Diss. ETH No. 25906

Preventing privilege abuse using policy analysis and policy mining

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich presented by

CARLOS MAURICIO COTRINI JIMENEZ

Master of Computer Science,
ETH Zurich
born on 19.05.1989
citizen of Colombia

accepted on the recommendation of
Prof. Dr. David Basin, examiner
Prof. Dr. Joachim Buhmann, co-examiner
Prof. Dr. Pierangela Samarati, co-examiner

2019

Abstract

Organizations define access control policies to prevent users abusing their privileges. In large organizations, such policies are highly complex as they administer thousands of permissions for thousands of users. In addition, these policies are currently manually maintained, which makes policies prone to mistakes. Such mistakes may deny users the permissions they need to perform daily tasks or, even worse, they may grant permissions that users should not have. The latter may have dire consequences for the organization, even when the employees themselves do not abuse those additional permissions, as hackers may gain internal access and then abuse them to perform nefarious acts.

Privilege abuse remains a major problem for organizations handling sensitive data. Even for healthcare companies, where access to patient data is critical, abuse by internal employees is a threat to patients' privacy. Indeed, Verizon's data breach report from 2018 shows that 54% of data breaches in healthcare involved internal actors. Perhaps the most famous case of internal abuse is Edward Snowden, which shows that giving the right access to each employee is challenging even for the strongest security agencies.

Two fields have proposed solutions to strengthen policy specification and maintenance. The first one, *policy analysis*, offer queries that policy administrators can execute to determine whether policies are granting permissions as intended. Policy analysis has helped to discover inconsistencies in policies or unnecessary assignments of permissions. The second one, *policy mining*, analyzes how permissions are being used in the organization and computes a policy that grants to each employee the permissions he needs. By observing what permissions are *not* actually used, policy miners can compute policies that prevent privilege abuse and that are also tighter than manually specified policies.

This thesis makes three contributions to these fields. First, we propose FORBAC, an extension for RBAC (Role-Based Access Control) that strikes a balance between expressiveness in policy specification and efficiency in policy analysis. Through a case study with a major European bank, we show that FORBAC is expressive enough for modern RBAC policies while simple enough to keep the complexity of policy analysis in NP. Second, we propose RHAPSODY, the first algorithm for mining ABAC (Attribute-Based Access Control) policies from logs that guarantees to mine precisely the set of all significant, reliable, and succinct rules. We also show how all other ABAC mining algorithms fail to provide these guarantees. Finally, we propose UNICORN, a universal method for building policy miners. Using UNICORN, we have built competitive policy miners for a wide variety of policy languages. In particular, the ABAC policy miner built with UNICORN outperforms RHAPSODY and, using UNICORN, we have been able to build the first policy miners for XACML (eXtensible Access Control Markup Language) and RBAC with spatio-temporal constraints, languages for which no miner was known before.

Zusammenfassung

Organisationen erstellen Zugriffskontrolrichtlinien, auch “Policies” genannt, um den Missbrauch von Zugriffsrechten zu verhindern. In grossen Organisationen werden diese Policies sehr komplex, da sie tausende Zugriffsrechte für tausende Benutzer verwalten. Zudem werden Policies zurzeit manuell gewartet, was sie anfällig für Fehler macht. Solche Fehler können Benutzer daran hindern ihre gewöhnliche Aufgabe zu erledigen. Im schlimmsten Fall können diese Fehler unberechtigte Zugriffsrechte an Benutzer vergeben. Dies könnte ernste Konsequenzen für die Organisation haben. Selbst wenn die Benutzer selbst nicht diese Rechte missbrauchen, so könnten Hacker internen Zugang erlangen und dann diese Rechte auf Kosten der Organisation missbrauchen.

Der Missbrauch von Zugriffsrechten ist nach wie vor ein relevantes Problem für Organisationen, die mit privaten Daten arbeiten. Selbst für Gesundheitsunternehmen, wo der Zugriff zu Patientendaten kritisch ist, bedroht der Missbrauch von Zugriffsrechten durch interne Fachkräfte die Privatsphäre der Patienten. Der Bericht von Verizon in 2018 zeigt, dass interne Fachkräfte in 54% der Datenlecke in Gesundheitsunternehmen involviert waren. Das wichtigste Beispiel internen Missbrauchs ist vielleicht Edward Snowden, der zeigt, dass eine richtige Zuteilung von Zugriffsrechten eine grosse Herausforderung ist, sogar für die besten Sicherheitsunternehmen.

Zwei Forschungsgebiete haben Lösungen hervorgebracht, um die Spezifizierung und Wartung von Policies zu erleichtern. Die erste, *policy analysis*, bietet Abfragen an, die Policyverwalter ausführen können, um dafür zu sorgen, dass die Policies Zugriffsrechte richtig zuteilen. Policy analysis hat dabei geholfen, Unstimmigkeiten in Policies und unnötige Zuteilungen von Zugriffsrechten aufzudecken. Das zweite Gebiet, *policy mining*, beschäftigt sich mit der Entwicklung von Algorithmen, die als *Policy Miners* bezeichnet werden. Diese Algorithmen analysieren wie Zugriffsrechte in der Organisation angefragt werden und errechnen daraus eine Policy, die jedem Benutzer die Zugriffsrechte zuteilt, die er braucht. Policy Miners streben danach, präzise Policies zu rechnen, die den Missbrauch von Zugriffsrechten vermeiden.

Diese Doktorarbeit liefert drei Beiträge zu diesen Gebieten. Erstens stellen wir FORBAC vor, eine Erweiterung von RBAC (Role-Based Access Control), die eine Balance zwischen Ausdrucksfähigkeit in der Policyspezifizierung und Effizienz in der Policyanalyse erreicht. Mithilfe einer Studie in einer grossen europäischen Bank zeigen wir, dass FORBAC expressiv genug ist, um moderne RBAC-Policies zu spezifizieren, und gleichzeitig einfach genug, um die Komplexität des Policyanalyseproblems innerhalb der NP Komplexitätsklasse zu halten. Zweitens stellen wir Rhapsody vor, der erste Policy Miner für ABAC (Attribute-Based Access Control), der exakt die Menge aller signifikanten, zuverlässigen und bündigen Regel rechnet. Wir zeigen auch wie alle anderen Policy Miners für ABAC darin scheitern, diese Menge von Regeln zu rechnen. Zuletzt stellen wir Unicorn vor, eine universelle Methode um Policy Miners zu entwickeln. Mithilfe von UNICORN haben wir fähige Policy Miners für eine breite Vielfalt von Policysprachen entwickelt. Insbesondere ist das Policy Miner für ABAC, das wir mit UNICORN entwickelten, RHAPSODY überlegen. Mithilfe von UNICORN haben wir auch die ersten Policy Miners für zwei komplexe Policysprachen entwickelt. Die erste Sprache ist XACML (eXtensible Access Control Markup Language) und die zweite ist eine Erweiterung von RBAC, mit der man die Zeit und den Ort einschränken kann, an denen Benutzer Zugriffsrechten erteilt werden. Bisher gab es keinen Policy Miner für diese beide Sprachen.

Acknowledgements

My time as a doctoral student has been the most rewarding and enriching period of my life. I would like to thank my advisor, David Basin, for having given me this opportunity. His support, mentorship, and diligence were an inspiring example for me during these years.

I would also like to thank my co-examiners Joachim Buhmann and Pierangela Samarati for taking the time to read this thesis, understand their ideas, and critically evaluate them.

I would also like to express my gratitude to those who in one way or another have mentored me throughout my time at ETH Zürich. I would like to thank Manuel Clavel for fruitful discussions and guidance on my first years of research at ETH, Felix Klaedtke and Eugen Zălinescu for advising me during my first research project at ETH, and Luis Jaime Corredor and Jorge Cuellar for advising and encouraging me to start my postgraduate studies at ETH Zürich.

Throughout my research I met several colleagues from the institute of machine learning who helped me to grasp many important concepts upon which this thesis was built. Many thanks to Luca Corinzia, Alex Gronskiy, Nico Gorbach, and Mario Lučić for their insights and patience.

Many thanks to my colleague Thilo for the many interesting discussions. His insights on work and life in general will be of value for me in the future. Many thanks also to my family and friends, for all the support and love throughout my career.

Finally, my deepest thanks to Klaus (.-*) for all the good moments we shared together these years.

Contents

Contents **3**

1 Introduction **5**
- 1.1 Background . . . 6
- 1.2 Challenges . . . 11
- 1.3 Contributions . . . 11
- 1.4 Overview . . . 13

2 Preliminaries **15**
- 2.1 Motivating example . . . 15
- 2.2 Many-sorted first-order logic . . . 16
- 2.3 Access control . . . 25
- 2.4 Policy analysis . . . 29
- 2.5 Policy mining . . . 30
- 2.6 Conclusion . . . 34

3 FORBAC: First-order Role-based Access Control **37**
- 3.1 Introduction . . . 37
- 3.2 Requirements for FORBAC . . . 38
- 3.3 Syntax and semantics of FORBAC . . . 41
- 3.4 Policy analysis in FORBAC . . . 45
- 3.5 Experimental results . . . 49
- 3.6 Related work . . . 50
- 3.7 Conclusion and future work . . . 51

4 Rhapsody: Reliable Apriori Subgroup Discovery **53**
- 4.1 Introduction . . . 53
- 4.2 The ABAC mining problem . . . 57
- 4.3 Quantifying over-permissiveness . . . 62
- 4.4 Rhapsody . . . 66

4.5 Evaluating generalization 71
4.6 Experiments 76
4.7 Related work 79
4.8 Conclusion 82

5 Unicorn: Universal access-control policy mining 85
5.1 Introduction 85
5.2 The problem of designing policy miners 88
5.3 A universal policy language 89
5.4 A probability distribution on permission assignments and policies 95
5.5 Approximating distributions with mean-field variational inference 99
5.6 Building the policy miner 103
5.7 Mining policies 105
5.8 Spatio-temporal RBAC policies 111
5.9 XACML policies 112
5.10 Experiments 120
5.11 Related work 124
5.12 Conclusion 125

6 Conclusion 127

A Appendix for FORBAC 129
A.1 Complexity results for F 129
A.2 Complexity results for FORBAC 130

B Appendix for Rhapsody 137
B.1 ABAC instances used for experiments 137
B.2 Proofs 139

C Appendix for Unicorn 141
C.1 Grid search for policy mining 141
C.2 Optimal parameters for the mean-field approximating distribution 141
C.3 Simplifying the computation of expectations 142
C.4 Modeling RBAC temporal constraints 143
C.5 Datasets and synthetic policies used for experiments 146

Bibliography 149

Chapter 1

Introduction

Any kind of human organization requires rules that restrict what their members are allowed to do in order to protect the organization's interests from the abuse of others. In the case of nations, these rules are called laws. Perhaps a marked feature of laws is that, being conceived by humans, they sometimes fail to consider all possible cases and implications, which some devious citizens exploit to get away with nefarious acts. The government must then amend the law and adapt it to changing circumstances. Similarly, the advent of new technologies and disruptive events in society require one to rethink and reconsider what citizens are allowed to do in certain circumstances. This is one of the reasons why laws become complex and why common citizens must recur to experts during litigations.

Organizations that use information technology must also protect their interests from abuse by their own members, so they define *access control policies* that restrict what their members are allowed to do. As in the case of law, organizations need to foresee the implications of an access control policy. Moreover, they continually need to edit the policy, as they discover unintended consequences of the currently implemented policy and as the organization experiences structural changes. As a result, policies become convoluted with time and it is then hard to ensure that the policy adequately restricts the organization's members' actions.

In access control literature, members of an organization are called *users*. Access control policies usually work by granting a set of permissions to each user. An incorrect assignment of permissions may have serious consequences for the organization. On one hand, users may lack the rights they need to perform their daily job. This is usually promptly recognized by the users themselves. On the other hand, users may be granted permissions that they do not need. This is more problematic, as certain users may see this as an opportunity for personal gain or for harming the organization. This is illustrated by the case of Barings Bank, United Kingdom's oldest merchant bank, which went bankrupt in 1995 after one of its employees engaged in a series of fraudulent and unauthorized investments [1].

Privilege abuse is still a problem for many organizations. Perhaps the most famous example of our time is Edward Snowden's disclosure of government surveillance, which shows that defining correct access control policies is challenging even for the most experienced security organizations. Verizon's data breach reports from the last 5 years show that privilege abuse is still a major cause for data breaches in healthcare and public organizations [50, 51]. Internal users exploit their access to private information for financial gain, espionage, or revenge. Verizon's reports advocate policy administrators to continually review access control policies and ensure that users are granted only those permissions that they need to perform their tasks and nothing else.

Given the impact that privilege abuse may have in current organizations, we devote ourselves to the problem of maintaining access control policies. We develop techniques, using results from policy analysis and policy mining, that help to maintain access control policies simple and precise. We provide next an overview of these fields and the techniques we developed.

1.1 Background

1.1.1 Access control

An *organization* is a collection of *users* and *resources*. Users perform *actions* on those resources. In order to exercise an action on a resource, a user requires a *permission* for that particular action and that particular resource. In this thesis, we identify a permission with a pair consisting of an action and a resource. To prevent abuse from their own users, organizations define *organizational security policies* that define which permissions are assigned to which users in the organization. These policies are usually described in a high-level language. To be machine enforceable, policy administrators must specify these policies as *access control policies* in a machine-readable format. These policies can be understood as an assignment of *permissions* to *users* and are formalized using *policy languages*.

Figure 1.1 illustrates a tiny organization with three users and three permissions. There are two departments: trading and technology. Each user and each permission is labeled with one of these departments. The organizational security policy dictates that users shall have exactly those permissions whose department label matches the user's department label. For example, users from the trading department can only exercise those permissions for the trading department. The access control policy implementing this security policy is a matrix that marks with a tick those entries where the corresponding user is assigned the corresponding permission. Observe that the

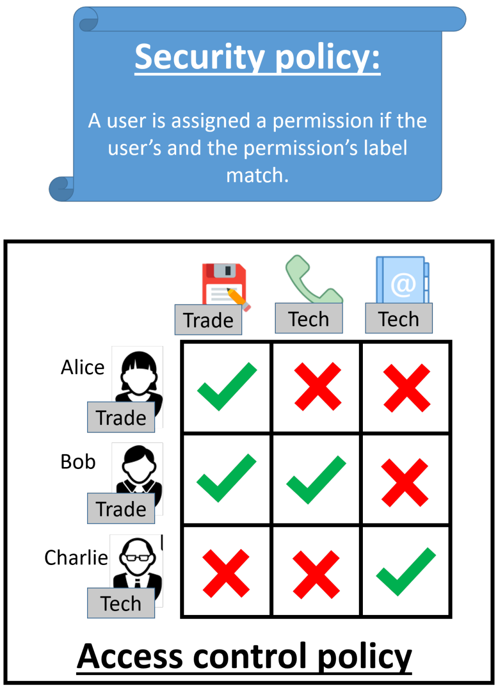

Security policy:
A user is assigned a permission if the user's and the permission's label match.
Trade
Tech
Tech
Alice
Trade
Bob
Trade
Charlie
Tech
Access control policy

Figure 1.1: A small organization with three users and permissions.

access control policy incorrectly implements the security policy. It assigns the green permission to Bob, although their labels do not match. Also, it does not assign the green permission to Charlie, although their labels do match. We discuss later methods to detect and correct those misconfigurations.

Whenever a user wants to perform a sensitive action, she must issue a *request* (a pair consisting of a user and a permission), which identifies the user and the permission she tries to exercise. The access control policy then decides if the user is authorized to exercise the permission. Observe then that we can also treat access control policies as functions mapping requests to Boolean values. When an access control policy outputs *true*, we say that the policy *authorizes* the request; otherwise, we say that it *denies* the request. We call a *decision* the value output by an access control policy after evaluating a request.

We assume that the organization keeps a *log* that records all requests issued, together with the respective decisions output by the access control policy. Figure 1.2 gives an example of a log for the scenario in Figure 1.1. Each entry in the log contains a request, a green tick if the request was authorized, and a red cross if the request was denied. Observe that logs may contain repeated occurrences of a same request, as users may try several times to exercise a permission.

