_\alpha \phi$. From this and the induction hypothesis, it follows that $secure_{\phi}^W (r, i - 1 \vdash_\alpha \phi)$ holds. Furthermore, from Proposition D.7, it follows that $t$ preserves the equivalence class with respect to $r^{i-1}$, $P$, and $\exists \Omega_M$. If the trigger's action is an INSERT or a DELETE operation, we claim that the operation does not change the content of any table $\phi$ for any $v \in \Omega_M$. We also claim that executing the trigger $t$ in any run $v \in \Omega_M$ does not generate security or integrity exceptions. From these claims, the fact that $t$ preserves the equivalence class with respect to $r^{i-1}$, $P$, and $\exists \Omega_M$, Lemma C.21, and $secure_{\phi}^W (r, i \vdash_\alpha \phi)$, it follows that also $secure_{\phi}^W (r, i \vdash_\alpha \phi)$ holds.

We now prove our claim. Assume that $t$'s action in either an INSERT or a DELETE operation. From the rule, it follows that $r, i - 1 \vdash_\alpha \neg \psi$. From this and the soundness of $\vdash_\alpha$ (see Proposition C.1), $\neg \psi$ holds. From $r, i - 1 \vdash_\alpha \neg \psi$ and the induction hypothesis, it follows that $secure_{\phi}^W (r, i - 1 \vdash_\alpha \psi)$ holds. From this and $\neg \psi$'s soundness, it follows that $Secure_{\phi}^W (r, i \vdash_\alpha \phi)$ holds. From this and $\neg \psi$'s soundness, it follows that $Secure_{\phi}^W (r, i - 1 \vdash_\alpha \phi)$ holds.
We now prove our claim that executing the trigger in any run $r \in [r^{-1}]^{W}_P$. does not generate security or integrity exceptions. This directly follows from (1) executing $t$ in $r^{-1}$ does not throw an exception, and (2) $t$ preserves the equivalence class with respect to $r^{-1}$, $P$, and $\geq^{W}_P$.

23. Propagate Forward Disabled Trigger. The proof for this case is similar to that of Propagate Forward Disabled Trigger.

24. Learn INSERT Forward. Let $i$ be such that $r^i = r^{-1} \cdot t \cdot s$, where $s = (db, U, sec, T, V, c) \in \Omega_M$, last($r^{-1}$) = $(db, U, sec, T, V, c)$, and $t$ is a trigger, and $\phi$ be $R(\overline{t})$. Furthermore, let $\psi$ be $t$'s condition where all free variables are replaced with $tpl(last(r^{-1}))$. From the rule's definition, it follows that $t$'s action is $(u', \text{INSERT}, R, \overline{t})$ and that $r, i - 1 \vdash u, \psi$ holds. From the soundness of $\vdash u$ (see Proposition C.1) and $r, i - 1 \vdash u, \psi$, it follows that $\psi_{\text{last}(r^{-1}), db} = \top$. From $r, i - 1 \vdash u, \psi$ and the induction hypothesis, it follows that $\text{secure}_{W}^{P,u}(r, i - 1 \vdash u, \psi)$. From this and $\psi_{\text{last}(r^{-1}), db} = \top$, it follows that $\psi_{\text{last}(r), db} = \top$ for any $r' \in [r^{-1}]_{P,u}$. From the rule definition, the trigger has been scheduled. From this, it follows that there has been a previous INSERT or DELETE command that has been authorized and executed such that this trigger has been executed in response to the command. Let $k$ be the index of this command and $r' \in [r^{-1}]_{P,u}$. From $r' \geq^{P,u} r^k$, the fact that the $k$-th command has been authorized in $r^{k+1}$, Proposition D.5, and the stability of $ifEnf$, it follows that the $k$-th command is authorized also in $r'$. From this and the soundness of $ifEnf$, it follows that the observations produced by executing the command (and the associated triggers) in $r'$ are the same as those produced by executing the command in $r^k$. From $\psi_{\text{last}(r'), db} = \top$ for any $r' \in [r^{-1}]_{P,u}$, it follows that $t$ is enabled in any $r' \in [r^{-1}]_{P,u}$. From this and the fact that the observations are the same, it follows that extending $r'$ with $t$ does not throw exceptions. From this and the fact that $t$ is enabled, it follows that $R(\overline{t})$ holds in $last(r')$. From this, $\text{secure}_{W}^{P,u}(r, i - 1 \vdash u, \psi)$. From this and the soundness of $\vdash u$, it follows that $\psi_{\text{last}(r^{-1}), db} = \top$. From $r, i - 1 \vdash u, \psi$ and the induction hypothesis, it follows $\text{secure}_{W}^{P,u}(r, i - 1 \vdash u, \psi)$. From this and $\psi_{\text{last}(r^{-1}), db} = \top$, it follows that $\psi_{\text{last}(r), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$. From the rule definition, it follows that executing $t$ in $r'$ does not violate $\gamma$. From this and the fact that $\phi$ is the weakest precondition for not violating $\gamma$ (see Proposition C.10), it follows that $\phi_{\text{last}(r^{-1}), db} = \top$. From Proposition D.5, the stability, and the soundness of $ifEnf$, it follows that $t$ is executed and authorized also in $r' \in [r^{-1}]_{P,u}$ without throwing exceptions. From this and $\psi_{\text{last}(r'), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$, it follows that $t$ is enabled in $r'(r', t)$. From this, Proposition C.10, and the fact that $t$ does not violate $\gamma$, it follows that $\phi_{\text{last}(r^{-1}), db} = \top$. From this, $\phi_{\text{last}(r'), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$. From this, $\text{secure}_{W}^{P,u}(r, i - 1 \vdash u, \phi)$.

25. Learn INSERT - FD. Let $i$ be such that $r^i = r^{-1} \cdot t \cdot s$, where $s = (db, U, sec, T, V, c) \in \Omega_M$, last($r^{-1}$) = $(db', U', sec', T', V', c')$, and $t \in TRIGGER D$, and $\phi$ be $\forall \overline{\tau}, \overline{\pi}, r(\overline{\tau}, \overline{\pi}, \overline{\tau}) \wedge \overline{\pi} \neq \overline{\tau}$ (and let $\gamma$ be the associate integrity constraint). Furthermore, let $\psi$ be $t$'s condition where all free variables are replaced with the values in $tpl(last(r^{-1}))$ and $(u', \text{INSERT}, R, (\overline{\tau}, \overline{\pi}, \overline{\tau}))$ be $t$'s actual action. From the rule, it follows that $r, i - 1 \vdash u, \psi$. From this and the soundness of $\vdash u$, it follows that $\psi_{\text{last}(r^{-1}), db} = \top$. From $r, i - 1 \vdash u, \psi$ and the induction hypothesis, it follows $\text{secure}_{W}^{P,u}(r, i - 1 \vdash u, \psi)$. From this and $\psi_{\text{last}(r^{-1}), db} = \top$, it follows that $\psi_{\text{last}(r), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$. From the rule definition, it follows that $t$ in $r'$ does not violate $\gamma$. From this and the fact that $\phi$ is the weakest precondition for not violating $\gamma$ (see Proposition C.10), it follows that $\phi_{\text{last}(r^{-1}), db} = \top$. From Proposition D.5, the stability, and the soundness of $ifEnf$, it follows that $t$ is executed and authorized also in $r' \in [r^{-1}]_{P,u}$ without throwing exceptions. From this and $\psi_{\text{last}(r'), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$, it follows that $t$ is enabled in $r'(r', t)$. From this, Proposition C.10, and the fact that $t$ does not violate $\gamma$, it follows that $\phi_{\text{last}(r^{-1}), db} = \top$. From this, $\phi_{\text{last}(r'), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$. From this, $\text{secure}_{W}^{P,u}(r, i - 1 \vdash u, \phi)$.

26. Learn INSERT - FD - 1. The proof of this case is similar to that of Learn INSERT - FD.

27. Learn INSERT - ID. The proof of this case is similar to that of Learn INSERT - FD. See also the proof of INSERT Success - ID.

29. Learn INSERT Backward - 1. Let $i$ be such that $r^i = r^{-1} \cdot t \cdot s$, where $s = (db', U', sec', T', V', c') \in \Omega_M$, last($r^{-1}$) = $(db, U, sec, T, V, c)$, and $t \in TRIGGER D$, and $\phi$ be $t$'s actual action when condition, where all free variables are replaced with the values in $tpl(last(r^{-1}))$. From the rule's definition, it follows that $t$ is authorized and it does not throw an exception. From this, $dbEnf$'s definition, Proposition D.5, and the stability and soundness of $ifEnf$, the same holds for $t$ in $e(r', t)$ such that $r' \in [r^{-1}]_{P,u}$. From the rule, $\vdash u$'s soundness (see Proposition C.1), and the induction hypothesis, it follows that $\psi_{\text{last}(r'), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$. From this and the fact that $t$ does not throw exceptions in $e(r', t)$, it follows that $t$ is enabled in $e(r', t)$. From this, $\phi_{\text{last}(r'), db} = \top$ for all $r' \in [r^{-1}]_{P,u}$. From this, $\text{secure}_{W}^{P,u}(r, i - 1 \vdash u, \psi)$.

30. Learn INSERT Backward - 2. Let $i$ be such that $r^i = r^{-1} \cdot t \cdot s$, where $s = (db', U', sec', T', V', c') \in \Omega_M$, last($r^{-1}$) = $(db, U, sec, T, V, c)$, and $t \in TRIGGER D$, and $\phi$ be $R(\overline{t})$. Furthermore, let act = $(u', \text{INSERT}, R, \overline{t})$ be $t$'s actual action and $\gamma$ be $t$'s actual action when condition obtained by replacing all free variables with the values in $tpl(last(r^{-1}))$. From the rule's definition, there is a $\psi$ such that $r, i - 1 \vdash u, \psi$ and $r, i \vdash u, \neg \psi$. From this, the rule's definition, it follows that $t$ is authorized and it does not throw an exception. From this, $dbEnf$'s definition, Proposition D.5, and the stability and soundness of $ifEnf$, the same holds for $t$ in $e(r', t)$ such that $r' \in [r^{-1}]_{P,u}$. From the rule, $\vdash u$'s soundness, and the induction hypothesis, it follows that $\psi_{\text{last}(r')}$ does not throw exceptions. From this, $\psi_{\text{last}(r')}$ does not throw exceptions.
D.1. From Database Access Control to Information-flow Control

249

31. Learn DELETE Forward. The proof of this case is similar to that of Learn INSERT Forward.

32. Learn DELETE - ID. The proof of this case is similar to that of Learn INSERT - FD. See also the proof of DELETE Success - ID.

33. Learn DELETE - ID - 1. The proof of this case is similar to that of Learn DELETE - ID.

34. Learn DELETE Backward - 1. The proof is similar to that of Learn INSERT Backward - 1.

35. Learn DELETE Backward - 2. The proof is similar to that of Learn INSERT Backward - 2.

36. Propagate Forward Trigger Action. Let \( i \) be such that \( r' = r'^{-1} - t.s \), where \( t \) is a trigger, \( s = \langle db, U, sec, T, V, c \rangle \in \Omega_M \) and \( last(r'^{-1}) = \langle \overline{db}', U', \overline{sec}', T', V', c' \rangle \). From the rule's definition, \( r, i-1 \vdash_u \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{wu}(r, i-1 \vdash_u \phi) \) holds. From Proposition D.7, the trigger \( t \) preserves the equivalence class with respect to \( r'^{-1} \), \( P \), and \( \Xi_{wu} \). We claim that (1) the execution of \( t \) does not alter the content of the tables in \( \text{tables}(\phi) \), and (2) executing the trigger \( t \) in any run \( v \in [r'^{-1}]_{wu} \) does not generate security or integrity exceptions. From these claims, Lemma C.21, and \( \text{secure}_{wu}(r, i-1 \vdash_u \phi) \), it follows that also the judgment \( r, i \vdash_u \phi \) is secure, i.e., \( \text{secure}_{wu}(r, i \vdash_u \phi) \) holds.

We now prove our claim that the execution of \( t \) does not alter the content of the tables in \( \text{tables}(\phi) \). If the trigger is not enabled, proving the claim is trivial. In the following, we assume the trigger is enabled. There are four cases:

- \( t \)'s action is an INSERT statement. This case amount to claiming that the INSERT statement \( \langle u', 'INSERT, R, T \rangle \) does not alter the content of the tables in \( \text{tables}(\phi) \) in case \( \text{revise}(r'^{-1}, \phi, r') = \top \). We proved the claim above in the Propagate Forward INSERT/DELETE Success case.

- \( t \)'s action is a DELETE statement. The proof is similar to that of the INSERT case.

- \( t \)'s action is a GRANT statement. In this case, the action does not alter the database state and the claim follows trivially.

- \( t \)'s action is a REVOKE statement. The proof is similar to that of the GRANT case.

We now prove our claim that executing the trigger \( t \) in any run \( v \in [r'^{-1}]_{wu} \) does not generate security or integrity exceptions. This directly follows from (1) executing \( t \) in \( r'^{-1} \) does not throw an exception, and (2) \( t \) preserves the equivalence class with respect to \( r'^{-1} \), \( P \), and \( \Xi_{wu} \).

37. Propagate Backward Trigger Action. The proof of this case is similar to Propagate Backward Trigger Action.

38. Propagate Forward INSERT Trigger Action. Let \( i \) be such that \( r' = r'^{-1} - t.s \), where \( t \) is a trigger, \( s = \langle db, U, sec, T, V, c \rangle \in \Omega_M \) and \( last(r'^{-1}) = \langle \overline{db}', U', \overline{sec}', T', V', c' \rangle \). From the rule's definition, \( r, i-1 \vdash_u \phi \) holds. From this and the induction hypothesis, it follows that \( \text{secure}_{wu}(r, i-1 \vdash_u \phi) \) holds. From Proposition D.7, the trigger \( t \) preserves the equivalence class with respect to \( r'^{-1} \), \( P \), and \( \Xi_{wu} \). We claim that (1) the execution of \( t \) does not alter the content of the tables in \( \text{tables}(\phi) \), and (2) executing the trigger \( t \) in any run \( v \in [r'^{-1}]_{wu} \) does not generate security or integrity exceptions. From these claims, Lemma C.21, and \( \text{secure}_{wu}(r, i-1 \vdash_u \phi) \), it follows that also the judgment \( r, i \vdash_u \phi \) is secure, i.e., \( \text{secure}_{wu}(r, i \vdash_u \phi) \).

We now prove our claim that the execution of \( t \) does not alter the content of the tables in \( \text{tables}(\phi) \). If the trigger is not enabled, the claim is trivial. In the following, we assume the trigger is enabled. Then, \( t \)'s action is an INSERT statement. This case amount to claiming that the INSERT statement \( \langle u', 'INSERT, R, T \rangle \) does not alter the content of the tables in \( \text{tables}(\phi) \) in case \( r, i-1 \vdash_u R(T) \) holds. We proved the claim above in the Propagate Forward INSERT Success - 1 case.

We now prove our claim that executing the trigger \( t \) in any run \( v \in [r'^{-1}]_{wu} \) does not generate security or integrity exceptions. This directly follows from (1) executing \( t \) in \( r'^{-1} \) does not throw an exception, and (2) \( t \) preserves the equivalence class with respect to \( r'^{-1} \), \( P \), and \( \Xi_{wu} \).

39. Propagate Forward DELETE Trigger Action. The proof of this case is similar to that of Propagate Forward INSERT Trigger Action.

40. Propagate Backward DELETE Trigger Action. The proof of this case is similar to that of Propagate Forward INSERT Trigger Action.

41. Propagate Backward DELETE Trigger Action. The proof of this case is similar to that of Propagate Forward INSERT Trigger Action.
42. **Trigger FD INSERT Disabled Backward.** Let \( i \) be such that \( r^i = r^{i-1} \cdot t \cdot s \), where \( s = (dv', U', sec', T', V', c') \in \Omega_u \), \( t \in TRIGGER_{D} \), last\((r^{i-1}) = (dv, U, sec, T, V, c) \), and \( \phi \) be \( t \)'s actual action. Then if \( E \) is sound with respect to the progress-sensitive variant of Definition 7.2 given last\((r^i) \) (cf. [185]). Let \( s' \) be \( s \)'s actual action and \( \psi = E(\pi, s') \land \neg \psi \). From the rule’s definition, it follows that (1) the trigger has been authorized, (2) \( r^i \) depends on the memory’s content. From this, it follows that \( s^i = (ву, u' \cdot s' \cdot 휶, 면, 개, 'ג) \) holds, where \( \psi \) is the weakest precondition associated with throwing an exception for one of the constraints \( \gamma \) (see Proposition C.10). From \( r^i \), \( i \leftarrow 1 \) \( \cdot u \) \ and the induction hypothesis, it follows that \( \text{secure}_W^\mu (r, i \leftarrow 1 \cdot u, \psi) \) holds for all runs \( r^i \in [r^{i-1}]_W^\mu \). From this and the fact that the trigger has been executed, it also follows that the exception is not thrown if we extend \( r^i \in [r^{i-1}]_W^\mu \) with \( t \). This happens if the trigger’s actual action \( \phi \) is not satisfied. From this, \( \text{secure}_W^\mu (r, i \leftarrow 1 \cdot u, \neg \psi) \).

43. **Trigger ID INSERT Disabled Backward.** The proof of this case is similar to that of Trigger FD INSERT Disabled Backward.

44. **Trigger ID DELETE Disabled Backward.** The proof of this case is similar to that of Trigger FD INSERT Disabled Backward.

This completes the proof of the induction step.

D.1.7 Security in the Truman model

We now prove that our construction for the Truman model is secure as well. Note that in the following we restrict our setting to non-boolean SELECT queries (the only one for which the Truman model semantics has been defined). For boolean queries, we can directly resort to the mechanism produced using Reduction 7.1.

**Theorem D.2.** Let \( M \) be a system configuration, ifEnf be an enforcement mechanism for WHILE SQL programs, \( u \) be a user, and dbEnf be the database access control mechanism for the Truman model semantics constructed using Reduction 7.1 and Reduction 7.2. If for all user \( u \in \text{UID} \), (1) ifEnf is sound with respect to the progress-sensitive variant of Definition 7.2 given \( E'_{\text{user}} \) and \( E''_{\text{user}} \), and (2) ifEnf is \( u \)-stable, then dbEnf is secure and sound for the Truman model, i.e., it satisfies Definitions 3.8 and 3.18.

**Proof.** We show security for a single non-boolean query. The security of the multi-query case follows directly (cf. [165]). Let \( \{ \pi | \varphi \} \) be a non-boolean query, \( s \) be a database state, and \( T \) be \( \{ \pi | \varphi \} \)’s result on \( s \). For each of the tuples in \( T \), we check whether the boolean query \( \varphi [\pi \rightarrow \pi] \) is authorized by the mechanism \( NTdbEnf \) constructed using Reduction 7.1. From Theorem 7.1, it follows that \( NTdbEnf \) provides data confidentiality. From this, it follows that the result of \( \varphi [\pi \rightarrow \pi] \) is the same for all indistinguishable database states. From this and \( T \in T \), it follows that \( T \in [\pi | \varphi] \) for all \( s' \) indistinguishable from \( s \). From this, it follows that \( dbEnf \) is secure in the sense of Definition 3.8. The soundness of \( dbEnf \) (see Definition 3.18) directly follows from the fact that the result returned by \( NTdbEnf \) is always a subset of the result of the query \( \{ \pi | \varphi \} \) for all \( s' \) that are indistinguishable from \( s \) for the user \( u \).}

D.1.8 From internal attackers to external attackers

Here, we prove the results from Section 7.5.5.

**Lemma D.1.** Let \( S \) be the sequential scheduler \( 0^\omega \), \( C \in \text{Com}_{\text{UID}} \) be a sequence of simple query-only programs, and \( s \) be a system state. For any two sequences of memories \( M_1, M_2 \in \text{Mem}_{\text{UID}} \), if \( M_1 = \mu_1 \), \( M_2 = \mu_2 \), \( M_1 \) and \( M_2 \) are for all \( 1 \leq i \leq |C| \), \( u_i = u_i \) and \( u_i = u_1 \) (where \( C(i) = \langle u_i, c_i \rangle \)), \( M_1(i) = \langle u_1, m_1 \rangle \), and \( M_2(i) = \langle u_1, m_2 \rangle \), the following conditions hold:

- If there are \( n, \tau, C', M'_1, s', ctx', S' \) such that \( (C, M_1, (s, ctx), S) \xrightarrow{n} (C', M'_1, (s', ctx'), S') \), then there exists an \( M'_2 \) such that \( (C, M_2, (s, ctx), S) \xrightarrow{n} (C', M'_2, (s', ctx'), S'). \)

- If there are \( n, \tau, C', M'_2, s', ctx', S' \) such that \( (C, M_2, (s, ctx), S) \xrightarrow{n} (C', M'_2, (s', ctx'), S') \), then there exists an \( M'_1 \) such that \( (C, M_1, (s, ctx), S) \xrightarrow{n} (C', M'_1, (s', ctx'), S'). \)

**Proof.** The lemma directly follows from (1) the programs in \( C \) are simple query-only programs, (2) the result of a query statement depends only on the content of the database, and (3) simple query-only programs do not use program variables inside queries. Hence, the computation does not depend on the memories in the initial configuration.
For simple query-only programs, we can simplify the security condition by ignoring memories in the global configurations, as stated in Lemma D.2. In the following, given a set of global configurations \( S \), we denote by \( DB(S) \) the set of databases in \( S \), i.e., \( DB(S) = \{ s \mid \exists M, (M, s) \in S \} \).

**Lemma D.2.** Let \( S \) be the sequential scheduler \( 0^\infty \), \( C \in Com^{*}_{\mathcal{M}} \) be a sequence of simple query-only programs, \( s \) be a system state, \( M \in Mem_{\mathcal{M}}^{\infty} \) be a sequence of memories, \( u \in UID \) be a user. The sequence of programs \( C \) is secure with respect to \( u \) for \( S, s, M, E^{\infty} \), and \( S^{\infty} \) for the progress-sensitive variant of Definition 7.2 iff for all epochs \((C_i, M_i, (s_i, ctx_i), S_i)\) in \( [E^{\infty}, S^{\infty}]_{\approx u}((s, \mathcal{C}, M, \tau)) \), \( DB(P_{\mathcal{D}(u)}(M_i, s_i), C_i, S_i, \tau)) = DB((M_i, s_i))_{u} \).

**Proof.** Let \( S \) be the sequential scheduler \( 0^\infty \), \( C \in Com^{*}_{\mathcal{M}} \) be a sequence of simple query-only programs, \( s \) be a system state, \( M \in Mem_{\mathcal{M}}^{\infty} \) be a sequence of memories, \( u \in UID \) be a user.

(\( \Rightarrow \)). Assume that \( C \) is secure with respect to \( u \) for \( S, s, M, E^{\infty} \), and \( S^{\infty} \) for the progress-sensitive variant of Definition 7.2. Assume, further, for contradiction’s sake, that there is an epoch \((C_i, M_i, (s_i, ctx_i), S_i)\) in \( [E^{\infty}, S^{\infty}]_{\approx u}((s, \mathcal{C}, M, \tau)) \) such that \( DB(P_{\mathcal{D}(u)}(M_i, s_i), C_i, S_i, \tau)) \neq DB((M_i, s_i))_{u} \). This happens iff \( P_{\mathcal{D}(u)}(M_i, s_i), C_i, S_i, \tau)) \neq (M_i, s_i), contradicting the security of \( C \).

(\( \Leftarrow \)). Assume that for all epochs \((C_i, M_i, (s_i, ctx_i), S_i)\) in \( [E^{\infty}, S^{\infty}]_{\approx u}((s, \mathcal{C}, M, \tau)) \), \( DB(P_{\mathcal{D}(u)}(M_i, s_i), C_i, S_i, \tau)) = DB((M_i, s_i))_{u} \). Assume, further, for contradiction’s sake, that \( C \) is not secure with respect to \( u \) for \( S, s, M, E^{\infty} \), and \( S^{\infty} \) for the progress-sensitive variant of Definition 7.2. Hence, there is an epoch \((C_i, M_i, (s_i, ctx_i), S_i)\) in \( [E^{\infty}, S^{\infty}]_{\approx u}((s, \mathcal{C}, M, \tau)) \) such that \( P_{\mathcal{D}(u)}(M_i, s_i), C_i, S_i, \tau)) \neq (M_i, s_i), contradicting the difference must have been caused by differences in the initial memories. Since the programs in \( C \) are simple and query-only, this contradicts Lemma D.1.

Lemma D.3 states that, for the sequential scheduler, simple query-only programs, and user-based epochs, if the program corresponding to each epoch is secure, then so is the original program.

**Lemma D.3.** Let \( S \) be the sequential scheduler \( 0^\infty \), \( C \in Com^{*}_{\mathcal{M}} \) be a sequence of simple query-only programs, \( s \) be a system state, \( M \in Mem_{\mathcal{M}}^{\infty} \) be a sequence of memories, \( u \in UID \) be a user. Furthermore, let \( r \) be the longest run obtained starting from \((C, M, \langle s, \mathcal{C}, M, \tau \rangle)\). If for all epochs \((C_i, M_i, (s_i, ctx_i), S_i)\) in \( [E^{\infty}, S^{\infty}]_{\approx u}((s, \mathcal{C}, M, \tau)) \), \( DB(P_{\mathcal{D}(u)}(M_i, s_i), C_i, S_i, \tau)) = DB((M_i, s_i))_{u} \), the sequence of programs \( C^m \) is secure with respect to \( u \) for \( S, s_i, M^m, E^{\infty} \), \( S^{\infty} \) for the progress-sensitive variant of Definition 7.2, and \( \mathcal{R}((C, M, \langle s, \mathcal{C}, M, \tau \rangle, m)) \) is a \( (E^{\infty}, S^{\infty}) \)-single-epoch configuration, then \( \mathcal{C} \) is secure with respect to \( u \) for \( S, s_i, M^m, E^{\infty} \), \( S^{\infty} \) for the progress-sensitive variant of Definition 7.2, where \( \mathcal{R}((C, M, s, S, m)) \) denotes the global configuration \((C^m, M^m, s, S)\).

**Proof.** Let \( S \) be the sequential scheduler \( 0^\infty \), \( C \in Com^{*}_{\mathcal{M}} \) be a sequence of simple query-only programs, \( s \) be a system state, \( M \in Mem_{\mathcal{M}}^{\infty} \) be a sequence of memories, \( u \in UID \) be a user. Furthermore, let \( r \) be the longest run obtained by starting from \((C, M, \langle s, \mathcal{C}, M, \tau \rangle)\). Assume that since \( r \) is the longest run obtained starting from \((C, M, \langle s, \mathcal{C}, M, \tau \rangle)\), all epochs \((C_i, M_i, (s_i, ctx_i), S_i)\) in \( [E, S]((s, \mathcal{C}, M, \tau)) \) have an even length (hence \( v/2 = \lceil v/2 \rceil \)). Assume that for all epochs \((C_i, M_i, (s_i, ctx_i), S_i)\) in \( [E, S]((s, \mathcal{C}, M, \tau)) \), (1) the sequence of programs \( C^m \) is secure with respect to \( u \) for \( S, s_i, M^m, E^{\infty} \), \( S^{\infty} \) for the progress-sensitive variant of Definition 7.2, and (2) \( \mathcal{R}((C, M, \langle s, \mathcal{C}, M, \tau \rangle, \lceil v/2 \rceil)) \) is a \( (E^{\infty}, S^{\infty}) \)-single-epoch configuration, where \( m = \lceil v/2 \rceil \). Observe that \( v \) is always even since executing each program in \( C \) takes two steps of the global semantics (one application of the M-EVAL-STEP rule and one application of the M-EVAL-END rule) and \( r \) is the longest run obtained by starting from \((C, M, \langle s, \mathcal{C}, M, \tau \rangle)\). Furthermore, assume, for contradiction’s sake, that \( C \) is not secure with respect to \( u \) for \( S, s, M, E^{\infty} \), and \( S^{\infty} \). From this and Lemma D.2, it follows that there is an epoch \((C_k, M_k, (s_k, ctx_k), S_k) \rightarrow (C_j, M_j, (s_j, ctx_j), S_j) \) in \([E, S]((s, \mathcal{C}, M, \tau))\) such that \( DB(P_{\mathcal{D}(u)}(M_k, s_k), C_k, S_k, \tau)) \neq DB((M_k, s_k))_{u} \). Let \( m = \lceil v/2 \rceil \). From (1), we know that \( C^m \) is secure with respect to \( u \) for \( S, s_k, M^m, E^{\infty} \), and \( S^{\infty} \). From (2), we know that \( \mathcal{R}((C_k, M_k, \langle s, \mathcal{C}, \tau \rangle, m)) \) is a \( (E^{\infty}, S^{\infty}) \)-single-epoch configuration. From the security of \( C^m \) and Lemma D.2, it follows that \( DB(P_{\mathcal{D}(u)}(M_k, s_k), C_k, S_k, \tau)) = DB((M_k, s_k))_{u} \). We claim that \( DB(P_{\mathcal{D}(u)}(M_k, s_k), C_k, S_k, \tau)) = DB(P_{\mathcal{D}(u)}((M_k, s_k), C_k, S_k, \tau)) \). From this and \( DB((M_k, s_k))_{u} = DB((M_k, s_k))_{u} \), it follows that \( DB(P_{\mathcal{D}(u)}(M_k, s_k), C_k, S_k, \tau)) = DB((M_k, s_k))_{u} \), leading to a contradiction.

We now prove our claim that \( DB(P_{\mathcal{D}(u)}((M_k, s_k), C_k, S_k, \tau)) = DB(P_{\mathcal{D}(u)}((M_k, s_k), C_k, S_k, \tau)) \). We first show that \( DB(P_{\mathcal{D}(u)}((M_k, s_k), C_k, S_k, \tau)) \subseteq DB(P_{\mathcal{D}(u)}((M_k, s_k), C_k, S_k, \tau)) \). We observe that \( C_k \) and \( C^m \) are simple query-only programs. Furthermore, we know that (1) the first \( m \) programs in \( C_k \) and \( C^m \) are identical by construction, (2) they all consist of commands whose observations are visible by the user \( u \) (since we consider user-based epochs), and (3) \( \tau_{\alpha(s_k)} = \tau \) since \( \tau \) has been produced inside one of the epochs associated with \( u \) (which contains only commands associated with \( u \)). Let \( s \) be a database in \( DB(P_{\mathcal{D}(u)}((M_k, s_k), C_k, S_k, \tau)) \). From this,
there is a sequence of memories \( M \) such that \( s \approx_u s_k, M \approx_u M_k^u \), and there are \( ct^{t'}, \tau', C', M', s', S' \) such that \( \langle C_m^u, M, (s, c, S) \rangle \xrightarrow{s} \langle C', M', (s', ct^{t'}), S' \rangle \) and \( \tau_{|_{db(u)}} = \tau'_{|_{db(u)}} \). We now need to show that \( s \in DB(PK_{db(u)}((M_k, s_k), C_k, S_k, \tau)) \). This happens iff there is a memory \( M_1 \) such that \( M_1 \approx_u M_k \), and there are \( ct^{t'}, \tau', C', M', s', S' \) such that \( \langle C_k, M_1, (s, c), S \rangle \xrightarrow{s} \langle C', M', (s', ct^{t'}), S' \rangle \) and \( \tau_{|_{db(u)}} = \tau'_{|_{db(u)}} \). Without loss of generality, we can assume that \( M_1^u = M_1^u \) and \( M_1 \approx_u M_k \) (this immediately follows from Lemma D.1 and the fact that the programs in \( C_k \) are simple and query-only). Hence, we just need to show that there are \( ct^{t'}, \tau', C', M', s', S' \) such that \( \langle C_k, M_1, (s, c), S \rangle \xrightarrow{s} \langle C', M', (s', ct^{t'}), S' \rangle \) and \( \tau_{|_{db(u)}} = \tau'_{|_{db(u)}} \).

Lemma D.4 and Lemma D.5 prove some useful intermediate results for the transformation in Definition 7.4.

**Lemma D.4.** Let \( C \) be a sequence of simple single-query programs (without ADD USER commands), \( S \) be the sequential scheduler \( 0^\infty \), \( \langle M_1, s_1 \rangle \) be a reachable global state, 0 \( \leq k \leq |C| \), \( C^u \) be the code in \( T(C, M_1, (s_1, c), S) \), and \( m = |C^u(0)| \). The following conditions hold (where \( T^{-1}(\tau_1) \) is the trace obtained by (1) dropping the first \(|C^u(0)|\) observations from \( \tau_1 \), (2) dropping observations associated with CREATE VIEW commands that involve predicate symbols of the form \( TMP_R \), and (3) replacing each occurrence of \( TMP_R \) with \( R \)):

1. If \( \langle C, M_1, (s_1, c), S \rangle \xrightarrow{2(k+m+1)} \langle C', M_1', (s'_1, ct^{t'}), S' \rangle \), then \( T(C, M_1, (s_1, c), S) \xrightarrow{2(k+m+1)} \langle C'_u, M'_u, (s'_1', ct^{t'}), S'_u \rangle \), \( s'_1 = T^{-1}(s_1') \), and \( \tau_1 = T^{-1}(\tau_1') \), where \( l \) is the number of CREATE VIEW queries occurring in \( C^u \).

2. If \( T(C, M_1, (s_1, c), S) \xrightarrow{2(k+m+1)} \langle C'_u, M'_u, (s'_1', ct^{t'}), S'_u \rangle \), then \( C, M_1, (s_1, c), S \xrightarrow{2(k+m+1)} \langle C'_u, M'_u, (s'_1', ct^{t'}), S'_u \rangle \), \( s'_1 = T^{-1}(s_1') \), and \( \tau_1 = T^{-1}(\tau_1') \), where \( l \) is the number of CREATE VIEW queries occurring in \( C^u \).

**Proof.** Let \( C \) be a sequence of simple single-query programs (without ADD USER commands), \( S \) be the sequential scheduler \( 0^\infty \), \( \langle M_1, s_1 \rangle \) be a reachable global state, 0 \( \leq k \leq |C| \), \( C^u \) be the code in \( T(C, M_1, (s_1, c), S) \), and \( m = |C^u(0)| \). We now prove the first claim by induction on \( k \). The proof of the second claim is similar.

**Base case.** We assume \( k = 0 \). From this, it also follows that \( l = 0 \). Then, in the computation \( T(C, M_1, (s_1, c), S) \xrightarrow{2(m)} \langle C'_u, M'_u, (s'_1', ct^{t'}) \rangle \), we execute only the program \( C^u(0) \), which consists only of REVOKE commands. The trace \( \tau_1 \) consists of exactly \( m \) observations (one per REVOKE command), therefore \( T^{-1}(\tau_1) = e \) which is equivalent to \( \tau_1 \). We now show that \( T^{-1}(s_1') = s_1 \). For the database content, this trivially follows from the fact that \( C^u(0) \) consists of just REVOKE commands. For the users, it follows from (1) \( T \) does not change the set of users, and (2) \( C \) does not contain ADD USER commands. For the triggers and the views, this follows from (1) \( T^{-1}(s_1') \) just inverts \( T \), and (2) \( C^u(0) \) consists of just REVOKE commands. For the security policy, the only difference may arise from GRANTS associated with SELECT privileges. Observe, however, that in \( C^u(0) \) we revoke all privileges to relation schemas of the form \( TMP_R \) that do not correspond to privileges on \( R \) in \( sec \). From this and \( \langle M_1, s_1 \rangle \) is a reachable global state, all REVOKE commands are authorized. Hence, after executing \( C^u(0) \), there is a privilege \( (op, u, (SELECT, TMP_R), u^0) \) in the final configuration iff \( (op, u, (SELECT, R), u^0) \) is in the initial configuration. Thus, by first removing all GRANTS associated with privileges associated with relation schemas of the form \( R \) and later replacing all occurrences of \( TMP_R \) with \( R \), we obtain again the initial security policy. This completes the proof of the base case.

**Induction step.** We assume that the claim holds for all \( k' < k \) and we show that the claim holds also for \( k \). There are two cases:

- The last command executed in \( C, M_1, (s_1, c), S \xrightarrow{t} C'_u, M'_u, (s'_1, ct^{t'}) \rangle \) is not a CREATE VIEW command. Hence, it is easy to see that we can always execute 2 steps of the computation and therefore execute the program \( C^u(k + 1) \) in the run that starts form \( T(C, M_1, (s_1, c), S) \).

There are several cases depending on the actual command.

If the command in \( C^u(k + 1) \) is a SELECT command, then \( s'_1 = T^{-1}(s_1') \) follows from (1) the induction hypothesis and (2) the semantics of SELECT commands. Instead, \( \tau_1 = T^{-1}(\tau_1') \) directly follows from the induction hypothesis (since the database where the SELECT query is executed is the same as in the original run).

If the command in \( C^u(k + 1) \) is a CREATE TRIGGER command, then \( s'_1 = T^{-1}(s_1') \) and \( \tau_1 = T^{-1}(\tau_1') \) follow from (1) the induction hypothesis, (2) the result of CREATE TRIGGER commands
Lemma D.5. Let $D.1$. From Database Access Control to Information-flow Control
such that $S⟩τ$ follow from (1) the induction hypothesis, and (2) the result of GRANT and REVOKE commands depend only on the non-SELECT privileges in the policy (which are the same in both runs before executing the command). If the command involves a SELECT privilege, then in the run starting from $(C, M_1, ⟨s_1, ϵ⟩, S)$ we attempt to modify the privilege associated with $R$ while in the run starting from $T((C, M_1, ⟨s_1, ϵ⟩, S))$ we attempt to modify the privilege associated with $T M P_R$. From the induction hypothesis, the database states in the $k$-th and $(k + m)$-th configurations in the two runs are equivalent after applying $T^−1$. This means that the privileges over predicate symbols of the form $T M P_R$ mirror exactly those on symbols of the form $R$. Hence, the result of a GRANT or REVOKE operation would be the same in both configurations since the only two things that may influence the result are (1) the policy (which is mirrored correctly), and (2) the views of the form $T M P_V$ (which in $T((C, M_1, ⟨s_1, ϵ⟩, S))$ refer only to the mirrored version of the symbols). Thus, $s'_1 = T^−1(s''_1)$ and $τ_1 = T^−1(τ''_1)$.

If the command in $C'(k + 1)$ is an INSERT or DELETE command, there are two cases. If the corresponding triggers do not involve GRANT or REVOKE commands, then $s'_1 = T^−1(s''_1)$ and $τ_1 = T^−1(τ''_1)$ directly follow from the induction hypothesis. If the triggers involve GRANT and REVOKE commands, then $s'_1 = T^−1(s''_1)$ and $τ_1 = T^−1(τ''_1)$ follow from (1) the induction hypothesis, (2) the fact that all triggers that modify SELECT privileges have been transformed to refer to the mirrored predicate symbols (of the form $T M P_R$), and (3) a similar argument to that for GRANT and REVOKE commands.

- The last command executed in $(C, M_1, ⟨s_1, ϵ⟩, S) −→^k (C'_1, M'_1, ⟨s'_1, ctxτ'_1⟩, S)$ is a CREATE VIEW command. From this, it follows that we can always execute 4 steps of the computation and therefore execute the program $C'(k + 1)$ in the run that starts form $T((C, M_1, ⟨s_1, ϵ⟩, S))$.

Observing that now the number of the executed CREATE VIEW commands is increased by 1. Hence, the computation is as follows: $T((C, M_1, ⟨s_1, ϵ⟩, S)) −→^k (C''_1, M''_1, ⟨s''_1, ctxτ''_1⟩, S)$. Thus, we can rewrite the computation as $T((C, M_1, ⟨s_1, ϵ⟩, S)) −→^k (C''_1, M''_1, ⟨s''_1, ctxτ''_1⟩, S)$. Furthermore, $s'_1 = T^−1(s''_1)$ and $τ_1 = T^−1(τ''_1)$ directly follow from (1) the induction hypothesis, and (2) the result of CREATE VIEW commands depend only on the database configuration (which is the same in both runs before executing the command). Observe that the observations executed by the two CREATE VIEW commands are identical modulo the use of temporary predicate symbols.

This completes the proof of the induction step. $\square$

Lemma D.5. Let $C$ be a sequence of simple single-query programs (without ADD USER commands) and $S$ be the sequential scheduler $0^m$. For any reachable global states $(M_1, s_1)$ and $(M_2, s_2)$ (with the same database configuration) and $0 \leq k \leq |C|$, there are $C'_1, C'_2, M'_1, M'_2, s'_1, s'_2, ctxτ'_1, τ_1, τ_2$ such that $(C, M_1, ⟨s_1, ϵ⟩, S) τ_1 −→^k (C'_1, M'_1, ⟨s'_1, ctxτ'_1⟩, S), (C, M_2, ⟨s_2, ϵ⟩, S) τ_2 −→^k (C'_2, M'_2, ⟨s'_2, ctxτ'_2⟩, S)$, and $τ_1 = τ_2$ iff there are $C''_1, C''_2, M''_1, M''_2, s''_1, s''_2, ctxτ''_1, ctxτ''_2, τ''_1, τ''_2$ such that $T((C, M_1, ⟨s_1, ϵ⟩, S)) τ''_1 −→^k (C''_1, M''_1, ⟨s''_1, ctxτ''_1⟩), S'_1), T((C, M_2, ⟨s_2, ϵ⟩, S)) τ''_2 −→^k (C''_2, M''_2, ⟨s''_2, ctxτ''_2⟩), S'_2)$, and $τ''_1 = τ''_2$ where $m$ is the number of statements in $C'(0)$ and $l$ is the number of CREATE VIEW queries in $C''$.

Proof. Let $C$ be a sequence of simple single-query programs (without ADD USER commands), $S$ be the sequential scheduler $0^m$, $(M_1, s_1)$ and $(M_2, s_2)$ be two reachable global states (with the same database configuration), $0 \leq k \leq |C|$, $m$ is the number of statements in $C'(0)$ and $l$ is the number of CREATE VIEW queries in $C''$.

$(⇒)$. Assume that there are $C'_1, C'_2, M'_1, M'_2, s'_1, s'_2, ctxτ'_1, τ_1, τ_2$ such that $(C, M_1, ⟨s_1, ϵ⟩, S) τ_1 −→^k (C'_1, M'_1, ⟨s'_1, ctxτ'_1⟩), S), (C, M_2, ⟨s_2, ϵ⟩, S) τ_2 −→^k (C'_2, M'_2, ⟨s'_2, ctxτ'_2⟩, S)$, and $τ_1 = τ_2$. Then, by applying Lemma D.4.(1) to both runs, it follows that there are $C''_1, C''_2, M''_1, M''_2, s''_1, s''_2, ctxτ''_1, τ''_1, τ''_2$ such that $T((C, M_1, ⟨s_1, ϵ⟩, S)) τ''_1 −→^k (C''_1, M''_1, ⟨s''_1, ctxτ''_1⟩), S'_1), T((C, M_2, ⟨s_2, ϵ⟩, S)) τ''_2 −→^k (C''_2, M''_2, ⟨s''_2, ctxτ''_2⟩), S'_2)$, $τ''_1 = T^−1(τ_1)$, and $τ''_2 = T^−1(τ_2)$. From $τ''_1 = T^−1(τ_1)$, $τ''_2 = T^−1(τ_2)$, and $τ''_1 = τ''_2$, it follows that the suffixes containing all but the first $m$ observations of $τ''_1$ and $τ''_2$ are equivalent. Furthermore, the first $m$ observations in $τ''_1$ and $τ''_2$ are the same since $(M_1, s_1)$ and
\((M_2, s_2)\) have the same database configuration. Hence, \(\tau_1^2 = \tau_2^2\) are equivalent. This completes the proof of the if direction.

\((\Leftarrow)\). Assume that there are \(C'_i, C''_i, M'_i, M''_i, s'_i, s''_i, ctx'_i, ctx''_i, \tau'_i, \tau''_i\) such that \(T((C, M_1, (s_1, e), S)) \xrightarrow{\tau''_i} (C'_i, M'_i, (s'_i, ctx'_i), S'_i), (C, M_2, (s_2, e), S) \xrightarrow{\tau'_i} (C'', M'', (s''_i, ctx''_i), S''_i),\) and \(\tau'_1 = \tau''_2\). By applying Lemma D.4.2, it follows that there are \(C'_1, C'_2, M'_1, M'_2, s'_1, s'_2, ctx'_1, ctx'_2,\) \(\tau'_1, \tau'_2\) such that \((C, M_1, (s_1, e), S) \xrightarrow{\tau'_1} 2^{k+m+1} ((C'_1, M'_1, (s'_1, ctx'_1), S'_1), (C, M_2, (s_2, e), S)) \xrightarrow{\tau'_2} 2^{k+m+1} ((C'_2, M'_2, (s'_2, ctx'_2), S''_2))\), and \(\tau'_1 = \tau'_2\). This completes the proof of the only if direction.

In Lemma D.6, we finally connect the security conditions for external and internal attackers.

**Lemma D.6.** Let \(S\) be the sequential scheduler \(0^\infty\), \(C \in \Com^{\pr} UID\) be a sequence of simple query-only programs (without ADD USER commands), \(s\) be a system state, \(M \in \Mem^{\pr} UID\) be a sequence of memories, \(u \in UID\) be a user, and \((C', M', (s', e), S')\) be the configuration \(T((C, M, (s, e), S))\). If \((C, M, (s, e), S)) is \((E^{user}, S^{user})\)-single-epoch and the sequence of programs \(C'\) is secure with respect to the user db(u) for \(S', s',\) and \(M'\) according to the progress-sensitive variant of Definition 7.3 (extended to handle users in UID \(\cup \{db(u) | u \in UID\}\)), then the sequence of programs \(C\) is secure with respect to the attacker \(u\) for \(S, s, M, E^{user}\), and \(S^{user}\) according to the progress-sensitive variant of Definition 7.2.

**Proof.** Let \(S\) be the sequential scheduler \(0^\infty\), \(C \in \Com^{\pr} UID\) be a sequence of simple query-only programs (without ADD USER commands), \(s\) be a system state, \(M \in \Mem^{\pr} UID\) be a sequence of memories, \(u \in UID\) be a user, and \((C', M', (s', e), S')\) be the configuration \(T((C, M, (s, e), S))\). We assume that (a) \((C, M, (s, e), S)) is \((E, S)\)-single-epoch, and (b) the sequence of programs \(C'\) is secure with respect to the user db(u) for \(S', s',\) and \(M'\) according to the progress-sensitive variant of Definition 7.3. Let \(r\) be the longest run obtained from \((C', M', (s', e), S')\) (whose length we denote by max) and \(r_1\) be the longest run obtained from \((C, M, (s, e), S))\) (whose length we denote by max) from the security of \(C'\) with respect to the progress-sensitive variant of Definition 7.3. It follows that whenever \(r = (C', M', (s', e), S') \xrightarrow{\tau} (C''', M'''', (s''', ctx'''', S'''))\), then for all \(1 \leq i \leq n\), \(PK_{\text{db}(u)}((M', s'), (C', S'), \text{trace}(r''_{i-1}))) \cap A_{\text{sec}}(C, M, s) \subseteq PK_{\text{db}(u)}((M', s'), C', S', \text{trace}(r''_{i}))\), where sec is the security policy in the i-th configuration in \(r\). Observe that \(A_{\text{sec}}(C, M, s)\) never contributes to the security of \(C'\). This follows from the transformation in Definition 7.4 since (a) the programs in \(C'\) modify only the SELECT privileges associated with predicate symbols of the form \(\text{TMP}_R\), and (b) the user \(u\) is initially allowed to read all the values in tables \(\text{TMP}_R\) (for any table \(R\) and user \(u\), the security policy \(\text{sec}_{\text{caol}}\) in \(T((C, M, (s, e), S))\) is such that \(\text{TMP}_R\) in \(\text{auth}_{\text{sec}_{\text{caol}}} u\)). Therefore, for all \(1 \leq i \leq n\), \(PK_{\text{db}(u)}((M', s'), C', S', e) \subseteq A_{\text{sec}}(C, M, s)\), where sec is the security policy in the i-th configuration in \(r\). From this and the security of \(C'\), it follows that whenever \(r = (C', M', (s', e), S') \xrightarrow{\tau} (C'', M'', (s'', ctx'', S''))\), then for all \(1 \leq i \leq n\), \(PK_{\text{db}(u)}((M', s'), (C', S', \text{trace}(r''_{i-1}))) \subseteq PK_{\text{db}(u)}((M', s'), C', S', \text{trace}(r''_{i})))\). From this, it follows that \(PK_{\text{db}(u)}((M', s'), C', S', e) \subseteq PK_{\text{db}(u)}((M', s'), C', S', \text{trace}(r''_{i}))\) for all \(1 \leq i \leq \text{max}\), where max is the length of the longest run obtained by starting from \((C', M', (s', e), S')). Observe also that (1) \(PK_{\text{db}(u)}((M', s'), C', S', e) \subseteq ((M', s')_{\text{sec}}\), and (2) \(PK_{\text{db}(u)}((M', s'), C', S', \tau) \subseteq ((M', s')_{\text{sec}}\) for all \(\tau\). Hence, we have that \((M', s')_{\text{sec}} = PK_{\text{db}(u)}((M', s'), C', S', \text{trace}(r''_{i}))\) for all \(1 \leq i \leq \text{max}\). Let \(\text{unwarp}(K)\) be the set \(\{(M, T^{-1}(s)) | (m, M, s) \in K\}\). We claim that for all \(0 \leq j \leq \text{max}\), there is a \(j \leq i \leq \text{max}\) such that \(\text{unwarp}(PK_{\text{db}(u)}((M', s'), C', S', \text{trace}(r''_{i}))) = PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r))\). By applying our claim to \(\text{max}\), we obtain that there is a \(\text{max} \leq i \leq \text{max}\) such that \(\text{unwarp}(PK_{\text{db}(u)}((M', s'), C', S', \text{trace}(r''_{i}))) = PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r))\). From this and \((M', s')_{\text{sec}} = PK_{\text{db}(u)}((M', s'), C', S', \text{trace}(r''_{i}))\) for all \(1 \leq i \leq \text{max}\), it follows that \(\text{unwarp}((M', s')_{\text{sec}}) = PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r))\). From this and \(\text{unwarp}((M, s)_{\text{sec}}\) = \((M, s)_{\text{sec}}\), it follows that \((M, s)_{\text{sec}} = PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r))\). From \((M, s)_{\text{sec}} = PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r))\), it follows that \((M, s)_{\text{sec}} = PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r))\) for all \(1 \leq j \leq \text{max}\), and \(PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r))\) for all \(1 \leq j \leq \text{max}\), it follows that \(\text{unwarp}((M, s)_{\text{sec}} = PK_{\text{db}(u)}((M, s), C, S, \text{trace}(r)))\) for all \(1 \leq j \leq \text{max}\). From this, \(r_1\) is the longest computation, and \((C, M, (s, e), S)\) is \((E^{user}, S^{user})\)-single-epoch, it follows that whenever \(r = (C, M, (s, e), S) \xrightarrow{\tau} (C, M, (s, ctx, S), S)\), for all epochs \((C, M, (s, e), S) \xrightarrow{\tau} (C, M, (s, ctx, S), S)\) in \((E^{user}, S^{user})\) then \(PK_{\text{db}(u)}((M, s), C, S, \tau) = ((M, s)_{\text{sec}}\). Hence, \(C\) is secure with respect to the progress-sensitive variant of Definition 7.2.

**Auxiliary results.** Let \(C\) be a sequence of simple query-only programs, \(M\) be a sequence of memories, and \((C', M', (s', e), S) = T((C, M, (s, e), S))\). We now show that \(\text{unwarp}((M', s')) = (M, s)\). By construction \(M' = (\text{admin}, m_0') - M\), where \(m_0\) is the memory that initializes every variable to 0. Furthermore, \(T^{-1}(s) = s\) (as we have already shown in Lemma D.4). Hence, \(\text{unwarp}((M', s')) = (M, s)\).
Lemma D.7. Let $\mathcal{S}$ be the sequential scheduler $0^\omega$, $u$ be a user, $C \in \mathcal{C}m^u_s$ be a sequence of simple query-only programs, $(M_0, s_0)$ be a reachable global state, and $\tau$ be a trace of observations such that $|\tau| \leq \#_\mathcal{S}(C)$ and $\tau_u = \tau$. If $C$ is safe for $\mathcal{S}$ and $[(M_0, s_0)]_u$, then $\mathcal{P}_u((M_0, s_0), C, \mathcal{S}, \tau) = K_u((M_0, s_0), C, \mathcal{S}, \tau)$.

Proof. Let $\mathcal{S}$ be the sequential scheduler $0^\omega$, $u$ be a user, $C \in \mathcal{C}m^u_s$ be a sequence of simple query-only programs, $(M_0, s_0)$ be a reachable global state, and $\tau$ be a trace such that $|\tau| \leq \#_\mathcal{S}(C)$ and $\tau_u = \tau$. Furthermore, we assume that $C$ is safe for $\mathcal{S}$ and $[(M_0, s_0)]_u$.

We first prove that $\mathcal{P}_u((M_0, s_0), C, \mathcal{S}, \tau) \supseteq K_u((M_0, s_0), C, \mathcal{S}, \tau)$. Let $(M, s) \in K_u((M_0, s_0), C, \mathcal{S}, \tau)$. From this, it follows that $M \approx_u (M_0, s_0)$ and for all $ctx', \tau', C', M', s'$ such that $(C, (s, ctx), S) \rightarrow^{\tau'} (C', (s', ctx'), S)$, then $\tau_u \leq \tau' |_u$ or $\tau' \geq \tau |_u$. From this, $C \in \mathcal{C}m^u_s$. Therefore, $\mathcal{P}_u((M_0, s_0), C, \mathcal{S}, \tau) \supseteq K_u((M_0, s_0), C, \mathcal{S}, \tau)$.
and $\tau |_u = \tau$, it follows that for all $ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{r} (C', M', (s', ctx'), S)$, then $\tau \leq \tau'$ or $\tau \geq \tau'$. From this, $C$ is a sequence of simple query-only programs, $C'$'s safety, and $|\tau| \leq \#_{\text{ctx}}(C)$, it follows that there are $n, ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{r} (C', M', (s', ctx'), \tau, \tau')$, and $|\tau| = |\tau'|$. From $\tau \leq \tau' \vee \tau \geq \tau'$ and $|\tau| = |\tau'|$, it follows that $\tau = \tau'$. Hence, there are $n, ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{r} (C', M', (s', ctx'), \tau, \tau')$ and $\tau = \tau'$. From this, $\tau |_u = \tau$, and $C \in \text{Com}_s^n$, it follows that there are $n, ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{n} (C', M', (s', ctx'), \tau, \tau')$. Hence, $(M, s) \in \text{PK}_u((M_0, s_0), C, S, \tau)$.

We now show that $\text{PK}_u((M_0, s_0), C, S, \tau) \subseteq K_u((M_0, s_0), C, S, \tau)$. Let $(M, s) \in \text{PK}_u((M_0, s_0), C, S, \tau)$. From this, it follows that $M \equiv M_0, s \equiv s_0$, and there are $r, ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{n} (C', M', (s', ctx'), \tau, \tau')$. From this, $C \in \text{Com}_s^n$, and $\tau |_u = \tau$, it follows that there are $n, ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{n} (C', M', (s', ctx'), \tau, \tau')$ and $\tau = \tau'$. Let $r$ be an arbitrary run such that $(C, M, (s, ctx), S) \xrightarrow{n} (C', M', (s', ctx'), S)$, and $\tau |_u = \tau'. From this and $\tau = \tau'$, $\tau' \geq \tau$. If $n = m$, then $\tau' = \tau$. From this and $\tau = \tau'$, $\tau' \geq \tau$. If $n > m$, then $\tau' \geq \tau$. From this and $\tau = \tau'$, $\tau' \geq \tau$. Hence, for all $n, ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{n} (C', M', (s', ctx'), S)$, $\tau' \geq \tau$. From this, $C \in \text{Com}_s^n$, and $\tau |_u = \tau$, it follows that for all $n, ctx', \tau', C', M', s'$ such that $(C, M, (s, ctx), S) \xrightarrow{n} (C', M', (s', ctx'), S)$, then $\tau |_u \geq \tau' \geq \tau$. Hence, $(M, s) \in K_u((M_0, s_0), C, S, \tau)$.

Theorem D.3 proves that the progress-sensitive and progress-insensitive security conditions for external attackers are equivalent for simple query-only programs and sequential schedulers.

**Theorem D.3.** Let $S$ be the sequential scheduler $\emptyset^n$, $u$ be a user, $C \in \text{Com}_s^n$ be a sequence of simple query-only programs, and $(M_0, s_0)$ be a reachable global state such that $C$ is safe for $S$ and $[(M_0, s_0)]_u$. Then, $C$ is secure with respect to Definition 7.3 for $S$, $M$, $s$, and $u$ if $C$ is secure with respect to the progress-sensitive variant of Definition 7.3 for $S$, $M$, $s$, and $u$.

**Proof.** Let $S$ be the sequential scheduler $\emptyset^n$, $u$ be a user, $C \in \text{Com}_s^n$ be a sequence of simple query-only programs, and $(M_0, s_0)$ be a reachable global state such that $C$ is safe for $S$ and $[(M_0, s_0)]_u$. ($\Rightarrow$) Assume that $C$ is secure with respect to Definition 7.3 for $S$, $M_0$, $s_0$, and $u$. From this, it follows that whenever $r = (C, M_0, (s_0, e), S) \xrightarrow{n} (C', M', (s', ctx'), S')$, then for all $1 \leq i \leq n$, $K_u((M_0, s_0), C, S, \text{trace}(r_{-i}^n)) \subseteq K_u((M_0, s_0), C, S, \text{trace}(r^n))$, where the database in $r$'s $(i-1)$-th configuration is $(db, U, sec, T, V)$. From this, $C$'s safety, and Lemma D.7, it follows that whenever $r = (C, M_0, (s_0, e), S) \xrightarrow{n} (C', M', (s', ctx'), S')$, then for all $1 \leq i \leq n$, $PK_u((M_0, s_0), C, S, \text{trace}(r_{-i}^n)) \cap A_u(M_0, s_0) \subseteq PK_u((M_0, s_0), C, S, \text{trace}(r^n))$, where the database in $r$'s $(i-1)$-th configuration is $(db, U, sec, T, V)$. From this, $C$'s safety, and Lemma D.7, it follows that whenever $r = (C, M_0, (s_0, e), S) \xrightarrow{n} (C', M', (s', ctx'), S')$, then for all $1 \leq i \leq n$, $K_u((M_0, s_0), C, S, \text{trace}(r_{-i}^n)) \subseteq K_u((M_0, s_0), C, S, \text{trace}(r^n))$, where the database in $r$'s $(i-1)$-th configuration is $(db, U, sec, T, V)$. Hence, $C$ is secure with respect to the progress-sensitive variant of Definition 7.3 for $S$, $M_0$, $s_0$, and $u$.