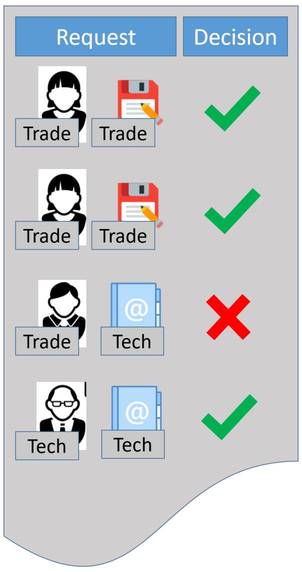


Figure 1.2: A log for the scenario in Figure 1.1.

The solutions presented in this thesis do not take into account a request's frequency in the log. A request that occurs 1,000 times is treated in the same way as a request that occurs only once. We admit that a request's frequency may provide valuable information, but the datasets we had access to did not provide each request's frequency. Hence, investigating solutions that use the requests' frequencies is left as future work.

1.1.2 Policy languages

We now introduce some of the most important policy languages for specifying access control policies.

Role-based access control (RBAC) [52]. A policy in this language, called *RBAC policy*, consists of three components:

- A set of roles describing a group of permissions that are often exercised by a group of users in the organization.
- An assignment of permissions to roles.
- An assignment of roles to users.

A user is authorized to exercise a permission if he or she has assigned a role that was assigned the permission.

Figure 1.3 illustrates an RBAC policy that (correctly) implements the organizational security policy of the organization in Figure 1.1. Observe that there are two roles corresponding to the organization's departments. Each user is assigned the role corresponding to that user's department label and each permission is assigned to the role corresponding to its label.

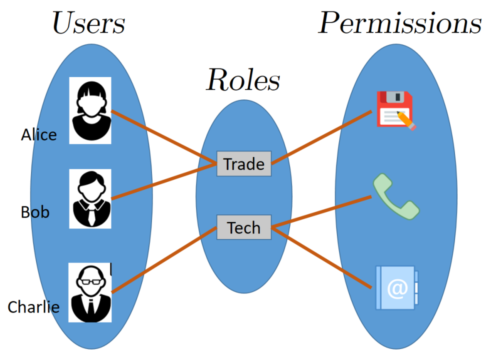
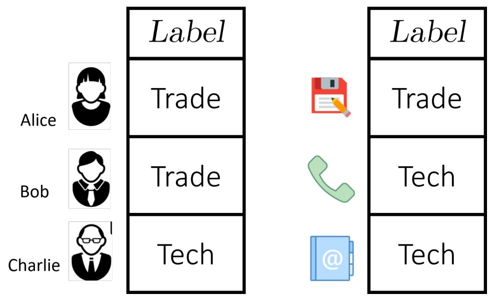


Figure 1.3: An RBAC policy that implements the organizational security policy of the organization in Figure 1.1.

Attribute-based access control (ABAC) [68]. ABAC requires that the organization defines *attributes* and that users and permissions have *attribute values*. A policy in this language, called *ABAC policy*, is a set of *rules*. A rule is a function that defines if a permission is assigned to a user. Rules can only depend on the user's and the permission's attribute values.

We now (informally) specify the organizational security policy of the organization in Figure 1.1 with an ABAC policy. A rigorous definition of ABAC policies is given in Chapter 2. First, we define one attribute, called *Label*, and define two attribute values for it: Trade and Tech. Figure 1.4 shows the assignment of attribute values to users and permissions in Figure 1.1. The desired ABAC policy contains only one rule specified by the following sentence: *A user is assigned a permission if their attribute values are the same.*



	Label		*Label*
Alice	Trade		Trade
Bob	Trade		Tech
Charlie	Tech		Tech

Figure 1.4: An assignment of attribute values to users and permissions.

RBAC with constraints [70, 78, 89]. RBAC is not expressive enough for many organizations. For example, an organization may wish that a role is assigned to a user only at certain times or when the user is in a particular location. It may also want to assign roles depending on the user attribute values. For this reason, extensions of RBAC have been proposed that allow the specification of spatio-temporal constraints or constraints that depend on the user's attribute values.

1.1.3 Policy analysis and policy mining

Privilege abuse happens when a user is authorized by the access control policy to exercise a permission, in contradiction to the organizational security policy. For example, if Bob attempts to exercise the green permission in Figure 1.1, he would commit privilege abuse, as the access control policy in Figure 1.1 authorizes this, but the security policy forbids it. Perhaps Bob had no malicious intention and just was not aware that he should not exercise the green permission. Even in this case, we classify his action as privilege abuse.

This thesis presents techniques for preventing privilege abuse in organizations. These techniques belong to the following two research fields:

Policy analysis [107, 126, 54, 10]. Techniques developed in this field take a currently implemented access control policy and a set of properties that we want to make sure the access control policy satisfies. Policy analysis uses logic-based frameworks to validate the properties against the policies. The properties are specified by policy administrators in formal languages and usually specify requirements described in the organizational security policy. Policy analysis counteracts privilege abuse by inspecting if a policy is granting permissions to users as intended by the security policy.

Policy mining [132, 133, 32, 30]. The main goal in this field is to design *policy miners*, algorithms taking as input a log containing previously decided requests. From these logs, policy miners compute (i.e., mine) policies that evaluate requests in a way that is consistent with the observed decisions in the log. Policy mining has various applications. It can be used for refactoring policies that have become convoluted after organizational changes and migrating policies to new policy languages. We give an overview of use cases for policy mining in Chapter 2. In particular, policy mining can be used to prevent privilege abuse. We will see in Chapter 4 that a mined policy can be compared with the currently implemented policy to identify permissions that are currently assigned to but rarely exercised by users.

1.2 Challenges

We identify the following three challenges in these two fields:

Tension between policy expressiveness and efficiency in policy analysis. In the last years, different works have proposed extensions for RBAC that allow the specification of more expressive policies, e.g. [63, 70, 78, 79, 89, 94]. However, the higher expressive power comes at the cost of more complicated policies. It becomes then harder to understand what the implications of a given access control policy are and policy analysis becomes computationally harder. The challenge here is to propose a language that is expressive enough to specify current access control policies while at the same time yielding an acceptable computational complexity for policy analysis.

Mining policies from sparse logs yields complicated or overly permissive policies. In our case studies with companies, logs contain less than 10% of all possible requests and 90% of them consist of authorized requests. Such logs do not give sufficient information to decide how all requests should be decided. As a result, we will show that, when applying standard policy mining techniques, one obtains policies that are overly permissive, which allows privilege abuse. Those techniques that guarantee not to mine overly permissive policies yield, however, unnecessarily complicated policies. The challenge here is to design mining algorithms that can mine simple and not overly permissive policies.

Policy mining requires expert knowledge in machine learning. Although competent policy miners exist for languages like RBAC and ABAC, it is very difficult to use the ideas behind them to design policy miners for other languages, even for relatively simple RBAC extensions. RBAC extensions with spatio-temporal constraints illustrate this difficulty. They have been researched for more than 15 years, but there is no known policy miner that can mine these type of RBAC policies. Also, no policy miner is known for XACML [64], which has been a popular and well-researched language in the access control literature. This is problematic as organizations often tailor policy languages to meet specific expressiveness requirements, meaning that they cannot use standard policy miners. The challenge here is to facilitate the design of policy miners for other languages.

1.3 Contributions

We address the challenges above with the following contributions:

FORBAC: a new extension for RBAC that strikes a balance between expressiveness and efficiency [40]. Other RBAC extensions are so expressive that their policy analysis problems are computationally harder than necessary, while other RBAC extensions tailored for policy analysis are so simplistic that they cannot fulfill expressiveness requirements of current organizations.

FORBAC was developed in collaboration with Thilo Weghorn. I proposed the FORBAC language, proposed the set of existential FORBAC-formulas as a language for specifying policy properties, and proved that deciding satisfiability of existential FORBAC formulas is NP-complete. Thilo Weghorn proposed relevant policy properties for policy analysis in FORBAC and formalized them as existential FORBAC-formulas. He also conducted the case study in a major European bank where we evaluated FORBAC's expressiveness and efficiency in policy analysis.

Rhapsody: an algorithm for mining ABAC policies from logs [39]. Since the 70s, there have been techniques that can be used to mine policies from logs. However, these techniques yield overly complicated or, even worse, overly permissive policies. Rhapsody is the first policy mining algorithm that guarantees never to mine overly permissive policies and to mine rules of minimal size.

Unicorn: a universal method to build policy miners. Designing policy miners is a challenging task as this requires expert knowledge of machine learning. Moreover, it is hard to use the ideas behind a policy miner to build policy miners for another policy language, even when that language is just an extension of the original. Unicorn presents a straightforward method to building policy miners that does not require knowledge of machine learning. The designer of a policy miner only needs to specify the policy language and an objective function describing the requirements for the policy miner. Given these inputs, Unicorn defines a policy miner as an iterative procedure that the designer can implement using a programming language. Using Unicorn, we have built competitive policy miners for a wide variety of policy languages. In particular, we have built the first XACML policy miner and the first policy miner for RBAC policies with spatio-temporal constraints.

1.3.1 Publications

This thesis's contents is based on the following articles:

- Carlos Cotrini, Thilo Weghorn, Manuel Clavel, and David Basin, *"Analyzing First-Order Role-Based Access Control"*, in Proceedings of the 28th IEEE Computer Security Foundations Symposium (**CSF 2015**)
- Carlos Cotrini, Thilo Weghorn, and David Basin, *"Mining ABAC Rules from Sparse Logs"*, in Proceedings of the 3rd European Symposium on Security and Privacy (**EuroS&P 2018**)

- Carlos Cotrini, Luca Corinzia, Thilo Weghorn, and David Basin, *"The Next 700 Policy Miners: A Universal Method for Building Policy Miners"*, unpublished manuscript, 2018.

1.4 Overview

This thesis is organized as follows. After this introduction (Chapter 1), we give preliminaries on many-sorted first-order logic and access control (Chapter 2). We then devote one chapter to each of our contributions: FORBAC (Chapter 3), RHAPSODY (Chapter 4), and UNICORN (Chapter 5). Finally, we draw conclusions (Chapter 6). Appendices A, B, and C contain proofs and technical details.

As this thesis makes contributions to both policy analysis and policy mining, there are two ways to read it. The reader interested in policy analysis may read Chapter 2 for an overview of many-sorted first-order logic and then read about FORBAC in Chapter 3. The reader interested in policy mining may read Chapter 2 and then read about RHAPSODY and UNICORN in Chapters 4 and 5.

Chapter 2

Preliminaries

In this chapter, we give an overview of access control, first-order logic, and the fields of policy analysis and policy mining.

2.1 Motivating example

Example 1 Figure 2.1 presents an example of an organization with 27 users and some policies that we will use throughout this section. For simplicity, we assume that there is only one permission, not shown in the figure. Each user has two *attributes* that define what he does in the organization. They are *function* and *level*. Each of them can take one of three values and the levels are assumed to be ordered. Observe that rows and columns in Figure 2.1 have a different meaning than in Figure 1.1 in Chapter 1. A row in Figure 2.1 contains all users who have a particular attribute value for the attribute "function", whereas a row in Figure 1.1 describes what permissions are assigned to a particular user. Similarly, a column in Figure 2.1 contains all users who have a particular attribute value for the attribute "level", whereas a column in Figure 1.1 describes to which users a particular permission is assigned.

The shaded rectangles describe the organizational security policy, which assigns the permission to all users fulfilling at least one of the following:

- The user's level is 2.
- The user's function is staff and level is at most 2.
- The user's function is teacher and level is at least 2.

The dotted rectangles describe the implemented access control policy. We assume here that the implemented policy is an ABAC policy and that the policy administrator inadvertently implemented a policy that, in compari-

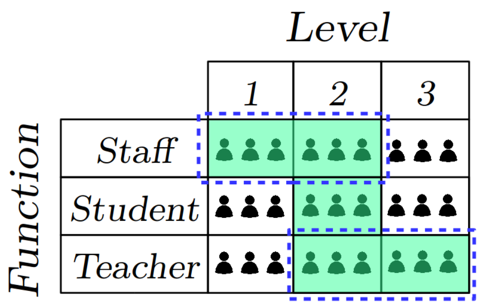


Figure 2.1: An example of an organization with 27 users and one permission (not shown in the figure). Each user has two attribute values describing his or her *function* and *level* in the organization, respectively. The shaded rectangles describe the organizational security policy and the dotted rectangles describe the implemented access control policy.

son with the organizational security policy, fails to assign the permission to students with level 2. □

There is a wide variety of policy languages that could have been used to define the access control policy from the example above. One of the most basic are access control matrices [24]. These are binary matrices, where each row represents a user and each column represents a permission. For an access control matrix M, a user u, and a permission p, $M[u, p] = 1$ if p is assigned to u and $M[u, p] = 0$ otherwise. More sophisticated policies use rules to define what permissions each user has. An example of such a policy is: "A user can access the main building on week days during working hours, provided that he has a valid ID card". In this thesis, we need a formalism that allows us to model organizations and the wide variety of access control policies.

2.2 Many-sorted first-order logic

Previous work in the field of policy analysis has shown that *many-sorted first-order logic* is a formalism powerful enough to describe organizations and policies in a wide variety of policy languages. Therefore, we start with a review of basic concepts of first-order logic and then use them to rigorously define the concepts of "organization", "user", "permission", and "access control policy".

We start by defining the concepts of "signature", "formula", and "structure". Signatures define the symbols used to build formulas, formulas are helpful to specify certain types of policies, and structures model organizations and other types of policies.

2.2.1 Syntax

Specifying formulas

Definition 2 A *signature* is a tuple $(\mathbb{S}, \mathbb{R}, \mathbb{F}, \mathbb{V})$ fulfilling the following.

- $\mathbb{S}$ is a finite non-empty set of *sorts*.
- $\mathbb{R}$ is a finite non-empty set of *relation symbols*.
- $\mathbb{F}$ is a finite non-empty set of *function symbols*.
- $\mathbb{V}$ is a countable set of *variables*.

Each relation and each function symbol has associated a *type*, which is a sequence of sorts. Each variable has also an associated type, which is a sort. Furthermore, we assume the existence of two sorts $\mathbf{USERS}, \mathbf{PERMS} \in \mathbb{S}$, denoting the users and the permissions in the organization, respectively. We also assume the existence of the sorts $\mathbf{BOOL}$, $\mathbf{INT}$, $\mathbf{STR}$, $\mathbf{2^{INT}}$, $\mathbf{2^{STR}}$, which represent Boolean values, integers, strings, sets of integers, and sets of strings, respectively. □

We denote sorts with **CAPITAL BOLD** letters, relation symbols with *CAPITAL ITALIC* letters, and function symbols and variables with *small italic* letters. To agree with standard notation, we write a relation symbol's type $(\mathbf{S}_1, \ldots, \mathbf{S}_k)$ as $\mathbf{S}_1 \times \ldots \times \mathbf{S}_k$ instead. We write a function's symbol's type $(\mathbf{S}_1, \ldots, \mathbf{S}_k)$ as $\mathbf{S}_1 \times \ldots \times \mathbf{S}_{k-1} \to \mathbf{S}_k$ instead. We allow $k = 1$ and, in that case, we call function symbols *constant symbols*. We denote constant symbols with small serif letters.

Example 3 We now define a signature $\Sigma_0 = (\mathbb{S}, \mathbb{R}, \mathbb{F}, \mathbb{V})$ that we use to formalize the concepts from Example 1.

- $\mathbb{S}$ contains the default sorts.
- $\mathbb{R}$ consists of only one relation symbol $\leq$: $\mathbf{INT} \times \mathbf{INT}$.
- $\mathbb{F}$ consists of two function symbols *Function* : $\mathbf{USERS} \to \mathbf{STR}$ and *Level* : $\mathbf{USERS} \to \mathbf{INT}$. It also contains six constant symbols: "staff", "student", "teacher" of type $\mathbf{STR}$ and 1, 2, and 3 of type $\mathbf{INT}$. These symbols are intended to represent the strings "staff", "student", "teacher", and the integers 1, 2, and 3, respectively.

□

Definition 4 Let Σ be a signature. We define *(first-order) terms* as those expressions obtained by finitely many applications of the following rules:

- Any variable is a term.
- Any constant symbol is a term.

- If $t_1, t_2, \ldots, t_n$ are terms of type $\mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_{k-1}$ and f is a function symbol of type $\mathsf{S}_1 \times \ldots \times \mathsf{S}_{k-1} \to \mathsf{S}_k$, then $f(t_1, \ldots, t_n)$ is a term of type S_k.

□

Example 5 Let Σ_0 be the signature from Example 3. If we let u be a variable of type **USERS**, then some first-order terms from this signature are the variable symbol u, the constant symbol "student" of type **STR**, and $Function(u)$.
□

Observe that, using Definition 4, we cannot build ill-formed terms like

$$Function(Level(u)). \tag{2.1}$$

This is because $Level(u)$ is of type **INT** but $Function$ is a symbol of type **USERS** $\to$ **STR**.

Definition 6 Let Σ be a signature. We define *(first-order) formulas* as those expressions obtained by finitely many applications of the following rules:

(F1) If t_1 and t_2 are terms of a same type, then $t_1 = t_2$ is a formula.

(F2) If $t_1, t_2, \ldots, t_n$ are terms of type $\mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_n$ and R is a relation symbol of type $\mathsf{S}_1 \times \mathsf{S}_2 \times \ldots \times \mathsf{S}_n$, then $R(t_1, \ldots, t_n)$ is a formula.

(F3) If φ_1 and φ_2 are formulas, then $\varphi_1 \wedge \varphi_2$ is also a formula.

(F4) If φ is a formula, then $\neg\varphi$ is also a formula.

(F5) If φ is a formula and x is a variable of type W, then $\exists x : \mathsf{W}.\, \varphi$ is also a formula.

□

Formulas may use the symbol $=$ in their construction. This can lead to awkward notation. For example, suppose that we define a formula φ and two terms t_1 and t_2. Suppose also that we claim that the formulas φ and $t_1 = t_2$ are the same. We could denote this as $\varphi = (t_1 = t_2)$, but we sometimes denote it instead as $\varphi \equiv t_1 = t_2$. That is, we use the symbol $\equiv$ to indicate that two formulas are the same, whereas we use the symbol $=$ to denote the equality symbol in a formula. We also use the symbol $=$ outside first-order logic to indicate that two objects are the same.

We use the following standard notation to abbreviate some formulas:

$$\begin{aligned}
&\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) && \varphi_1 \to \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2. \\
&\bigwedge_{1 \le i \le n} \varphi_i \equiv \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n && \bigvee_{1 \le i \le n} \varphi_i \equiv \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n. \\
&\varphi_1 \neq \varphi_2 \equiv \neg(\varphi_1 = \varphi_2). && \forall x : \mathsf{W}.\varphi \equiv \neg\exists x : \mathsf{W}.\, \neg\varphi.
\end{aligned}$$

Definition 7 An *atomic formula* is a formula that was obtained only by rules **(F1)** and **(F2)**. □

Definition 8 A *quantifier-free formula* is a formula that was obtained only by rules **(F1)–(F4)**. □

Example 9 Let u be a variable of type **USERS** and let Σ_0 be the signature from Example 3. Some examples of formulas are $Function\,(u) = ``\mathsf{student}"$ and $\exists x : \mathbf{INT}.\, Level\,(u) \leq x$. The formula $Function\,(u) = ``\mathsf{student}"$ is an atomic and quantifier-free formula. □

Occurrences in a formula

Definition 10 below formally defines the following concept. A variable x or a formula φ *occurs* in another formula ψ if x or φ is a substring of ψ, when they are all written as strings. For example, let

$$\psi \equiv \exists u : \mathbf{USERS}.\, Function(u) = ``\mathsf{student}" \vee Function(u) = ``\mathsf{teacher}". \quad (2.2)$$

The variable u occurs in ψ. Whereas the formula $Function(u) = ``\mathsf{student}" \wedge Function(u) = ``\mathsf{teacher}"$ does not occur in ψ.

Definition 10 A term t *occurs in a term* t_0 if

- $t_0 \equiv t$ or
- $t_0 \equiv f(t_1, \ldots, t_n)$ and t occurs in t_i, for some $1 \leq i \leq n$.

A term t *occurs in a formula* ψ if any of the following holds:

- $\psi \equiv t_1 = t_2$ and t occurs in t_1 or t_2.
- ψ is an atomic formula of the form $R\,(t_1, \ldots, t_n)$ and t occurs in t_i, for some $i \leq n$.
- $\psi \equiv \neg\psi'$ and t occurs in ψ'.
- $\psi \equiv \psi_1 \wedge \psi_2$ and t occurs in ψ_1 or in ψ_2.
- $\psi \equiv \exists x : \mathbf{W}.\psi'$ and t occurs in ψ'.

A formula φ *occurs in another formula* ψ if any of the following holds:

- $\varphi \equiv \psi$.
- $\psi \equiv \neg\psi'$ and φ occurs in ψ.
- $\psi \equiv \psi_1 \wedge \psi_2$ and φ occurs in ψ_1 or in ψ_2.
- $\psi \equiv \exists x : \mathbf{W}.\psi'$ and φ occurs in ψ'.

□

Definition 11 For a term t, *the multiset* $\mathit{TerOcc}\,(t)$ *of term occurrences of* t is inductively defined as follows:

- If t is a variable x, then $\mathit{TerOcc}\,(t) = \{x\}$.
- If $t = f(t_1, \ldots, t_n)$, then $\mathit{TerOcc}\,(t) = \{t\} \cup \mathit{TerOcc}\,(t_1) \cup \ldots \cup \mathit{TerOcc}\,(t_n)$.

For a formula ψ, *the multiset* $\mathit{TerOcc}\,(\psi)$ *of term occurrences of* ψ is inductively defined as follows:

- If $\psi \equiv t_1 = t_2$, then $\mathit{TerOcc}\,(\psi) = \mathit{TerOcc}\,(t_1) \cup \mathit{TerOcc}\,(t_2)$.
- If $\psi \equiv R\,(t_1, \ldots, t_n)$, then $\mathit{TerOcc}\,(\psi) = \bigcup_{i \leq n} \mathit{TerOcc}\,(t_i)$.
- If $\psi \equiv \neg\psi'$, then $\mathit{TerOcc}\,(\psi) = \mathit{TerOcc}\,(\psi')$.
- If $\psi \equiv \psi_1 \wedge \psi_2$, then $\mathit{TerOcc}\,(\psi) = \mathit{TerOcc}\,(\psi_1) \cup \mathit{TerOcc}\,(\psi_2)$.
- If $\psi \equiv \exists x : \mathbf{W}.\psi'$, then $\mathit{TerOcc}\,(\psi) = \mathit{TerOcc}\,(\psi')$.

The multiset $\mathit{VarOcc}\,(\psi)$ *of variable occurrences in* ψ is obtained from $\mathit{TerOcc}\,(\psi)$ by removing all non-variable terms.

The multiset $\mathit{FlaOcc}\,(\psi)$ *of formula occurrences in* ψ is inductively defined as follows:

- If ψ is an atomic formula, then $\mathit{FlaOcc}\,(\psi) = \{\psi\}$.
- If $\psi \equiv \neg\psi'$, then $\mathit{FlaOcc}\,(\psi) = \{\psi\} \cup \mathit{FlaOcc}\,(\psi')$.
- If $\psi \equiv \psi_1 \wedge \psi_2$, then $\mathit{FlaOcc}\,(\psi) = \{\psi\} \cup \mathit{FlaOcc}\,(\psi_1) \cup \mathit{FlaOcc}\,(\psi_2)$.
- If $\psi \equiv \exists x : \mathbf{W}.\psi'$, then $\mathit{FlaOcc}\,(\psi) = \{\psi\} \cup \mathit{FlaOcc}\,(\psi')$.

□

We use squared brackets when defining multisets by enumeration of its elements. For example, $[a, a, b]$ is the multiset containing twice the element a and once the element b.

Example 12 Let $\psi \equiv \big(\mathit{Function}\,(u) = \text{“staff''} \vee \mathit{Level}\,(u) \leq 2\big) \rightarrow \mathit{Function}\,(u) \neq$ “teacher”. Then

- $\mathit{VarOcc}\,(\psi) = [u,\, u,\, u]$.
- $\mathit{TerOcc}\,(\psi) = \left[\begin{array}{c} u,\, u,\, u, \text{“staff''}, 2, \text{“teacher''} \\ \mathit{Function}\,(u)\,, \mathit{Function}\,(u)\,, \mathit{Level}\,(u) \end{array}\right]$.
- $\mathit{FlaOcc}\,(\psi) = \left[\begin{array}{c} \mathit{Function}\,(u) = \text{“staff''},\, \mathit{Level}\,(u) \leq c_2, \\ \mathit{Function}\,(u) \neq \text{“teacher''}, \\ \mathit{Function}\,(u) = \text{“staff''} \vee \mathit{Level}\,(u) \leq c_2,\, \psi \end{array}\right]$.

□

Definition 13 For a formula ψ, an element of any of the multisets *VarOcc* (ψ), *TerOcc* (ψ), *FlaOcc* (ψ) is called *an occurrence in* ψ. A formula φ *occurs in* another formula ψ if $\varphi \in$ *FlaOcc* (ψ). A formula φ is a *subformula* of another formula ψ if φ occurs in ψ. □

Observe that a variable, a term, or a formula may have *multiple* occurrences in a formula.

We now define the concept of a "free variable". Intuitively, a variable x is a free variable of a formula ψ if it occurs in ψ, but not within the range of a quantifier of x.

Definition 14 *The set of free variables of a formula* ψ is denoted by $FV(\psi)$ and is inductively defined as follows:

- If ψ is an atomic formula, then $FV(\psi)$ is the set of all variables occurring in ψ.
- If $\psi \equiv \neg\psi'$, then $FV(\psi) = FV(\psi')$.
- If $\psi \equiv \psi_1 \wedge \psi_2$, then $FV(\psi) = FV(\psi_1) \cup FV(\psi_2)$.
- If $\psi \equiv \exists x : \mathbf{W}.\psi'$, then $FV(\psi) = FV(\psi') \setminus \{x\}$.

A variable x *is a free variable of a formula* ψ if $x \in FV(\psi)$. □

Sometimes it is important to emphasize the free variables of a formula ψ. In those cases, instead of ψ, we write $\psi(x_1, \ldots x_n)$, where $\{x_1, \ldots, x_n\} = FV(\psi)$.

We measure the complexity of our algorithms using a notion of "size" for formulas. We will later see that formulas of "large" size increase the time and space required by our algorithms. In this thesis, we define the size as the total number of occurrences of all atomic formulas.

Definition 15 The *size* of a formula φ is denoted by $|\varphi|$ and is defined inductively as follows:

- If t_1 and t_2 are terms, then $|t_1 = t_2| = 1$.
- If $t_1, \ldots, t_n$ are terms and R is a relation symbol, then we define $|R(t_1, \ldots, t_n)| = 1$.
- If φ_1 and φ_2 are formulas, then $|\varphi_1 \wedge \varphi_2| = |\varphi_1| + |\varphi_2|$.
- If φ is a formula, then $|\neg\varphi| = |\varphi|$.
- If φ is a formula and x is a variable of type $\mathbf{W}$, then $|\exists x : \mathbf{W}.\varphi| = |\varphi|$.

□

Observe that, for a formula φ, its size equals the size of the multiset of all occurrences of atomic formulas in φ. Observe also that our notion of size does not take into account the presence of the Boolean operator $\neg$ nor the presence of quantifiers. This is because the formulas we encounter in this thesis rarely use quantifiers or negation, so they have a negligible impact on our algorithm's complexities.

Example 16

- $\left|\mathit{Function}\left(u_1\right) = \text{“student''}\right| = 1$.
- $\left|\mathit{Function}\left(u_1\right) = \text{“student''} \wedge \mathit{Level}\left(u_2\right) \leq 3\right| = 2$.
- $\left|\mathit{Function}\left(u_1\right) = \text{“student''} \wedge \mathit{Function}\left(u_1\right) = \text{“student''}\right| = 2$. This is because

$$\begin{aligned}&\left|\mathit{Function}\left(u_1\right) = \text{“student''} \wedge \mathit{Function}\left(u_1\right) = \text{“student''}\right|\\ &= \left|\left[\mathit{Function}\left(u_1\right) = \text{“student''}, \mathit{Function}\left(u_1\right) = \text{“student''}\right]\right|\\ &= 2.\end{aligned}$$

□

2.2.2 Semantics

Consider the formula $\varphi \equiv \forall u : \textbf{USERS}.\, \mathit{Level}\left(u\right) \leq 3$. The reader familiar with logic knows that this formula means that every user's level is at most 3. However, with our current definitions, this formula is just a sequence of symbols constructed according to some rules. We recall here how to give meaning to first-order formulas. Defining what a formula means helps us to decide whether a formula holds or not for an organization. For instance, in the organization from Example 1, the formula φ holds, as every user's level is not larger than 3. In this case, we say that the organization in Example 1 *satisfies* φ. Observe that not every organization satisfies φ, as one can conceive organizations where users may have higher levels.

In this section, we formalize the notion of an "organization" using the standard concept of a "structure". Afterwards, we explain how to give meaning to first-order formulas. Finally, we define the notion of "satisfaction", which plays a crucial role in this thesis. Indeed, the problems solved by FORBAC, Rhapsody, and Unicorn can be cast as the problem of finding a structure that satisfies a given formula or as the problem of finding a formula that is satisfied by a given structure.

Giving meaning to a signature's symbols

Definition 17 Let Σ be a signature. A Σ-*structure* is a pair $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ fulfilling the following.

- $\mathfrak{S}$ is a function mapping each sort S in Σ to a finite non-empty set $\mathsf{S}^{\mathfrak{S}}$, called S*'s carrier set*. $\mathfrak{S}$ must map $\mathbf{BOOL}$, $\mathbf{INT}$, $\mathbf{STR}$, $\mathbf{2^{INT}}$, $\mathbf{2^{STR}}$ to the sets of Boolean values, integers, strings, finite subsets of integers, and finite subsets of strings, respectively.
- $\mathfrak{I}$ is a function mapping (i) each relation symbol R in Σ of type $\mathsf{S}_1 \times \ldots \times \mathsf{S}_k$ to a relation $R^{\mathfrak{I}} \subseteq \mathsf{S}_1^{\mathfrak{S}} \times \ldots \times \mathsf{S}_k^{\mathfrak{S}}$ and (ii) each function symbol f in Σ of type $\mathsf{S}_1 \times \ldots \times \mathsf{S}_{k-1} \to \mathsf{S}_k$ to a function $f^{\mathfrak{I}} : \mathsf{S}_1^{\mathfrak{S}} \times \ldots \times \mathsf{S}_{k-1}^{\mathfrak{S}} \to \mathsf{S}_k^{\mathfrak{S}}$.

For any function or relation symbol W in Σ, we call $W^{\mathfrak{I}}$, $\mathbb{K}$*'s interpretation of* W. The function $\mathfrak{I}$ is called an *interpretation function*. □

When Σ is irrelevant or clear from the context, we simply say structure instead of Σ-structure. We denote elements of carrier sets with small serif letters: a, b, ...

Example 18 Recall the signature Σ_0 from Example 3. We now define the organization from Example 1 as a Σ_0-structure $\mathbb{K}_0 = (\mathfrak{S}_0, \mathfrak{I}_0)$ defined as follows.

- $\mathbf{USERS}^{\mathfrak{S}_0} = \left\{ \mathsf{u}_{i,j} \mid 0 \leq i \leq 2, 0 \leq j \leq 8 \right\}$.
- $\mathbf{PERMS}^{\mathfrak{S}_0} = \{\mathsf{p}\}$.
- $\leq^{\mathfrak{S}_0}$ is the standard order relation on integers.
- For $\mathsf{u}_{i,j} \in \mathbf{USERS}^{\mathfrak{S}_0}$, $\mathit{Function}^{\mathfrak{I}_0}\left(\mathsf{u}_{i,j}\right) = \begin{cases} \text{"staff"} & \text{if } i = 0, \\ \text{"student"} & \text{if } i = 1, \text{ and} \\ \text{"teacher"} & \text{if } i = 2. \end{cases}$
- For $\mathsf{u}_{i,j} \in \mathbf{USERS}^{\mathfrak{S}_0}$, $\mathit{Level}^{\mathfrak{I}_0}\left(\mathsf{u}_{i,j}\right) = \begin{cases} 1 & \text{if } 0 \leq j < 3, \\ 2 & \text{if } 3 \leq j < 6, \text{ and} \\ 3 & \text{if } 6 \leq j < 9. \end{cases}$
- $\text{"staff"}^{\mathfrak{I}_0} = \text{"staff"}$, $\text{"student"}^{\mathfrak{I}_0} = \text{"student"}$, $\text{"teacher"}^{\mathfrak{I}_0} = \text{"teacher"}$.
- $1^{\mathfrak{I}_0} = 1$, $2^{\mathfrak{I}_0} = 2$, $3^{\mathfrak{I}_0} = 3$.