($\Leftarrow$) Assume that $C$ is secure with respect to the progress-sensitive variant of Definition 7.3 for $S$, $M_0$, $s_0$, and $u$. From this, it follows that whenever $r = (C, M_0, (s_0, e), S) \xrightarrow{n} (C', M', (s', ctx'), S')$, then for all $1 \leq i \leq n$, $PK_u((M_0, s_0), C, S, \text{trace}(r_{-i}^n)) \cap A_u(M_0, s_0) \subseteq PK_u((M_0, s_0), C, S, \text{trace}(r^n))$, where the database in $r$'s $(i-1)$-th configuration is $(db, U, sec, T, V)$. From this, $C$'s safety, and Lemma D.7, it follows that whenever $r = (C, M_0, (s_0, e), S) \xrightarrow{n} (C', M', (s', ctx'), S')$, then for all $1 \leq i \leq n$, $K_u((M_0, s_0), C, S, \text{trace}(r_{-i}^n)) \subseteq K_u((M_0, s_0), C, S, \text{trace}(r^n))$, where the database in $r$'s $(i-1)$-th configuration is $(db, U, sec, T, V)$. Hence, $C$ is secure with respect to Definition 7.3 for $S$, $M_0$, $s_0$, and $u$.

Theorem D.4 proves that the progress-sensitive and progress-insensitive security conditions for internal attackers are equivalent for simple query-only programs and sequential schedulers.

**Theorem D.4.** Let $S$ be the sequential scheduler $\emptyset^n$, $u$ be a user, $C \in \text{Com}_s^n$ be a sequence of simple query-only programs, and $(M_0, s_0)$ be a reachable global state such that $C$ is safe for $M$ and all $1 \leq i \leq |M|$, $M(i) = (u_i, m_i)$ and $C(i) = (u_i, c_i)$. Furthermore, let $r_{\text{max}}$ be the longest run obtained starting from $(C, M_0, (s_0, e), S)$. If $C|M_{[r_{\text{max}}]}^{\text{sec}}$ is safe for $S$ and $[(M_0, s_0)]_u$ for all $(C, M_0, (s_0, ctx), S_0) \xrightarrow{r} (C, M_0, (s_0, ctx), S_0)$ in $\mathcal{E}_{\text{user}}$, $\mathcal{S}_{\text{user}}$, then $C$ is secure with respect to Definition 7.2 for $S$, $M$, $s$, $u$, $E_{\text{user}}$, and $\mathcal{S}_{\text{user}}$ iff $C$ is secure with respect to the progress-sensitive variant of Definition 7.2 for $S$, $M$, $s$, $u$, $E_{\text{user}}$, and $\mathcal{S}_{\text{user}}$.

**Proof.** Let $S$ be the sequential scheduler $\emptyset^n$, $u$ be a user, $C \in \text{Com}_s^n$ be a sequence of simple query-only programs, and $(M_0, s_0)$ be a reachable global state such that $C$ is safe for $M$ and all $1 \leq i \leq |M|$, $M(i) = (u_i, m_i)$ and $C(i) = (u_i, c_i)$. Furthermore, let $r_{\text{max}}$ be the longest run obtained starting from
Proof of claim (1).

Let \( r = (C, M_0, \langle s_0, \epsilon \rangle, S) \) be an arbitrary run and \( \langle C, M_1, (s_1, ctx_1), S \rangle \) be the next state. Then, we have that for all epochs \((C, M_1, (s_1, ctx_1), S) \) in \( E_{\text{user}} S_{\text{user}} \), we have that \( DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C, S, \tau))) = DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C', S, \tau))) \).

Proof of claim (2).

Let \( \tau = (C, M_0, (s_0, ctx_0), S) \) be an arbitrary run and \( \langle C, M_1, (s_1, ctx_1), S \rangle \) be the next state. Then, we have that for all epochs \((C, M_1, (s_1, ctx_1), S) \) in \( E_{\text{user}} S_{\text{user}} \), we have that \( DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C, S, \tau))) = DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C', S, \tau))) \).

Proof of claim (3).

Let \( \tau = (C, M_0, (s_0, ctx_0), S) \) be an arbitrary run and \( \langle C, M_1, (s_1, ctx_1), S \rangle \) be the next state. Then, we have that for all epochs \((C, M_1, (s_1, ctx_1), S) \) in \( E_{\text{user}} S_{\text{user}} \), we have that \( DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C, S, \tau))) = DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C', S, \tau))) \).

Proof of claim (4).

Let \( \tau = (C, M_0, (s_0, ctx_0), S) \) be an arbitrary run and \( \langle C, M_1, (s_1, ctx_1), S \rangle \) be the next state. Then, we have that for all epochs \((C, M_1, (s_1, ctx_1), S) \) in \( E_{\text{user}} S_{\text{user}} \), we have that \( DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C, S, \tau))) = DB(K_{\text{dial}}((\langle M_1, s_1 \rangle, C', S, \tau))) \).
Proof of claim (2). Let \( r \) be an arbitrary run \( r = (C, M_0, (s_0, \epsilon), S) \xrightarrow{\tau_0} (C_n, M_n, (s_n, \text{ctx} x), S_n) \). We now prove our claim that for all epochs \((C_i, M_i, (s_i, \text{ctx} x), S_i)\) in \([E_{\text{user}}, S^\text{user}^+](r)\), then \( DB(\mathcal{PK}_{\Phi}(M_i, s_i, C_i, S_i, \tau)) = DB(\mathcal{PK}_{\phi}(M_i, s_i, C_i, S_i, \tau)) \), where \( M_i = M^{(\tau_i)} \) and \( C_i = C^{(\tau_i)} \). Let \((C_i, M_i, (s_i, \text{ctx} x), S_i)\) be an arbitrary epoch in \([E_{\text{user}}, S^\text{user}^+](r)\), where \( M_i = M^{(\tau_i)} \) and \( C_i = C^{(\tau_i)} \).

We show that \( DB(\mathcal{PK}_{\Phi}(M_i, s_i, C_i, S_i, \tau)) \subseteq DB(\mathcal{PK}_{\phi}(M_i, s_i, C_i, S_i, \tau)) \). Let \( s \) be a database state in \( DB(\mathcal{PK}_{\Phi}(M_i, s_i, C_i, S_i, \tau)) \). Then, there is a memory \( M \) such that \( M \approx_u M_i \), \( s \approx_u s_i \), and there are \( c \in C_i \), \( C', M', s', \tau' \) such that \((C_i, M_i, (s_i, \epsilon), S) \xrightarrow{\tau_i} (C', M', (s', \text{ctx} x'), S) \) and \( \tau'_u = \tau_u \). Therefore, the first \( |\tau| \) observations in \( \tau' \) are produced by the programs in the epoch associated with the user \( u \) and therefore \( \tau'_u = \tau_u \). Then, there is a run that produces exactly the trace \( \tau \) and executes only the programs in the epoch (i.e., \( C_i \)). Namely, there are \( c \in C', C', M', s', \tau' \) such that \((C_i, M^{(\tau_i)}, (s_i, \epsilon), S) \xrightarrow{\tau_i} (C', M', (s', \text{ctx} x'), S) \) and \( \tau'_u = \tau_u \). Hence, \((M^{(\tau_i)}, \epsilon), s_i) \in \mathcal{PK}_{\Phi}((M, s_i, C_i, S_i, \tau)) \) and \( s \in DB(\mathcal{PK}_{\Phi}(M_i, s_i, C_i, S_i, \tau)) \).

We show that \( DB(\mathcal{PK}_{\Phi}(M_i, s_i, C_i, S_i, \tau)) \supseteq DB(\mathcal{PK}_{\phi}(M_i, s_i, C_i, S_i, \tau)) \). Let \( s \) be a database state in \( DB(\mathcal{PK}_{\phi}(M_i, s_i, C_i, S_i, \tau)) \). Then, there is a memory \( M \) such that \( M \approx_u M_i \), \( s \approx_u s_i \), and there are \( c \in C \), \( C', M', s', \tau' \) such that \((C_i, M_i, (s_i, \epsilon), S) \xrightarrow{\tau_i} (C', M', (s', \text{ctx} x'), S) \) and \( \tau'_u = \tau_u \). From this, it follows that there are \( c \in C', C', M', s', \tau' \) such that \((C_i, M_i, (s_i, \epsilon), S) \xrightarrow{\tau_i} (C', M', (s', \text{ctx} x'), S) \) and \( \tau'_u = \tau_u \) where \( M'' = M \cup M'' \) (where \( M_i = M_i'' \) such that \( |M_i| = |M| \)). Hence, \((M'', \epsilon), s_i) \in \mathcal{PK}_{\phi}((M, s_i, C_i, S_i, \tau)) \) and \( s \in DB(\mathcal{PK}_{\phi}(M_i, s_i, C_i, S_i, \tau)) \).

\[\square\]

D.2 Monitor’s transparency

Here, we show that the monitor of Section 7.6 is transparent. In more detail, we prove that the monitor’s local semantics is transparent. Furthermore, we also show that for sequential schedulers the monitor’s global semantics is transparent as well.

D.2.1 Local semantics

Before proving the correctness of the local semantics, we introduce some terminology and notation. Given an extended WHILE program \( c \), we denote by \( \text{strip}(c) \) the program obtained by (1) removing statements of the form \text{set} p \text{c} to \( l \), \text{dbout}(u, v, o, \tau), \| x \leftarrow q \| \), and \text{asuser}(u, c') and (2) replacing \([c] \) with \( c \). Furthermore, we write \text{safe}(c) iff \( \text{strip}(c) \) is not of the form \( x \leftarrow q, x \leftarrow q' \); \text{asuser}(u, c'), \text{asuser}(u, c') \) \text{dbout}(u, v, o, \tau), \text{dbout}(u, v, o, \tau) \) \text{dbout}(u, v, o, \tau) : c', \| x \leftarrow q \|, \| x \leftarrow q' \| \).

In Lemma D.8 we prove the correctness of the weakest precondition operator for \text{INSERT} and \text{DELETE} commands.

**Lemma D.8.** Let \( \phi \) be a sentence that does not refer to views, \( m \) be a memory, and \( db \) a database state. For all well-formed assignments \( \nu \) for \( \phi \), the following facts hold:

1. \[\text{wp}(\phi, T \uplus T) \downarrow db \] holds iff \( [\phi]^\nu \downarrow db \) holds, where \( db(R) = db(R) \) for all \( R \neq T \) and \( db'(T) = db'(T) \cup \{\nu\(T\)\} \).
2. \[\text{wp}(\phi, T \uplus T) \downarrow db \] holds iff \( [\phi]^\nu \downarrow db \) holds, where \( db(R) = db(R) \) for all \( R \neq T \) and \( db'(T) = db'(T) \cup \{\nu\(T\)\} \).
3. \[\neg \wp(\phi, c') \downarrow db \] holds iff \( [\neg \phi, c'] \downarrow db \).

**Proof.** Let \( \phi \) be a sentence that does not refer to views, \( m \) be a memory, and \( db \) a database state.

**Proof of (1).** Let \( db' \) be the database state such that \( db'(R) = db(R) \) for all \( R \neq T \) and \( db'(T) = db'(T) \cup \{\nu\(T\)\} \).

For the if direction, we show that for all well-formed assignments \( \nu \) for \( \phi \), \( \text{wp}(\phi, T \uplus T) \downarrow db' \). We now prove, by structural induction on \( \phi \), that \( [\phi]^\nu \downarrow db' = T \). There are two base cases:

- \( \phi \) is \( R(T) \). If \( R \neq T \), then \( \text{wp}(\phi, T \uplus T) = \phi \) and the claim trivially holds since \( db'(T) = db'(R) \). If \( R = T \), then \( \text{wp}(\phi, T \uplus T) = R(T) \uplus T = T \). From this and \( \text{wp}(\phi, T \uplus T) \downarrow db' = T \), it follows that \( \text{wp}(\phi, T \uplus T) \downarrow db' = T \) as well. If \( \nu(T') \in db'(T) \), then \( \nu(T') \in db'(T) \) by construction. Hence, \( [\phi]^\nu \downarrow db' = T \).
- \( \phi \) is \( x = x \). Then, \( \text{wp}(\phi, T \uplus T) = \phi \) and the claim trivially holds.

For the induction step, we assume that the claim holds for all sub-formulae of \( \phi \) and we show that it holds for \( \phi \) as well. There are several cases:

Lemma D.9. Let \( m \in \text{Mem} \) be a memory, \( \langle s, ctx \rangle \) be a runtime state such that \( \text{trigger}(cctx) = \epsilon \), \( u \in \text{UID} \) be a user, and \( \Delta \) be a monitor state. Whenever \( \langle \Delta, x \leftarrow q, m, \langle s, ctx \rangle \rangle \overset{\text{\( \Delta', \epsilon, m', \langle s', ctx' \rangle \)}}{\to} \), then \( \langle x \leftarrow q, m, \langle s', ctx' \rangle \rangle \) and the following conditions hold: (1) \( m'(x) = m''(x) \), (2) \( s' = s'' \), (3) \( \text{trigger}(cctx') = \text{trigger}(cctx) = \epsilon \), and (4) \( r' = r'' \).

Proof. Let \( m \in \text{Mem} \) be a memory, \( \langle s, ctx \rangle \) be a runtime state such that \( \text{trigger}(cctx) = \epsilon \), \( u \in \text{UID} \) be a user, and \( \Delta \) be a monitor state. Furthermore, we assume that \( \langle \Delta, x \leftarrow q, m, \langle s, ctx \rangle \rangle \overset{\text{\( \Delta', \epsilon, m', \langle s', ctx' \rangle \)}}{\to} \). In the computation, we applied the rule F-EXPAND-ONCE, the rule F-EXPANDEDCODE
multiple times, and the rule F-RemoveExpandedCode once. Hence, $\langle \Delta', x \leftarrow q, m, \langle s, cxz \rangle \rangle \xrightarrow{\alpha} \langle \Delta', \epsilon, m', \langle s', cxz' \rangle \rangle$ if $\langle \Delta, [c], m, \langle s, cxz \rangle \rangle \xrightarrow{\alpha} \langle \Delta', [c], m', \langle s', cxz' \rangle \rangle$, where $c = \text{expand}(s, cxz), x, q, u$. This, in turn, happens if $\langle \Delta, c, m, \langle s, cxz \rangle \rangle \xrightarrow{\alpha} \langle \Delta', c, m', \langle s', cxz' \rangle \rangle$, where $c = \text{expand}(s, cxz), x, q, u$.

From expand’s definition, it follows that $c = \text{decls}(s, cxz, m, u, x \leftarrow q) \iff \text{decls}(s, cxz, m, u, x \leftarrow q)$, where decls$(s, cxz, m, u, x \leftarrow q)$ is a sequence of SELECT queries and body$(s, m, u, x \leftarrow q)$ is a sequence of IF statements. We claim that (1) at least one of the conditions of the IF statements in body$(s, m, u, x \leftarrow q)$ is satisfied, and (2) the conditions in the IF statements in body$(s, m, u, x \leftarrow q)$ are mutually exclusive, i.e., in each execution we execute only one of the IF statements. Let $c'$ be the IF statement associated with the satisfied condition. From expand’s definition, $c' = \text{cond}(s, cxz) \land \text{body}(s, cxz) \implies \text{decls}(s, cxz, m, u, x \leftarrow q)$ holds, which is a configuration-consistent execution path. We additionally claim that (3) if $\text{cond}(s, cxz) \land \text{body}(s, cxz)$ is satisfied, then the configuration-consistent execution path $\tilde{F}$ represents an actual execution of the command and the corresponding triggers, and (4) $\text{body}(s, cxz)$ is a configuration-consistent execution path $\tilde{F}$ correctly implements the semantics of the execution path $F$.

At least one condition is satisfied. Here we prove our claim that at least one condition in the IF statements in body$(s, m, u, x \leftarrow q)$ is satisfied. First, observe that there is always at least one configuration-consistent execution path. If $\langle q, \text{secEx} \rangle$ is configuration-consistent, then the claim trivially holds as there is an IF statement with condition $\top$, which is trivially satisfied. Assume now that $\langle q, \text{secEx} \rangle$ is not configuration-consistent. We observe that the encoding is such that all possible combinations of variables are covered. Namely, for a query $q$, there are two IF statements: one checking whether the trigger is enabled and one checking whether the constraints are not satisfied. Similarly, for a trigger $t$, there are four possible IF statements: one checking if the trigger is enabled and the constraints are satisfied, one checking if the trigger is enabled and the constraints are not satisfied, one checking if the trigger is enabled, and one checking if the trigger is not enabled. From these observations, it follows that there is always at least one condition satisfied.

Mutually exclusive conditions. Here we prove our claim that the conditions in the IF statements in body$(s, m, u, x \leftarrow q)$ are mutually exclusive. Assume, for contradiction’s sake, that this is not the case. This requires that there are two distinct configuration-consistent execution paths $\tilde{F}$ and $\tilde{F}'$ such that both $\text{cond}(s, cxz) \land \text{body}(s, cxz)$ and $\text{cond}(s, cxz) \land \text{body}(s, cxz)$ are satisfied. Since $\text{cond}(s, cxz) \land \text{body}(s, cxz)$ and $\text{cond}(s, cxz) \land \text{body}(s, cxz)$ are configuration-consistent, this requires that all $c_{s, u, x, q}(i)$ and $c_{s, u, x, q}(j)$ are satisfied for $1 \leq i \leq \lvert \tilde{F} \rvert$ and $1 \leq j \leq \lvert \tilde{F}' \rvert$. Let $k$ be the first position where $\tilde{F}$ and $\tilde{F}'$ differ. There are two cases:

1. $k = 1$. Then, the two paths differ on the initial query $q$. There are 6 cases depending on the values of $\tilde{F}(1)$ and $\tilde{F}'(1)$:

   a. $\tilde{F}(1) = (q, \text{ok})$ and $\tilde{F}'(1) = (q, \text{ok})$. Since $\tilde{F}$ and $\tilde{F}'$ are configuration-consistent paths, it follows that $\text{allowed}(s, u, q)$ and $\text{allowed}(s, u, q)$, leading to a contradiction.

   b. $\tilde{F}(1) = (q, \text{ok})$ and $\tilde{F}'(1) = (q, \text{ex})$. Therefore, $c_{s, u, x, q}(1) = k_{1, \gamma_1} \ldots k_{1, \gamma_n}$ and $c_{s, u, x, q}(1) = (x_{1, \gamma_1} \ldots x_{1, \gamma_n})$. From decls’s definition, it follows that $x_{1, \gamma_1}$ and $x_{1, \gamma_n}$ are respectively initialized by the statements $x_{1, \gamma_1} \leftarrow \text{SELECT} w_{\gamma}(\gamma_1)$ and $x_{1, \gamma_n} \leftarrow \text{SELECT} w_{\gamma}(\gamma_{n-1})$. From this, $\tilde{F}(1) = (q, \text{ok})$, $\tilde{F}(1) = (q, \text{ex})$, and wp’s definition, it follows that $w_{\gamma}(\gamma_1) = w_{\gamma}(\gamma_{n-1})$ and $c_{s, u, x, q}(1) = (x_{1, \gamma_1} \ldots x_{1, \gamma_n})$, which leads to a contradiction (since the result of the queries are the same given that in decls we just executed SELECT queries).

   c. $\tilde{F}(1) = (q, \text{secEx})$ and $\tilde{F}(1) = (q, \text{ok})$. Since $\tilde{F}$ and $\tilde{F}'$ are configuration-consistent paths, it follows that $\text{allowed}(s, u, q)$ and $\text{allowed}(s, u, q)$, leading to a contradiction.

   d. $\tilde{F}(1) = (q, \text{secEx})$ and $\tilde{F}(1) = (q, \text{ex})$. Since $\tilde{F}$ and $\tilde{F}'$ are configuration-consistent paths, it follows that $\text{allowed}(s, u, q)$ and $\text{allowed}(s, u, q)$, leading to a contradiction.

   e. $\tilde{F}(1) = (q, \text{ex})$ and $\tilde{F}(1) = (q, \text{ex})$. Then, $c_{s, u, x, q}(1) = (x_{1, \gamma_1} \ldots x_{1, \gamma_n})$ and $c_{s, u, x, q}(1) = (x_{1, \gamma_1} \ldots x_{1, \gamma_n})$. From decls’s definition, it follows that $x_{1, \gamma_1}$ and $x_{1, \gamma_n}$ are respectively initialized by the statements $x_{1, \gamma_1} \leftarrow \text{SELECT} w_{\gamma}(\gamma_1)$ and $x_{1, \gamma_n} \leftarrow \text{SELECT} w_{\gamma}(\gamma_{n-1})$. From this, $\tilde{F}(1) = (q, \text{ok})$, $\tilde{F}(1) = (q, \text{ex})$, and wp’s definition, it follows that $w_{\gamma}(\gamma_1) = w_{\gamma}(\gamma_{n-1})$ for all $\gamma_i \in \Gamma$. This combined with $c_{s, u, x, q}(1) = (x_{1, \gamma_1} \ldots x_{1, \gamma_n})$ and $c_{s, u, x, q}(1) = (x_{1, \gamma_1} \ldots x_{1, \gamma_n})$, leads to a contradiction (since the result of the queries are the same given that in decls we just executed SELECT queries).
(f) \( T(1) = (q, \text{ex}) \) and \( T(1) = (q, \text{secEx}) \). Since \( T \) and \( T' \) are configuration-consistent paths, it follows that \( \lnot \text{allowed}(s, u, q) \) and \( \text{allowed}(s, u, q) \), leading to a contradiction.

2. \( k > 1 \). Then the two paths differ on the \((k-1)\)-th scheduled trigger \( t \). There are 9 cases depending on the values \( T(k) \) and \( T'(k) \):

(a) \( T(k) = (t, \text{ok}) \) and \( T(k) = (t, \text{secEx}) \). Since \( T \) and \( T' \) are configuration-consistent paths, it follows that \( \text{allowed}(s, u, t(m)) \) and \( \lnot \text{allowed}(s, u, t(m)) \), leading to a contradiction.

(b) \( T(k) = (t, \text{ok}) \) and \( T(k) = (t, \text{ex}) \). Then, \( c_{s, \text{m, u, x, q}} T(k) \) and \( c_{s, \text{m, u, x, q}} T'(k) \). From \( \text{decls} \)'s definition, the variables are initialized as follows: \( x_{\text{k, cond}} \leftarrow \text{select} wp(\varphi, t^{k-1}) \), \( x_{\text{k, cond}} \leftarrow \text{select} wp(\varphi, t^{k-1}) \), and \( x_{\text{k, cond}} \leftarrow \text{select} wp(\gamma_i, T') \) for all \( \gamma_i \in \Gamma \). From this, \( T \) and \( T' \) differ only on the \( k \)-th element, it follows that \( wp(\varphi, t^{k-1}) = wp(\varphi, t^{k-1}) \). From this and \( T \) differ only on the \( k \)-th element, it follows that \( wp(\varphi, t^{k-1}) = wp(\varphi, t^{k-1}) \). Hence, the all variables are initialized using the same queries. Since during \( \text{decls} \) we execute only \( \text{SELECT} \) queries, the variables are initialized to the same values. This combined with \( c_{s, \text{m, u, x, q}} T(k) = x_{\text{k, cond}} \land x_{\text{k, cond}} \land \ldots \land x_{\text{k, cond}} \) leads to a contradiction.

(c) \( T(k) = (t, \text{ok}) \) and \( T(k) = (t, \text{dis}) \). Then, \( c_{s, \text{m, u, x, q}} T(k) = x_{\text{k, cond}} \land x_{\text{k, cond}} \land \ldots \land x_{\text{k, cond}} \) and \( c_{s, \text{m, u, x, q}} T'(k) = x_{\text{k, cond}} \land x_{\text{k, cond}} \land \ldots \land x_{\text{k, cond}} \) leads to a contradiction. The proof of this case is similar to that of \( T(k) = (t, \text{dis}) \) and \( T'(k) = (t, \text{dis}) \).
then the configuration-consistent execution path $\mathcal{I}$ represents an actual execution of the command and the corresponding triggers. Let $\mathcal{I}$ be an execution path such that $\text{cond}_{(s,c,tz)}(\mathcal{I})$ is satisfied in the local state $(m, s)$. We now show that each prefix of $\mathcal{I}$ corresponds to a computation on the database (i.e., correspond to a run in the database). We show this by induction on the prefix's length.

For the base case, let $\mathcal{I}'$ be the prefix of length 1. Then, $\mathcal{I}' = (q, r)$, where $r \in \{\text{secEx}, \text{ok}, \text{ex}\}$.

If $r = \text{secEx}$, it follows that $\text{allowed}(s, u, q) = \bot$. From this and $\text{allowed}$'s definition, it follows that executing the command on the database throws a security exception.

If $r = \text{ex}$, it follows that $\text{allowed}(s, u, q) = \top$. From this and $\text{allowed}$'s definition, it follows that executing the command on the database does not throw a security exception. Furthermore, from $r = \text{ex}$, it follows that (1) $\mathcal{I}_r = \mathcal{I}'$, and (2) $\text{cond}_{(s,c,tz), m, u, x, q}(\mathcal{I})$ is satisfied in the local state $(m, s)$, it follows that all variables in $\{x_1^1, \ldots, x_{k,\gamma_1}\}$ evaluates to $\bot$.

Then, $\mathcal{I}_r$ represents a computation. From this and $\text{decls}$'s definition, the result of $\text{SELECT} \ wp(\gamma, \mathcal{I}_r)$ on the database $s$ is $\bot$. From this and Lemma D.8, executing the query $q$ throws an integrity exception.

For the induction step, we assume that all prefixes of length less than $k$ correspond to actual computations. We now show that the same holds for prefixes of length $k$. Then, $\mathcal{I}' = \mathcal{I}'(t, r)$, where $r \in \{\text{secEx}, \text{ok}, \text{ex}, \text{dis}\}$.

If $r = \text{secEx}$, it follows that $\text{allowed}(s, u, q) = \bot$. From this and $\text{allowed}$'s definition, it follows that executing the command on the database throws a security exception. Furthermore, from $r = \text{secEx}$, it follows that one of the conjuncts in $\text{cond}_{(s,c,tz), m, u, x, q}(\mathcal{I})$ is $x_1^1 \wedge \ldots \wedge x_{k,\gamma_1}$. From (1), (2), and $\text{cond}_{(s,c,tz), m, u, x, q}(\mathcal{I})$ is satisfied in the local state $(m, s)$, it follows that all variables in $\{x_1^1, \ldots, x_{k,\gamma_1}\}$ evaluates to $\bot$. From this and $\text{decls}$'s definition, it follows that for all $\gamma_i \in \Gamma$, the result of $\text{SELECT} \ wp(\gamma_i, \mathcal{I}_r)$ on the database $s$ is $\bot$. From this and Lemma D.8, executing the query $q$ does not throw an integrity exception.

Finally, if $r = \text{ok}$, it follows that $\text{allowed}(s, u, q) = \top$. From this and $\text{allowed}$'s definition, it follows that executing the command on the database does not throw a security exception. Furthermore, from $r = \text{ok}$, it follows that the trigger's condition. From the induction hypothesis, it follows that $\mathcal{I}_r$ represents an actual execution of the command $t$, where $r \in \{\text{secEx}, \text{ok}, \text{ex}, \text{dis}\}$.
Encoding and execution paths. Here, we show that \( \text{body}_{\{u,\text{ctx}\},m,u,u,q}(\mathcal{T}) \) correctly implements the semantics of the execution path \( \mathcal{T} \). Let \( \mathcal{T} \) be a configuration-consistent execution path. If \( \mathcal{T} = (q, \text{secEx}) \), then the produced code only assigns \( \text{secEx} \) to \( x \) and does not modify the database. If \( \mathcal{T} = (q, \text{ex}) \), then the produced code only assigns \( \text{intEx} \) to \( x \), where \( \theta \) is the set of violated constraints, and does not modify the database. Furthermore, in both cases the produced code creates the corresponding database-level event for the user \( db(u) \). Since we use the weakest precondition, \( \theta \) contains exactly the violated constraints. If \( \mathcal{T} \) does not ends in \( \text{ex} \) or \( \text{secEx} \) and \( |\mathcal{T}| = 1 \), the generated code produces the database-level event (for the user \( db(u) \) or \( public \) depending on the command), modifies the database state as described in \( \mathcal{T} \), and sets the correct value for \( x \). If \( \mathcal{T} \) does not ends in \( \text{ex} \) or \( \text{secEx} \) and \( |\mathcal{T}| = 1 \), the generated code produces the database-level event (for the user \( db(u) \) or \( public \) depending on the command) that contains also the observations produced by triggers, modifies the database state as described in \( \mathcal{T} \), and sets the correct value for \( x \). If \( |\mathcal{T}| > 1 \) and \( \mathcal{T} \) ends in \( \text{ex} \), then the generated code stores in \( x \) the error message \( t, \text{intEx}, \theta \) (which contains the trigger \( t \) that has thrown the exception and the set \( \theta \) of all violated integrity constraints) and produces the associated database-level event. Finally, if \( |\mathcal{T}| > 1 \) and \( \mathcal{T} \) ends in \( \text{secEx} \), then the generated code stores in \( x \) the error message \( t, \text{secEx}, \theta \) (which contains the trigger \( t \) that has thrown the exception) and produces the associated database-level event.

Lemma D.10 states that the local semantics of the security monitor correctly mimics the operational semantics of WhISeSQL for all statements that are not queries \( x \leftarrow q \).