□

We defined the structure $\mathbb{K}_0$ in a way that fit our purpose of modeling the organization from Example 1. We assume that $\mathsf{u}_{i,j}$, for $0 \leq i < 3$ and $0 \leq j < 9$, denotes the user at the $(i+1)$-th row and $(j+1)$-th column, when seeing the arrangement of users in Figure 2.1 as a 3×9 matrix.

One could also have defined a structure that interprets the symbols in Σ_0 in a very different way. The next example illustrates this.

Example 19 We define a Σ_0-structure $\tilde{\mathbb{K}}_0 = \left(\tilde{\mathfrak{S}}_0, \tilde{\mathfrak{I}}_0\right)$ as follows.

- $\mathsf{USERS}^{\tilde{\mathfrak{S}}_0} = \{\mathsf{u}\}$.
- $\mathsf{PERMS}^{\tilde{\mathfrak{S}}_0} = \{\mathsf{p}\}$.
- $\leq^{\tilde{\mathfrak{S}}_0}$ is the standard order relation on integers.
- $\mathit{Function}^{\tilde{\mathfrak{I}}_0}(\mathsf{u}) = ``\mathsf{staff}"$.
- $\mathit{Level}^{\tilde{\mathfrak{I}}_0}(\mathsf{u}) = 1$.
- $``\mathsf{staff}"^{\tilde{\mathfrak{I}}_0} = ``\mathsf{staff}"$, $``\mathsf{student}"^{\tilde{\mathfrak{I}}_0} = ``\mathsf{student}"$, $``\mathsf{teacher}"^{\tilde{\mathfrak{I}}_0} = ``\mathsf{teacher}"$.
- $1^{\tilde{\mathfrak{I}}_0} = 1$, $2^{\tilde{\mathfrak{I}}_0} = 2$, $3^{\tilde{\mathfrak{I}}_0} = 3$.

□

Observe that in our examples, $``\mathsf{staff}"^{\mathfrak{I}_0} = ``\mathsf{staff}"^{\tilde{\mathfrak{I}}_0} = ``\mathsf{staff}"$. However, it is possible to define a strange structure $\left(\tilde{\mathfrak{S}}, \tilde{\mathfrak{I}}\right)$ where $``\mathsf{student}"^{\tilde{\mathfrak{I}}} = ``\mathsf{teacher}"$. Such structures never appear and are not relevant in this thesis.

Giving meaning to terms and formulas

Definition 17 explains how structures give meaning to a signature's symbols. We now explain in Definitions 20 and 24 how they also give meaning to terms and formulas.

Definition 20 Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a structure. A *substitution* σ is a function mapping each variable x of type W to an element in $\mathsf{W}^{\mathfrak{S}}$.

Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a structure, let σ be a substitution, and let t be a term. *The interpretation of t under $(\mathfrak{I}, \sigma)$* is denoted by $[\![t]\!]_{(\mathfrak{I},\sigma)}$ and is inductively defined as follows:

- If t is a variable x of type W, then $[\![t]\!]_{(\mathfrak{I},\sigma)} = \sigma(x)$.
- If t is a term of the form $f(t_1, \ldots, t_n)$, then

$$[\![t]\!]_{(\mathfrak{I},\sigma)} = f^{\mathfrak{I}}\left([\![t_1]\!]_{(\mathfrak{I},\sigma)}, \ldots, [\![t_n]\!]_{(\mathfrak{I},\sigma)}\right).$$

□

Example 21 We use the notation from Example 18. Let σ be a substitution mapping the variable u to $u_{0,0}$. Then $[\![\mathit{Function}(u)]\!]_{(\mathfrak{I}_0,\sigma)} = \mathsf{staff}$ and $[\![\mathit{Level}(u)]\!]_{(\mathfrak{I}_0,\sigma)} = 1$ □

Definition 22 Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a structure and assume that $x_1, \ldots, x_n$ are of types $\mathsf{W}_1 \times \ldots \times \mathsf{W}_n$, respectively. *The satisfaction relation* $\vDash$ is a ternary relation on structures, substitutions, and formulas that is inductively defined as follows:

$$
\begin{array}{lll}
\mathbb{K}, \sigma \models t_1 = t_2 & \text{if} & [\![t_1]\!]_{(\mathfrak{I},\sigma)} = [\![t_2]\!]_{(\mathfrak{I},\sigma)}. \\
\mathbb{K}, \sigma \models R\,(t_1, \ldots, t_n) & \text{if} & R^{\mathfrak{I}}\left([\![t_1]\!]_{(\mathfrak{I},\sigma)}, \ldots, [\![t_n]\!]_{(\mathfrak{I},\sigma)}\right). \\
\mathbb{K}, \sigma \models \neg\varphi & \text{if} & \mathbb{K}, \sigma \not\models \varphi. \\
\mathbb{K}, \sigma \models \varphi_1 \wedge \varphi_2 & \text{if} & \mathbb{K}, \sigma \models \varphi_1 \text{ and } \mathbb{K}, \sigma \models \varphi_2. \\
\mathbb{K}, \sigma \models \exists x : \mathbf{W}.\varphi & \text{if} & \text{there is } \mathsf{w} \in \mathbf{W}^{\mathfrak{S}} \text{ such that } \mathbb{K}, \sigma\,[x \mapsto \mathsf{w}] \models \varphi.
\end{array}
$$

Here, $\sigma\,[x \mapsto \mathsf{w}]$ is the substitution that maps x to w and any other variable y to $\sigma\,(y)$. □

If φ has no free variables, then we usually write $\mathbb{K} \models \varphi$ instead of $\mathbb{K}, \sigma \models \varphi$, as the fact that $\mathbb{K}, \sigma \models \varphi$ holds does not depend on σ in this case. Additionally, we say that $\mathbb{K}$ *satisfies* φ whenever $\mathbb{K} \models \varphi$.

Example 23 Let $\mathbb{K}_0$ be the structure from Example 18 and let σ be a substitution mapping the variables u and p to $\mathsf{u}_{0,0}$ and $\mathsf{p}_{0,0}$, respectively. Then

- $\mathbb{K}_0, \sigma \models \mathit{Function}\,(u) = \text{“staff”}$.
- $\mathbb{K}_0, \sigma \models \mathit{Level}\,(u) \leq 2$.
- $\mathbb{K}_0, \sigma \models \neg\exists \ell : \mathbf{INT}.\, \exists \ell' : \mathbf{INT}.\, \ell \neq \ell' \wedge \mathit{Level}\,(u) = \ell \wedge \mathit{Level}\,(u) = \ell'$.
- $\mathbb{K}_0 \models \forall u : \mathbf{INT}.\, 0 \leq \mathit{Level}\,(u) \leq 2$.

□

Definition 24 Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a structure and let $\varphi\,(x_1, \ldots, x_n)$ be a formula. Assume that φ's free variables are of type $\mathbf{W}_1, \ldots, \mathbf{W}_n$, respectively. *The interpretation of* φ *under* $\mathfrak{I}$ is the relation $\varphi^{\mathfrak{I}} \subseteq \mathbf{W}_1^{\mathfrak{S}} \times \ldots \times \mathbf{W}_n^{\mathfrak{S}}$ such that $(\mathsf{w}_1, \ldots, \mathsf{w}_n) \in \varphi^{\mathfrak{I}}$ iff $\mathbb{K}, \sigma \models \varphi$, where σ is any substitution such that $\sigma\,(x_i) = \mathsf{w}_i$, for $i \leq n$. □

Example 25 Let $\mathbb{K}_0$ be the structure from Example 18. Let also $\varphi_1 \equiv \mathit{Level}\,(u) \neq \mathit{Level}\,(u)$ and $\varphi_2 \equiv \mathit{Function}\,(u) = \text{“student”} \wedge \mathit{Level}\,(u) \leq 2$. Then $\varphi_1^{\mathfrak{I}_0} = \emptyset$ and $\varphi_2^{\mathfrak{I}_0} = \{\mathsf{u}_{1,0}, \mathsf{u}_{1,1}, \ldots, \mathsf{u}_{1,5}\}$. □

2.3 Access control

We now provide background on access control. We focus here on two of the most popular access control paradigms: RBAC (role-based access control) and ABAC (attribute-based access control).

2.3.1 RBAC: Role-based access control

One of the first mechanisms for specifying access control policies were *access control matrices* [24]. These matrices can be understood as binary relations between the users and permissions in an organization. In large organizations, however, access control matrices become impractical. Whenever a new user joins the organization, the policy administrator must decide for each permission, whether it should be assigned to the user. Also, whenever there are organizational changes, the administrator needs to review the matrix and decide how to change the permissions that have been assigned to each user.

To facilitate the maintenance of policies during organizational changes, *RBAC (Role-Based Access Control) policies* were proposed [52]. The key insight in RBAC is that several users in an organization require very similar permissions. For example, in a hospital, all nurses have essentially the same permissions and the same can be said of all administrative staff and all doctors. Hence, an RBAC policy decouples the assignment of permissions to users into two parts. It first assigns roles to users and then permissions to roles. Users who do similar tasks in an organization are assigned a same role and this role is assigned all permissions that those users need. The number of roles is substantially smaller than the number of users or permissions, which facilitates the maintenance of policies during organizational changes. For example, whenever a new user joins the organization, the policy administrator only needs to decide what roles shall be assigned to the new user.

Role-based access control has been one of the most popular ways to specify access control policies in large organizations. We now recall how RBAC policies are defined.

Definition 26 An *RBAC policy* is a tuple $\pi = (U, P, Ro, Ua, Pa)$. U and P are two non-empty sets denoting, respectively, the sets of users and permissions in an organization. Ro is a set denoting the roles in the organization. $Ua \subseteq U \times Ro$ and $Pa \subseteq Ro \times P$ are binary relations. The policy π *assigns* a permission p to a user u if there is a role $r \in Ro$ such that $(u, r) \in Ua$ and $(r, p) \in Pa$. □

Example 27 Figure 2.2 shows an RBAC policy. It consists of 6 users, 2 roles, and 3 permissions. □

Observe that our formalization of RBAC policies does not use first-order logic at all. However, we will see later in Chapter 3, that many extensions that have been proposed for RBAC are actually easier to formalize using first-order logic.

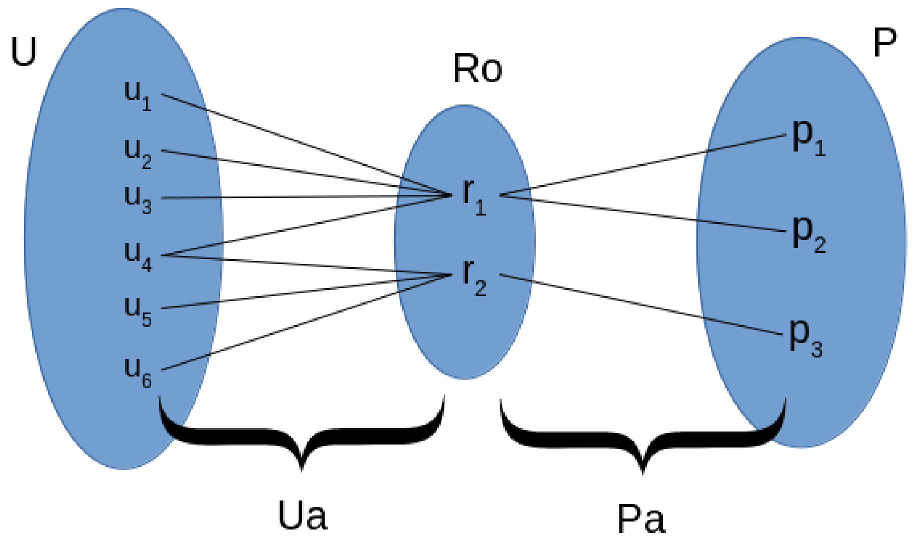


Figure 2.2: An RBAC policy with 6 users, 2 roles, and 3 permissions.

2.3.2 ABAC: Attribute-based access control

RBAC suffers from an issue called the *role explosion* problem, which we illustrate with the following example.

Example 28 Assume we want to develop an RBAC policy for a university. In this organization, each user denotes a student and each permission denotes a course. We assume that there are 10,000 students and 100 courses. The RBAC policy should authorize each student to view the contents of only those courses in which the student is enrolled.

Two different students rarely enroll in exactly the same set of courses. In the worst case, the RBAC policy would require one role for each student. Hence, an RBAC policy can become as impractical as an access control matrix in this scenario as the number of roles is substantially large. □

To solve this problem, *ABAC (Attribute-Based Access Control) policies* were proposed. An ABAC policy is a set of *rules*, where each rule describes a set of conditions based on the *attribute values* of *users* and *permissions* [77, 69]. An example of an ABAC policy can be seen in stores that sell alcoholic beverages only to people who are at least 21 years old. The user (i.e., the client) is granted permission to buy alcohol only if his age attribute has a value greater than or equal to 21. Observe here that the policy grants permissions according the user's attribute values.

We now give a definition of ABAC policies based on first-order logic and then show how to formalize an ABAC policy for Example 28.

Definition 29 An *ABAC policy* is a first-order formula $\varphi(u,p)$ built from the following BNF grammar:

$$\begin{array}{rlll} t & ::= f(u) \mid f(p) \mid \mathsf{c} & /* & \texttt{Terms} \quad */ \\ \alpha & ::= t = t \mid Q(t,t) & /* & \texttt{Atoms} \quad */ \\ r & ::= \alpha \mid \alpha \wedge r & /* & \texttt{Rules} \quad */ \\ \pi & ::= r \vee \ldots \vee r. & /* & \texttt{Policies} \quad */ \end{array}$$

Here, u and p range over variables of sorts **USERS** and **PERMS**, respectively, whereas f, c, and Q range over unary function, constant, and binary relation symbols of the appropriate sorts, respectively. We restrict atoms, rules, and policies to have at most two free variables of sorts **USERS** and **PERMS**, respectively.

We call *attribute* any unary function symbol with domain sort **USERS** or **PERMS**.

Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a structure and $\varphi(u,p)$ be an ABAC policy. For an attribute f, a user u, and a permission p, we call $f^{\mathfrak{I}}(\mathsf{u})$ the *attribute value of* u for f. Similarly, we call $f^{\mathfrak{I}}(\mathsf{p})$ the *attribute value of* p for f. We say that φ assigns a permission $\mathsf{p} \in \mathbf{PERMS}^{\mathfrak{S}}$ to a user $\mathsf{u} \in \mathbf{USERS}^{\mathfrak{S}}$ if $\mathbb{K}, \sigma_{\mathsf{u},\mathsf{p}} \models \varphi$, where $\sigma_{\mathsf{u},\mathsf{p}}$ is the substitution that maps u and p to u and p, respectively. $\square$

In the context of ABAC policies, we refer to atomic formulas as *atoms*. We call *rule* any conjunction of atoms. Observe that ABAC policies are disjunctions of rules.

Example 30 The RBAC policy from Example 28 can be specified with the ABAC policy $\mathit{courseId}(p) \in \mathit{courses}(u)$, where $\mathit{courseId} : \mathbf{PERMS} \to \mathbf{STR}$ represents a function that assigns an identifier to each course and $\mathit{courses} : \mathbf{USERS} \to \mathbf{2}^{\mathbf{STR}}$ represents a function that assigns to each student the identifiers of the courses in which the student is enrolled. Observe that this policy requires less administration effort in comparison to an RBAC policy. $\square$

Example 31 We give an ABAC policy that formalizes the access control policy from Example 1.

$$\begin{gathered}\left(\mathit{Function}(u) = \text{“staff''} \wedge \mathit{Level}(u) \leq 2\right) \vee \\ \left(\mathit{Function}(u) = \text{“teacher''} \wedge \mathit{Level}(u) \geq 2\right).\end{gathered}$$

$\square$

2.4 Policy analysis

The specification and maintenance of access control policies is a manual task. In large organizations, where there are thousands of users and thousands of permissions, it is likely that organizational security policies are complex and, hence, the implemented access control policy does not properly act as dictated by the organizational security policy. As a result, when evaluating a request (i.e., a user-permission pair), the access control policy may make two types of mistakes: (i) denying a legitimate request or (ii) authorizing a request that, according to the organizational security policy, should not be authorized. We call these an *incorrect denial* and an *incorrect authorization*, respectively. Incorrect denials prevent users from performing their tasks in the organization. Incorrect authorizations are especially dangerous as they violate the principle of least privilege and can be abused by users. Reports from Verizon in 2017 [50] show that privilege abuse is still a major issue for many companies, especially for banking and healthcare companies, where data is often highly sensitive.