Lemma D.10. Let \( c \in \text{Com} \) be an extended WhISeSQL program such that safe(c), \( m \in \text{Mem} \) be a memory, \( (s, \text{ctx}) \) be a runtime state such that trigger\((\text{ctx}) = \epsilon \), and \( u \in \text{UID} \) be a user. Furthermore, let \( \Delta \) be a monitor state. If \( \langle \Delta, c, m, (s, \text{ctx}) \rangle \xrightarrow{\text{F-Skip}} \langle \Delta', c', m', (s', \text{ctx}') \rangle \) and the following conditions hold: (1) for all variables \( x \) occurring in \( \text{strip}(c) \), we obtain \( m'(x) = m^x(x) \), (2) \( s' = s'' \), (3) \( \text{trigger}(\text{ctx}') = \epsilon \), and (4) \( \tau' = \tau'' \). Then \( \langle \Delta, c, m, (s, \text{ctx}) \rangle \xrightarrow{\text{F-Skip}} \langle \Delta', c', m', (s', \text{ctx}') \rangle \).

Proof. Let \( c \in \text{Com} \) be an extended WhISeSQL program such that safe(c), \( m \in \text{Mem} \) be a memory, \( (s, \text{ctx}) \) be a runtime state such that trigger\((\text{ctx}) = \epsilon \), \( u \in \text{UID} \) be a user, and \( \Delta \) be a monitor state. Assume that \( \langle \Delta, c, m, (s, \text{ctx}) \rangle \xrightarrow{\text{F-Skip}} \langle \Delta', c', m', (s', \text{ctx}') \rangle \). We prove our claim by structural induction on the rules defining \( \xrightarrow{\text{F-Skip}} \).

Base case. There are several cases depending on the rule used in the computation:

- Rule F-Skip. From the rule, it follows that \( c = \text{skip}, \Delta = \Delta', c' = \epsilon, m = m', (s, \text{ctx}) = (s', \text{ctx}') \), and \( \tau' = \tau'' \). By applying the E-Skip rule to \( (c, m, (s, \text{ctx})) \), we obtain \( (c, m, (s, \text{ctx})) \xrightarrow{\text{E-Skip}} \langle \text{strip}(c'), m'', (s'', \text{ctx}'') \rangle \), where \( \tau'' = \epsilon = c', m'' = m, (s, \text{ctx}) = (s'', \text{ctx}'') \). Therefore, \( c = c', m = m'', (s, \text{ctx}) = (s'', \text{ctx}'') \), \( \text{trigger}(\text{ctx}') = \epsilon \), and \( \tau'' = \tau'' \).

- Rule F-Assign. From the rule, it follows that \( c = x := \epsilon, \Delta = \Delta', c' = \epsilon, m = m[x \mapsto \text{e}[m]], (s', \text{ctx}') = (s, \text{ctx}), \) and \( \tau' = \tau'' \). By applying the E-Assign rule to \( (c, m, (s, \text{ctx})) \), we obtain \( (c, m, (s, \text{ctx})) \xrightarrow{\text{E-Assign}} \langle c''[m], m'', (s'', \text{ctx}'') \rangle \), where \( \tau'' = \epsilon = c', m'' = m, (s, \text{ctx}) = (s'', \text{ctx}'') \). Therefore, \( c = c', m = m'', (s, \text{ctx}) = (s'', \text{ctx}'') \), \( \text{trigger}(\text{ctx}') = \epsilon \), and \( \tau'' = \tau'' \).

- Rule F-OUT. From the rule, it follows that \( c = \text{out}(u', \epsilon), \Delta = \Delta', c' = \epsilon, m = m', (s', \text{ctx}') = (s, \text{ctx}), \) and \( \tau' = \epsilon \). By applying the E-OUT rule to \( (c, m, (s, \text{ctx})) \), we obtain \( (c, m, (s, \text{ctx})) \xrightarrow{\text{E-OUT}} \langle c''[m], m''[\text{e}[m]], (s'', \text{ctx}'') \rangle \), where \( \tau'' = c', \epsilon = m', (s, \text{ctx}) = (s'', \text{ctx}'') \). Therefore, \( c = c', m = m'', (s, \text{ctx}) = (s'', \text{ctx}'') \), \( \text{trigger}(\text{ctx}') = \epsilon \), and \( \tau'' = \tau'' \).

- Rule F-IFTrue. From the rule, it follows that \( c = \text{if } e \text{ then } c_1 \text{ else } c_2, \epsilon)[m] = \Delta, \epsilon = [c_1 \text{ set } \text{pc} \text{ to } \Delta](\text{pc}_0), m' = m, (s', \text{ctx}') = (s, \text{ctx}) \), and \( \tau' = \epsilon \). By applying the E-IFTrue rule to \( (c, m, (s, \text{ctx})) \), we obtain \( (c, m, (s, \text{ctx})) \xrightarrow{\text{E-IFTrue}} \langle \text{strip}(c'), m''[\text{e}[m]], (s'', \text{ctx}'') \rangle \), where \( \tau'' = \epsilon = m', (s, \text{ctx}) = (s'', \text{ctx}'') \). Therefore, \( m = m'', (s', \text{ctx}') = (s'', \text{ctx}'') \), \( \text{trigger}(\text{ctx}') = (s', \text{ctx}') \), and \( \tau'' = \tau'' \).

- Rule F-IFFalse. The proof of this case is similar to that of the F-IFTrue case.

- Rule F-WHILETrue. The proof of this case is similar to that of the F-IFTrue case.

- Rule F-WHILEFalse. The proof of this case is similar to that of the F-IFTrue case.

- Rule F-SEQEMPTY. From the rule, it follows that \( c = \epsilon \text{ ; } c_1 \), \( c' = c_1, m = m', (s, \text{ctx}) = (s', \text{ctx}') \), and \( \tau' = \epsilon \). By applying the E-SEQEMPTY rule to \( (c, m, (s, \text{ctx})) \), we obtain \( (c, m, (s, \text{ctx})) \xrightarrow{\text{E-SEQEMPTY}} \langle c''[m], m'', (s'', \text{ctx}'') \rangle \), where \( \tau'' = \epsilon = c', m'' = m, \) and
(s, ctx) = (s″, ctx″). Therefore, c′ = c″, m′ = m″, (s′, ctx′) = (s″, ctx″), triggers(ctx′) = triggers(ctx″) = \epsilon, and \tau′ = \tau″.

- Rule F-REMOVEEXPANDED_CODE. The proof of this case is similar to that of the F-SEQEMPTY case.

- Rule F-UPDATELABELS. From the rule, it follows that c = set pc to l, c′ = ε, m′ = m, (s′, ctx′) = (s, ctx), and \tau′ = \tau″. Since the rule modifies only the monitor configuration and strip(c) = strip(c′) = ε, it follows that (strip(c), m, (s, ctx)) \rightarrow \langle \langle (s′, ctx′) \rangle \rangle where m′ = m″, (s′, ctx′) = (s″, ctx″), and triggers(ctx′) = triggers(ctx″) = \epsilon.

This completes the proof of the base step.

**Induction Step.** For the induction step, we consider only the F-SEQ and F-EXPANDED_CODE rules.

- Rule F-SEQ. From the rule, it follows that c = c1 ; c2 and (Δ, c1, m, (s, ctx)) \xrightarrow{\epsilon} s, m, (s′, ctx′). Furthermore, from safe(c), it follows that we can apply the induction hypothesis. From (Δ, c1, m, (s, ctx)) \xrightarrow{\epsilon} (Δ, c1, m, (s′, ctx′)) and the induction’s hypothesis, it follows that (strip(c1), m, (s, ctx)) \xrightarrow{\epsilon} (strip(c1), m, (s′, ctx′)) such that m′ and m″ agree on all variables occurring in strip(c1), \tau″ = \tau′, s″ = s″, and triggers(ctx″) = triggers(ctx″). Observe also that strip(c1; c2) = strip(c1; strip(c2)). There are two cases:
  - the first statement executed in c1 is of the form set pc to l. Therefore, m′ = m′ = m, (s, ctx) = (s, ctx′) = (s″, ctx″), and \tau′ = \tau″ = \epsilon. From this, it directly follows that (\langle (strip(c1); strip(c2)), m, (s, ctx) \rangle \xrightarrow{\epsilon} (\langle strip(c1) \rangle; strip(c2), m, (s′, ctx′))).
  - the first statement executed in c1 is not of the form set pc to l. By applying the E-SEQ rule to (Δ, strip(c1; c2), m, (s, ctx)) (given that (1) (\langle strip(c1) \rangle; m, (s, ctx)) \xrightarrow{\epsilon} (\langle strip(c1) \rangle; m, (s′, ctx′)) and (2) strip(c1; c2) = strip(c1; strip(c2)) we obtain ((\langle strip(c1) \rangle; strip(c2)), m, (s, ctx)) \xrightarrow{\epsilon} (\langle strip(c1) \rangle; strip(c2), m, (s′, ctx′)). Our claim directly follows from (1) m′ and m″ agree on all variables modified in the computation, (2) \tau″ = \tau′, (3) s″ = s″, and (4) triggers(ctx″) = triggers(ctx″).

This completes the proof of the induction step.

Finally, Theorem D.5 states that the local semantics of the security monitor correctly implements the local semantics of WHILESQL.

**Theorem D.5.** Let c ∈ Com be a WHILESQL program (without extended commands from Section 7.6), m ∈ Mem be a memory, (s, ctx) be a runtime state such that trigger(ctx, s) = ε, and u ∈ UID be a user. Furthermore, let Δ be a monitor state. If (Δ, c, m, (s, ctx)) \xrightarrow{\epsilon} (Δ, c′, m′, (s′, ctx′)), then (c, m, (s, ctx)) \xrightarrow{\epsilon} (c′, m′, (s′, ctx′)) and the following conditions hold: (1) for all variables x occurring in c, then m′(x) = m″(x), (2) s′ = s″, (3) triggers(ctx′) = triggers(ctx″) = ε, and (4) \tau″ = \tau′.

**Proof.** This claim directly follows from Lemma D.9, Lemma D.10. In particular, Lemma D.9 is used to handle statements of the form x ← q, whereas Lemma D.10 is used to handle all other statements.

**D.2.2 Global semantics**

Theorem D.6 shows that the monitor’s global semantics is transparent for sequential schedulers. We remark, however, that the monitor’s global semantics is, in general, not transparent as the monitor modifies the scheduling of commands to avoid timing leaks that may be introduced by the parallel execution of multiple WHILESQL programs.

**Theorem D.6.** Let C ∈ Com_UID be a sequence of WHILESQL programs (without the extended commands from Section 7.6), M ∈ Mem_UID be a sequence of memories, s be a system state, and S be the sequential scheduler \(0^\infty\). Whenever (Δ, C, M, (s, e), S) \xrightarrow{\epsilon} (Δ′, e, M′, (s′, ctx′), S′) then (C, M, (s, e), S) \xrightarrow{\epsilon} (e, M″, (s″, ctx″), S″) and the following conditions hold: (1) for all \(1 \leq i \leq |M|\), for all variables that occur in C(i), then M′(i)(x) = M″(i)(x), (2) s′ = s″, (3) triggers(ctx′) = triggers(ctx″) = ε, and (4) S′ = S″.
D.3 Monitor’s soundness

Here, we prove the main result of Section 7.6, namely that our enforcement mechanism is sound with respect to our security condition for external attackers. In the following, let $\langle D, \Gamma \rangle$ be a system configuration such that the constraints in $\Gamma$ are well-formed.

D.3.1 Auxiliary notation

In the following, we lift the trace projection operator $\tau |_U$ to sets of users. The projection of a trace $\tau$ for a set of users $U$, written $\tau |_U$, is as follows: $\epsilon|_U = \epsilon$, $\langle (u', e) \cdot \tau' \rangle |_U = \langle u', e \rangle \cdot \tau' |_U$ if there exists $u \in U$ such that $u' \leq_u u$, $\langle (u', q, o, \tau'') \cdot \tau' \rangle |_U = \langle u', q, o, \tau'' |_U \cdot \tau' \rangle |_U$ if there exists a user $u \in U$ such that $1) u' \leq_u u$, and $2) \langle (u', q, o, \tau'') \cdot \tau' \rangle |_U = \langle (u', q, o, \tau'') \cdot \tau' \rangle |_U = \tau' |_U$ otherwise, where $\text{users}(\tau)$ is the set of all users appearing in the observations in $\tau$.

Let $\text{obs}$ be either a program-level or a database-level observation. We denote by $\text{user}(\text{obs})$ the user associated with the observation. Namely, $\text{user}(\langle \text{public}, q, o, \tau \rangle) = u$, $\text{user}(\langle \text{public}, q, o, \tau \rangle) = u$, and $\text{user}(\langle \text{public}, q, o, \tau \rangle) = \text{ATK}$ otherwise.

Let $\text{obs}$ be either a program-level or a database-level observation and $\text{conf} \xrightarrow{\text{obs}} \text{conf}'$ be a step of the local or global semantics. The direct dependencies of $\text{obs}$ given $\text{conf} \xrightarrow{\text{obs}} \text{conf}'$ are defined in Figure D.1.

Let $c$ be a WHILESql extended program. The function $\text{first}(c)$ returns the first statement to be executed in $c$. Formally:

\[
\text{first}(c) = \begin{cases} 
\text{first}(c_1) & \text{if } \exists c_1. c = [c_1] \\
\text{first}(c_1) & \text{if } \exists c_1, c_2. c = c_1; c_2 \\
\text{first}(c_1) & \text{if } \exists u, c_1. c = \text{asuser}(u, c_1) \\
\text{else} & \text{otherwise}
\end{cases}
\]

D.3.2 Equivalence definitions

We now introduce a number of equivalence relations that we use throughout the proofs. We first introduce equivalence relations between monitor state, database states, and memories.

Definition D.2. Let $\Delta, \Delta'$ be two monitor states, $(m, s), (m', s')$ be two local states, and $u$ be a user.

We say that $\Delta$ and $\Delta'$ are $L$-equivalent, where $L$ is a subset of $\text{Vars} \cup RC_{\text{priv}} \cup \{pc_u | u \in UID\}$, written $\Delta \approx_L \Delta'$, iff for all $x \in L$, $\Delta(x) = \Delta'(x)$.

Proof. The claim directly follows from (1) the use of the sequential scheduler, (2) the application of Theorem D.5 to the execution of each program in $C$, and (3) the fact that the monitor’s global semantics does not add new observations to the trace. \qed
We say that \( s = \langle db, U, S, T, V \rangle \) and \( s' = \langle db', U', S', T', V' \rangle \) are configuration equivalent, written \( s \equiv_{cs} s' \), if \( U = U' \), \( S = S' \), \( T = T' \), and \( V = V' \).

We say that \( \langle m, s \rangle \) and \( \langle m', s' \rangle \) are \( (V, Q) \)-equivalent, where \( V \subseteq \text{Vars} \) and \( Q \subseteq \text{RC} \), written \( \langle m, s \rangle \approx_{V, Q} \langle m', s' \rangle \), if (1) for all \( x \in V \), \( m(x) = m'(x) \), and (2) for all \( q \in Q \), \( [q]^{db} = [q']^{db} \), where \( s = \langle db, U, \text{sec}, T, V \rangle \) and \( s' = \langle db', U', S', T', V' \rangle \).

We now formalize equivalence of local configurations.

**Definition D.3.** Let \( \langle \Delta, c, m, (s, ctx) \rangle, \langle \Delta', c', m', (s', ctx') \rangle \) be two local configurations and \( \ell \in \mathbb{L} \) be a label. We say that \( \langle \Delta, c, m, (s, ctx) \rangle \) and \( \langle \Delta', c', m', (s', ctx') \rangle \) are \( \ell \)-equivalent, written \( \Delta, c, m, \langle s, ctx \rangle \approx_{\ell} \Delta', c', m', \langle s', ctx' \rangle \) if

1. for all \( x \in \text{Vars} \cup \{ pc_u \mid u \in \text{UID} \} \), \( \Delta(x) \not\subseteq \ell \) iff \( \Delta'(x) \not\subseteq \ell \),
2. for all \( q \in \text{RC}^{\text{pred}} \), \( \Delta(q) \not\subseteq \ell \) iff \( \Delta'(q) \not\subseteq \ell \),
3. \( \langle m, s \rangle \approx_{V, Q} \langle m', s' \rangle \), where \( V = \{ x \in \text{Vars} \mid \Delta(x) \not\subseteq \ell \} \) and \( Q = \{ q \in \text{RC} \mid \text{Lc}^\ell(\Delta, q) \not\subseteq \ell \} \),
4. \( \Delta \approx_L \Delta' \), where \( L = \{ x \in \text{Vars} \mid \Delta(x) \subseteq \ell \} \cup \{ q \in \text{RC}^{\text{pred}} \mid \Delta(q) \subseteq \ell \} \cup \{ pc_u \mid u \in \text{UID} \} \).

Similarly, we say that two global configurations \( \langle \Delta, C, M, (s, ctx), S \rangle \approx_{\ell} \langle \Delta', C', M', (s', ctx'), S' \rangle \), iff \( |C| = |C'| \), \( |M| = |M'| \), and for all \( 1 \leq i \leq |C| \), \( \langle \Delta, C(i), M(i), (s, ctx) \rangle \approx_{\ell} \langle \Delta', C'(i), M'(i), (s', ctx') \rangle \).
D.3.4 Results about $L_Q$

Here we state some simple facts about $L_Q$.

**Proposition D.9.** Let $\Delta$ and $\Delta'$ be two monitor states. If $\Delta(q) \subseteq \ell$ iff $\Delta'(q) \subseteq \ell$ for all $q \in RC^{pred}$, then $L_Q(\Delta, q) \subseteq \ell$ iff $L_Q(\Delta', q) \subseteq \ell$ for all $q \in RC$.

**Proof.** Let $\Delta$ and $\Delta'$ be two monitor states such that $\Delta(q) \subseteq \ell$ iff $\Delta'(q) \subseteq \ell$ for all $q \in RC^{pred}$. 

($\Rightarrow$). Assume that $L_Q(\Delta, q) \subseteq \ell$ holds. From this, it follows that $\bigcup_{q' \in supp_D,T(q)} \bigcup_{q'' \in Q} \Delta(q'') \subseteq \ell$. From this, it follows that $\bigcup_{q' \in supp_D,T(q)} \bigcup_{q'' \in Q} \Delta(q'') \subseteq \ell$. From this, it follows that $\bigcup_{q' \in supp_D,T(q)} \bigcup_{q'' \in Q} \Delta(q'') \subseteq \ell$. From this, it follows that $L_Q(\Delta', q) \subseteq \ell$.

($\Leftarrow$). Assume that $L_Q(\Delta', q) \subseteq \ell$ holds. From this, it follows that $\bigcup_{q' \in supp_D,T(q)} \bigcup_{q'' \in Q} \Delta(q'') \subseteq \ell$. From this, it follows that $\bigcup_{q' \in supp_D,T(q)} \bigcup_{q'' \in Q} \Delta(q'') \subseteq \ell$. From this, it follows that $\bigcup_{q' \in supp_D,T(q)} \bigcup_{q'' \in Q} \Delta(q'') \subseteq \ell$. From this, it follows that $L_Q(\Delta, q) \subseteq \ell$. \qed

**Proposition D.10.** Given a monitor state $\Delta$ and a predicate query $q \in RC^{pred}$, $\Delta(q) = L_Q(\Delta, q)$.

**Proof.** Let $\Delta$ be a monitor state and $q \in RC^{pred}$ be a predicate query. From the definition of $L_Q(\Delta, q)$, it follows that $L_Q(\Delta, q) = \bigcup_{q' \in supp_D,T(q)} \bigcup_{q'' \in Q} \Delta(q'')$. Since $\Gamma$ is a set of well-formed integrity constraints and $q \in RC^{pred}$, $supp_D,T(q) = \{\{q\}\}$. From this, $L_Q(\Delta, q) = \Delta(q)$. \qed

D.3.5 Results about relaxed NSU checks

We now prove some simple results about relaxed NSU checks.

**Proposition D.11.** Let $sec_0$ be the initial policy, $\ell$ be a label such that $cl(auth(\text{sec}_0, ATK)) \subseteq \ell$, and $\Delta$ be a monitor state. If $\text{nsu}(x, \text{pc}_u)$ is satisfied for $\Delta$ and $\Delta(x) \subseteq \ell$, then $\Delta(\text{pc}_u) \subseteq \ell$.

**Proof.** Let $sec_0$ be the initial policy, $\ell$ be a label such that $cl(auth(\text{sec}_0, ATK)) \subseteq \ell$, and $\Delta$ be a monitor state. Furthermore, assume that $\text{nsu}(x, \text{pc}_u)$ is satisfied for $\Delta$ and $\Delta(x) \subseteq \ell$. From this, it follows that $\Delta(\text{pc}_u) \subseteq cl(auth(\text{sec}_0, ATK)) \lor \Delta(\text{pc}_u) \subseteq \Delta(x)$. There are two cases:

1. $\Delta(\text{pc}_u) \subseteq cl(auth(\text{sec}_0, ATK))$ holds. From this and $cl(auth(\text{sec}_0, ATK)) \subseteq \ell$, it follows that $\Delta(\text{pc}_u) \subseteq \ell$.

2. $\Delta(\text{pc}_u) \subseteq \Delta(x)$ holds. From this and $\Delta(x) \subseteq \ell$, it follows that $\Delta(\text{pc}_u) \subseteq \ell$.

This completes the proof. \qed

D.3.6 Lemmas about the local semantics

Here we present some auxiliary results about the local semantics of our enforcement mechanism.

Lemma D.11 states that whenever the security monitor produces an output, the labels associated with $\text{pc}$ and with the events for $\text{ATK}$ occur only in low contexts, i.e., in contexts such that $\Delta(\text{pc}_u) \subseteq L_L(s, ATK)$, and (2) all direct flows are authorized, namely $\text{ATK}$ observes only events that directly depend on information at a lower (or equal) level in the security lattice.

**Lemma D.11.** Whenever $r = \langle \Delta, c, m, s \rangle \xrightarrow{\text{obs}_u} \langle \Delta', c', m', s' \rangle$ and $\text{obs} \neq \epsilon$, we have $\Delta(\text{deps}(\text{obs}, r)) \cup \Delta(\text{pc}_u) \subseteq L_L(s, \text{user}(\text{obs}))$.

**Proof.** Let $\langle \Delta, c, m, s \rangle$ and $\langle \Delta', c', m', s' \rangle$ be local configurations such that $\langle \Delta, c, m, s \rangle \xrightarrow{\text{obs}_u} \langle \Delta', c', m', s' \rangle$ and $\text{obs} \neq \epsilon$. In the following, we denote $\langle \Delta, c, m, s \rangle \xrightarrow{\text{obs}_u} \langle \Delta', c', m', s' \rangle$ as $r$. We now prove, by structural induction on $\text{obs}_u$, that $\Delta(\text{deps}(\text{obs}, r)) \cup \Delta(\text{pc}_u) \subseteq L_L(s, \text{user}(\text{obs}))$. In the following we consider only on those rules that produce observations. Proving the claim for rules not producing outputs is trivial.

**Base Case.** There are two cases depending on the rule used to produce the event.

- Rule F-OUT. From the rule, it directly follows that (1) $\text{obs} = \langle u', [e][m] \rangle$, and (2) $\Delta(e) \cup \Delta(\text{pc}_u) \subseteq L_L(s, u')$. From $\text{user}$'s definition, it follows that $\text{user}(u', [e][m]) = u'$. From this and $\Delta(\text{e}) \cup L_L(s, u')$, it follows that $\Delta(\text{e}) \cup L_L(s, \text{user}(u', [e][m]))$. From this, $\text{deps}(u', [e][m], r) = \text{free}(e)$, and $\Delta(\text{free}(e)) = \Delta(e)$, it follows that $\Delta(\text{deps}(u', [e][m], r)) \cup \Delta(\text{pc}_u) \subseteq L_L(s, \text{user}(u', [e][m]))$. Hence, $\Delta(\text{deps}(\text{obs}, r)) \cup \Delta(\text{pc}_u) \subseteq L_L(s, \text{user}(\text{obs}))$. 


Proof. Let \( \text{Induction Step.} \) This completes the proof of the base case.

Lemma D.12 states that, given a label \( \ell \), whenever \( pc_a \) becomes high with respect to \( \ell \), this is caused by a branching statement, a loop statement, or a set-label statement.

**Lemma D.12.** Let \( \ell \in \mathcal{L} \) be a label. Whenever \( (\Delta, c, m, \langle s, ctx \rangle) \sim_{u_a} (\Delta', c', m', \langle s', ctx' \rangle) \), if \( \Delta(pc_a) \subseteq \ell \) and \( \Delta'(pc_a) \not\subseteq \ell \), then one of the following conditions hold:

1. first\( (c) = \text{if } e \text{ then } c_1 \text{ else } c_2 \) and \( \Delta(e) \not\subseteq \ell \),
2. first\( (c) = \text{while } e \text{ do } c_1 \) and \( \Delta(e) \not\subseteq \ell \), or
3. first\( (c) = \text{set } pc \to \ell_1 \) and \( \ell_1 \not\subseteq \ell \).

Proof. Let \( c \) be a WHILE loop program, \( \ell \in \mathcal{L} \) be a label, and \( (\Delta, c, m, \langle s, ctx \rangle) \) and \( (\Delta', c', m', \langle s', ctx' \rangle) \) be two local configurations such that (1) \( (\Delta, c, m, \langle s, ctx \rangle) \sim_{u_a} (\Delta', c', m', \langle s', ctx' \rangle) \), and (2) \( \Delta(pc_a) \subseteq \ell \) and \( \Delta'(pc_a) \not\subseteq \ell \). We now prove, by structural induction on \( \sim_{u_a} \), that our claim holds. In the following, we focus only on the rules that directly modify \( \Delta(pc_a) \). The proof for the other cases is trivial.

**Base Case.** There are several cases depending on the rule used to derive \( (\Delta, c, m, \langle s, ctx \rangle) \sim_{u_a} (\Delta', c', m', \langle s', ctx' \rangle) \):

1. Rule F-UpdateLabels. Then, first\( (c) = \text{set } pc \to \ell_1 \). From the rule, it follows that \( \Delta'(pc_a) = \ell_1 \). From this and \( \Delta'(pc_a) \not\subseteq \ell \), it follows that \( \ell_1 \not\subseteq \ell \).
2. Rule F-IfTrue. Then, first\( (c) = \text{if } e \text{ then } c_1 \text{ else } c_2 \). Furthermore, from \( \Delta(pc_a) \subseteq \ell \), \( \Delta'(pc_a) \not\subseteq \ell \), and \( \Delta'(pc_a) \not\subseteq \ell \), it follows that \( \Delta(e) \not\subseteq \ell \).
3. Rule F-IfFalse. The proof of this case is similar to that of F-IfTrue.
4. Rule F-WhileTrue. The proof of this case is similar to that of F-IfTrue.
5. Rule F-WhileFalse. The proof of this case is similar to that of F-IfTrue.

**Induction Step.** The proof of the induction step directly follows from the induction hypothesis (since the rules do not further introduce events).

Lemma D.13 states that, given a label \( \ell \), whenever \( pc_a \) becomes high with respect to \( \ell \), this is caused by a branching statement, a loop statement, or a set-label statement.

**Lemma D.13.** Let \( sec_0 \) be the policy used to initialize the monitor and \( \ell \in \mathcal{L} \) be a security label such that \( cl(auth(sec_0, ATK)) \subseteq \ell \). Whenever \( (\Delta, c, m, \langle s, ctx \rangle) \sim_{u_a} (\Delta', c', m', \langle s', ctx' \rangle) \), if \( \Delta(pc_a) \not\subseteq \ell \), then \( (m, s) \approx_{V,Q} (m', s') \), where \( V = \{ x \in \text{Vars} \mid \Delta(x) \subseteq \ell \} \) and \( Q = \{ q \in RC \mid L_Q(\Delta, q) \subseteq \ell \} \).

Proof. Let \( sec_0 \) be the policy used to initialize the monitor and \( \ell \in \mathcal{L} \) be a security label such that \( cl(auth(sec_0, ATK)) \subseteq \ell \). Furthermore, let \( u \in UID \) be a user and \( (\Delta, c, m, \langle s, ctx \rangle) \), \( (\Delta', c', m', \langle s', ctx' \rangle) \) be two local configurations such that (1) \( (\Delta, c, m, \langle s, ctx \rangle) \sim_{u_a} (\Delta', c', m', \langle s', ctx' \rangle) \), and (2) \( \Delta(pc_a) \not\subseteq \ell \). Finally, let \( V = \{ x \in \text{Vars} \mid \Delta(x) \subseteq \ell \} \) and \( Q = \{ q \in RC \mid L_Q(\Delta, q) \subseteq \ell \} \). We now show, by structural induction on the rules defining \( \sim_{u_a} \), that \( (m, s) \approx_{V,Q} (m', s') \). In the following, we consider only those rules that modify the memory \( m \) or the database state \( s \). For the other rules, the claim holds trivially (since \( (m, s) = (m', s') \)).

**Base Case.** There are several cases depending on the applied rule:

1. Rule F-Assign. Assume, for contradiction’s sake, that \( (m, s) \not\approx_{V,Q} (m', s') \). From the rule, it follows that \( s = s' \) and \( m = m[x \mapsto e[m]] \). From this, \( (m, s) \not\approx_{V,Q} (m', s') \), and \( \approx_{V,Q} \)’s definition, it follows that (1) \( x \in V \) and, therefore, \( \Delta(x) \subseteq \ell \), and (2) \( m(x) \neq m'(x) \).

2. Rule F-While. From the rule, it follows that \( \text{nsu}(x, pc_a) \) holds. From this, \( cl(auth(sec_0, ATK)) \subseteq \ell \), \( \Delta(x) \subseteq \ell \), and Proposition D.11, it follows that \( \Delta(pc_a) \not\subseteq \ell \). This, however, contradicts our assumption that \( \Delta(pc_a) \not\subseteq \ell \).
2. Rule F-SELECT. Assume, for contradiction’s sake, that \( \langle m, s \rangle \not\in V,Q \langle m', s' \rangle \). From the rule, it follows that \( s = s' \) and \( m' = m[x \mapsto r] \), where \( \langle (s, ctx'), r, e, \ell \rangle = \langle [y][s, u] \rangle \). From this, \( \langle m, s \rangle \not\in V,Q \langle m', s' \rangle \), and \( \approx_{V,Q} \)’s definition, it follows that (1) \( x \in V \) and, therefore, \( \Delta(x) \not\subseteq \ell \), and (2) \( m(x) \not\equiv m'(x) \). From this, it follows that \( \nsu(x, pc_\ell) \). From this, \( \mathit{cl}(auth(\mathit{sec}, ATK)) \subseteq \ell \), \( \Delta(x) \subseteq \ell \), and Proposition D.11, it follows that \( \Delta(pc_\ell) \subseteq \ell \). This, however, contradicts our assumption that \( \Delta(pc_\ell) \not\subseteq \ell \).

3. Rule F-UPDATE DATABASE OK. Without loss of generality, we assume that \( q = T' \oplus \pi \) and that \( \pi \) is the tuple inserted in the database (after evaluating all the expressions). Assume, for contradiction’s sake, that \( \langle m, s \rangle \not\in V,Q \langle m', s' \rangle \). From the rule, it follows that \( \approx_{T,Q} \)’s definition, it follows that there are two cases:

(a) \( x \in V \) and \( m(x) \not\equiv m'(x) \). From \( x \in V \) and \( V \)’s definition, it follows that \( \Delta(x) \subseteq \ell \). From the rule, it follows that \( \nsu(x, pc_\ell) \). From this, \( \mathit{cl}(auth(\mathit{sec}, ATK)) \subseteq \ell \), \( \Delta(x) \subseteq \ell \), and Proposition D.11, it follows that \( \Delta(pc_\ell) \subseteq \ell \). This, however, contradicts our assumption that \( \Delta(pc_\ell) \not\subseteq \ell \).

(b) There is a query \( q' \in Q \) such that \( \approx_{T,Q} \)’s definition, it follows that \( \Delta(x) \subseteq \ell \). From this, it follows that \( \nsu(x, pc_\ell) \). From this, \( \mathit{cl}(auth(\mathit{sec}, ATK)) \subseteq \ell \), \( \Delta(x) \subseteq \ell \), and Proposition D.11, it follows that \( \Delta(pc_\ell) \subseteq \ell \). This, however, contradicts our assumption that \( \Delta(pc_\ell) \not\subseteq \ell \).

We now prove our claim that for all \( q \in supp_{D,T}(q') \), \( T' \) determines \( q' \)’s definition, it follows that the predicate queries in \( Q \) determine \( q' \). From the database semantics, the result of all queries in \( Q \) is the same in \( s \) and \( s' \), since we modify only \( T' \). From this and \( Q \) determines \( q' \), it follows that the result of \( q' \) is the same in \( s \) and \( s' \). This, however, contradicts \( \approx_{T,Q} \)’s definition.

4. Rule F-UPDATE CONFIGURATION OK. The proof of this case is similar to the F-SELECT case.

Induction Step. The proof for the induction step directly follows from the induction hypothesis (since the rules do not further modify the memory and the database). \( \square \)

Lemma D.14 states that, given a label \( \ell \), whenever we are in a high context (i.e., \( \Delta(pc_\ell) \not\subseteq \ell \)), then (1) there are no changes to the labels associated with variables and queries whose labels are initially below \( \ell \), and (2) the label of a variable (or query) is initially below \( \ell \) iff it is below \( \ell \) also at the end of the computation.

Lemma D.14. Let \( \mathit{sec}_0 \) be the policy used to initialize the monitor and \( \ell \in \mathcal{L} \) be a security label such that \( \mathit{cl}(\mathit{sec}, ATK) \subseteq \ell \). Whenever \( \langle \mathcal{D}, c, m, (s, ctx) \rangle \not\approx_{0} \langle \mathcal{D}, c', m', (s', ctx') \rangle \), if \( \Delta(pc_\ell) \not\subseteq \ell \), then the following conditions hold:

- for all \( x \in \mathit{Vars} \), \( \Delta(x) \subseteq \ell \) iff \( \Delta(x') \subseteq \ell \),
- for all \( q \in \mathit{RCP} \), \( \Delta(q) \subseteq \ell \) iff \( \Delta(q') \subseteq \ell \),
- \( \approx_{L_D} \), where \( L_D = \{ x \in \mathit{Vars} | \Delta(x) \subseteq \ell \} \cup \{ q \in \mathit{RCP} | \Delta(q) \subseteq \ell \} \).

Proof. Let \( \mathit{sec}_0 \) be the policy used to initialize the monitor, \( \ell \in \mathcal{L} \) be a security label such that \( \mathit{cl}(\mathit{sec}, ATK) \subseteq \ell \), \( u \in \mathit{UID} \) be a user, and \( \langle \mathcal{D}, c, m, (s, ctx), \langle \mathcal{D}, c', m', (s', ctx') \rangle \rangle \) be two local configurations such that (1) \( \langle \mathcal{D}, c, m, (s, ctx) \rangle \not\approx_{\mathit{sec}_0} \langle \mathcal{D}, c', m', (s', ctx') \rangle \), and (2) \( \Delta(pc_\ell) \not\subseteq \ell \). We prove our claim by structural induction on the rules defining \( \approx_{\mathit{sec}_0} \). In the following, we consider only those rules that modify the monitor state \( \Delta \) for the identifiers in \( \mathit{Vars} \cup \mathit{RCP} \). For the other rules, the claim holds trivially.

Base Case. There are several cases depending on the applied rule:

1. Rule F-ASSIGN. From the rule, it follows that (1) \( \mathcal{D}' = \Delta(x \mapsto \Delta(pc_\ell) \cup \Delta(e)) \), and (2) \( \nsu(x, pc_\ell) \). Assume, for contradiction’s sake, that our claim does not hold. From \( \mathcal{D}' = \Delta(x \mapsto \Delta(pc_\ell) \cup \Delta(e)) \), there are three cases:

- \( \Delta(x) \subseteq \ell \) and \( \Delta(x) \not\subseteq \ell \). From \( \Delta(x) \subseteq \ell \), \( \nsu(x, pc_\ell) \), \( \mathit{cl}(\mathit{sec}, ATK) \subseteq \ell \), and Proposition D.11, it follows that \( \Delta(pc_\ell) \subseteq \ell \). This contradicts our assumption that \( \Delta(pc_\ell) \not\subseteq \ell \).
- \( \Delta(x) \subseteq \ell \) and \( \Delta(x) \not\subseteq \ell \). From \( \mathcal{D}'(x) = \Delta(pc_\ell) \cup \Delta(e) \) and \( \Delta(x) \subseteq \ell \), it follows that \( \Delta(pc_\ell) \cup \Delta(e) \subseteq \ell \). From this, it follows that \( \Delta(pc_\ell) \not\subseteq \ell \). This contradicts our assumption that \( \Delta(pc_\ell) \subseteq \ell \).
- \( \Delta(x) \subseteq \ell \) and \( \Delta(x) \not\subseteq \ell \). From \( \Delta(x) \subseteq \ell \), \( \nsu(x, pc_\ell) \), \( \mathit{cl}(\mathit{sec}, ATK) \subseteq \ell \), and Proposition D.11, it follows that \( \Delta(pc_\ell) \subseteq \ell \). This contradicts our assumption that \( \Delta(pc_\ell) \subseteq \ell \).
2. Rule F-SELECT. From the rule, it follows that $\Delta' = \Delta[x \mapsto \Delta(\text{pc}_u)]$, where $\ell_x = L_{\text{Q}}(\Delta, \text{SELECT} \phi) \cup \bigcup_{v \in \text{mis}, \text{act}} \Delta(v)$, and $\text{nsu}(x, \text{pc}_u)$. Assume, for contradiction’s sake, that our claim does not hold. From $\Delta' = \Delta[x \mapsto \Delta(\text{pc}_u)]$, there are three cases:

- $\Delta(x) \subseteq \ell$ and $\Delta'(x) \not\subseteq \ell$. From $\Delta(x) \subseteq \ell$, $\text{nsu}(x, \text{pc}_u)$, $\text{cl(auth(seco, ATK))} \subseteq \ell$, and Proposition D.11, it follows that $\Delta(\text{pc}_u) \subseteq \ell$. This contradicts our assumption that $\Delta(\text{pc}_u) \not\subseteq \ell$.
- $\Delta(x) \not\subseteq \ell$ and $\Delta'(x) \subseteq \ell$. From $\Delta'(x) = \Delta(\text{pc}_u) \cup \ell_x$, it follows that $\Delta(\text{pc}_u) \not\subseteq \ell$. From this, it follows that $\Delta(\text{pc}_u) \not\subseteq \ell$. This contradicts our assumption that $\Delta(\text{pc}_u) \not\subseteq \ell$.
- $\Delta(x) \subseteq \ell$, $\Delta'(x) \subseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \subseteq \ell$, $\text{nsu}(x, \text{pc}_u)$, $\text{cl(auth(seco, ATK))} \subseteq \ell$, and Proposition D.11, it follows that $\Delta(\text{pc}_u) \subseteq \ell$. This contradicts our assumption that $\Delta(\text{pc}_u) \not\subseteq \ell$.

3. Rule F-UPDATEDATABASEOK. From the rule, it follows that (1) $\Delta' = \Delta[T(\pi)] \Rightarrow \Delta(\text{pc}_u) \cup \ell_x$, $x \mapsto \Delta(\text{pc}_u) \cup \ell_x$, where $\ell_x = \bigcup_{0 \leq i < |\pi|} \Delta(e_i)$, (2) $\text{nsu}(T(\pi), \text{pc}_u)$, and (3) $\text{nsu}(x, \text{pc}_u)$. Assume, for contradiction’s sake, that our claim does not hold. From $\Delta' = \Delta[T(\pi)] \Rightarrow \Delta(\text{pc}_u) \cup \ell_x$, $x \mapsto \Delta(\text{pc}_u) \cup \ell_x$, there are six cases:

- $\Delta(x) \subseteq \ell$ and $\Delta'(x) \not\subseteq \ell$. From $\Delta(x) \subseteq \ell$, $\text{nsu}(x, \text{pc}_u)$, $\text{cl(auth(seco, ATK))} \subseteq \ell$, and Proposition D.11, it follows that $\Delta(\text{pc}_u) \subseteq \ell$. This contradicts our assumption that $\Delta(\text{pc}_u) \not\subseteq \ell$.
- $\Delta(x) \not\subseteq \ell$ and $\Delta'(x) \subseteq \ell$. From $\Delta'(x) = \Delta(\text{pc}_u) \cup \ell_x$, it follows that $\Delta(\text{pc}_u) \not\subseteq \ell$. This contradicts our assumption that $\Delta(\text{pc}_u) \not\subseteq \ell$.
- $\Delta(x) \subseteq \ell$, $\Delta'(x) \subseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \subseteq \ell$, $\text{nsu}(x, \text{pc}_u)$, $\text{cl(auth(seco, ATK))} \subseteq \ell$, and Proposition D.11, it follows that $\Delta(\text{pc}_u) \subseteq \ell$. This contradicts our assumption that $\Delta(\text{pc}_u) \not\subseteq \ell$.

4. Rule F-UPDATECONFIGURATION-OK. The proof of this case is similar to that of F-SELECT.

**Induction Step.** The proof for the induction step follows directly from the induction hypothesis (since the rules do not further modify the memory and the database). \qed

**Lemma D.15.** Let $\text{seco}$ be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label. Whenever $(\Delta', c', m', \{s, ctz\}) \leftrightarrow_a \Delta_a (\Delta', c', m', \{s', ctz\})$, if $\Delta(\text{pc}_u) \not\subseteq \ell$ and $\text{cl(auth(seco, ATK))} \subseteq \ell$, then $s \equiv^\ell s'$.

**Proof.** Let $\text{seco}$ be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label. Furthermore, let $u \in \text{UID}$ be a user and $(\Delta, c, m, \{s, ctz\}), (\Delta', c', m', \{s', ctz\})$ be two local configurations such that (1) $(\Delta, c, m, \{s, ctz\}) \leftrightarrow_a \Delta_a (\Delta', c', m', \{s', ctz\})$, (2) $\Delta(\text{pc}_u) \not\subseteq \ell$, and (3) $\text{cl(auth(seco, ATK))} \subseteq \ell$. We now show, by structural induction on the rules defining $\leftrightarrow_a$, that $s \equiv^\ell s'$. In the following, we consider only those rules that modify the database configuration. For the other rules, the claim holds trivially (since the configuration is the same in $s$ and $s'$).

**Base Case.** There only interesting case is the rule F-UPDATECONFIGURATION-OK. From the rule, it follows that $\Delta(\text{pc}_u) \subseteq \text{cl(auth(seco, ATK))}$. From this and $\text{cl(auth(seco, ATK))} \subseteq \ell$, it follows that $\Delta(\text{pc}_u) \subseteq \ell$, leading to a contradiction.

**Induction Step.** The proof of the induction step follows from the induction hypothesis. \qed

**Lemma D.16.** Let $\text{seco}$ be the policy used to initialize the monitor, $(\Delta_1, c_1, m_1, \{s_1, ctz_1\})$, $(\Delta_1', c_1', m_1', \{s_1', ctz_1'\})$, $(\Delta_2, c_2, m_2, \{s_2, ctz_2\})$, and $(\Delta_2', c_2', m_2', \{s_2', ctz_2'\})$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:

1. $s_1 \equiv^\ell s_2$.
2. $(\Delta_1, c_1, m_1, \{s_1, ctz_1\}) \approx_{\ell} (\Delta_2, c_2, m_2, \{s_2, ctz_2\})$.
3. $\text{L}_{\ell}(\Delta_1, \text{ATK}) \subseteq \ell$ and $\text{L}_{\ell}(\Delta_2, \text{ATK}) \subseteq \ell$.
4. $c_1 = c_2$.
5. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$,  
6. $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_2, c_2, m_2', \langle s_2', ctx_2' \rangle \rangle$,  
then $\tau_1|_{\text{ATK}} = \tau_2|_{\text{ATK}}$.

Proof. Let $sec_0$ be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle$, $\langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$, and $\langle \Delta_2', c_2, m_2', \langle s_2', ctx_2' \rangle \rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold: (1) $s_1 \equiv_s s_2$, (2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \equiv_{\ell} \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$, (3) $L_0(s_1, \text{ATK}) \subseteq \ell$ and $L_0(s_2, \text{ATK}) \subseteq \ell$, (4) $c_1 = c_2$, (5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$, (6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_2', c_2, m_2', \langle s_2', ctx_2' \rangle \rangle$. We prove our claim by induction on the rules defining $\sim_u$. Without loss of generality, we focus only on the rules producing observations. The claim trivially holds for all rules that do not produce observations.

**Base Case.** There are a number of cases depending on the rule applied to derive $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$.

- **Rule F-OUT.** From the rule, it follows that $c_1 = \text{out}(u, e)$. From this and (4), it follows that $c_2 = \text{out}(u, e)$. In the following, we assume that $u = \text{ATK}$. This is not the case the proof is trivial. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$ and $\mathcal{L} = \text{out}(u, e)$, it follows that $\Delta_1 = \Delta_1'$, $m_1 = m_1'$, $c_1 = c_1'$, and $\langle s_1, ctx_1 \rangle = \langle s_1', ctx_1' \rangle$.

- **Rule F-DINOUT.** From the rule, it follows that $c_1 = \text{dbout}(u, v, o, \tau)$. From this and (4), it follows that $c_2 = \text{dbout}(u, v, o, \tau)$.

- **Induction Step.** The proof of the induction step directly follows from the induction hypothesis. □

Lemma D.17 states that, under appropriate conditions, executing the same command on two $\ell$-equivalent states modifies the database configuration in the same way.

**Lemma D.17.** Let $sec_0$ be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle$, $\langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$, and $\langle \Delta_2', c_2, m_2', \langle s_2', ctx_2' \rangle \rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:

1. $s_1 \equiv_s s_2$,
2. $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \equiv_{\ell} \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$,
3. $c_1 = c_2$,  
then $\tau_1|_{\text{ATK}} = \tau_2|_{\text{ATK}}$. 

Proof. Let $sec_0$ be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle$, $\langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$, and $\langle \Delta_2', c_2, m_2', \langle s_2', ctx_2' \rangle \rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold: (1) $s_1 \equiv_s s_2$, (2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \equiv_{\ell} \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$, (3) $L_0(s_1, \text{ATK}) \subseteq \ell$ and $L_0(s_2, \text{ATK}) \subseteq \ell$, (4) $c_1 = c_2$, (5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_1', c_1, m_1', \langle s_1', ctx_1' \rangle \rangle$, (6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\text{rules}} \langle \Delta_2', c_2, m_2', \langle s_2', ctx_2' \rangle \rangle$. We prove our claim by induction on the rules defining $\sim_u$. Without loss of generality, we focus only on the rules producing observations. The claim trivially holds for all rules that do not produce observations.
4. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \xrightarrow{\Delta} (\Delta_1', c_1', m_1', (s_1', ctx_1'))$
5. $(\Delta_2, c_2, m_2, (s_2, ctx_2)) \xrightarrow{\Delta} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$
6. $\text{cl}(\text{auth}(sec_0, ATK)) \subseteq \ell$
then $s_1' \equiv_{\text{hash}} s_2'$.

**Proof.** Let $sec_0$ be the policy used to initialize the monitor, $(\Delta_1, c_1, m_1, (s_1, ctx_1))$, $(\Delta_1', c_1', m_1', (s_1', ctx_1'))$, $(\Delta_2, c_2, m_2, (s_2, ctx_2))$, and $(\Delta_2', c_2', m_2', (s_2', ctx_2'))$ be four local configurations, and $\ell \in L$ be a label such that the following conditions hold: 1. $s_1 \equiv_{\text{hash}} s_2$, 2. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \equiv_{\ell} (\Delta_2, c_2, m_2, (s_2, ctx_2))$, 3. $c_1 = c_2$, 4. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \equiv_{\ell} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$. 5. $(\Delta_2, c_2, m_2, (s_2, ctx_2)) \xrightarrow{\Delta} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$. 6. $\text{cl}(\text{auth}(sec_0, ATK)) \subseteq \ell$. We prove our claim by induction on the rules defining $\equiv_{\text{hash}}$. In the following, we focus only on rules that modify the database configuration. For rules that do not modify the database configuration, the claim directly follows from $s_1 \equiv_{\text{hash}} s_2$.

**Base Case.** The only interesting case is the rule $\text{F-UPDATEOk}$. From the rule, it follows that $c_1 = \|x \leftarrow q\|$, where $q$ is a configuration command. From this (and (3)), it follows that $c_2 = \|x \leftarrow q\|$. From $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \xrightarrow{\Delta} (\Delta_1', c_1', m_1', (s_1', ctx_1'))$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(pc_u) \cup \bigcup_{v \in \text{vars}(q)} \Delta_1(v)]$, $m_1' = m_1[x \mapsto r_1]$, $\text{vars}(q) = \{v_1, \ldots, v_n\}$. Similarly, from $(\Delta_2, c_2, m_2, (s_2, ctx_2)) \xrightarrow{\Delta} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$ and $c_2 = \|x \leftarrow q\|$, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(pc_u) \cup \bigcup_{v \in \text{vars}(q)} \Delta_2(v)]$, $m_2' = m_2[x \mapsto r_2]$, $\text{vars}(q) = \{v_1, \ldots, v_n\}$. From the rules, it follows that $\bigcup_{v \in \text{vars}(q)} \Delta_1(v) \subseteq \text{cl}(\text{auth}(sec_0, ATK))$. From this and (6), it follows that $\bigcup_{v \in \text{vars}(q)} m_1(v) = m_2(v)$. From this, (1), $[q_1](s_1, ctx_1) = (s_1', ctx_1')$, $r_1, \epsilon, \epsilon$, and $[q_2](s_2, ctx_2) = (s_2', ctx_2')$, $r_2, \epsilon, \epsilon$, it directly follows that $s_1' \equiv_{\text{hash}} s_2'$ (since the initial configuration is the same and the database semantics is deterministic).

**Induction Step.** The proof of the induction step directly follows from the induction hypothesis. \(\square\)

**Lemma D.18.** Let $sec_0$ be the policy that, given a label $\ell$, whenever we are in a low context (i.e., $\Delta(pc_u) \subseteq \ell$), then executing the same command on two $\ell$-equivalent states produces to $\ell$-equivalent states.

**Lemma D.18.** Let $sec_0$ be the policy used to initialize the monitor, $(\Delta_1, c_1, m_1, (s_1, ctx_1))$, $(\Delta_1', c_1', m_1', (s_1', ctx_1'))$, $(\Delta_2, c_2, m_2, (s_2, ctx_2))$, and $(\Delta_2', c_2', m_2', (s_2', ctx_2'))$ be four local configurations, and $\ell \in L$ be a label. If the following conditions hold:

1. $s_1 \equiv_{\text{hash}} s_2$,
2. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \equiv_{\ell} (\Delta_2, c_2, m_2, (s_2, ctx_2))$,
3. $\Delta_1(pc_u) \subseteq \ell$ and $\Delta_2(pc_u) \subseteq \ell$,
4. $c_1 = c_2$,
5. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \xrightarrow{\Delta} (\Delta_1', c_1', m_1', (s_1', ctx_1'))$
6. $(\Delta_2, c_2, m_2, (s_2, ctx_2)) \xrightarrow{\Delta} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$
then $(\Delta_1', c_1', m_1', (s_1', ctx_1')) \equiv_{\ell} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$.

**Proof.** Let $sec_0$ be the policy used to initialize the monitor, $(\Delta_1, c_1, m_1, (s_1, ctx_1))$, $(\Delta_1', c_1', m_1', (s_1', ctx_1'))$, $(\Delta_2, c_2, m_2, (s_2, ctx_2))$, and $(\Delta_2', c_2', m_2', (s_2', ctx_2'))$ be four local configurations, and $\ell \in L$ be a label such that the following conditions hold:

1. $s_1 \equiv_{\text{hash}} s_2$,
2. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \equiv_{\ell} (\Delta_2, c_2, m_2, (s_2, ctx_2))$,
3. $\Delta_1(pc_u) \subseteq \ell$ and $\Delta_2(pc_u) \subseteq \ell$,
4. $c_1 = c_2$,
5. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \xrightarrow{\Delta} (\Delta_1', c_1', m_1', (s_1', ctx_1'))$,
6. $(\Delta_2, c_2, m_2, (s_2, ctx_2)) \xrightarrow{\Delta} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$.

We prove our claim by induction on the rules defining $\equiv_{\text{hash}}$. In the following, we focus only on those rules that modify the monitor state, the database, or the memory. For the other rules, the claim directly follows from (2).

**Base Case.** There are a number of cases depending on the rule applied to derive $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \xrightarrow{\Delta} (\Delta_1', c_1', m_1', (s_1', ctx_1'))$.

- **Rule F-UPDATELABELS.** From the rule, it follows that $c_1 = \text{set pc to } l$. From this and (4), it follows that $c_2 = \text{set pc to } l$. From $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \xrightarrow{\Delta} (\Delta_1', c_1', m_1', (s_1', ctx_1'))$ and $c_1 = \text{set pc to } l$, it follows that $\Delta_1' = \Delta_1[pc_u \mapsto l]$, $m_1' = m_1'$, $\tau_1 = \epsilon$, $c_1' = \epsilon$, and $(s_1, ctx_1) = (s_1', ctx_1')$. Similarly, from $(\Delta_2, c_2, m_2, (s_2, ctx_2)) \xrightarrow{\Delta} (\Delta_2', c_2', m_2', (s_2', ctx_2'))$ and
D.3. Monitor’s soundness

Follows that $\tau' \neq \tau$, and $\langle \Delta'$, $c_1', m_1', (s_1', ctzx') \rangle \not\preceq \langle \Delta, c_2, m_2, (s_2', ctzx') \rangle$. From this, $m_1 = m_1'$, $m_2 = m_2'$, $(s_1, ctzx1) = (s_1', ctzx')$, $(s_2, ctzx2) = (s_2', ctzx')$, and (2), there are two cases:

- $\Delta'(pc_x) \nsubseteq \ell$ and $\Delta'(pc_y) \nsubseteq \ell$ (or vice versa). From $\Delta_1' = \Delta_1[pc_r \mapsto \ell]$ and $\Delta_2' = \Delta_2[pc_r \mapsto \ell]$, it follows that $\Delta_1'(pc_x) = \Delta_2'(pc_x)$. From this and $\Delta_1'(pc_x) \subseteq \ell$, it follows that $\Delta_2'(pc_x) \subseteq \ell$, leading to a contradiction.

- $\Delta'(pc_x) \subseteq \ell$, $\Delta_2'(pc_x) \subseteq \ell$, and $\Delta_1'(pc_y) \neq \Delta_2'(pc_y)$. From $\Delta_1' = \Delta_1[pc_r \mapsto \ell]$ and $\Delta_2' = \Delta_2[pc_r \mapsto \ell]$, it follows that $\Delta_1'(pc_y) = \Delta_2'(pc_y)$, and $\Delta_1'(pc_y) \subseteq \ell$, leading to a contradiction.

- Rule F-ASSIGN. From the rule, it follows that $c_1 = x := e$. From this and (4), it follows that $c_2 = x := e$. From $\langle \Delta_1, c_1, m_1, (s_1, ctzx1) \rangle \overset{\Delta_e}{\rightarrow} \langle \Delta_1', c_1', m_1', (s_1', ctzx') \rangle$ and $c_1 = x := e$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(pc_1) \cup \Delta_1(e)]$, $m_1' = m_1[x \mapsto [\ell(m_1)]]$, $\tau_1 = \epsilon, c_1' = c_2$, and $(s_1, ctzx1) = (s_1', ctzx')$. Similarly, from $\langle \Delta_2, c_2, m_2, (s_2, ctzx2) \rangle \overset{\Delta_e}{\rightarrow} \langle \Delta_2', c_2', m_2', (s_2', ctzx') \rangle$ and $c_2 = x := e$, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(pc_2) \cup \Delta_2(e)]$, $m_2' = m_2[x \mapsto [\ell(m_2)]]$, $\tau_2 = \epsilon, c_2' = c_2$, and $(s_2, ctzx2) = (s_2', ctzx')$. Assume, for contradiction’s sake, that $\langle \Delta_1', c_1', m_1', (s_1', ctzx') \rangle \preceq \langle \Delta_2', c_2', m_2', (s_2', ctzx') \rangle$. From this, $(s_1, ctzx1) = (s_1', ctzx')$, $(s_2, ctzx2) = (s_2', ctzx')$, and (2), there are three cases:

- $\Delta_1(x) \subseteq \ell$, and $\Delta_2(x) \subseteq \ell$ (or vice versa). From the rule, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(pc_1) \cup \Delta_1(e)]$ and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(pc_2) \cup \Delta_2(e)]$. From this and $\Delta_1(x) \subseteq \ell$, it follows that $\Delta_2(x) \subseteq \ell$, which contradicts $\Delta_1'(x) \neq \Delta_2'(x)$.

- $\Delta_1(x) \subseteq \ell, \Delta_2(x) \subseteq \ell$, and $m_1(x) \neq m_2(x)$. From the rule, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(pc_1) \cup \Delta_1(e)]$ and $\Delta_2'(pc_2) = \Delta_2[pc_2 \mapsto \Delta_2(e)]$. From this and $\Delta_1(x) \subseteq \ell$, it follows that $\Lambda_{\Delta_e[pc_2]}(\Delta_2(y)) = \Delta_2(y)$, which contradicts $\Delta_1'(pc_2) = \Delta_2'[pc_2] \cup \Delta_2'(e)$. From this, $\Delta_1'(pc_2) = \Delta_1[pc_2 \mapsto \Delta_1(e)]$, and $\Delta_2'(pc_2) = \Delta_2[pc_2 \mapsto \Delta_2(e)]$. From this and (2), it follows that $\Lambda_{\Delta_e[pc_2]}(m_1(y)) = m_2(y)$. From this, it follows that $[\ell(m_1)] = [\ell(m_2)]$. From this, $m_1 = m_1[x \mapsto [\ell(m_1)]]$, and $m_2 = m_2[x \mapsto [\ell(m_2)]]$, which contradicts $\Delta_1'(pc_2) \neq \Delta_2'(pc_2)$.

- Rule F-IFTRUE. From the rule, it follows that $c_1 = \text{if } e \text{ then } c' \text{ else } c''$. From this and (4), it follows that $c_2 = \text{if } e \text{ then } c' \text{ else } c''$. From $\langle \Delta_1, c_1, m_1, (s_1, ctzx1) \rangle \overset{\Delta_e}{\rightarrow} \langle \Delta_1', c_1, m_1', (s_1', ctzx') \rangle$ and $c_1 = \text{if } e \text{ then } c' \text{ else } c''$, it follows that $\Delta_1' = \Delta_1[pc_u \mapsto \Delta_1(pc_u) \cup \Delta_1(e)]$, $m_1 = m_1'$, $\tau_1 = \epsilon, c_1' = c_1 \text{ set } pc \text{ to } \Delta_1(pc_u)$, and $(s_1, ctzx1) = (s_1', ctzx')$. From the rule, it also follows that $[\ell(m_1)] = m_1$. There are two cases:

- $\exists \Delta'(pc_x) \subseteq \ell$ and $\Delta'(pc_y) \subseteq \ell$ (or vice versa). From the rule, it follows that $\Delta_1' = \Delta_1[pc_u \mapsto \Delta_1(pc_u) \cup \Delta_1(e)]$ and $\Delta_2' = \Delta_2[pc_u \mapsto \Delta_2(pc_u) \cup \Delta_2(e)]$. From this and $\Delta_1'(pc_u) \subseteq \ell$, it follows that $\Delta_2'(pc_u) \subseteq \ell$. From this and (2), it follows that $\Delta_1'(pc_u) = \Delta_2'(pc_u)$.