To mitigate the risk of incorrect authorizations and denials *policy analyzers* have been proposed. We give an informal overview here and refer to Chapter 3 for a more rigorous presentation. Policy analyzers are tools that receive as input a policy and a security property that the policy should fulfill. A policy analyzer transforms these inputs into a formula ψ that is satisfied only by structures that encode a violation of the security property. Afterwards, the policy analyzer searches for a structure $\mathbb{K}$ that satisfies ψ. If found, the policy analyzer builds a counterexample from $\mathbb{K}$ that illustrates to the policy administrator how the policy fails to fulfill the security property.

Example 32 Let us consider the organization and the policy presented in Example 1, which were formalized in Examples 18 and 31. Recall that the organizational security policy dictates that every user with level 2 is assigned the permission. We wish to verify if the implemented access control policy (i.e., the ABAC policy φ from Example 31) fulfills that. We can achieve this by specifying two formulas ψ_1 and ψ_2. The formula ψ_1 is

$$\exists u : \mathsf{USERS}. \mathit{Level}\left(u\right) = 2 \land \neg\varphi\left(u\right). \tag{2.3}$$

Observe this formula is satisfied by a structure $\mathbb{K}$ iff there is a user with level 2 who is not assigned the permission. Finally, the formula ψ_2 is a formula that is only satisfied by $\mathbb{K}_0$, the structure representing the organization. We do not give the details of ψ_2, as they are quite technical, but computing such a formula is a standard procedure in first-order logic [14].

Let now $\psi \equiv \psi_1 \land \psi_2$. Observe that ψ is satisfied only by $\mathbb{K}_0$ and only if there is a user whose level is 2 and is not granted the permission by φ. One can see that $\mathbb{K}_0$ indeed satisfies ψ. Therefore, the implemented policy does not act as dictated by the organizational security policy. □

Admittedly, we could have discovered the deficiencies of the implemented policy with a manual inspection and without going through the hassle of building the formulas ψ_1 and ψ_2. In practice, however, manual inspections do not scale, as organizations have thousands of users and permissions, each with thousands of different attribute values. In these scenarios, automated tools like policy analyzers are more efficient when it comes to verify that policies fulfill security properties. Moreover, in our case with the organization from Example 1, policy analyzers can point to a user u in $\mathbb{K}_0$ who has level 2 and is not assigned the permission by φ. We elaborate on this in Chapter 3.

The satisfiability problem for first-order logic.

In Example 18, the problem of verifying whether the implemented access control policy fulfills organizational security policy reduced to check if there exists a structure that satisfies the first-order formula ψ. We will see in Chapter 3 that most problems in policy analysis reduce to verify if a first-order formula is *satisfiable*; that is, if there is a structure that satisfies it. The following important result illustrates one of the main problems of using first-order logic as a language for specifying policies and properties.

Definition 33 A *fragment of first-order logic* is a set S of first-order formulas. The *decidability problem for a fragment S of first-order logic* is the problem of deciding if a given formula in S is satisfiable. $\square$

Theorem 34 The decidability problem for first-order logic is undecidable.

As a consequence, there is no algorithm that can decide for every policy and every property, whether the policy fulfills the property. Fortunately, we seldom require the full expressive power of first-order logic to specify policies and properties. Previous work has shown that there exists fragments of first-order logic that, on one hand, are powerful enough to express realistic policies and properties and, on the other hand, are simple enough so that the decidability problem is tractable [66].

Many works have proposed fragments that are rich enough to express relevant concepts in first-order and also simple enough so that the decidability problem has an acceptable computational complexity [28, 13, 44]. These works also propose algorithms, called *SMT solvers*, that can decide for some of the formulas in the fragment, whether they are satisfiable [43, 46].

2.5 Policy mining

Another field that assists with the specification and maintenance of policies and mitigates the risk of incorrect authorizations and denials is *policy mining*. Algorithms from this field receive as input the current *permission assignment*, which is a relation between the set of users and the set of permissions that reflects how the implemented access control policy has assigned permissions to users. The permission assignment might be given as an access control matrix or a log of access requests showing the access decisions made for each request so far. The miner constructs a policy that is as consistent as possible with the permission assignment and can be expressed using the organization's policy language.

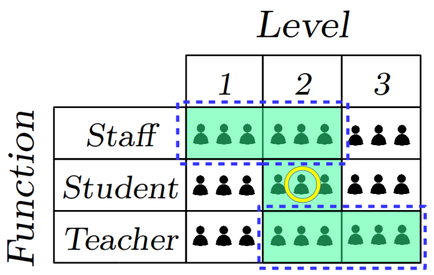
2.5.1 Applications

We present some examples from previous works of how policy miners assist the specification and maintenance of policies.

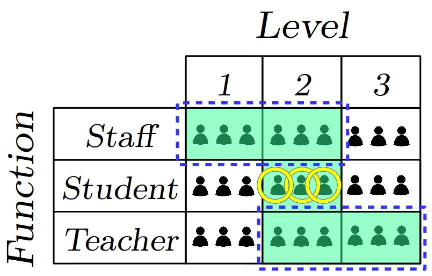
Handling incorrect denials

An incorrect denial is usually discovered when a user attempts to exercise a legitimate permission, but is forbidden by the policy. The user reports the issue to the administrator, who changes the policy in a way that the user is now assigned the permission. One way to change the policy is to add an *exceptional assignment* that grants the permission to the user in question. For example, an exceptional assignment can be, in the case of RBAC, a new role that solely assigns the permission to the user or, in the case of ABAC, a new rule achieving the same purpose. This is usually the preferred solution when neither the user nor the administrator can infer from this particular situation what exactly is wrong with the implemented access control policy. After several users experience the same problem, the administrator might be able to deduce what is wrong from the accumulated exceptional assignments and provide a general fix.

Example 35 Let us consider the case of user $\mathsf{u}_{1,3}$ in the organization presented in Example 1 and formalized as a structure in Example 18, which we illustrate in Figure 2.3a. Recall that $\mathsf{u}_{1,3}$ denotes the user in the second row and in the fourth column in the figure, when seeing the arrangement of users as a 3×9 matrix. According to the organizational security policy, $\mathsf{u}_{1,3}$ should be authorized as his level is 2. However, the implemented access control policy does not authorize him. If $\mathsf{u}_{1,3}$ requests the permission, then the request will be denied by the policy, yielding an incorrect denial. The user $\mathsf{u}_{1,3}$ contacts the policy administrator and after convincing her that his request to exercise the permission should be authorized, the administrator adds an exceptional assignment to the currently ABAC policy that authorizes $\mathsf{u}_{1,3}$ to exercise the permission. More formally, the new ABAC policy



(a) The user in the circle represents an incorrect denial. From just this incorrect denial, it is difficult to see what the implemented policy is missing with respect to the organizational security policy.



(b) The users in the circles are more examples of incorrect denials. They support the hypothesis that students with level 2 should also be assigned the permission.

Figure 2.3

is the following:

$$\begin{aligned}\big(\mathit{Function}\,(u) = \text{“staff''} \wedge \mathit{Level}\,(u) \leq 2\big) \vee \\ \big(\mathit{Function}\,(u) = \text{“teacher''} \wedge \mathit{Level}\,(u) \geq 2\big) \vee \\ \mathit{userID}\,(u) = \text{“u\_1\_3''}.\end{aligned}$$

Here, we added new symbols to the signature. *UserID* is a function symbol of type **USERS** $\rightarrow$ **STR** that assigns a unique identifier string to each user. We also added a constant symbol “u_1_3” of type **STR** that denotes $\mathsf{u}_{1,3}$'s identifier. We assume that $\mathit{UserID}^{\mathfrak{I}_0}\,(\mathsf{u}_{1,3}) = \text{“u\_1\_3''}$.

It may not possible for the policy administrator, from just this exceptional assignment, to deduce what the implemented access control policy is missing, as several explanations may be possible. Perhaps all users whose level is 2 should also be granted the permission or perhaps all users who are students should be granted the permission instead. Even the explanation that all students with level 2 should be assigned the permission is preposterous, as there are always cases where a user must be granted a permission in exceptional circumstances that go against the organizational security policy. For example, a nurse may not be allowed to access a patient's data, but if the patient is in a life-threatening situation, the nurse may be granted exceptional but temporal access to his or her data.

Let us consider again Figure 2.3a. As time goes by, one can assume that more users who should legitimately be authorized by the policy are denied. This yields more incorrect denials that make the policy more complicated, but these incorrect denials also give insights on what the implemented access control policy is missing. Assume, for example that $\mathsf{u}_{1,3}$ and $\mathsf{u}_{1,5}$ also unsuccessfully attempt to exercise the permission and the policy administrator manually adds exceptional assignments to the policy, as shown in Figure 2.3b by the two new yellow circles. These exceptional assignments support the conjecture that the policy should also assign the permission to all users whose level is 2. This is indeed what makes the implemented policy different from the organizational security policy. □

In practice, in large organizations, there may be thousands of users and dozens of attributes, each with thousands of possible attribute values. In such cases, a manual inspection of the exceptional assignments is impractical and *policy miners* that automatically inspect these exceptions and suggest policy improvements are needed.

From the above discussion, one can see that policy miners can simplify policies that have become convoluted by the addition of several exceptional assignments. Even more importantly, the miner can discover what the implemented policy is missing and this can prevent future incorrect denials. Indeed, for a mined policy to be useful, it must be able to compute policies that minimize the risk of incorrect denials and authorizations. This ability is called *generalization* and is formalized later in Chapters 4 and 5.

Other use cases

Another important use case for policy mining concerns organizational change. For example, when an organization acquires a smaller organization, the smaller organization's access control policy must be translated into the larger organization's policy language and merged with the organization's access control policy. As another example, when an organization wants to reimplement its access control policy in a different language, it must translate its own access control policy. Policy miners offer a practical solution to these kinds of problems, as they can mine a policy in the target language from the current permission assignment. For the mined policy to be useful in these cases, it must also have *low complexity*; that is, it must be as simple as possible so that a human can review it and validate it. We discuss complexity metrics in Chapters 4 and 5.

Another use case for policy mining is to identify policies that are *overly permissive* [39]. That is, policies that assign to users more permissions than what they actually need. Overly-permissive policies are prone to have incorrect authorizations. By mining from a log that has recorded the permissions exercised by the users so far, one can compute a policy that better reflects what each user needs. This policy can be compared using policy comparison tools [55] with the currently implemented access control policy to discover unnecessary assignments of permissions.

2.5.2 Quality criteria for policy miners

Policy miners can be regarded as machine-learning algorithms. Therefore, they are evaluated by the quality of the policies they mine, and here three criteria are used:

Generalization [57, 39, 103] A mined policy should not only authorize requests in a way that is consistent with the given permission assignment. It must also correctly decide what *other* permissions should be granted to users who perform similar functions in the organization. This is particularly important when mining from logs. For example, if most of students in a university have requested and been granted access to a computer room, then the mined policy should grant all students access to the computer room rather than just to those who have requested access to it. One popular machine-learning method to evaluate generalization is cross-validation [60, 23], which we recall in Chapter 4.

Complexity [30, 133] A mined policy should not be unnecessarily complicated, as the policies are usually reviewed and audited by humans. This is especially important when mining with the goal of refactoring an existing policy or migrating to a new policy language. One of the simplest and most popular complexity metrics for ABAC policies is the size (Definition 15). Other works have defined its own metrics to quantify complexity [39, 132, 133, 58], not only for ABAC but also for other policy languages. We will illustrate some of these metrics in Chapter 5.

Precision [39] When mining from a log, we need to consider another quality criterion. We will later see in Chapter 4 that it is possible to mine a policy that generalizes well and has low complexity, but *authorizes significant sets of requests for which the log provides no evidence.* We call such a policy an *overly-permissive policy*. To quantify over-permissiveness we use the standard precision metric [111]. Intuitively, the precision of a policy measures the ratio of the number of requests authorized by the access control policy to the number of requests that should be authorized according to the organizational security policy. We discuss this and other metrics to quantify over-permissiveness in Chapter 4.

2.6 Conclusion

We now know how to formalize the most relevant access control concepts using many-sorted first-order logic. We have also learned the essentials of policy analysis and policy mining. The reader interested in policy analysis can continue with Chapter 3. There we present FORBAC, an extension of

RBAC that is expressive enough to formalize a wide range of access control policies, while at the same time being simple enough so that relevant policy analysis queries are in the complexity class NP. The reader interested in policy mining can continue with Chapters 4 and 5, where we present Rhapsody and Unicorn. Rhapsody is the first algorithm for mining ABAC policies that guarantees to mine from sparse logs the set of all significant, reliable, and succinct rules. Unicorn is a universal method for building policy miners. Using Unicorn, we have built competitive policy miners for a wide variety of policy languages. In particular, the policy miner for ABAC built with Unicorn outperforms Rhapsody. Moreover, using Unicorn, we have built the first policy miners for XACML and RBAC with spatio-temporal constraints. In spite of active research in policy mining, no miner was known for any of these languages for several years.

Chapter 3

FORBAC: First-order Role-based Access Control

3.1 Introduction

RBAC [52] is one of the most popular access control paradigms. However, different extensions for RBAC have been proposed in the last decades that aim to make RBAC capable of expressing more complex policies, e.g. [63, 70, 78, 79, 89, 94]. The added expressive power has also motivated research on policy analysis for these extensions, e.g. [8, 15, 53, 122].

Most of these RBAC extensions build upon first-order logic. However, deciding whether a formula is valid in first-order logic is known to be undecidable [48]. This has serious consequences for policy analysis for these extensions, as its computational complexity is directly influenced by the computational complexity of deciding whether formulas in the RBAC extension's syntax are valid. As a result, researchers have made efforts to discover fragments (i.e., sets of first-order formulas) that are rich enough to express relevant policy properties while, at the same time, simple enough so that the complexity of deciding whether a formula is satisfiable is tractable [28, 66].

A closer look at the syntax of these RBAC extensions shows that only a fragment of first-order logic is sufficient to capture the syntax of those extensions. For example, their syntax does not involve disjunction or arbitrary quantifier alternation [66, 16]. By restricting the syntax to just a fragment of first-order logic, we hope to propose a language that is expressive enough to specify RBAC policies in these extensions and that is simple enough to keep the complexity of policy analysis in NP. Although policy analysis should ideally have a polynomial-time complexity, we argue later that policy analysis for any sufficiently expressive policy language is NP-hard.

In this chapter, we propose FORBAC, an extension of RBAC that strikes that desired balance between expressiveness in policy specification and efficiency in policy analysis. FORBAC incorporates the main features of different RBAC extensions from the literature, e.g. [7, 63, 70, 78]. Moreover, we prove that, for a large class of properties, the complexity of policy analysis in FORBAC NP-complete. To verify properties of FORBAC policies, we reduce them to satisfiability modulo theories and use the SMT-solver Z3 [43].

To illustrate FORBAC's expressive power and its ability to efficiently analyze policies, we conducted a case study on a major European bank, approximately containing 50,000 users, 5,400 roles, and 57,000 permissions. We were able to express 10 of the bank's most complex access-control policies in FORBAC and we verified a variety of relevant policy properties. Using Z3, most of the properties were verified within seconds.

The remainder of this chapter is organized as follows. In Section 3.2 we establish FORBAC's expressiveness requirements for FORBAC. In Section 3.3, we define FORBAC's syntax and semantics and in Section 3.4 we show how to specify policy properties for FORBAC policies. In Section 3.5, we present the results from our case study with the bank. In Section 3.6, we discuss related work and in Section 3.7 we draw conclusions.

3.2 Requirements for FORBAC

FORBAC is an RBAC extension that strikes a balance among the following three factors:

- An expressive language for specifying RBAC policies.
- An expressive language for specifying properties of RBAC policies.
- A low complexity for verifying policies against properties.

In the remainder of this section, we discuss language requirements and complexity classes for policy analysis.

3.2.1 Requirements for policy specification

Numerous extensions for RBAC have been proposed and the syntax of many of them (e.g. [70, 78, 94]) includes fragments of first-order logic that make policy analysis undecidable, or at best highly intractable. In the following, we review some of their features in order to elicit the central requirements for an expressive RBAC extension. Based on these requirements, we present in Section 3.3 a fragment of first-order logic that is simple, but expressive enough to formalize realistic policies.