- $\Delta_1'(pc_u) \subseteq \ell$, $\Delta_2'(pc_u) \subseteq \ell$, and $\Delta_1'(pc_y) \neq \Delta_2'(pc_y)$. We have already shown above that from $\Delta_1'(pc_u) \subseteq \ell$ and (2), it follows $\Delta_2'(pc_u) = \Delta_2'(pc_u)$, which contradicts $\Delta_1'(pc_u) \neq \Delta_2'(pc_u)$.

- $[\ell(m_2)] = \text{ff}$. From this, $\langle \Delta_2, c_2, m_2, (s_2, ctzx2) \rangle \overset{\Delta_e}{\rightarrow} \langle \Delta_2', c_2', m_2', (s_2', ctzx') \rangle$ and $c_2 = \text{if } e \text{ then } c' \text{ else } c''$, it follows that $\Delta_2' = \Delta_2[pc_u \mapsto \Delta_2(pc_u) \cup \Delta_2(e)]$, $m_2 = m_2'$, $\tau_2 = \epsilon, c_2' = c_2', \text{ set } pc \text{ to } \Delta_2(pc_u)$, and $(s_2, ctzx2) = (s_2', ctzx')$. The rest of the proof is similar to the case above.
Rule F-IfFalse. The proof of this case is similar to that of F-IfTrue.

Rule F-WhileFalse. The proof of this case is similar to that of F-IfTrue.

Rule F-Select. From the rule, it follows that \( c_1 = [x \leftarrow \text{SELECT } \varphi] \). From this and (4), it follows that \( c_1 = [x \leftarrow \text{SELECT } \varphi] \). From \((\Delta_1, c_1, m_1, (s_1, ctx_1)) \leftrightarrow_n\Delta'\) \((\Delta'_1, c'_1, m'_1, (s'_1, ctx'_1))\) and \( c_1 = [x \leftarrow q] \), it follows that \( \Delta'_1 = \Delta_1[x \rightarrow \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q)] \cup \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \), \( m'_1 = m_1[x \rightarrow r_1], \tau_1 = \epsilon, c'_1 = \epsilon, \) where \( \text{free}(\varphi) = \{v_1, \ldots, v_n\} \), \( q = \text{SELECT } \varphi[v_1 \rightarrow v_1(m_1), \ldots, v_n \rightarrow v_n(m_n), m) \), and \( [q_1](s_1, ctx_1) = (s'_1, ctx'_1), r_1, \epsilon, \). From \((\Delta_2, c_2, m_2, (s_2, ctx_2)) \leftrightarrow_n\Delta_2\) \((\Delta'_2, c'_2, m'_2, (s'_2, ctx'_2))\) and \( c_2 = [x \leftarrow q] \), it follows that \( \Delta'_2 = \Delta_2[x \rightarrow \Delta_2(pc_u) \cup L_\varphi(\Delta_2, q)] \cup \bigcup_{m \in \text{free}(\varphi)} \Delta_2(v) \), \( m'_2 = m_2[x \rightarrow r_2], \tau_2 = \epsilon, c'_2 = \epsilon, \) where \( \text{free}(\varphi) = \{v_1, \ldots, v_n\} \), \( q = \text{SELECT } \varphi[v_1 \rightarrow v_1(m_1), \ldots, v_n \rightarrow v_n(m_n), m) \), and \( [q_2](s_2, ctx_2) = (s'_2, ctx'_2), r_2, \epsilon, \). Assume, for contradiction's sake, that \((\Delta_1, c_1, m_1, (s_1, ctx_1)) \not\equiv_\ell (\Delta_2, c_2, m_2, (s_2, ctx_2))\). From this and (2), there are three cases:

- \( \Delta_1(x) \subseteq \ell \text{ and } \Delta_2(x) \not\subseteq \ell \) (or vice versa). From \( \Delta_1(x) \subseteq \ell \) and \( \Delta_2(x) \not\subseteq \ell \), it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \), it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \subseteq \ell \). From this, it follows that \( \Delta_1(pc_u) \subseteq \ell \) and \( \Delta_2(pc_u) \subseteq \ell \). From \( \text{propD.9.9}, \) it follows that \( L_\varphi(\Delta_1, q) \subseteq \ell \). From this and point (3) in the notion of disclosure order, it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \subseteq \ell \). From this and \( \Delta_2(x) \subseteq \ell, \) leading to a contradiction.

- \( \Delta_1(x) \not\subseteq \ell, \Delta_2(x) \subseteq \ell \), and \( \Delta_1(x) \neq \Delta_2(x) \). From \( \Delta_1(x) \not\subseteq \ell \) and \( \Delta_2(x) \subseteq \ell \), it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \), it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \subseteq \ell \). From this, it follows that \( \Delta_1(pc_u) \subseteq \ell \) and \( \Delta_2(pc_u) \subseteq \ell \). From \( \text{propD.9.9}, \) it follows that \( L_\varphi(\Delta_1, q) \subseteq \ell \). From this and point (3) in the notion of disclosure order, it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \subseteq \ell \). From this and (2), it follows that \( \Delta_1(q) \not\subseteq \ell \) and \( \Delta_2(q) \subseteq \ell \). From this and \( \Delta_1(x) \not\subseteq \ell \), leading to a contradiction.

- \( \Delta_1(x) \subseteq \ell, \Delta_2(x) \not\subseteq \ell \), and \( m_1(x) \neq m_2(x) \). From \( \Delta_1(x) \subseteq \ell \) and \( \Delta_2(x) \not\subseteq \ell \), it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \), it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \subseteq \ell \). From this, it follows that \( \Delta_1(pc_u) \cup L_\varphi(\Delta_1, q) \subseteq \bigcup_{m \in \text{free}(\varphi)} \Delta_1(v) \subseteq \ell \). From this and (2), it follows that \( \Delta_1(q) \not\subseteq \ell \) and \( \Delta_2(q) \subseteq \ell \). From this and \( \Delta_1(x) \subseteq \ell \), leading to a contradiction.

Rule F-UpdateDatabaseOK. Without loss of generality, we assume that the query is an INSERT query. From the rule, it follows that \( c_2 = [x \leftarrow \text{UPDATE } \varphi]. \) From this and (4), it follows that \( c_2 = [x \leftarrow \text{UPDATE } \varphi]. \) From \( \Delta_1, c_1, m_1, (s_1, ctx_1) \leftrightarrow_n\Delta_1, c_1, m_1, (s_1, ctx_1)\) and \( c_1 = [x \leftarrow \text{UPDATE } \varphi] \), it follows that \( \Delta'_1 = \Delta_1[T(v)] \rightarrow \Delta_1(pc_u) \cap \bigcup_{s \leq n} \Delta_1(e_1) \), \( x \rightarrow \Delta_1(pc_u) \cap \bigcup_{s \leq n} \Delta_1(e_1) \), \( m_1 = m_1[x \rightarrow r_1], \tau_1 = \epsilon, c'_1 = \epsilon, \) where \( \text{free}(\varphi) = \{e_1, \ldots, e_n\} \), \( \Delta_1, (s_1, ctx_1) = (s'_1, ctx'_1), r_1, \epsilon, \). From \( \Delta_2, c_2, m_2, (s_2, ctx_2) \leftrightarrow_n\Delta_2, c_2, m_2, (s_2, ctx_2)\) and \( c_2 = [x \leftarrow \text{UPDATE } \varphi] \), it follows that \( \Delta'_2 = \Delta_2[T(v)] \rightarrow \Delta_2(pc_u) \cap \bigcup_{s \leq n} \Delta_2(e_1) \), \( x \rightarrow \Delta_2(pc_u) \cap \bigcup_{s \leq n} \Delta_2(e_1) \), \( m'_2 = m_2[x \rightarrow r_2], \tau_2 = \epsilon, c'_2 = \epsilon, \) where \( \text{free}(\varphi) = \{e_1, \ldots, e_n\} \), \( \tau_2 = \{e_2(m_2), \ldots, e_n(m_2)\} \), \( q_2 = [x \leftarrow \text{UPDATE } \varphi] \). Assume, for contradiction's sake, that \( \Delta_1, c_1, m_1, (s_1, ctx_1) \not\equiv_\ell (\Delta_2, c_2, m_2, (s_2, ctx_2))\). From this and
D.3. Monitor's soundness

1. \( \Delta_1(x) \subseteq \ell \) and \( \Delta_1'(x) \not\subseteq \ell \) (or vice versa). From \( \Delta_1'(x) \not\subseteq \ell \) and \( \Delta_1'(x) = \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \), it follows that \( \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \subseteq \ell \). From this, it follows that \( \Delta_1(\pc_1) \subseteq \ell \) and \( \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \subseteq \ell \). From \( \Delta_1(\pc_1) \subseteq \ell \) and (2), it follows that \( \Delta_2(\pc_2) \subseteq \ell \). From \( \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \subseteq \ell \) and (2), it follows that \( \Delta_2(\pc_2) \subseteq \ell \). From this, \( \Delta_2(\pc_2) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \subseteq \ell \). From this and \( \Delta_2'(x) = \Delta_2(\pc_2) \cup \bigcup_{1 \leq i \leq n} \Delta_2(e_i) \), it follows that \( \Delta_2'(x) \subseteq \ell \), leading to a contradiction.

2. \( \Delta_1(x) \not\subseteq \ell \), \( \Delta_2(x) \not\subseteq \ell \), and \( \Delta_1(x) \neq \Delta_2(x) \). From \( \Delta_1(x) \not\subseteq \ell \) and \( \Delta_1'(x) = \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \), it follows that \( \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \). From this, it follows that \( \Delta_2(\pc_2) \subseteq \ell \) and \( \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \). From \( \Delta_1(\pc_1) \subseteq \ell \) and (2), it follows that \( \Delta_2(\pc_2) = \Delta_1(\pc_1) \). From \( \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \) and (2), it follows that \( \bigcup_{1 \leq i \leq n} \Delta_2(e_i) = \Delta_1(e_i) \). From this, \( \Delta_2(\pc_2) \cup \bigcup_{1 \leq i \leq n} \Delta_2(e_i) = \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \). From this and \( \Delta_2(x) = \Delta_2(\pc_2) \cup \bigcup_{1 \leq i \leq n} \Delta_2(e_i) \), it follows that \( \Delta_2(x) = \Delta_1'(x) \), leading to a contradiction.

3. \( \Delta_1(x) \not\subseteq \ell \), \( \Delta_1'(x) \not\subseteq \ell \), and \( m_1(x) \neq m_2(x) \). From the database semantics, both \( r_1 \) and \( r_2 \) are \( T \). From this, \( m_1(x) = r_1 \) and \( m_2(x) = r_2 \), it follows that \( m_1(x) = m_2(x) \), leading to a contradiction.

4. \( \Delta_1(T(\pi_1)) \not\subseteq \ell \) and \( \Delta_1'(T(\pi_1)) \not\subseteq \ell \). From \( \Delta_1(T(\pi_1)) \not\subseteq \ell \) and \( \Delta_1'(T(\pi_1)) = \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \), it follows that \( \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \). From (3), it follows that \( \Delta_2(\pc_2) \not\subseteq \ell \). From this and \( \Delta_2(\pc_2) \cup \bigcup_{1 \leq i \leq n} \Delta_2(e_i) \not\subseteq \ell \), it follows that \( \bigcup_{1 \leq i \leq n} \Delta_2(e_i) \not\subseteq \ell \). From the rule, it follows that \( \bigcup_{1 \leq i \leq n} \Delta_2(e_i) \not\subseteq \ell \). From this and \( \bigcup_{1 \leq i \leq n} \Delta_2(e_i) \not\subseteq \ell \), it follows that \( \Delta_2(T(\pi_2)) \not\subseteq \ell \). From this and (2), it follows that \( \Delta_1(T(\pi_1)) \not\subseteq \ell \). From \( \Delta_1(T(\pi_1)) \not\subseteq \ell \) and \( \Delta_1(\pi_1) = \Delta_1(\pi_1) \), it follows that \( \Delta_1(\pi_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \). From this, \( \pi_1 = \pi_2 \), it follows that \( \Delta_1(T(\pi_2)) \not\subseteq \ell \), leading to a contradiction.

5. \( \Delta_1(T(\pi_2)) \not\subseteq \ell \) and \( \Delta_1'(T(\pi_2)) \not\subseteq \ell \). The proof of this case is similar to that of case 4.

6. \( \Delta_2(T(\pi_2)) \not\subseteq \ell \) and \( \Delta_2'(T(\pi_2)) \not\subseteq \ell \). The proof of this case is similar to that of case 5.

7. \( \Delta_2(T(\pi_1)) \not\subseteq \ell \) and \( \Delta_2'(T(\pi_1)) \not\subseteq \ell \). The proof of this case is similar to that of case 5.

8. \( \Delta_1'(T(\pi_1)) \neq \Delta_2(T(\pi_1)) \). From \( \Delta_1'(T(\pi_1)) \neq \Delta_2(T(\pi_1)) = \Delta_1'(T(\pi_1)) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \), it follows that \( \Delta_1(\pc_1) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \). From this, \( \Delta_1(\pc_1) \not\subseteq \ell \) and \( \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \). From this and (2), it follows that \( \Delta_2(\pc_2) = \Delta_1(\pc_1) \). From \( \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \) and (2), it follows that \( \Delta_2(\pc_2) \cup \bigcup_{1 \leq i \leq n} \Delta_1(e_i) \not\subseteq \ell \). From this, \( \pi_1 = \pi_2 \), it follows that \( \Delta_2(T(\pi_1)) \not\subseteq \ell \). From this and \( \Delta_1(T(\pi_2)) \not\subseteq \ell \), it follows that \( \Delta_1(T(\pi_2)) \not\subseteq \ell \). This contradicts \( \Delta_1'(T(\pi_1)) \not\subseteq \ell \).

9. \( \Delta_1(T(\pi_1)) \neq \ell \) and \( \Delta_1'(T(\pi_1)) \neq \ell \). The proof is similar to that of case 8.

10. There is a query \( q \) such that \( L_\phi(\Delta_1, q) \not\subseteq \ell \), \( L_\phi(\Delta_2, q) \not\subseteq \ell \), and \( [q]^{\Delta_1} \neq [q]^{\Delta_2} \), where \( db_1' \) and \( db_2' \) are the databases in \( s_1 \) and \( s_2 \) respectively. From \( L_\phi \)’s definition, it follows that \( [q]^{\Delta_1} \not\subseteq \ell \) and \( [q]^{\Delta_2} \not\subseteq \ell \). There are two cases:

- There is a \( Q \in supp_{\phi}(q) \) such that \( T(\pi_1) \not\subseteq Q \) and \( T(\pi_2) \not\subseteq Q \). From \( L_\phi(\Delta_1, q) \not\subseteq \ell \), it follows that \( \bigcup_{Q' \in supp_{\phi}(q)} L_\phi(\Delta_1, q'') \not\subseteq \ell \). From that and \( \Delta_1(T(\pi_1)) \not\subseteq \ell \), it follows that \( \Delta_1(T(\pi_1)) \not\subseteq \ell \). From this, \( Q \subseteq RC^{pred} \), and \( L_\phi(\Delta_1, q') = \Delta(q') \) for any \( q'' \in RC^{pred} \) (Proposition D.10), it follows that \( \bigcup_{Q' \in supp_{\phi}(q)} \Delta(q'') \not\subseteq \ell \). From this and (2), it follows that \( \bigcup_{Q' \in supp_{\phi}(q)} [q]^{db_1'} = [q]^{db_2'} \), where \( db_1' \) and \( db_2' \) are the database in \( s_1 \) and \( s_2 \) respectively. From this, \( T(\pi_1) \not\subseteq Q \), \( T(\pi_2) \not\subseteq Q \), and the fact that we modify only the values of \( T(\pi_1) \) and \( T(\pi_2) \), it follows that \( \bigcup_{Q' \in supp_{\phi}(q)} [q]^{db_1'} = [q]^{db_2'} \). From this and \( Q \) determines \( q \), it follows that \( [q]^{s_1} = [q]^{s_2} \).
leading to a contradiction.

- For all $Q \in \text{supp}_{D,F}(q)$, $T(\tau_1) \in Q$ or $T(\tau_2) \in Q$. Assume that $T(\tau_1) \in Q$ (the proof in case $T(\tau_2) \in Q$ is analogous). From $\bigcup_{Q \in \text{supp}_{D,F}(q)} \bigcup_{q' \in Q} \Delta_i'(q') \subseteq \ell$, it follows that $\bigwedge_{q' \in Q} \Delta_i'(q') \subseteq \ell$. From this and $T(\tau_1) \in Q$, it follows that $\Delta_i'(q') \subseteq \ell$. From this and $\Delta_i(T(\tau_1)) = \Delta_i(\text{pc}_u) \cup \bigcup_{1 \leq n \leq \ell} \Delta_i(e_n)$, it follows that $\Delta_i(\text{pc}_u) \cup \bigcup_{1 \leq n \leq \ell} \Delta_i(e_n) \subseteq \ell$. From this, it follows that $\bigwedge_{1 \leq n \leq \ell} \Delta_i(e_n) \subseteq \ell$. From this and (2), it follows that $\bigwedge_{1 \leq n \leq \ell} \Delta_i(e_n) \subseteq \ell$.

- Let $q''$ be a query in $Q \setminus \{ T(\tau_1) \}$. From this and $q'' \in \text{supp}_{D,F}(q)$ (Proposition D.10), it follows that $\bigwedge_{q' \in Q \setminus \{ T(\tau_1) \}} \Delta_i'(q') \subseteq \ell$. From this and (2), it follows that $\Delta_i'(q'') \subseteq \ell$. From this, it follows that $\bigwedge_{q'' \in Q \setminus \{ T(\tau_1) \}} \Delta_i'(q'') \subseteq \ell$. From this, it follows that $\bigwedge_{q'' \in Q \setminus \{ T(\tau_1) \}} \Delta_i'(q'') \subseteq \ell$. From this, it follows that $\Delta_i(q'') \subseteq \ell$. From this, it follows that $\bigwedge_{q'' \in Q \setminus \{ T(\tau_1) \}} \Delta_i(q'') \subseteq \ell$.

- For all $q'' \in \text{supp}_{D,F}(q)$ (and therefore $Q$ determines $q$), it follows that $\bigwedge_{q'' \in Q \setminus \{ T(\tau_1) \}} \Delta_i(q'') \subseteq \ell$.

- Rule $\text{F-UpdateConfigurationOK}$. From the rule, it follows that $c_1 = [x \leftarrow q]$. From this and (4), it follows that $c_2 = [x \leftarrow q]$. From $\Delta_1, c_1, m_1, (s_1, ctx_1)$ and $c_1 = [x \leftarrow q]$, it follows that $\Delta_1 = \Delta_1[x \leftarrow \Delta_1(\text{pc}_u) \cup \bigcup_{q' \in \text{vars}(q')} \Delta_i(q')]$, and $\bigwedge_{q' \in \text{vars}(q')} \Delta_i(q') \subseteq \ell$. From this and (2), it follows that $\bigwedge_{q' \in \text{vars}(q')} \Delta_i(q') \subseteq \ell$. From this, it follows that $\bigwedge_{q' \in \text{vars}(q')} \Delta_i(q') \subseteq \ell$. From this, it follows that $\bigwedge_{q' \in \text{vars}(q')} \Delta_i(q') \subseteq \ell$. From this and $\Delta_i(\text{pc}_u) \cup \bigcup_{q' \in \text{vars}(q')} \Delta_i(q') \subseteq \ell$. From this and (2), it follows that $\bigwedge_{q' \in \text{vars}(q')} \Delta_i(q') \subseteq \ell$. From this and $\Delta_i(\text{pc}_u) \cup \bigcup_{q' \in \text{vars}(q')} \Delta_i(q') \subseteq \ell$. From this and $\Delta_i(x) \subseteq \ell$ and $\Delta_i(x) \subseteq \ell$ (or vice versa). From $\Delta_i(x) \subseteq \ell$ and $\Delta_i(x) \subseteq \ell$, it follows that $\Delta_i(x) \subseteq \ell$. From this, it follows that $\Delta_i(x) \subseteq \ell$. From this, it follows that $\Delta_i(x) \subseteq \ell$. From this and $\Delta_i(x) \subseteq \ell$, it follows that $\Delta_i(x) \subseteq \ell$. From this and (2), it follows that $\Delta_i(x) \subseteq \ell$. From this, it follows that $\Delta_i(x) \subseteq \ell$. From this, it follows that $\Delta_i(x) \subseteq \ell$. From this and (2), it follows that $\Delta_i(x) \subseteq \ell$. From this, it follows that $\Delta_i(x) \subseteq \ell$. From this, it follows that $\Delta_i(x) \subseteq \ell$. From this and $\Delta_i(x) \subseteq \ell$, it follows that $\Delta_i(x) \subseteq \ell$. From this and (2), it follows that $\Delta_i(x) \subseteq \ell$. From this, it follows that $\Delta_i(x) \subseteq \ell$. From this and $\Delta_i(x) \subseteq \ell$, it follows that $\Delta_i(x) \subseteq \ell$. From this and (2), it follows that $\Delta_i(x) \subseteq \ell$.

**Induction Step.** The proof of the induction step directly follows from the induction hypothesis for all rules except $\text{F-AsUser}$. For the $\text{F-AsUser}$ rule, the proof can be done by case distinction on the executed query. The proofs for the various cases are similar to that of the rules $\text{F-Select}$, $\text{F-UpdateDatabaseOK}$, and $\text{F-UpdateConfigurationOK}$.

Lemma D.19 states that, under appropriate conditions, performing a step of execution in two $\ell$-equivalent states with the same initial code results in the same code.
Lemma D.19. Let $s_{sec}$ be the policy used to initialize the monitor, $(\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle)$, $(\Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle)$, $(\Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle)$, and $(\Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle)$ be four local configurations, and $\ell \in L$ be a label. If the following conditions hold:

1. \( s_1 \overset{\ell}{\Rightarrow} s_2 \),
2. \( (\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle) \approx_{\ell} (\Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle) \),
3. \( \Delta'_1(\mathit{pc}_u) \subseteq \ell \) and \( \Delta'_2(\mathit{pc}_u) \subseteq \ell \),
4. \( c_1 = c_2 \),
5. \( (\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle) \vdash_{\mathcal{D}} (\Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle) \),
6. \( (\Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle) \vdash_{\mathcal{D}} (\Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle) \),

then \( c'_1 = c'_2 \).

Proof. Let $s_{sec}$ be the policy used to initialize the monitor, $(\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle)$, $(\Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle)$, $(\Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle)$, and $(\Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle)$ be four local configurations, and $\ell \in L$ be a label such that the following conditions hold:

1. \( s_1 \overset{\ell}{\Rightarrow} s_2 \),
2. \( (\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle) \approx_{\ell} (\Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle) \),
3. \( \Delta'_1(\mathit{pc}_u) \subseteq \ell \) and \( \Delta'_2(\mathit{pc}_u) \subseteq \ell \),
4. \( c_1 = c_2 \),
5. \( (\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle) \vdash_{\mathcal{D}} (\Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle) \),
6. \( (\Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle) \vdash_{\mathcal{D}} (\Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle) \).

We prove our claim by induction on the rules defining $\vdash_{\mathcal{D}}$.

**Base Case.** The proof is similar to that for the F-Assign or F-Out case. The only interesting and the expansion procedure:

- **Rule F-IF-True.** From the rule, it follows that $c_1 = \text{if } e \text{ then } c' \text{ else } c''$. From this and (4), it follows that $c_2 = \text{if } e \text{ then } c' \text{ else } c''$. From $(\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle) \vdash_{\mathcal{D}} (\Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle)$ and $c_1 = \text{if } e \text{ then } c' \text{ else } c''$, it follows that $\Delta'_1 = \Delta_1[\mathit{pc}_u \Rightarrow \Delta_1(\mathit{pc}_u) \cup \Delta_1(e)]$, $m_1 = m'_1$, $\tau_1 = e$, $\Delta'_1 = c' \mid \mathit{set pc to} \Delta_1(\mathit{pc}_u)$, and $(\langle s_1, ctx_1 \rangle) = \langle s'_1, ctx'_1 \rangle$. From this, it also follows that $\mathit{tt} \in \mathit{var}(c_1, m_1)$.
- **Rule F-IF-False.** The proof is similar to that for the F-IF-True case.
- **Rule F-WHILE-True.** The proof is similar to that for the F-IF-True case.
- **Rule F-WHILE-False.** The proof is similar to that for the F-IF-False case.
- **Rule F-EXPAND.** From the rule, it follows that $c_1 = x \leftarrow q$. From this and (4), it follows that $c_2 = x \leftarrow q$. From $(\Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle) \vdash_{\mathcal{D}} (\Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle)$ and $c_1 = x \leftarrow q$, it follows that $\Delta'_1 = \Delta_1, m_1 = m'_1, \tau_1 = e, c'_1 = \text{expand}(s_1, x, q, u)$, and $(\langle s_1, ctx_1 \rangle) = \langle s'_1, ctx'_1 \rangle$. From $(\Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle) \vdash_{\mathcal{D}} (\Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle)$ and $c_2 = x \leftarrow q$, it follows that $\Delta'_2 = \Delta_2, m_2 = m'_2, \tau_2 = e, c'_2 = \text{expand}(s_2, x, q, u)$, and $(\langle s_2, ctx_2 \rangle) = \langle s'_2, ctx'_2 \rangle$. There are a number of cases depending on $q$:
- $q$ is SELECT $\varphi$. For SELECT queries, the result of the expansion procedure is the same for any two database states $s_1$ and $s_2$. Therefore, it follows that $\text{expand}(s_1, x, q, u) = \text{expand}(s_2, x, q, u)$, and, therefore, $c'_1 = c'_2$.
- $q$ is INSERT $\varphi$ INTO $T$. The expansion procedure relies only on the allowed and apply procedures, which, in turn, depend only on the configuration of the database state. From this and (1), it follows that $\text{expand}(s_1, x, q, u) = \text{expand}(s_2, x, q, u)$, and, therefore, $c'_1 = c'_2$.
- $q$ is DELETE $\varphi$ FROM $T$. The proof of this case is similar to that of INSERT $\varphi$ INTO $T$.
- $q$ is GRANT $p$ TO $u$. The expansion procedure relies only on the allowed and apply procedures, which, in turn, depend only on the configuration of the database state. From this and (1), it follows that $\text{expand}(s_1, x, q, u) = \text{expand}(s_2, x, q, u)$, and, therefore, $c'_1 = c'_2$.
- $q$ is GRANT $p$ TO $u$ WITH GRANT OPTION. The proof of this case is similar to that of GRANT $p$ TO $u$.
- $q$ is REVOKE $p$ FROM $u$. The proof of this case is similar to that of REVOKE $p$ FROM $u$.
- $q$ is a CREATE queries. For CREATE queries, the result of the expansion procedure is the same for any two database states $s_1$ and $s_2$. Therefore, it follows that $\text{expand}(s_1, x, q, u) = \text{expand}(s_2, x, q, u)$, and, therefore, $c'_1 = c'_2$.
\( q \) is ADD USER \( u' \). For ADD USER queries, the result of the expansion procedure is the same for any two database states \( s_1 \) and \( s_2 \). Therefore, it follows that \( \text{expand}(s_1, x, q, u) = \text{expand}(s_2, x, q, u) \) and, therefore, \( c'_1 = c'_2 \).

**Induction Step.** The proof of the induction step directly follows from the induction hypothesis.
From this, it follows that $(\Delta_1, c_1', m_1', (s_1', ctx_1')) \xrightarrow{\ast_{a_u}} (\Delta''_1, [\varepsilon ; \text{set } pc \text{ to } \Delta_1(pc_u)], m''_1, (s''_1, ctx_1'))$ and $(\Delta_2, c_2, m_2, (s_2, ctx_2')) \xrightarrow{\ast_{a_u}} (\Delta''_2, [\varepsilon ; \text{set } pc \text{ to } \Delta_2(pc_u)], m''_2, (s''_2, ctx_2'))$. By applying the F-EXPANCODE and the F-SEQEMPTY rules, we obtain $(\Delta_1, c_1, m_1', (s_1', ctx_1')) \xrightarrow{\ast_{a_u}} (\Delta''_1, [\varepsilon ; \text{set } pc \text{ to } \Delta_1(pc_u)], m''_1, (s''_1, ctx_1'))$ and $(\Delta_2, c_2, m_2, (s_2, ctx_2')) \xrightarrow{\ast_{a_u}} (\Delta''_2, [\varepsilon ; \text{set } pc \text{ to } \Delta_2(pc_u)], m''_2, (s''_2, ctx_2'))$. By applying the F-EXPANCODE and the F-UPDATELABELS rules, we obtain $(\Delta_1, c_1, m_1', (s_1', ctx_1')) \xrightarrow{\ast_{a_u}} (\Delta''_1, [\varepsilon ; \text{set } pc \text{ to } \Delta_1(pc_u)], m''_1, (s''_1, ctx_1'))$ and $(\Delta_2, c_2, m_2, (s_2, ctx_2')) \xrightarrow{\ast_{a_u}} (\Delta''_2, [\varepsilon ; \text{set } pc \text{ to } \Delta_2(pc_u)], m''_2, (s''_2, ctx_2'))$ (observe that $\Delta''_1[pc_u \mapsto \Delta_1(pc_u)] = \Delta''_1$ and $\Delta''_2[pc_u \mapsto \Delta_2(pc_u)] = \Delta''_2$). From this, it directly follows our claim (since the code is the same in both final configurations and from (3), it follows that the pc_u is below l in both configurations).