Attributes A common feature of RBAC extensions is the association of attributes to users, roles, and permissions. This stems from the need to add fine-grained access control to RBAC. For example, a user in a physician role should be authorized to access patient information, but only for those patients he supervises. Instead of defining one role for every subset of patients, an attribute is added to the role that specifies the set of patients under the physician's supervision.

Role and permission assignments specified in first-order logic Another common feature of RBAC extensions is the use of rules to assign roles to users and permissions to roles. This feature is motivated by the difficulty of manually administering these relations in large environments where users' and permissions' attribute values frequently change. Many RBAC extensions, such as [76, 63, 70], use first-order logic to specify user-role and role-permission assignment relations. However, they do not limit the fragment of first-order logic used for these specifications.

We propose restrictions on the first-order fragment we use in FORBAC. For instance, we do not allow the arbitrary nesting of quantifiers. In practice, access control permissions simply require the presence or absence of values in the user's, role's, and permission's attributes. This is reflected in the syntax of logic-based policy languages that have been used in practice. For example, Lithium [66] forbids quantifier alternation and yet it can still express various parts of U.S. legislation, including fragments of the Privacy Rule, which governs access to electronic medical files, and Title 42, Chapter 7 of the U.S. Code, which determines who is eligible for Social Security. Another example is Cassandra [17], an earlier version of SecPAL, which does not allow quantifier alternation, but can express the policies for the national electronic health record system of the United Kingdom.

Numeric constraints We incorporate this kind of constraints as they often occur in authorization policies. For example, Title 29 of the U.S. Code §1181, which belongs to the HIPAA rule, says:

> A period of creditable coverage shall not be counted, with respect to enrollment of an individual under a group health plan, if, after such period and before the enrollment date, there was a 63-day period during all of which the individual was not covered under any creditable coverage.

In electronic health record systems, health organizations are authorized to request a credential asserting patient/EHR-service bindings if they can provide an RA-approved NHS health organization credential [17]. Such credentials are valid only for fixed time intervals. More generally, functions within an organization may have a limited duration. For instance, vendors may be authorized to access vendor contracts only in the second week of every quarter of every year, and vendor contracts must be submitted within two weeks of that time [22]. Such numerical constraints can usually be expressed as inequalities between two integer values.

This concludes the requirements for our language for specifying RBAC policies. There are other access control features that have received attention in the literature that we have not included as requirements for our language. These include role hierarchies [115], delegation [12], and separation-of-duty constraints [5]. We leave these as future work and focus on the core features explained above.

3.2.2 A complexity class for policy analysis

Ideally, the time complexity of policy analysis should be polynomial. However, we argue that it is NP-hard for any sufficiently expressive policy language. To support this, we present a simple policy language F that can be embedded into languages have been successfully applied in industry like Margrave [106] and Gem-RBAC-DSL [18] and show that checking even the simple query of whether every access request is permitted in a given policy in F is NP-hard.

Definition 36 (Syntax of *F*) Let Σ be a signature where all its relation symbols are binary. Let F be the set of formulas of the form $P_1(u,p) \wedge \ldots \wedge P_k(u,p) \wedge \neg Q_1(u,p) \wedge \ldots \wedge \neg Q_n(u,p)$. Here, $k, n \geq 0$ and u and p are variables of types $\textsf{\textbf{USERS}}$ and $\textsf{\textbf{PERMS}}$, respectively. P_i and Q_j, for $i \leq k$ and $j \leq n$, are binary relation symbols. A *policy in F* consists of a disjunction of the form $\varphi_1(u,p) \vee \ldots \vee \varphi_\ell(u,p)$, where φ_i, for $i \leq \ell$, is a formula in F. $\square$

Definition 37 (Semantics of *F*) Let T be a policy in F and $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a Σ-structure. For a request $(\mathsf{u}, \mathsf{p}) \in \textsf{\textbf{USERS}}^{\mathfrak{S}} \times \textsf{\textbf{PERMS}}^{\mathfrak{S}}$, we say that (u, p) is *authorized by T* if $\mathbb{K}, \sigma_{(\mathsf{u},\mathsf{p})} \models T$, where $\sigma_{(\mathsf{u},\mathsf{p})}$ is any substitution mapping u and p to u and p, respectively. $\square$

For a policy $T \equiv \varphi_1(u,p) \vee \ldots \vee \varphi_\ell(u,p)$ in F, suppose that we want to verify if every access request is authorized. This can be done by checking the validity of $\forall u : \textsf{\textbf{USERS}} \,.\, \forall p : \textsf{\textbf{PERMS}} \,.\, (\varphi_1(u,p) \vee \ldots \vee \varphi_\ell(u,p))$; that is, checking that every Σ-structure satisfies it. However, we prove in Appendix A.1 that checking this for an arbitrary T is NP-hard.

F is extremely simple. It uses only binary relation symbols and it can be embedded into state-of-the-art analysis frameworks used for analyzing realistic policies like Margrave [106] (see Section A.1 in the Appendix for details). Nevertheless, despite its low expressiveness, even basic policy analysis queries are NP-hard. For this reason, we believe P is too restrictive (unless P = NP) and we set our sights on performing policy analysis in NP.

3.3 Syntax and semantics of FORBAC

Given the requirements for our framework, we start by defining the signature for writing FORBAC-policies.

Definition 38 A *FORBAC-signature* is a signature $\Sigma = (\mathbb{S}, \mathbb{R}, \mathbb{F}, \mathbb{V})$ where:

- $\mathbb{S} = \mathbb{S}_{RBAC} \cup \{\mathsf{INT}, \mathsf{STR}, 2^{\mathsf{INT}}, 2^{\mathsf{STR}}\}$, with

$$\mathbb{S}_{RBAC} = \{\mathsf{USERS}, \mathsf{Roles}_1, \mathsf{Roles}_2, \ldots, \mathsf{Roles}_T, \mathsf{PERMS}\}\,. \tag{3.1}$$

- $\mathbb{R}$ is the set containing only the following binary relation symbols:

$$\begin{array}{ccl@{\qquad}ccl} \leq & : & \mathsf{INT} \times \mathsf{INT} & & & \\ \in_{\mathsf{STR}} & : & \mathsf{STR} \times 2^{\mathsf{STR}} & \in_{\mathsf{INT}} & : & \mathsf{INT} \times 2^{\mathsf{INT}} \\ \subseteq_{\mathsf{STR}} & : & 2^{\mathsf{STR}} \times 2^{\mathsf{STR}} & \subseteq_{\mathsf{INT}} & : & 2^{\mathsf{INT}} \times 2^{\mathsf{INT}} \end{array} \tag{3.2}$$

- $\mathbb{F} = \mathbb{A} \cup \mathbb{C}$. $\mathbb{A}$ is a set of unary function symbols. Every $f \in \mathbb{A}$ has a type $\mathbf{W}_f \to \mathbf{V}_f$, where $\mathbf{W}_f \in \mathbb{S}_{RBAC}$ and $\mathbf{V}_f \in \{\mathsf{INT}, \mathsf{STR}, 2^{\mathsf{INT}}, 2^{\mathsf{STR}}\}$. $\mathbb{C}$ is a set of constant symbols, each of some sort in $\mathbb{S}$.

$\square$

We assume that $\mathbb{C}$ contains enough constant symbols to represent all integers, strings, sets of integers, and sets of strings. For simplicity, we use $\in$ instead of $\in_{\mathsf{STR}}$ or $\in_{\mathsf{INT}}$ when it is clear from the context. We do the same with the symbol $\subseteq$. When writing formulas, we sometimes use standard abbreviations like $x \notin F(y)$ and $x \neq y$ to denote the formulas $\neg\,(x \in F(y))$ and $\neg\,(x = y)$.

The symbols in $\mathbb{A}$ with domain in $\mathcal{S}_{RBAC}$ and codomain in $\{\mathsf{INT}, \mathsf{STR}\}$ denote *single-valued attributes* and those with domain in $\mathcal{S}_{RBAC}$ and codomain in $\{2^{\mathsf{INT}}, 2^{\mathsf{STR}}\}$ denote *set-valued attributes*. We call *attribute* any single or set-valued attribute. We define $RT(\Sigma)$ as the set $\{\mathsf{Roles}_1, \mathsf{Roles}_2, \ldots, \mathsf{Roles}_T\}$ and call it *the set of role templates of* Σ [2, 63, 27, 62].

Example 39 We present a simple FORBAC signature $\Sigma_B = (\mathbb{S}, \mathbb{R}, \mathbb{F}, \mathbb{V})$ for specifying the access control policies of a bank's account administration tool. Here, $RT(\Sigma_B) = \{\mathsf{R}_{\texttt{Student}}, \mathsf{R}_{\texttt{Employee}}\}$ represents two different kinds of roles a user can be assigned: student and employee.

We define in $\mathbb{A}$ the following single-valued attributes for the sort USERS:

- *name* : $\mathsf{USERS} \to \mathsf{STR}$.
- *age* : $\mathsf{USERS} \to \mathsf{INT}$.
- *nationality* : $\mathsf{USERS} \to \mathsf{STR}$.

- *salary* : $\mathsf{USERS} \to \mathsf{INT}$.

The role template $\mathsf{R}_{\texttt{Employee}}$ has one single-valued attribute called *limit*:

- *limit* : $\mathsf{R}_{\texttt{Employee}} \to \mathsf{INT}$.

The attribute *limit* specifies the maximal amount that the role's owner is allowed to use in one transaction on her bank account.

We also have a set-valued attribute for the role template $\mathsf{R}_{\texttt{Student}}$:

- *country* : $\mathsf{R}_{\texttt{Student}} \to 2^{\mathsf{STR}}$

It specifies in which countries a user with a student role can order transactions. Having this attribute only for one role template shows why two different sorts of role templates are needed, since this restriction will only apply to student roles, but not for employee roles.

The sort PERMS is provided with three single-valued attributes:

- *action* : $\mathsf{PERMS} \to \mathsf{STR}$
- *amount* : $\mathsf{PERMS} \to \mathsf{INT}$
- *location* : $\mathsf{PERMS} \to \mathsf{STR}$

The attribute *action* denotes what kind of transaction can be executed (e.g., withdrawing or transferring money from a bank account), *amount* denotes how much money is issued in a transaction and *location* denotes the country in which the transaction is placed. □

Definition 40 Let $\Sigma = (\mathbb{S}, \mathbb{R}, \mathbb{F}, \mathbb{V})$ be a FORBAC-signature. A Σ-*FORBAC-structure* is a Σ-structure $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ such that $\mathfrak{I}$ interprets the binary relation symbols in $\mathbb{R}$ in the standard way. That is, $\leq^{\mathfrak{I}}$ is the standard order relation on the integers, $\subseteq^{\mathfrak{I}}_{\mathsf{STR}}$ is the standard subset relation on the sets of strings, and so on. □

We call an element of $\mathsf{USERS}^{\mathfrak{S}}$ a *user* of $\mathbb{K}$. For a role template $\mathsf{R} \in RT(\Sigma)$, we call an element of $\mathsf{R}^{\mathfrak{S}}$ a *role instance of* R. We call an element of $\mathsf{PERMS}^{\mathfrak{S}}$ a *permission* of $\mathbb{K}$.

Example 41 Let Σ_B be the FORBAC-signature from Example 39. Figure 3.1 shows a Σ_B-FORBAC-structure $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ with three users, two role instances of $\mathsf{R}_{\texttt{Student}}$, one role instance of $\mathsf{R}_{\texttt{Employee}}$, and three permissions. □

Definition 42 An *atomic* FORBAC-*formula* is any atomic formula built from a FORBAC-signature. A FORBAC-*formula* is a quantifier-free formula built from a FORBAC-signature.

□

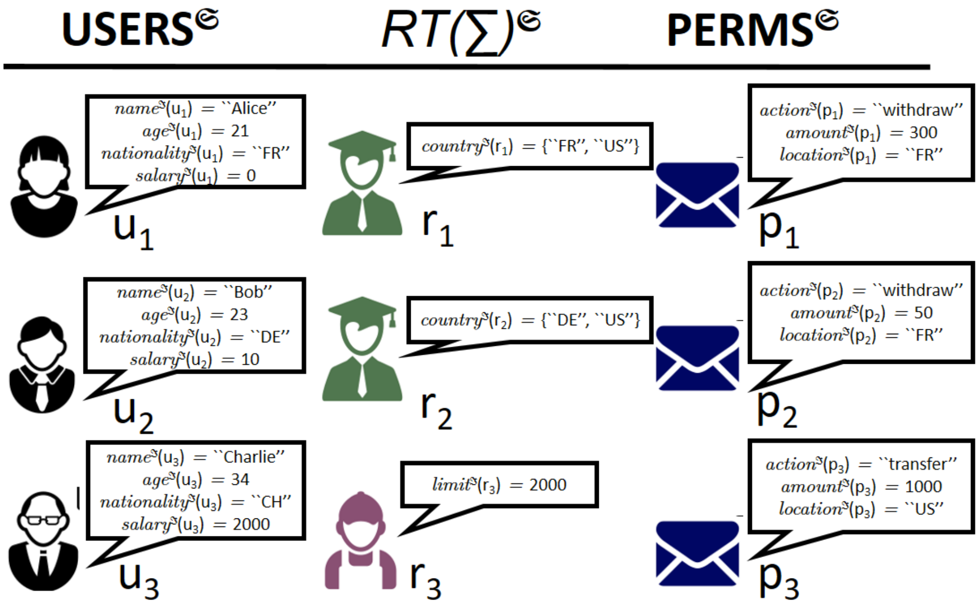

USERS$^{\mathfrak{S}}$
$RT(\Sigma)^{\mathfrak{S}}$
PERMS$^{\mathfrak{S}}$
$name^{\mathfrak{S}}(u_1)$ = "Alice" $age^{\mathfrak{S}}(u_1)$ = 21 $nationality^{\mathfrak{S}}(u_1)$ = "FR" $salary^{\mathfrak{S}}(u_1)$ = 0
u_1
$country^{\mathfrak{S}}(r_1)$ = {"FR", "US"}
r_1
$action^{\mathfrak{S}}(p_1)$ = "withdraw" $amount^{\mathfrak{S}}(p_1)$ = 300 $location^{\mathfrak{S}}(p_1)$ = "FR"
p_1
$name^{\mathfrak{S}}(u_2)$ = "Bob" $age^{\mathfrak{S}}(u_2)$ = 23 $nationality^{\mathfrak{S}}(u_2)$ = "DE" $salary^{\mathfrak{S}}(u_2)$ = 10
u_2
$country^{\mathfrak{S}}(r_2)$ = {"DE", "US"}
r_2
$action^{\mathfrak{S}}(p_2)$ = "withdraw" $amount^{\mathfrak{S}}(p_2)$ = 50 $location^{\mathfrak{S}}(p_2)$ = "FR"
p_2
$name^{\mathfrak{S}}(u_3)$ = "Charlie" $age^{\mathfrak{S}}(u_3)$ = 34 $nationality^{\mathfrak{S}}(u_3)$ = "CH" $salary^{\mathfrak{S}}(u_3)$ = 2000
u_3
$limit^{\mathfrak{S}}(r_3)$ = 2000
r_3
$action^{\mathfrak{S}}(p_3)$ = "transfer" $amount^{\mathfrak{S}}(p_3)$ = 1000 $location^{\mathfrak{S}}(p_3)$ = "US"
p_3

Figure 3.1: An example of a Σ_B-FORBAC-structure. **©2015, IEEE. Reprinted with permission.**

Definition 43 A FORBAC-*policy* is a triple $(\Sigma, \mathcal{UA}, \mathcal{PA})$, where Σ is a FORBAC-signature. The *user-assignment specification*

$$\mathcal{UA} = \{\mathcal{UA}_{\mathbf{R}}(u, r) : \mathbf{R} \in RT(\Sigma)\}$$

and the *permission-assignment specification*

$$\mathcal{PA} = \{\mathcal{PA}_{\mathbf{R}}(r, p) : \mathbf{R} \in RT(\Sigma)\}$$

are sets of FORBAC-formulas over Σ. The *user-assignment formulas* $\mathcal{UA}_{\mathbf{R}}(u, r)$ have (just) the two free variables u and r of sorts **USERS** and **R**, respectively, and the *permission-assignment formulas* $\mathcal{PA}_{\mathbf{R}}(r, p)$ have (just) the two free variables r and p of sorts **R** and **PERMS**, respectively. □

Example 44 Consider the FORBAC-signature Σ_B from Example 41 and suppose that we have the following policy.

- Users no older than 25 are assigned an instance of $\mathbf{R}_{\texttt{Student}}$, which entitles them to withdraw up to \$1,000 in the user's home country or in the US.
- Users whose salary exceeds \$1,500 are assigned an instance of $\mathbf{R}_{\texttt{Employee}}$, which entitles them to withdraw and transfer money in any country provided that the sum does not exceed the user's salary.

We present a FORBAC-policy $(\Sigma_B, \mathcal{UA}, \mathcal{PA})$ that models this. Σ_B was already specified in Example 39, so we just present $\mathcal{UA}$ and $\mathcal{PA}$:

$$\begin{aligned}
\mathcal{UA}_{\mathbf{R}_{\texttt{Student}}}(u,r) &\equiv \begin{pmatrix} age(u) \leq 25 \wedge \\ country(r) = \{nationality(u), \text{``US''}\} \end{pmatrix} \\
\mathcal{PA}_{\mathbf{R}_{\texttt{Student}}}(r,p) &\equiv \begin{pmatrix} action(p) \in \{\text{``withdraw''}\} \wedge \\ amount(p) \leq 1000 \wedge \\ location(p) \in country(r) \end{pmatrix} \\
\mathcal{UA}_{\mathbf{R}_{\texttt{Employee}}}(u,r) &\equiv \begin{pmatrix} salary(u) > 1500 \wedge \\ limit(r) = salary(u) \end{pmatrix} \\
\mathcal{PA}_{\mathbf{R}_{\texttt{Employee}}}(r,p) &\equiv \begin{pmatrix} action(p) \in \{\text{``withdraw''}, \text{``transfer''}\} \wedge \\ amount(p) \leq limit(r) \end{pmatrix} .
\end{aligned} \tag{3.3}$$

□

Let Σ be a FORBAC-signature and $\mathbb{K}$ be a Σ-FORBAC-structure. Let u, r, and p be a user, a role instance of $\mathbf{R}$, and a permission of $\mathbb{K}$, respectively. We say that u *is assigned* r if $\mathbb{K}, \sigma_{\mathsf{u},\mathsf{r}} \vDash \mathcal{UA}_{\mathbf{R}}(u,r)$, where $\sigma_{\mathsf{u},\mathsf{r}}$ is the substitution mapping u and r to u and r, respectively. We say that r *is assigned* p if $\mathbb{K}, \sigma_{\mathsf{r},\mathsf{p}} \vDash \mathcal{PA}_{\mathbf{R}}(r,p)$, where $\sigma_{\mathsf{r},\mathsf{p}}$ is the substitution mapping r and p to r and p, respectively.

Example 45 Figure 3.2 illustrates, in the context of the Σ_B-FORBAC-structure of Example 41 and the FORBAC-policy of Example 44, which role instances are assigned to which users and which permissions are assigned to which role instances. □

Definition 46 For a FORBAC-policy $(\Sigma, \mathcal{UA}, \mathcal{PA})$ and a role template $\mathbf{R} \in RT(\Sigma)$, let $Auth_{\mathbf{R}}(u,p)$ denote the formula

$$\exists r : \mathbf{R} \,.\, \mathcal{UA}_{\mathbf{R}}(u,r) \;\wedge\; \mathcal{PA}_{\mathbf{R}}(r,p). \tag{3.4}$$

Let $Auth(u,p)$ denote the formula $\bigvee_{\mathbf{R} \in RT(\Sigma)} Auth_{\mathbf{R}}(u,p)$. For a Σ-FORBAC-structure $\mathbb{K}$, we say that *a user* u *of* $\mathbb{K}$ *is authorized for a permission* p *of* $\mathbb{K}$ if $\mathbb{K}, \sigma_{\mathsf{u},\mathsf{p}} \vDash Auth(u,p)$. Here, $\sigma_{\mathsf{u},\mathsf{p}}$ is the substitution mapping u and p to u and p, respectively. □

Example 47 Consider the FORBAC-signature presented in Example 39, the Σ_B-FORBAC-structure $\mathbb{K}$ presented in Example 41, and the FORBAC-policy presented in Example 44. User u_1 is authorized for permissions p_1 and p_2 and user u_3 is authorized for permissions p_1, p_2, and p_3. □

When quantifying over variables, we do not specify the sorts **USERS** and **PERMS**, as these should be clear from the context. For example, instead of writing

$$\forall u : \mathbf{USERS}\, \exists r_1 : \mathbf{R}_1 \,.\, \mathcal{UA}_{\mathbf{R}_1}(u, r_1), \tag{3.5}$$

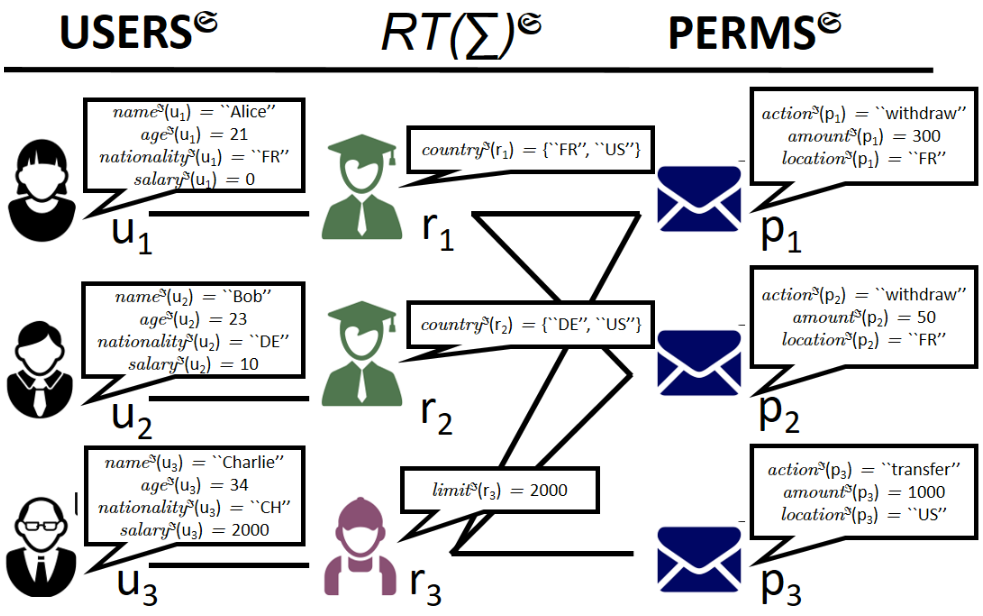

USERS^𝔖
RT(Σ)^𝔖
PERMS^𝔖
name^ℑ(u1) = "Alice" age^ℑ(u1) = 21 nationality^ℑ(u1) = "FR" salary^ℑ(u1) = 0
u1
name^ℑ(u2) = "Bob" age^ℑ(u2) = 23 nationality^ℑ(u2) = "DE" salary^ℑ(u2) = 10
u2
name^ℑ(u3) = "Charlie" age^ℑ(u3) = 34 nationality^ℑ(u3) = "CH" salary^ℑ(u3) = 2000
u3
country^ℑ(r1) = {"FR", "US"}
r1
country^ℑ(r2) = {"DE", "US"}
r2
limit^ℑ(r3) = 2000
r3
action^ℑ(p1) = "withdraw" amount^ℑ(p1) = 300 location^ℑ(p1) = "FR"
p1
action^ℑ(p2) = "withdraw" amount^ℑ(p2) = 50 location^ℑ(p2) = "FR"
p2
action^ℑ(p3) = "transfer" amount^ℑ(p3) = 1000 location^ℑ(p3) = "US"
p3

Figure 3.2: User and permission-assignments in the Σ_B-FORBAC-structure $\mathbb{K}$. **©2015, IEEE. Reprinted with permission.**

we write

$$\forall u\,.\,\exists r_1 : \mathbf{R}_1\,.\,\mathcal{UA}_{\mathbf{R}_1}(u, r_1). \tag{3.6}$$

We conclude our presentation of FORBAC by observing that authorization can be decided in polynomial time. The proof is given in Appendix A.2.

Theorem 48 Given a FORBAC-policy $(\Sigma, \mathcal{UA}, \mathcal{PA})$, a Σ-FORBAC-structure $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$, a user $\mathsf{u} \in \mathbf{USERS}^{\mathfrak{S}}$ and a permission $\mathsf{p} \in \mathbf{PERMS}^{\mathfrak{S}}$, deciding whether u is authorized for p takes time

$$O\left(\left|\mathbf{ROLES}^{\mathfrak{S}}\right| \cdot |\mathbb{K}|^2 \cdot (|\mathcal{UA}| + |\mathcal{PA}|)\right), \tag{3.7}$$

where $\mathbf{ROLES}^{\mathfrak{S}}$ is the set of role instances in $\mathbb{K}$, $|\mathbb{K}|$ is the length of $\mathbb{K}$ encoded as a string[1], $|\mathcal{UA}| = \sum_{\mathbf{R} \in RT(\Sigma)} |\mathcal{UA}_{\mathbf{R}}|$, and $|\mathcal{PA}| = \sum_{\mathbf{R} \in RT(\Sigma)} |\mathcal{PA}_{\mathbf{R}}|$.

3.4 Policy analysis in FORBAC

We now define a language for specifying properties of FORBAC-policies. Although first-order logic is the natural choice for property specification, its undecidability makes it unsuitable for efficiently verifying properties. To reduce the complexity of verifying properties, we must then use a fragment of

[1]For computational purposes, Σ-structures need to be encoded as strings in order to be given as input to programs. See [73] for details.

first-order logic. Many works have made efforts to find fragments that are decidable while still sufficiently powerful to express certain properties [28]. One of the most popular works is by Halpern and Weissman [66], where they evaluated the complexity of policy analysis for several fragments. They showed that even after severely limiting the fragment's expressiveness by reducing the number of quantifier alternations and removing function symbols, the remaining fragment has an unacceptably high complexity for verifying properties. They managed to propose a fragment that can be efficiently decided and is powerful enough to express access-control policies from diverse university libraries and US government policies.

To strike a balance between expressiveness in property specification and efficiency in policy analysis for extensions of RBAC that use first-order logic, we propose the set of existential FORBAC-formulas as the language for specifying policy properties.

Definition 49 An *existential* FORBAC-*formula* is a first-order formula of the form $\exists x_1 : \mathbf{S}_1\, \exists x_2 : \mathbf{S}_2 \ldots \exists x_n : \mathbf{S}_n \,.\, \varphi(x_1, x_2, \ldots, x_n)$, where $\varphi(x_1, x_2, \ldots, x_n)$ is a boolean combination of FORBAC-formulas over a FORBAC-signature. □

Theorem 50 Deciding the satisfiability of an existential FORBAC-formula is NP-complete.

Proof. See Appendix A.2. □

The low complexity, NP, is not free. There are relevant policy properties like "observational equivalence" and "conflict" [10] that cannot be expressed as existential FORBAC-formulas. However, they can still be expressed in first-order logic and can be passed as input to an SMT-solver. We simply cannot provide guarantees on the complexity of verifying such types of properties.

To verify a property, we build an existential FORBAC-formula that describes a counterexample; that is, a FORBAC-structure that violates the property. The formula is given as input to the SMT solver Z3, which attempts to compute a counterexample.

Having shown that deciding satisfiability of formulas in the language of existential FORBAC-formulas has an acceptable complexity, we argue now why this language is expressive enough to specify relevant policy properties. We showcase four types of policy properties that can be expressed as existential FORBAC-formulas. Afterwards, we show the main findings of a case study on a major European bank, where we evaluate these properties on some of the bank's applications. We provide here only a major overview, as this work belongs to Thilo Weghorn's doctoral thesis. A more detailed account is found in the original FORBAC paper [40].

Authorization inspection: This type of properties help to verify that a FORBAC-policy is granting the right permissions to the right users. A property of this type checks if there is a user u and a permission p with certain characteristics such that the FORBAC-policy assigns p to u. Properties of this type are formalized with FORBAC-formulas of the following form:

$$\exists u \,.\, \exists p \,.\, \left(\psi_{user}\left(u\right) \;\wedge\; \psi_{perm}\left(p\right) \;\wedge\; Auth\left(u,p\right)\right) . \tag{3.8}$$

Here, $\psi_{user}\left(u\right)$ and $\psi_{perm}\left(p\right)$ restrict the user and the permission we are interested in. The formula $Auth\left(u,p\right)$ is the formula from Definition 46 for some FORBAC-policy.

Observe that Formula 3.8 is not an existential FORBAC-formula, as the subformula $Auth\left(u,p\right)$ contains existential quantifiers. However, those quantifiers can be moved to the front of Formula 3.8 without affecting its validity, yielding an existential FORBAC-formula.

We illustrate an instance of authorization inspection in the context of the FORBAC-policy from Example 44. Suppose that we fix

$$\begin{aligned} \psi_{user}(u) &\equiv age(u) \leq 25 \\ \psi_{perm}(p) &\equiv \begin{pmatrix} action(p) = \text{“withdraw”} \wedge \\ amount(p) > 1000 \end{pmatrix} . \end{aligned} \tag{3.9}$$

The resulting property holds if there is a way for a user who is at most 25 years old to get a permission to withdraw amounts larger than \$1,000.

Assignment simplification: This type of properties help to verify if parts of a FORBAC-policy can be simplified. Let $(\Sigma, \mathcal{UA}, \mathcal{PA})$ be a FORBAC-policy. A property of this type specifies, for a formula ψ in $\mathcal{UA} \cup \mathcal{PA}$, if it can be rewritten into another simpler formula ψ'. Such a property can be formalized as the following existential FORBAC-formula:

$$\exists x_1 \ldots \exists x_k \,.\, \neg \left(\psi\left(x_1,\ldots,x_k\right) \leftrightarrow \psi'\left(x_1,\ldots,x_k\right)\right) . \tag{3.10}$$

Observe that the formula ψ can be rewritten as ψ' iff an SMT-solver, when given as input this formula, finds the formula to be unsatisfiable.

We illustrate an instance of assignment simplification. Consider a FORBAC-policy with a role template **R** such that

$$\mathcal{UA}_{\mathbf{R}}\left(u,p\right) \equiv \psi_1\left(u,r\right) \;\vee\; \psi_2\left(u,r\right) , \tag{3.11}$$

where

$$\begin{aligned} &\psi_1\left(u,r\right) \equiv unit\left(r\right) = 45 \;\wedge\; level\left(u\right) = 23 \text{ and} \\ &\psi_2\left(u,r\right) \equiv unit\left(r\right) = 45 \;\wedge\; level\left(u\right) > 20. \end{aligned}$$

Observe that whenever $\psi_1\left(u,r\right)$ holds, then $\psi_2\left(u,r\right)$ also holds. This means that $\psi_2\left(u,r\right)$ is redundant in $\mathcal{UA}_{\mathbf{R}}$. To confirm this, one can observe that the existential FORBAC-formula $\exists u \,.\, \exists r : \mathbf{R} \,.\, \neg \left(\left(\psi_1\left(u,r\right) \;\vee\; \psi_2\left(u,r\right)\right) \leftrightarrow \psi_1\left(u,r\right)\right)$ is unsatisfiable.

Role equivalence: RBAC policies that are administered by different entities may end up with equivalent roles. This type of properties help to detect such roles, which leads to simpler policies. For example, consider a FORBAC-policy with two role templates $\mathbf{R}_1$ and $\mathbf{R}_2$ such that:

$$\begin{aligned}&mPA_{\mathbf{R}_1}(r,p) \equiv level(r) = level(p) \;\vee\; level(r) > level(p)\\&mPA_{\mathbf{R}_2}(r,p) \equiv level(r) \geq level(p)\,.\end{aligned}$$

Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a Σ-FORBAC-structure and suppose that r_1 and r_2 are two role instances of $\mathbf{R}_1$ and $\mathbf{R}_2$, respectively, such that $level^{\mathfrak{I}}(\mathsf{r}_1) = level^{\mathfrak{I}}(\mathsf{r}_2)$. Observe, that, any permission assigned to r_1 is also assigned to r_2. They are essentially equivalent. Hence, we can remove role template $\mathbf{R}_2$ from the FORBAC-policy.