- $[e](m_1) = [e](m_2) = \mathbf{tt}$. The proof of this case is similar to that of $[e](m_1) = [e](m_2) = \mathbf{tt}$.
- $[e](m_1) = \mathbf{tt}$ and $[e](m_2) = \mathbf{ff}$. The proof of this case is similar to that of $[e](m_1) = \mathbf{tt}$.

This completes the proof of our claim.

Lemma D.20 presents some results about computations involving while statements.

**Lemma D.21.** Let sec0 be the policy used to initialize the monitor, $(\Delta_1, c_1, m_1, (s_1, ctx_1)), (\Delta'_1, c_1', m_1', (s_1', ctx_1')), (\Delta_2, c_2, m_2, (s_2, ctx_2)), (\Delta'_2, c_2', m_2', (s_2', ctx_2'))$ be four local configurations, and $\ell \in L$ be a label. If the following conditions hold:

1. $s_1 \equiv_{c_1} s_2$.
2. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \approx_{\ast} (\Delta_2, c_2, m_2, (s_2, ctx_2))$.
3. $\Delta_1(pc_u) \subseteq \ell$ and $\Delta_2(pc_u) \nsubseteq \ell$.
4. $\Delta'_1(pc_u) \subseteq \ell$ or $\Delta'_2(pc_u) \nsubseteq \ell$.
5. $c_1 = c_2$.
6. first$(c_1) = \text{while } c \text{ do } c'$.
7. $(\Delta_1, c_1, m_1, (s_1, ctx_1)) \xrightarrow{\ast_{a_u}} (\Delta'_1, c_1', m_1', (s_1', ctx_1'))$.
8. $(\Delta_2, c_2, m_2, (s_2, ctx_2)) \xrightarrow{\ast_{a_u}} (\Delta'_2, c_2', m_2', (s_2', ctx_2'))$.

Then one of the following conditions hold:

(a) $[e](m_1) = \mathbf{tt}$ and for all $(\Delta''_1, c''_1, m''_1, (s''_1, ctx''_1))$ such that $(\Delta'_1, c''_1, m''_1, (s''_1, ctx''_1)) \xrightarrow{\ast_{a_u}} (\Delta''_1, c''_1, m''_1, (s''_1, ctx''_1))$, $c''_1 \neq \varepsilon$.

(b) $[e](m_2) = \mathbf{tt}$ and for all $(\Delta''_2, c''_2, m''_2, (s''_2, ctx''_2))$ such that $(\Delta'_2, c''_2, m''_2, (s''_2, ctx''_2)) \xrightarrow{\ast_{a_u}} (\Delta''_2, c''_2, m''_2, (s''_2, ctx''_2))$, $c''_2 \neq \varepsilon$.

(c) there are $(\Delta'_1, c_1', m_1', (s_1', ctx_1'))$ and $(\Delta'_2, c_2', m_2', (s_2', ctx_2'))$ such that $(\Delta_1, c_1', m_1', (s_1', ctx_1')) \xrightarrow{\ast_{a_u}} (\Delta''_1, c''_1, m''_1, (s''_1, ctx''_1))$ and $(\Delta_2, c_2', m_2', (s_2', ctx_2')) \xrightarrow{\ast_{a_u}} (\Delta''_2, c''_2, m''_2, (s''_2, ctx''_2))$, $c''_1 = c''_2$, and $\Delta''_1(pc_u) \subseteq \ell$ and $\Delta''_2(pc_u) \nsubseteq \ell$.

**Proof.** The proof is similar to that of Lemma D.20.

### 3.7 Lemmas about the global semantics

Here we present some auxiliary results about the global semantics of our enforcement mechanism.

**Lemma D.22.** Let sec0 be the policy used to initialize the monitor, $(\Delta_1, C_1, M_1, (s_1, ctx_1), S_1)$, $(\Delta'_1, C'_1, M'_1, (s'_1, ctx'_1), S'_1)$, $(\Delta_2, C_2, M_2, (s_2, ctx_2), S_2)$, and $(\Delta'_2, C'_2, M'_2, (s'_2, ctx'_2), S'_2)$ be four global configurations, $\ell \in L$ be a label, $n'$ be the first value in $S$, $n = (n' \mod |C_1|) + 1$, and $u$ be the user associated with the n-th program in $C_1$. If the following conditions hold:

1. $s_1 \equiv_{c_1} s_2$.
2. $(\Delta_1, C_1, M_1, (s_1, ctx_1), S_1) \approx_{\ast} (\Delta_2, C_2, M_2, (s_2, ctx_2), S_2)$.
3. $\Delta_1(pc_u) \subseteq \ell$ and $\Delta_2(pc_u) \nsubseteq \ell$.
4. $\Delta'_1(pc_u) \subseteq \ell$ and $\Delta'_2(pc_u) \nsubseteq \ell$.
5. $C_1 = C_2$.
6. $S_1 = S_2$.
7. $(\Delta_1, C_1, M_1, (s_1, ctx_1), S_1) \xrightarrow{\ast_{a_u}} (\Delta'_1, C'_1, M'_1, (s'_1, ctx'_1), S'_1)$,
8. \(\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, S_2 \rangle \xrightarrow{\omega_2} \langle \Delta'_2, C'_2, M'_2, \langle s'_2, ctx'_2 \rangle, S'_2 \rangle\), then \(C'_2 = C'_2\) and \(S'_2 = S'_2\).

Proof. The claim directly follows from (4), (5), and Lemma D.19 (together with (1) and (3)).

Lemma D.23 states some properties of the execution of if statements in the global semantics.

**Lemma D.23.** Let \(sec_0\) be the policy used to initialize the monitor, \(\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, S_1 \rangle\), \(\langle \Delta_1, C'_1, M'_1, \langle s'_1, ctx'_1 \rangle, S'_1 \rangle\), \(\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, S_2 \rangle\), and \(\langle \Delta'_2, C'_2, M'_2, \langle s'_2, ctx'_2 \rangle, S'_2 \rangle\) be four global configurations, and \(\ell \in L\) be a label, \(n'\) be the first value in \(S, n = (n' \mod |C_1|) + 1,\) and \(u\) be the user associated with the n-th program in \(C_1\). If the following conditions hold:

1. \(s_1 \equiv s_2\),
2. \(\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, S_1 \rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, S_2 \rangle\),
3. \(\Delta_1(\text{pc}_1) \subseteq \ell\) and \(\Delta_2(\text{pc}_2) \subseteq \ell\),
4. \(\Delta_1(\text{pc}_1) \not\subseteq \ell\) or \(\Delta_2(\text{pc}_2) \not\subseteq \ell\),
5. \(C_1 = C_2\),
6. first\((C_1(n)) = \text{if } e \text{ then } c' \text{ else } c'\),
7. \(S_1 = S_2\),
8. \(\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, S_1 \rangle \xrightarrow{\omega_1} \langle \Delta'_1, C'_1, M'_1, \langle s'_1, ctx'_1 \rangle, S'_1 \rangle\),
9. \(\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, S_2 \rangle \xrightarrow{\omega_2} \langle \Delta'_2, C'_2, M'_2, \langle s'_2, ctx'_2 \rangle, S'_2 \rangle\),

then one of the following conditions holds:

(a) \[\llbracket e \rrbracket(m_1) = \text{tt} \text{ and for all } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \text{ such that } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \xrightarrow{\omega_1} \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle, c'_1 \neq \varepsilon,\]

(b) \[\llbracket e \rrbracket(m_1) = \text{ff} \text{ and for all } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \text{ such that } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \xrightarrow{\omega_1} \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle, c'_1 \neq \varepsilon,\]

(c) \[\llbracket e \rrbracket(m_2) = \text{tt} \text{ and for all } \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle \text{ such that } \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle \xrightarrow{\omega_2} \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle, c'_2 \neq \varepsilon,\]

(d) \[\llbracket e \rrbracket(m_2) = \text{ff} \text{ and for all } \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle \text{ such that } \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle \xrightarrow{\omega_2} \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle, c'_2 \neq \varepsilon,\]

(e) \[\text{there are } \langle \Delta'_1, c'_1, M'_1, \langle s'_1, ctx'_1 \rangle, S'_1 \rangle \text{ and } \langle \Delta'_2, c'_2, M'_2, \langle s'_2, ctx'_2 \rangle, S'_2 \rangle \text{ such that } \langle \Delta'_1, c'_1, M'_1, \langle s'_1, ctx'_1 \rangle, S'_1 \rangle \xrightarrow{\omega_1} \langle \Delta'_2, c'_2, M'_2, \langle s'_2, ctx'_2 \rangle, S'_2 \rangle, \]

\(S'_1 = C'_1, S'_2 = C'_2, S'_1 = S'_2,\) and \(\Delta'_1(\text{pc}_1) \subseteq \ell\) and \(\Delta'_2(\text{pc}_2) \subseteq \ell\).

Proof. From (6) and the rules F-IFTrue and F-IFFalse, it follows that the only applicable rule in the global semantics is F-ATOMIC-STATEMENT. Our claim directly follows from this, (3), (4), Lemma D.20, and the fact that the F-ATOMIC-STATEMENT rule does not modify the scheduler.

Lemma D.24 states some properties of the execution of while statements in the global semantics.

**Lemma D.24.** Let \(sec_0\) be the policy used to initialize the monitor, \(\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, S_1 \rangle\), \(\langle \Delta_1, C'_1, M'_1, \langle s'_1, ctx'_1 \rangle, S'_1 \rangle\), \(\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, S_2 \rangle\), and \(\langle \Delta'_2, C'_2, M'_2, \langle s'_2, ctx'_2 \rangle, S'_2 \rangle\) be four global configurations, and \(\ell \in L\) be a label, \(n'\) be the first value in \(S, n = (n' \mod |C_1|) + 1,\) and \(u\) be the user associated with the n-th program in \(C_1\). If the following conditions hold:

1. \(s_1 \equiv s_2\),
2. \(\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, S_1 \rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, S_2 \rangle\),
3. \(\Delta_1(\text{pc}_1) \subseteq \ell\) and \(\Delta_2(\text{pc}_2) \subseteq \ell\),
4. \(\Delta_1(\text{pc}_1) \not\subseteq \ell\) or \(\Delta_2(\text{pc}_2) \not\subseteq \ell\),
5. \(C_1 = C_2\),
6. first\((C_1(n)) = \text{while } e \text{ do } c'\),
7. \(S_1 = S_2\),
8. \(\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1 \rangle, S_1 \rangle \xrightarrow{\omega_1} \langle \Delta'_1, C'_1, M'_1, \langle s'_1, ctx'_1 \rangle, S'_1 \rangle\),
9. \(\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2 \rangle, S_2 \rangle \xrightarrow{\omega_2} \langle \Delta'_2, C'_2, M'_2, \langle s'_2, ctx'_2 \rangle, S'_2 \rangle\),

then one of the following conditions holds:

(a) \[\llbracket e \rrbracket(m_1) = \text{tt} \text{ and for all } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \text{ such that } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \xrightarrow{\omega_1} \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle, c'_1 \neq \varepsilon,\]

(b) \[\llbracket e \rrbracket(m_1) = \text{ff} \text{ and for all } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \text{ such that } \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle \xrightarrow{\omega_1} \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle, c'_1 \neq \varepsilon,\]

(c) \[\llbracket e \rrbracket(m_2) = \text{tt} \text{ and for all } \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle \text{ such that } \langle \Delta'_2, c'_2, m'_2, \langle s'_2, ctx'_2 \rangle \rangle \xrightarrow{\omega_2} \langle \Delta'_1, c'_1, m'_1, \langle s'_1, ctx'_1 \rangle \rangle, c'_2 \neq \varepsilon,\]
D.3. Monitor’s soundness

\[(d) \quad \{c\}(m_2) = \texttt{ff} \quad \text{and for all } \langle \Delta', c', m', (s', ctz') \rangle \quad \text{such that } \langle \Delta', c', m', (s', ctz') \rangle \rightarrow^*_{\mathcal{u}} \langle \Delta'_2, c'_2, m'_2, (s'_2, ctz'_2) \rangle, \]
\[(e) \quad \text{there are } \langle \Delta'_1, c'_1, M'_1, (s'_1, ctz'_1), S'_1 \rangle \quad \text{and } \langle \Delta'_2, C'_2, M'_2, (s'_2, ctz'_2), S'_2 \rangle \quad \text{such that } \langle \Delta'_1, c'_1, M'_1, (s'_1, ctz'_1), S'_1 \rangle \quad \text{and } \langle \Delta'_2, C'_2, M'_2, (s'_2, ctz'_2), S'_2 \rangle \rightarrow^*_{\mathcal{u}} \langle \Delta'_2, C'_2, M'_2, (s'_2, ctz'_2), S'_2 \rangle.\]

Proof. From (6) and the rules F-WhileTrue and F-WhileFalse, it follows that the only applicable rule in the global semantics is F-Atomic-Statement. Our claim directly follows from this, (3), (4), Lemma D.21, and the fact that the F-Atomic-Statement rule does not modify the schedulers. □

D.3.8 Bisimulations

Here we introduce bisimulations for our setting, and we prove some key results about them.

We first introduce some machinery. Let \(\sigma_1 = \langle \Delta_1, C_1, M_1, (s_1, ctz_1), S_1 \rangle\) and \(\sigma_2 = \langle \Delta_2, C_2, M_2, (s_2, ctz_2), S_2 \rangle\) be two global configurations, and \(i \in \mathbb{N}\) be an integer. We say \(\sigma_2\) is reachable in at most \(i\) steps from \(\sigma_1\), denoted \(\text{reach}^i(\sigma_1, \sigma_2)\), if there exists an \(i' \leq i\) such that \(\sigma_1 \rightarrow_{(i')}^* \sigma_2\).

The current program in \(\sigma_1\), denoted \(\text{currProg}(\sigma_1)\), is \(c\) and the current memory \(\text{currMem}(\sigma_1)\) is \(m\), where \(n\) is the first element in \(S_1\), \(n = \langle n' \text{ mod } |C_1| + 1, C_1(n) = (u, c)\rangle\) and \(M(n) = (u, c)\). Furthermore, the current pc in \(\sigma_1\), denoted \(\text{currPC}(\sigma_1)\), is \(\Delta_1(\text{pc}_1)\), where \(\text{pc}_1\) is the first element in \(S_1\), \(\text{pc}_1 = \langle u' \text{ mod } |C_1| + 1, C_1(\text{pc}_1) = (u, c)\rangle\). Finally, the current user in \(\sigma_1\), denoted \(\text{currUser}(\sigma_1)\), is \(u\), where \(\text{currProg}(\sigma_1) = (u, c)\). Given a local configuration and a user \(\sigma, u\), \(\text{term}(\sigma, u) = \top\) if there exists a \(\langle \Delta, c, m, (s, ctz) \rangle\) such that \(\sigma 
abla^N \langle \Delta, c, m, (s, ctz) \rangle\) and \(c = e\). Given a label \(\ell\) and a user \(u\), we denote by \(\text{notBelow}(\sigma, u) = \top\) if for all \(\langle \Delta, c, m, (s, ctz) \rangle\) such that \(\sigma 
abla^N \langle \Delta, c, m, (s, ctz) \rangle\), then \(\text{currPC}(\sigma) \not\subseteq \ell\).

We are now ready to formalize bisimulations.

**Definition D.4.** Let \(\sigma_1 = \langle \Delta_1, C_1, M_1, (s_1, ctz_1), S_1 \rangle\) and \(\sigma_2 = \langle \Delta_2, C_2, M_2, (s_2, ctz_2), S_2 \rangle\) be two global configurations, \(i, j \in \mathbb{N}\) be integers, and \(\ell \in \mathcal{L}\) be a label. Furthermore, let \(R\) be a binary relation over global configurations. We say that \(R\) is a \((\sigma_1, \sigma_2, i, j, \ell)\)-bisimulation iff for all \(\sigma'_1 = \langle \Delta_1', C_1', M_1', (s'_1, ctz'_1), S'_1 \rangle\) and \(\sigma'_2 = \langle \Delta_2', C_2', M_2', (s'_2, ctz'_2), S'_2 \rangle\) such that \(\sigma'_1 R \sigma'_2\), then the following conditions hold:

1. \(\text{reach}^i(\sigma_1, \sigma'_1)\) and \(\text{reach}^j(\sigma_2, \sigma'_2)\).
2. \(\sigma'_1 \equiv_{(i,j)} \sigma'_2\).
3. \(\text{currPC}(\sigma'_1) \subseteq \ell\) and \(\text{currPC}(\sigma'_2) \subseteq \ell\).
4. \(\text{currPC}(\sigma'_1) \not\subseteq \ell \text{ or } \text{currPC}(\sigma'_2) \not\subseteq \ell\).

**Lemma D.25.** Let \(\text{sec}_0\) be the policy used to initialize the monitor, \(\sigma'_0 = \langle \Delta'_0, C'_0, M'_0, (s'_0, ctz'_0), S'_0 \rangle\), \(\sigma'_2 = \langle \Delta'_2, C'_2, M'_2, (s'_2, ctz'_2), S'_2 \rangle\), \(\sigma'_1 = \langle \Delta'_1, C'_1, M'_1, (s'_1, ctz'_1), S'_1 \rangle\), \(\sigma'_2 = \langle \Delta'_2, C'_2, M'_2, (s'_2, ctz'_2), S'_2 \rangle\) be four global configurations, \(\tau_1, \tau_2\) be two traces, \(u\) be a user, and \(\ell \in \mathcal{L}\) be a label. If the following conditions hold:

1. \(\sigma'_0 \rightarrow_{(0,0)} \sigma'_1\).
2. \(\sigma'_0 \rightarrow_{(0,0)} \sigma'_2\).
3. \(\sigma'_1 \equiv_{(0,0)} \sigma'_2\).
4. \(\text{currPC}(\sigma'_1) \not\subseteq \ell \text{ or } \text{currPC}(\sigma'_2) \not\subseteq \ell\).
5. \(\text{currPC}(\sigma'_1) \subseteq \ell\) and \(\text{currPC}(\sigma'_2) \subseteq \ell\).
6. \(\text{auth}(\text{sec}_0, ATK) \subseteq \ell\).

Lemmas D.25 and D.26 state that, under certain conditions, we can construct bisimulations.
then $\{(\sigma_0^1, \sigma_0^2), (\sigma_1^1, \sigma_1^2)\}$ is a $(\sigma_0^1, \sigma_0^2, 1, 1, \ell)$-bisimulation.

Proof. Let $sec_0$ be the policy used to initialize the monitor, $\sigma_0^1 = (\Delta_0^1, C_0^1, M_0^1, (s_0^1, ctx_0^1), S_0^1)$, $\sigma_0^2 = (\Delta_0^2, C_0^2, M_0^2, (s_0^2, ctx_0^2), S_0^2)$, $\sigma_1^1 = (\Delta_1^1, C_1^1, M_1^1, (s_1^1, ctx_1^1), S_1^1)$, $\sigma_1^2 = (\Delta_1^2, C_1^2, M_1^2, (s_1^2, ctx_2^2), S_1^2)$ be four global configurations, $\tau_1, \tau_2$ be two traces, $u$ be a user, and $\ell \in \mathcal{L}$ be a label. Furthermore, we assume that following conditions hold:

1. $\sigma_0^1 \xrightarrow{\Delta_0^1} \sigma_1^1$,
2. $\sigma_0^2 \xrightarrow{\Delta_0^2} \sigma_2^2$,
3. $\sigma_0^1 \sim^\Delta \sigma_0^2$,
4. $\sigma_0^1 \equiv^\ell \sigma_0^2$,
5. $C_0^1 = C_0^2$,
6. $S_0^1 = S_0^2$,
7. $currPc(\sigma_0^1) \subseteq \ell$ and $currPc(\sigma_0^2) \subseteq \ell$,
8. $currPc(\sigma_1^1) \subseteq \ell$ and $currPc(\sigma_1^2) \subseteq \ell$,
9. $cl(auth(sec_0, ATK)) \subseteq \ell$.

We now show that $\{(\sigma_0^1, \sigma_0^2), (\sigma_1^1, \sigma_1^2)\}$ is a $(\sigma_0^1, \sigma_0^2, 1, 1, \ell)$-bisimulation. We first need to show that for all $\sigma_1^1 = (\Delta_1^1, C_1^1, M_1^1, (s_1^1, ctx_1^1), S_1^1)$ and $\sigma_2^2 = (\Delta_2^2, C_2^2, M_2^2, (s_2^2, ctx_2^2), S_2^2)$ such that $\sigma_1^1 \mathcal{R} \sigma_2^2$, the following conditions hold: (a) $reach^1(\sigma_1^1, \sigma_1^2)$ and $reach^1(\sigma_2^2, \sigma_2^1)$, (b) $\sigma_0^1 \approx \sigma_0^2$, (c) $\sigma_1^1 \equiv^\ell \sigma_1^2$, (d) $C_1^1 = C_2^2$, (e) $S_1^1 = S_2^2$, (f) $currPc(\sigma_1^1) \subseteq \ell$ and $currPc(\sigma_2^2) \subseteq \ell$. There are two cases:

- (1) is obtained by applying the M-EVAL-END rule. From this and (5), it follows that also (2) is obtained using the M-EVAL-END rule. From this, (5), and the rule, we eliminate in both run the same components. From this and (3)–(7), it directly follows that (b)–(f) are satisfied.

- (1) is obtained by applying the M-EVAL-STEP or M-Atomic-Statement rules. From this, (5), and (6), it follows that we perform one step of the local semantics for the same program in both runs. From this and (6), (e) directly follows. From (3)–(10), Lemmas D.17, D.18, and D.19, conditions (b)–(d) follow. Finally, condition (f) immediately follows from (8).

Therefore, the fact that a bisimulation directly follows from (i) the fact that (a)–(f) hold for $(\sigma_0^1, \sigma_0^2)$ and $(\sigma_1^1, \sigma_1^2)$, (ii) assumptions (1), (2), and $R = \{(\sigma_0^1, \sigma_0^2), (\sigma_1^1, \sigma_1^2)\}$, and (3) there are no configurations that are reachable in 1 step from $\sigma_0^1$ and $\sigma_0^2$ other than $\sigma_1^1$ and $\sigma_1^2$.

Lemma D.26. Let $sec_0$ be the policy used to initialize the monitor, $\sigma_0^1 = (\Delta_0^1, C_0^1, M_0^1, (s_0^1, ctx_0^1), S_0^1)$, $\sigma_0^2 = (\Delta_0^2, C_0^2, M_0^2, (s_0^2, ctx_0^2), S_0^2)$, $\sigma_1^1 = (\Delta_1^1, C_1^1, M_1^1, (s_1^1, ctx_1^1), S_1^1)$, $\sigma_1^2 = (\Delta_1^2, C_1^2, M_1^2, (s_1^2, ctx_2^2), S_1^2)$ be four global configurations, $\tau_1, \tau_2$ be two traces, $u = currUsr(\sigma_1^1)$ be a user, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:

1. $\sigma_0^1 \xrightarrow{\Delta_0^1} \sigma_1^1$,
2. $\sigma_0^2 \xrightarrow{\Delta_0^2} \sigma_2^2$,
3. $\sigma_0^1 \sim^\Delta \sigma_0^2$,
4. $\sigma_0^1 \equiv^\ell \sigma_0^2$,
5. $C_0^1 = C_0^2$,
6. $S_0^1 = S_0^2$,
7. $currPc(\sigma_0^1) \subseteq \ell$ and $currPc(\sigma_0^2) \subseteq \ell$,
8. $currPc(\sigma_1^1) \subseteq \ell$ or $currPc(\sigma_1^2) \subseteq \ell$,
9. $cl(auth(sec_0, ATK)) \subseteq \ell$,
10. for all users $u' \neq currUsr(\sigma_0^1)$, $\Delta_0^0(\text{pc}_{u'}) \subseteq \ell$ and $\Delta_0^0(\text{pc}_{u'}) \subseteq \ell$,
11. whenever first(currPrg(\sigma_0^1)) = if $e$ then $c'$ else $c''$ and $[e](M_0^1) = \top$, then term([$\Delta_1^1, c'$, currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle]), u = T and notBelow([$\Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle], \ell, u) = T,
12. whenever first(currPrg(\sigma_0^2)) = if $e$ then $c'$ else $c''$ and $[e](M_0^2) = \top$, then term([$\Delta_1^2, c'$, currMem(\sigma_1^2), \langle s_1^2, ctx_2^2 \rangle]), u = T and notBelow([$\Delta_1^2, c', currMem(\sigma_1^2), \langle s_1^2, ctx_2^2 \rangle], \ell, u) = T,
13. whenever first(currPrg(\sigma_0^1)) = if $e$ then $c'$ else $c''$ and $[e](M_0^1) = \top$, then term([$\Delta_1^1, c'$, currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle]), u = T and notBelow([$\Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle], \ell, u) = T,
14. whenever first(currPrg(\sigma_0^2)) = if $e$ then $c'$ else $c''$ and $[e](M_0^2) = \top$, then term([$\Delta_1^2, c'$, currMem(\sigma_1^2), \langle s_1^2, ctx_2^2 \rangle]), u = T and notBelow([$\Delta_1^2, c', currMem(\sigma_1^2), \langle s_1^2, ctx_2^2 \rangle], \ell, u) = T,
15. whenever first(currPrg(\sigma_0^1)) = while $u$ do $c'$ and $[e](M_0^1) = \top$, then term([$\Delta_1^1, c'$, while $u$ do $c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle], u = T and notBelow([$\Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle], \ell, u) = T,
16. whenever first(currPrg(\sigma_0^2)) = while $u$ do $c'$ and $[e](M_0^2) = \top$, then term([$\Delta_1^2, c'$, while $u$ do $c', currMem(\sigma_1^2), \langle s_1^2, ctx_2^2 \rangle], u = T and notBelow([$\Delta_1^2, c', currMem(\sigma_1^2), \langle s_1^2, ctx_2^2 \rangle], \ell, u) = T,
17. whenever first(currPrg(\sigma_0^1)) = set $pc$ to $\ell'$, $\ell' \subseteq currPc(\sigma_0^1)$,
then there are $i,j$ such that $\sigma^0_1 \xrightarrow{\ell} \sigma^0_1$, $\sigma^0_2 \xrightarrow{\ell} \sigma^0_2$, and $\{(\sigma^0_1, \sigma^0_2), (\sigma^0_1, \sigma^0_2)\}$ is a $(\sigma^0_1, \sigma^0_2, i,j, \ell)$-bisimulation.

Proof. Let $sec_0$ be the policy used to initialize the monitor, $\sigma^0_1 = (\Delta_1^0, C_1^0, M_1^0, (s_1^0, ctx_{1}^0), S_1^0)$, $\sigma^0_2 = (\Delta_2^0, C_2^0, M_2^0, (s_2^0, ctx_{2}^0), S_2^0)$, $\sigma_1 = (\Delta_1, C_1, M_1, (s_1, ctx_1), S_1)$, $\sigma_2 = (\Delta_2, C_2, M_2, (s_2, ctx_2), S_2)$ be four global configurations, $\tau_1, \tau_2$ be two traces, $u = curUr\sigma(\sigma_1)$ be a user, and $\ell \in L$ be a label.

Furthermore, we assume the following conditions hold:

1. $\sigma^0 \xrightarrow{\ell} \sigma^1$,
2. $\sigma^0 \xrightarrow{\ell} \sigma^1$,
3. $\sigma^0 \equiv_{fs} \sigma^0$,
4. $C_1 = C_2$,
5. $S_1 = S_2$,
6. currPc($\sigma^0_1$) $\subseteq \ell$ and currPc($\sigma^0_2$) $\subseteq \ell$,
7. currPc($\sigma_1$) $\not\subseteq \ell$ or currPc($\sigma_2$) $\not\subseteq \ell$,
8. $\chi(|auth|, sec_0, ATK) \not\subseteq \ell$,
9. for all users $u \neq curUr\sigma(\sigma_1)$, $\Delta_1^i(pc_{u}) \not\subseteq \ell$ and $\Delta_2^i(pc_{u}) \not\subseteq \ell$.

11. whenever first(currPrg($\sigma^0_1$)) = if e then c' else c'' and $\|\| M^0_1 \|\| = tt$, then term(\(\Delta_1, c', currMem(\sigma^0_1), (s_1, ctx_1))$, $u = T$ and notBelow($\Delta_1, c', currMem(\sigma^0_1), (s_1, ctx_1))$, $u = T$),
12. whenever first(currPrg($\sigma^0_1$)) = if e then c' else c'' and $\|\| M^0_1 \|\| = ff$, then term(\(\Delta_1, c'', currMem(\sigma^0_1), (s_1, ctx_1))$, $u = T$ and notBelow($\Delta_1, c'', currMem(\sigma^0_1), (s_1, ctx_1))$, $u = T$),
13. whenever first(currPrg($\sigma^0_2$)) = if e then c' else c'' and $\|\| M^0_2 \|\| = tt$, then term(\(\Delta_2, c', currMem(\sigma^0_2), (s_2, ctx_2))$, $u = T$ and notBelow($\Delta_2, c', currMem(\sigma^0_2), (s_2, ctx_2))$, $u = T$),
14. whenever first(currPrg($\sigma^0_2$)) = if e then c' else c'' and $\|\| M^0_2 \|\| = ff$, then term(\(\Delta_2, c'', currMem(\sigma^0_2), (s_2, ctx_2))$, $u = T$ and notBelow($\Delta_2, c'', currMem(\sigma^0_2), (s_2, ctx_2))$, $u = T$),
15. whenever first(currPrg($\sigma^0_1$)) = while do c' and $\|\| M^0_1 \|\| = tt$, then term(\(\Delta_1, c', while do c', currMem(\sigma^0_1), (s_1, ctx_1))$, $u = T$ and notBelow($\Delta_1, c', while do c', currMem(\sigma^0_1), (s_1, ctx_1))$, $u = T$),
16. whenever first(currPrg($\sigma^0_2$)) = while do c' and $\|\| M^0_2 \|\| = tt$, then term(\(\Delta_2, c', while do c', currMem(\sigma^0_2), (s_2, ctx_2))$, $u = T$ and notBelow($\Delta_2, c', while do c', currMem(\sigma^0_2), (s_2, ctx_2))$, $u = T$),
17. whenever first(currPrg($\sigma^0_1$)) = set pc to c', $\ell' \subseteq \ell$.

Let $c_1 = currPrg(\sigma^0_1)$, $u_1 = curUr\sigma(\sigma^0_1)$, $c_2 = currPrg(\sigma^0_2)$, and $u_2 = curUr\sigma(\sigma^0_2)$. From (5) and (6), it follows that $(u_1, c_1) = (u_2, c_2)$. In the following, we denote $u_1$ and $u_2$ using $u$ and $c_1$ and $c_2$ using $c$. Furthermore, we denote by $n$ the value such that $C^0_n(u) = \langle curUr\sigma(\sigma^0_1), currPrg(\sigma^0_1) \rangle$.

From Lemma D.12 and (17), there are only two cases:

• first($c_1$) = if e then c' else c''. We assume that $c = if e then c' else c''c_3$, which $c_3$ can be an empty program (the proof for the other cases is almost identical). From this and the F-IfTrue and F-IfFalse rules, it follows that $currPrg(\sigma) = [c'_3; set pc to \(\Delta_1^0(pc_{u})\);]c_3$ and $currPrg(\sigma') = [c'_3; set pc to \(\Delta_2^0(pc_{u})\);]c_3$, where $c'_3 \in \{c', c''\}$, and $c'_3 \in \{c', c''\}$. Furthermore, from (3) and (7), it follows that $\Delta_1^0(pc_{u}) = \Delta_2^0(pc_{u}) = \ell'$. Therefore, $currPrg(\sigma) = [c'_3; set pc to \ell'c_3]$, and $currPrg(\sigma') = [c'_3; set pc to \ell'c_3]$, where $c'_3 \in \{c', c''\}$, and $c'_3 \in \{c', c''\}$. From this and (11)–(14), it follows that there are $\Delta_1, [c' \in \{c', c''\}]$, and $\Delta_2, [c' \in \{c', c''\}]$, where $\Delta_1^i(pc_{u}) = \ell'$ and $\Delta_2^i(pc_{u}) = \ell'$. From (11)–(14), it follows that during both computations $pc_{c}$ is never below $\ell$ before executing the last set pc to $\ell'$ statement. From this and (8), it follows that $\Delta_1^i(pc_{u}) \not\subseteq \ell$ and $\Delta_2^i(pc_{u}) \not\subseteq \ell$ for all $1 \leq i' \leq i - 1$ and $1 \leq j' \leq j - 1$. By repeatedly applying Lemma D.13 and Lemma D.14 and $\Delta_1^i(pc_{u}) \not\subseteq \ell$ and $\Delta_2^i(pc_{u}) \not\subseteq \ell$ for all $1 \leq i' \leq i - 1$ and $1 \leq j' \leq j - 1$, we obtain that $\Delta_1^i(c^1, m^1, (s^1, ctx^1)) \approx_{\ell} (\Delta_2^i, c^2, m^2, (s^2, ctx^2))$ for all $1 \leq i' \leq i - 1$ and $1 \leq j' \leq j - 1$. From this and the fact that in the last step of the execution we set pc to $\ell'$ in both runs, $\Delta_1^i(c^1, m^1, (s^1, ctx^1)) \approx_{\ell} (\Delta_2^i, c^2, m^2, (s^2, ctx^2))$. Similarly, by repeatedly applying Lemma D.15, we obtain that $\Delta_1^i(c^1, m^1, (s^1, ctx^1)) \equiv_{fs} (\Delta_2^i, c^2, m^2, (s^2, ctx^2))$. Subsequently, by repeatedly applying the F-Atomic-Statement rule both to $\sigma_0$ and $\sigma_0'$, we obtain that $\sigma^0_1 \xrightarrow{\ell} (\Delta_1, C_1, M_1, (s_1, ctx_1), S_1)$ and $\sigma^0_2 \xrightarrow{\ell} (\Delta_2, C_2, M_2, (s_2, ctx_2), S_2)$, where $C_1 = C(1) \cdot \cdots \cdot C(1(n - 1))$ and $C_2 = C(1) \cdot \cdots \cdot C(1(n - 1))$, and $\sigma^0_1 \xrightarrow{\ell} (\Delta_1, C_1, M_1, (s_1, ctx_1), S_1)$ and $\sigma^0_2 \xrightarrow{\ell} (\Delta_2, C_2, M_2, (s_2, ctx_2), S_2)$. We now show that $R = \{(\sigma_0^1, \sigma_0^2), (\sigma^0_1, \sigma^0_2)\}$ is a $(\sigma_0^1, \sigma_0^2, i,j, \ell)$-bisimulation. We first need to show that for all $\sigma_0^1 = (\Delta_1, C_1, M_1, (s_1, ctx_1), S_1$ and $\sigma_0^2 = (\Delta_2, C_2, M_2, (s_2, ctx_2), S_2)$ that such $\sigma_0^1R\sigma_0^2$, the following conditions hold:
(a) $reach^i(σ^0_1, σ^i_1)$ and $reach^i(σ^0_2, σ^i_2)$, (b) $σ^0_i ≈_{fs} σ^i_2$, (c) $σ^0_2 ≈_{fs} σ^i_2$, (d) $C_i = C^i_2$, (e) $S_0 = S^0_2$, (f) $currPc(σ^0_i) ⊆ ℓ$ and $currPc(σ^i_2) ⊆ ℓ$. There are two cases:

- $σ^0_2 = (σ^i_1, σ^i_2)$. Then, (a) trivially follows since $reach^k(σ, a)$ always holds for all $k > 0$ and both $i, j > 0$. Moreover, (b)–(f) directly follow from (3)–(7).

- $(σ^0_1, σ^1_2) = (σ^i_1, σ^i_2)$. Then, (a) directly follows from $σ^0_1 ∪ σ^i_1$ and $σ^0_2 ∪ σ^i_2$. Condition (b) follows from (3), $(Δ^i_2, c^i_2, m^i_2, (σ^i_1, ctx^i_2))$ if $(Δ^i_2, c^i_2, m^i_2, (σ^i_1, ctx^i_2))$, and $Δ^i_1(pec^i_0) = Δ^i_2(pec^i_0) = ℓ$. Condition (c) follows from $(Δ^i_1, c^i_1, m^i_1, (σ^i_1, ctx^i_1))$ if $(Δ^i_1, c^i_1, m^i_1, (σ^i_1, ctx^i_1)) = Δ^i_1(pec^i_0) = Δ^i_2(pec^i_0)$.

Therefore, the fact that $R$ is a bisimulation directly follows from (i) the fact that (a)–(f) hold for $(σ^0_2, σ^i_2)$ and $(σ^i_1, σ^i_2)$, (ii) assumptions (1), (2), and $(σ^0_1, σ^i_0), (σ^i_1, σ^i_2)$, and (iii) there are no configurations that are reachable in $i$ steps from $σ^0_0$ and $j$ steps from $σ^i_0$ other than $σ^0_2$ and $σ^i_2$.

- $first(c^i_1) = \textbf{while } c \textbf{ do } c'$. The proof of this case is similar to that of $first(c^i_1) = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. This completes the proof of our claim.

Finally, Lemma D.27 states a composition result for bisimulations.

**Lemma D.27.** Let $R_1$ be a $(σ^0_0, σ^1_0, i, j, ℓ)$-bisimulation and $R_2$ be a $(ρ^0_0, ρ^i_0, x, y, ℓ')$-bisimulation. If the following conditions hold:

1. $(σ^0_0, σ^1_0) ∈ R_1$,
2. $(ρ^0_0, ρ^i_0) ∈ R_1 ∩ R_2$,
3. $ℓ = ℓ'$,
4. $σ^0_0 ≈_{fs} ρ^0_0$, and
5. $σ^1_0 ≈_{fs} ρ^i_0$,

then $R_1 ∪ R_2$ is a $(σ^0_0,σ^1_0, x + j, y + ℓ)$-bisimulation.

**Proof.** Let $R_1$ be a $(σ^0_0, σ^1_0, i, j, ℓ)$-bisimulation and $R_2$ be a $(ρ^0_0, ρ^i_0, x, y, ℓ')$-bisimulation. Assume, for contradiction’s sake, that $R_1 ∪ R_2$ is not a $(σ^0_0, σ^1_0, i + x, j + y, ℓ)$-bisimulation. This happens iff there is a $(ν_0, ν_1) ∈ R_1 ∪ R_2$ that violates one of the following constraints:

1. $¬reach^{i + x}(σ^0_0, ν_0)$ or $¬reach^{i + y}(σ^1_0, ν_1')$. Without loss of generality, we assume that $reach^{i + x}(σ^0_0, ν_0)$ does not hold. If $(ν_0, ν_1) ∈ R_1$, then $reach^{i + x}(σ^0_0, ν_0)$ and $reach^{i + x}(σ^0_0, ν_0)$ follow,

leading to a contradiction. If $(ν_0, ν_1) ∈ R_2 \setminus R_1$, then from $σ^0_0 ≈_{fs} ρ^0_0$ and $reach^{i + x}(ρ^0_0, ν_0)$, it follows that $reach^{i + x}(σ^0_0, ν_0)$, leading to a contradiction.

2. $ν_0 \neq_ℓ ν_1$. This contradicts $(ν_0, ν_1) ∈ R_1$ or $(ν_0, ν_1) ∈ R_2$.

3. $ν_0 \neq_ℓ ν_1$. This contradicts $(ν_0, ν_1) ∈ R_1$ or $(ν_0, ν_1) ∈ R_2$.

4. $C_0 \neq C_1$, where $C_0$ is the code in $ν_0$ and $C_1$ is the code in $ν_1$. This contradicts $(ν_0, ν_1) ∈ R_1$ or $(ν_0, ν_1) ∈ R_2$.

5. $S_0 \neq S_1$, where $S_0$ is the scheduler in $ν_0$ and $S_1$ is the scheduler in $ν_1$. This contradicts $(ν_0, ν_1) ∈ R_1$ or $(ν_0, ν_1) ∈ R_2$.

6. $currPc(ν_0) \not{∈} ℓ$ or $currPc(ν_1) \not{∈} ℓ$. This contradicts $(ν_0, ν_1) ∈ R_1$ or $(ν_0, ν_1) ∈ R_2$.

7. $ν_0 \approx_{can} ν_0, ν_1 \approx_{can} ν_1, reach^{i + x}(σ^0_0, ν_0), reach^{i + y}(σ^1_0, ν_1), and currPc(ν_0) \subseteq ℓ$, but $(ν_0, ν_1) \not{∈} R_1 \cup R_2$. There are three cases:

- $(ν_0, ν_1) ∈ R_0, reach^{i}(σ^0_0, ν_0)$, and $reach^{i}(σ^1_0, ν_1)$. From this and $R_1$ is a $(σ^0_0, σ^1_0, i, j, ℓ)$-bisimulation, it follows that $(ν_0, ν_1) ∈ R_1$. Hence, $(ν_0, ν_1) ∈ R_1 ∪ R_2$, leading to a contradiction.

- $(ν_0, ν_1) ∈ R_1, reach^{i}(σ^0_0, ν_0)$, and $reach^{i}(ρ^i_0, ν_1)$. From this and $R_2$ is a $(ρ^i_0, ρ^i_0, x, y, ℓ)$-bisimulation, it follows that $(ν_0, ν_1) ∈ R_2$. Hence, $(ν_0, ν_1) ∈ R_1 ∪ R_2$, leading to a contradiction.

- $(ν_0, ν_1) ∈ R_0, reach^{i + x}(σ^0_0, ν_0), reach^{i + y}(σ^1_0, ν_1)$, or $¬reach^{i}(σ^0_0, ν_0)$). This happens iff $(ν_0, ν_1) = (ρ_0, ρ_1)$. From this, $(ρ_0, ρ_1) ∈ R_1 ∩ R_2, σ^0_0 \approx_{fs} ρ_0, σ^1_0 \approx_{fs} ρ_1, reach^{i + x}(σ^0_0, ν_0)$, and $reach^{i + y}(σ^1_0, ν_1)$, it follows that $reach^{i + x}(ρ_0, ν_0)$ and $reach^{i + y}(ρ_1, ν_1)$. From this and $R_2$ is a $(ρ^0_0, ρ^i_0, x, y, ℓ)$-bisimulation, it follows that $(ν_0, ν_1) ∈ R_2$. Hence, $(ν_0, ν_1) ∈ R_1 ∪ R_2$, leading to a contradiction.

8. $ν_0 \approx_{can} ν_0, ν_1 \approx_{can} ν_1, reach^{i + x}(σ^0_0, ν_0)$, reach^{i + y}(σ^1_0, ν_1), $currPc(ν_0) \not{∈} ℓ$, and there are $ν''_0$ and $ν''_1$ such that $ν_0 \approx_{can} ν''_0, ν_1 \approx_{can} ν''_1, reach^{i + x}(σ^0_0, ν''_0)$, reach^{i + y}(σ^1_0, ν''_1), $currPc(ν''_0) \not{∈} ℓ$, and $(ν_0'', ν_1') \not{∈} R_1 ∪ R_2$. If $reach^{i + x}(σ^0_0, ν''_0)$ and $reach^{i + y}(σ^1_0, ν''_1)$, then $(ν''_0, ν''_1) ∈ R_1$ and, therefore, $(ν_0'', ν_1') ∈ R_1 ∪ R_2$, leading to a contradiction. If $reach^{i + x}(ρ''_0, ν''_0)$ and $reach^{i + y}(ρ''_1, ν''_1)$, then $(ν''_0, ν''_1) ∈ R_2$ and, therefore, $(ν_0'', ν_1') ∈ R_1 ∪ R_2$, leading to a contradiction. Note that from $(ρ_0, ρ_1) ∈ R_1 ∩ R_2$, it follows that the above cases are the only possible. Since all cases lead to a contradiction, this completes the proof of our claim.

\qed
D.3. Monitor’s soundness

We are now ready to prove the main result of Section 7.6, namely that our mechanism provides security with respect to an external attacker ATK.

**Theorem D.7.** For all programs $C = c_1, \ldots, c_k \in Com^L_{BID}$, scheduler $S$, memories $M = m_1, \ldots, m_k \in Mem^L_{BID}$, and initial runtime state $s$, whenever $\tau = (\Delta_0, C, (s, \epsilon), S) \not\in S \not\sim_{\Delta, C', M', (s', c'tx')} S'$, then for all $1 \leq i \leq n$, $K_{ATK}((M, s), C, S, \text{trace}(r^{-1})) \cap A_{ATK, sec}(M, s) \subseteq K_{ATK}((M, s), C, S, \text{trace}(r'))$, where the database in $\tau$’s $(i-1)$-th configuration is $\langle d, u, sec, T, V \rangle$ and $K_{ATK}$ refers to Definition 7.3 with \( \sim \) as the underlying evaluation relation.

Proof. Let $k \in \mathbb{N}$, $C_0 = c_1, \ldots, c_k \in Com^L_{BID}$ be WHILESQL programs, $S_0$ be a scheduler, $M_0 = m_1, \ldots, m_k \in Mem^L_{BID}$ be memories, $s_0$ be a database state, $s_0$ be the global state $(\Delta_0, C_0, M_0, (s_0, \epsilon), S_0)$. Let $\sigma_1$ be a global state and $n$ be a value in $\mathbb{N}$ such that $\sigma = \sigma_0 \stackrel{\ell}{\rightarrow} \sigma_1$. Assume, for contradiction’s sake, that our claim does not hold. Namely, there exists a value $1 \leq i \leq n$ such that $K_{ATK}((M_0, s_0), C_0, S_0, \text{trace}(r^{-1})) \cap A_{ATK, sec}(M_0, s_0) \not\subseteq K_{ATK}((M_0, s_0), C_0, S_0, \text{trace}(r'))$, where sec is the security policy in the $(i-1)$-th configuration in $\tau$. From (4), it follows that there is a state $(M_1, s_1)$ such that $(M_1, s_1) \not\in K_{ATK}((M_0, s_0), C_0, S_0, \text{trace}(r^{-1})) \cap A_{ATK, sec}(M_0, s_0)$, and $(M_1, s_1) \not\in K_{ATK}((M_0, s_0), C_0, S_0, \text{trace}(r'))$. From $(M_1, s_1) \in K_{ATK}((M_0, s_0), C_0, S_0, \text{trace}(r^{-1}))$, it follows that $\sigma_0 \approx_{\Delta, \tau', \Delta', C, M, (s', c'tx'), S'} S' \approx_{\Delta, C', \Delta', (s', c'tx'), S'} \text{trace}(r')$. From $(M_1, s_1) \in A_{ATK, sec}(M_0, s_0)$, it follows that $\sigma_0 \approx_{\Delta, \tau', \Delta', C, M, (s', c'tx'), S'} S' \approx_{\Delta, C', \Delta', (s', c'tx'), S'} \text{trace}(r')$. Note that only the last case is interesting (since $\sigma_0 \approx_{\Delta, \tau', \Delta', C, M, (s', c'tx'), S'} S' \approx_{\Delta, C', \Delta', (s', c'tx'), S'} \text{trace}(r')$). Therefore, the following conditions hold:

1. $\sigma_0 \approx_{\Delta, \tau', \Delta', C, M, (s', c'tx'), S'} S'$. From (4), it follows that the only interesting cases are those for which $\text{trace}(r') \not\approx_{\Delta, C', \Delta', (s', c'tx'), S'} S'$. From (4), it follows that the only interesting cases are those for which $\text{trace}(r') \not\approx_{\Delta, C', \Delta', (s', c'tx'), S'} S'$. From (4), it follows that the only interesting cases are those for which $\text{trace}(r') \not\approx_{\Delta, C', \Delta', (s', c'tx'), S'} S'$. From (4), it follows that the only interesting cases are those for which $\text{trace}(r') \not\approx_{\Delta, C', \Delta', (s', c'tx'), S'} S'$.
definition, it follows that $\Delta_0(q) = cl(q)$ for all $q \in RC_{prop}$. From this and $L_C(\Delta_0, q) = \bigcup_{Q \in supp_D(q)} \bigcup_{q' \in Q} \Delta_0(q')$, it follows that $L_C(\Delta_0, q) = \bigcup_{Q \in supp_D(q)} \bigcup_{q' \in Q} cl(q')$. From this and $L_C(\Delta_0, q) \subseteq cl(\text{auth(secc, ATK) or auth(sec, ATK)})$, it follows $\bigcup_{Q \in supp_D(q)} \bigcup_{q' \in Q} cl(q') \subseteq cl(\text{auth(secc, ATK) or auth(sec, ATK)})$. Furthermore, from the definition of $supp_D(q)$, it follows that $cl(q) \subseteq \bigcup_{Q \in supp_D(q)} \bigcup_{q' \in Q} cl(q')$. Therefore, the tables and views in $\text{auth(secc, ATK) or auth(sec, ATK)}$ determine the values of the queries in $supp_D(q)$, which in turn determine the value of $q$, i.e., $cl(q) \subseteq cl(\text{auth(secc, ATK) or auth(sec, ATK)})$. From $s_1 \approx_{ATK} s_0$, it follows that the content of all tables and views in $\text{auth(secc, ATK)}$ is the same in $s_0$ and $s_1$. From $s_1 \approx_{sec, ATK} s_0$, it follows that the content of all tables and views in $\text{auth(sec, ATK)}$ is the same in $s_0$ and $s_1$.

From this and $cl(q) \subseteq cl(\text{auth(secc, ATK) or auth(sec, ATK)})$, it follows that $[q]_{\text{obs}} = [q]_{\text{obs}}$

- For all variables $x \in Vars$ such that $\Delta_0(x) \subseteq \ell$, $[M_o](x) = [M_1](x)$. From $\ell = cl(\text{auth(secc, ATK) or auth(sec, ATK)} \cup \bigcup_{x \in \text{Vars}_{ATK}} \text{MEM}_x)$, it follows that $\Delta_0(x) \subseteq cl(\text{auth(secc, ATK) or auth(sec, ATK)} \cup \bigcup_{x \in \text{Vars}_{ATK}} \text{MEM}_x)$. From this and $\Delta_0(x) = \top$ if $x \notin \text{Vars}_{ATK}$ and $\Delta_0(x) = \text{MEM}_x$ otherwise, it follows that $\Delta_0(x) = \text{MEM}_x$. From this, it follows that $x \in \text{Vars}_{ATK}$.

From this and $s_1 \approx_{ATK} s_0$, it follows that $[M_o](x) = [M_1](x)$.

These facts together with the fact that the monitor state is in the same in $s_0$ and $s_1$, leads to $s_0^0 \approx s_1^0$.

**Fact 2.** We now prove that $s_0 \equiv_n^{|n|} s_1$. Let $s_0 = (\text{db}_0, U_0, S_0, T_0, V_0)$ and $s_1 = (\text{db}_1, U_1, S_1, T_1, V_1)$. From $s_1 \approx_{ATK} s_0$ and $s_1 \approx_{ATK}$'s definition, it follows that $U_0 = U_1$, $S_0 = S_1$, $T_0 = T_1$, and $V_0 = V_1$. From this and $s_1 \approx_{ATK}$'s definition, it follows that $s_0 \equiv_n^{|n|} s_1$.

**Fact 3.** We now show that $\text{currPc}(\sigma_0^{-1}) \subseteq \ell$ and $L_d(s_0^{-1}, ATK) \subseteq \ell$. From $\ell = cl(\text{auth(secc, ATK) or auth(sec, ATK)} \cup \text{auth(secc, ATK)} \cup \text{auth(sec, ATK)} \cup \bigcup_{x \in \text{Vars}_{ATK}} \text{MEM}_x)$ and $sec$ is the policy in $\sigma_0^{-1}$, it immediately follows that $L_d(s_0^{-1}, ATK) \subseteq \ell$. Assume, for contradiction's sake, that $\text{currPc}(\sigma_0^{-1}) \not\subseteq \ell$. From $\sigma_0^{-1} \approx_{\text{obs}} \sigma_0^0$ and $\text{obs}_0 \not\in \ell$, it follows that the executed rule produced an event. In the following, let $\text{currUsr}(\sigma_0^{-1}) = u$.

Observe that $\Delta_0^{-1}(pc_u) = cl(\text{currPc}(\sigma_0^{-1}))$. There are two cases:

- Rule F-EVAL-STEP. From the rule, it follows that $\Delta_0^{-1}(\sigma_0^{0}, m_0, (s_0^{-1}, ctx_0^{-1})) \approx_{\text{obs}} F-Eval\text{-Step}$. From this and Lemma D.11, it follows that $\Delta_0^{-1}(\text{obs}(\text{obs}_0, r)) \cup \Delta_0^{-1}(\text{pc}_u) \subseteq L_d(s_0^{-1}, \text{user}(\text{obs}_0))$. From this, it follows that $\Delta_0^{-1}(\text{pc}_u) \subseteq L_d(s_0^{-1}, \text{user}(\text{obs}_0))$. Furthermore, since $\text{trace}(r')|_{\text{ATK}} = \text{trace}(r'|_{\text{ATK}})$, it follows that $\text{user}(\text{obs}_0) = \text{ATK}$. From this and $\Delta_0^{-1}(\text{pc}_u) \subseteq L_d(s_0^{-1}, \text{user}(\text{obs}_0))$, it follows that $\Delta_0^{-1}(\text{pc}_u) \subseteq L_d(s_0^{-1}, \text{ATK})$. From this and $L_d(s_0^{-1}, \text{ATK}) \subseteq \ell$, it follows that $\Delta_0^{-1}(\text{pc}_u) \subseteq \ell$, leading to a contradiction.

- Rule F-ATOMIC-STATEMENT. The proof is similar to that of F-EVAL-STEP. Since all cases lead to a contradiction, this completes the proof of Fact 3.

**Fact 4.** We prove that $\sigma_0^{-1} \equiv_n^{|n|} \sigma_1^{-1}$. From $\sigma_0^{0} \approx_n^{|n|} \sigma_0^{-1} \approx_{\text{obs}} \sigma_0^0$, $\sigma_0^{-1} \approx_{\text{obs}} \sigma_1^{-1} \approx_{\text{obs}} \sigma_1^0$, $\tau_0|_{\text{ATK}} = \tau_1|_{\text{ATK}}$, and all configuration changes are associated with public events, and the code produced by the expansion process either terminates or gets stuck, it follows that the configuration has been modified in the same way in $\sigma_0^{-1} \approx_{\text{obs}} \sigma_0^0$ and $\sigma_1^{-1} \approx_{\text{obs}} \sigma_1^0$. Therefore, $\sigma_0^{-1} \equiv_n^{|n|} \sigma_1^{-1}$.

**Fact 5.** We now prove that $\text{currPc}(\sigma_1^{-1}) \subseteq \ell$. From $\sigma_1^{-1} \approx_{\text{obs}} \sigma_1^0$, there are two cases:

1. We applied the F-OUT rule. From the rule, it follows that $\text{currPc}(\sigma_1^{-1}) \subseteq L_d(s_1^{-1}, u')$. Furthermore, since $\text{obs}_1$ is visible to $\text{ATK}$, it follows that $u' = \text{ATK}$. Hence, $\text{currPc}(\sigma_1^{-1}) \subseteq L_d(s_1^{-1}, \text{ATK})$. From this and $\sigma_1^{-1} \approx_{\text{obs}} \sigma_1^0$ (Fact 4), it follows that $\text{currPc}(\sigma_1^{-1}) \subseteq L_d(s_0^{-1}, \text{ATK})$. From this and $L_d(s_0^{-1}, \text{ATK}) \subseteq \ell$ (Fact 3), it follows that $\text{currPc}(\sigma_1^{-1}) \subseteq \ell$.

2. We applied the F-DIOUT rule. The proof of this case is similar to that of the F-OUT rule.

**Fact 6.** We now prove that (1) $\sigma_0^{-1} \approx_l \sigma_1^{-1}$, (2) $C_0^{-1} = C_1^{-1}$, and (3) $S_0^{-1} = S_1^{-1}$. From Facts 1 and 2, it follows that initially $\sigma_0^0 \approx_{\text{obs}} \sigma_1^0$, $\sigma_0^0 = C_0^{-1}$, $S_0^{-1} = S_1^{-1}$, and $\sigma_0^0 \equiv_n^{|n|} \sigma_0^0$. Furthermore, $\text{currPc}(\sigma_0^0) = \text{currPc}(\sigma_1^0) = \bot$ given that both runs start from the initial monitor state $\Delta_0$. Therefore, we can repeatedly apply Lemmas D.25 and D.26 (depending on whether we are in a low or high context given $\ell$) to construct bisimulations among states in the two runs and use Lemma D.27 to compose the various bisimulations in a unique bisimulation $R$. We remark that during the construction of the bisimulation we can always apply either Lemma D.25 or Lemma D.26. In particular, $\text{currPc}(\sigma_0^{-1}) \subseteq \ell$ (Fact 3) and $\text{currPc}(\sigma_1^{-1}) \subseteq \ell$ (Fact 5) ensure that the execution of branching statements leading to high contexts with respect to $\ell$ always terminates before $\sigma_1^{-1}$ and $\sigma_1^{-1}$. Finally, observe that $(\sigma_0^0, \sigma_1^0, (\sigma_1^0, \sigma_1^{-1})) \subseteq R$ by construction. From this, $\sigma_0^{-1} \approx_l \sigma_1^{-1}$, $C_0^{-1} = C_1^{-1}$, and $S_0^{-1} = S_1^{-1}$ directly follow. □
Bibliography


<table>
<thead>
<tr>
<th>Bibliography</th>
<th>Page</th>
</tr>
</thead>
</table>


Resume

MARCO GUARNIERI
Born on 6 March 1988 in Ponte San Pietro, Italy.
Citizen of Italy.

Education

ETH Zurich, Zurich, Switzerland
PhD in Computer Science
October 2012 – present

Università degli Studi di Bergamo, Bergamo, Italy
Laurea Magistrale in Ingegneria Informatica
(Master of Science in Computer Engineering)
September 2010 – July 2012

Laurea in Ingegneria Informatica
(Bachelor of Science in Computer Engineering)
September 2007 – September 2010

Professional Experience

ETH Zurich, Zurich, Switzerland
Research Assistant
October 2012 – present

Università degli Studi di Bergamo, Bergamo, Italy
Research Assistant
August 2012 – September 2012

SAP Labs France, Sophia Antipolis, France
R&D Intern
June 2010 – September 2010

Salvaneschi & Partners, Software and System Engineering, Bergamo, Italy
Testing Consultant
October 2009 – March 2010