To confirm this observation, it suffices to verify that the following existential FORBAC-formula is unsatisfiable:

$$\exists r_1 : \mathbf{R}_1 \,.\, \exists r_2 : \mathbf{R}_2 \,.\, \exists p \,.\, \left(\begin{array}{l} level(r_1) = level(r_2) \;\wedge \\ \neg\left(\mathcal{PA}_{\mathbf{R}_1}(r_1, p) \leftrightarrow \mathcal{PA}_{\mathbf{R}_2}(r_2, p)\right)\end{array}\right). \tag{3.12}$$

Finally, equivalent roles are not always redundant. For example, in an IT organization, the roles of programmer and tester may have the same set of permissions, but they need to be distinguished for organizational purposes.

Redundant role templates: In large organizations, where RBAC policies are administered by different entities, the entities in charge of the assignment of roles to users may be different from those in charge of the assignment of permissions to roles. This may lead to situations like the one depicted in Figure 3.3, where for two role instances r_1 and r_2, the set of users who get assigned r_2 is a subset of those who get assigned r_1 and the set of permissions assigned to r_2 is a subset of those assigned to r_1. In this case, r_2 is redundant. We illustrate next how we can detect these situations with existential FORBAC-formulas.

Consider a FORBAC-policy with two role templates $\mathbf{R}_1$ and $\mathbf{R}_2$ such that

$$\begin{aligned}&\mathcal{UA}_{\mathbf{R}_1}(u,r) \equiv level(r) = age(u)\,, \qquad (3.13)\\&\mathcal{UA}_{\mathbf{R}_2}(u,r) \equiv level(r) = age(u)\,,\\&\mathcal{PA}_{\mathbf{R}_1}(r,p) \equiv level(r) \geq level(p)\,, \text{ and}\\&\mathcal{PA}_{\mathbf{R}_2}(r,p) \equiv level(r) = level(p)\,.\end{aligned}$$

Let r_1 and r_2 be two roles instances of $\mathbf{R}_1$ and $\mathbf{R}_2$, respectively. Observe that whenever a user is assigned r_2, he is also assigned r_1. Also, whenever a permission is assigned to r_2, that permission is also assigned to r_1. As a result, $\mathbf{R}_2$ is redundant and can be removed from the FORBAC-policy.

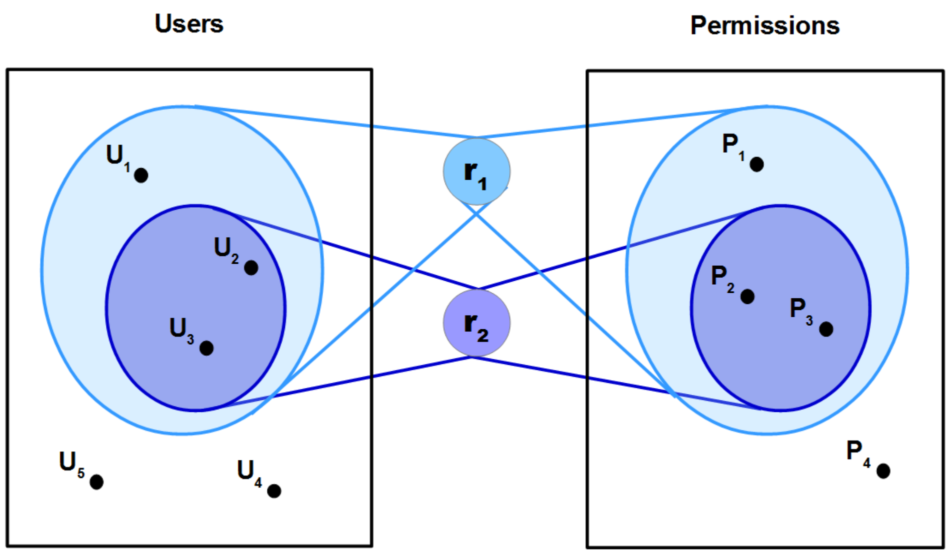


Figure 3.3: The role r_2 is redundant: the permissions assigned to r_2 are contained in those assigned to r_1 and the users assigned to r_2 are also assigned to r_1. **©2015, IEEE. Reprinted with permission.**

We can prove $\mathbf{R}_2$'s redundancy by verifying that the following two existential FORBAC-formulas are unsatisfiable:

$$\exists u \,.\, \exists r_1 : \mathbf{R}_1 \,.\, \exists r_2 : \mathbf{R}_2 \,.\, \left(\begin{array}{l} \mathit{level}\,(r_1) = \mathit{level}\,(r_2) \;\wedge \\ \mathcal{UA}_{\mathbf{R}_2}\,(r_2, p) \;\wedge\; \neg \mathcal{UA}_{\mathbf{R}_1}\,(r_1, p) \end{array} \right) \text{ and}$$
$$\exists r_1 : \mathbf{R}_1 \,.\, \exists r_2 : \mathbf{R}_2 \,.\, \exists p \,.\, \left(\begin{array}{l} \mathit{level}\,(r_1) = \mathit{level}\,(r_2) \;\wedge \\ \mathcal{PA}_{\mathbf{R}_2}\,(r_2, p) \;\wedge\; \neg \mathcal{PA}_{\mathbf{R}_1}\,(r_1, p) \end{array} \right) . \tag{3.14}$$

3.5 Experimental results

We conducted a case study on a major European bank to evaluate whether (i) the FORBAC language is powerful enough to express realistic access control policies and (ii) whether realistic policy properties can be verified within a reasonable amount of time. We provide here only a major overview as the results here belong to Thilo Weghorn's thesis.

For our study, we chose, from among the 350 applications we had access to, the 10 applications with the most complex access-control policies. We translated their policies into FORBAC-policies. For each of the obtained FORBAC policies and for each of the property types presented in Section 3.4, we randomly generated 10 different existential FORBAC-formulas and checked their satisfiability using Z3.

For role equivalence, we picked two role templates at random and then built an existential FORBAC-formula that held iff the two role templates were not equivalent. For authorization inspection, we proceeded as follows. First, we selected uniformly at random a subset of user attribute values and a subset of permission attribute values. Then we built an existential FORBAC-formula that holds iff there is a user with the selected attribute values who would be assigned a permission with the selected attribute values. For the other property types, we built the existential FORBAC-formulas analogously. Table 3.1 shows the time it took Z3 to decide satisfiability for the formulas we built.

Two of the FORBAC-policies were not complex enough to make meaningful experiments with existential FORBAC-formulas. In those cases, we marked the corresponding entry in Table 3.1 with "NA".

Type	App1	App2	App3	App4	App5	App6	App7	App8	App9	App10
AI	0,11	4,97	1,56	0,15	0,13	0,12	0,40	0,45	0,31	0,12
AS	0,61	0,63	0,57	0,54	*NA*	0,75	0,87	0,5	0,49	*NA*
RE	0,53	0,55	0,43	0,43	0,45	0,47	0,46	0,47	0,47	0,44
RR	0,73	0,47	0,46	0,49	*NA*	0,58	0,53	0,59	0,49	*NA*

Table 3.1: Average time (in seconds) needed by Z3 to decide satisfiability of an existential FORBAC-formula for each application and each property type. AI: authorization inspection, AS: assignment simplification, RE: role equivalence, RR: role redundancy.

The fact that we were able to express the main parts of these policies in FORBAC demonstrates that this policy language can be applied to model realistic access control policies. Also, the fact that we verified almost all properties in average in less than 1 second shows that FORBAC can perform policy analysis with reasonable overhead.

3.6 Related work

RBAC is an access control paradigm that was proposed to make access-control maintenance more scalable. As organizations defined more complex and fine-grained access control policies, different extensions of RBAC were proposed that allowed to specify conditions for assigning roles to users and permissions to roles.

3.6.1 Frameworks for policy specification and analysis

Some researchers noted that the increasing expressiveness was also making harder to determine whether the policies were granting access as desired.

Therefore, they proposed frameworks that not only allowed to specify policies, but also to analyze and verify properties of those policies [107, 10]. Their research focused on generic languages for policy specification. So those frameworks can specify RBAC and ABAC policies and verify standard properties against such policies.

The genericity of such frameworks makes them unable to specify complex policies like FORBAC policies. For example, Margrave, one of such frameworks, cannot specify arithmetic constraints and, therefore, cannot specify policies like "Alice is granted permission p if her clearance level is greater than p's clearance level". These frameworks are also unable to verify some natural properties that arise when analyzing FORBAC policies. Also, for some of these frameworks (e.g., [10]), the language for specifying policies and properties is so rich that Diophantine equations can be expressed within their policies, which yields an unnecessarily high computational complexity for policy analysis.

In comparison of the frameworks above, FORBAC guarantees that deciding satisfiability of existential FORBAC-formulas is in NP and we show that many relevant policy properties can be expressed as existential FORBAC-formulas. We remark, however, that there are still other relevant properties that cannot be expressed as existential FORBAC-formulas and that can be verified by the frameworks we presented above. Some of these properties are observational equivalence, conflict detection, and change-impact analysis. This does not mean that we cannot use FORBAC to verify these properties, as they can still be expressed as first-order logic formulas and analyzed by an SMT-solver. We can use FORBAC, but we cannot guarantee that the complexity of verifying them is in NP.

3.7 Conclusion and future work

New extensions for RBAC have been proposed over the past decades in order to fulfill new expressiveness requirements from different types of organizations. The added expressiveness to these extensions came at the cost of higher complexity in policy analysis. To address this issue, we proposed FORBAC, a framework that strikes a balance between expressiveness and efficient policy analysis for extensions of RBAC. With a case study, we have shown that FORBAC can express complex modern policies for extensions of RBAC. Moreover, the set of existential FORBAC formulas is also powerful enough to contain relevant policy properties. We also proved that deciding the satisfiability of an existential FORBAC-formula is in NP and argued why this is a desirable complexity class for policy analysis.

As future work, we consider extending FORBAC with some popular RBAC idioms. For example, we intend to incorporate role hierarchies and separation-of-duty constraints. Regarding role hierarchies, there is the challenge of agreeing on when a role instance inherits from another role instance. Some works have proposed ways to define role hierarchies when roles use attributes [63, 18, 19] but other ways are still possible. In particular, there is the challenge of how to proceed when the role instances come from role templates that have different attributes. A first approach would be to let the parent role instance inherit the attribute values of the child role instance. Similarly, there is no agreement on how to define separation-of-duty constraints in the presence of role instances with attribute values.

We also consider defining an extension of the FORBAC language that allows us to express other popular policy analysis properties like "conflict" and "change-impact analysis" [54]. We conjecture that the problem of verifying these properties in that extension would belong to a complexity class in the lower echelons of the polynomial-time hierarchy, as these verification problems can be cast as satisfiability problems for first-order formulas with fixed quantified alternation. If this is true, then this would mean that in contrast to the rapid increase of expressive power of RBAC extensions during the last decades, the complexity of policy verification of these extensions has actually remained in a limited complexity class smaller than PSPACE (unless NP = PSPACE).

Chapter 4

Rhapsody: Reliable Apriori Subgroup Discovery

4.1 Introduction

Attribute-based access control (ABAC) is a popular access control paradigm. ABAC policies are sets of rules that define which permissions are assigned to which users. Rules in ABAC policies depend on the user's and the permission's attribute values. The National Institute of Standards and Technology (NIST) issued in 2014 a special publication recommending ABAC over role-based access control (RBAC) and access control matrices [68]. Gartner Inc. estimates that by 2020 roughly 70% of all businesses will use ABAC to protect critical assets [105, 86].

4.1.1 Problem context

The specification and maintenance of ABAC policies brings new challenges. Manually migrating to ABAC is more difficult than migrating to RBAC, which is already time-consuming and cumbersome [68]. Moreover, even after specification, an ABAC policy must be maintained and audited, as organizational changes like mergers and acquisitions can make the policies convoluted or inaccurate.

An alternative to manually specifying and maintaining an ABAC policy is to automatically *mine* ABAC policies from access logs [131, 132, 32, 100]. Since logs reflect both the implemented access control policies and user behavior within the organization, mining from logs can help to refactor and simplify policies that become overly-complex due to organizational changes.

Mining from logs can also help to identify *overly permissive rules*; that is, rules in the policy that assign permissions to users who, according to the log, are not using them. Overly permissive rules violate the principle of least privilege and enable privilege abuse. Verizon's 2018 data breach investigation report shows that privilege misuse is a major issue for many companies [51]. Even for healthcare companies, where access to data is critical, Verizon's 2017 report recommends to ensure that employees in an organization have access only to the information they need [50].

In this chapter, we examine one aspect of this problem that is essential to using ABAC policy mining in practice: *mining ABAC policies from sparse logs*. We call a log *sparse* when it contains less than 10% of all possible requests. Users generally access only those resources they are authorized. As a consequence, real world logs typically contain only a small subset of all possible access requests. We illustrate this with three real-world logs from different representative domains, where each of these logs contains less than 10% of all possible requests and more than 95% of those requests were authorized by the policy.

This problem setting subsumes other previously considered settings, like mining from non-sparse logs and mining where permissions are given by formalisms such as access control matrices.

4.1.2 Limitations of the state-of-the-art

Despite intensive research in this area [133, 103, 29, 112, 37, 71, 80, 84], previous work has serious limitations.

Sparse logs. When the logs are sparse, some competing approaches [131, 132] cannot mine useful ABAC policies. This is because these approaches are intended only for mining from access control matrices or from logs containing a large fraction of all possible access requests.

Over-permissiveness. Mining algorithms [84, 80] resort to quality measures like confidence [4] and weighted relative accuracy [91] to guide the search and selection of policies. We show in Section 4.5.2 that, for sparse logs, these measures lead to the mining of *overly permissive* rules. These are rules in the policy that authorize a significant number of requests without any evidence of authorization in the log.

Rule size. Current algorithms mine policies containing unnecessarily large rules. Succinct rules are desirable from the administrative perspective as they are easier to audit and maintain.

Cross-validation. When evaluating models learned from logs, the standard cross-validation method splits the log into a training log and a testing log. We show that this approach validates policies containing overly permissive rules, which is undesirable from the security perspective. This happens because the logs reflect expected access requests. Denied access requests are therefore rare and mostly due to human error, rather than malice. A testing log does not include a representative set of requests that would be issued by malicious users. As a result, evaluating a mined policy on such a testing log validates policies with overly permissive rules.

4.1.3 Our approach

Rhapsody. We introduce a new ABAC mining algorithm that addresses the first three limitations of the state-of-the-art as follows.

> *Sparse logs*. Rhapsody uses association rule mining [3, 4] to effectively mine from sparse logs.
>
> *No overly permissive rules*. Rhapsody avoids mining overly-permissing rules by using *reliability* (see Definition 57), a new rule quality measure. We demonstrate that reliability gives low scores to rules that are overly permissive or have low confidence (see Observation 1 in Section 4.3.2). We also show that standard rule quality measures cannot quantify over-permissiveness.
>
> *Rule size*. Rhapsody only mines rules that have no shorter equivalent rule.

Rhapsody is guaranteed to mine all rules that have a reliability above a given threshold, do not authorize any denied request, and have no shorter equivalent rule (see Theorem 60 in Section 4.4.2). Table 4.1 shows how Rhapsody improves upon state-of-the-art ABAC mining algorithms.

Algorithm	Sparse logs	No overly perm. rules	Shortest rules
Xu & Stoller's miner [133]	✗	✓	✗
APRIORI-C [80]	✓	✗	✓
APRIORI-SD [84]	✓	✗	✓
Classsification-tree [117]	✓	✓	✗
CN2 [37]	✓	✓	✗
Rhapsody	✓	✓	✓

Table 4.1: State-of-the-art ABAC mining algorithms.

Universal cross-validation. We present an alternative to cross-validation for evaluating ABAC mining algorithms. In contrast to standard cross-validation, universal cross-validation evaluates mined policies with requests not occurring in the log. In this way, universal cross-validation improves the quality of the policies mined by *any* ABAC mining algorithm, as it rejects those containing overly permissive rules.

4.1.4 Case studies and evaluation

We experimentally compare Rhapsody with other methods for mining ABAC policies from logs. Our experimental evaluation is the most comprehensive to date in access control mining. We use logs from a large Swiss university (ETH Zurich) and two logs provided by Amazon [82, 93], where the last two are available for research purposes. Previous researchers used only logs from one enterprise, which are not publicly available [103, 59] or just synthetic data [132, 100].

Our experimental results, presented in Section 4.6.4 and summarized in Table 4.1, compare the policies mined by competing approaches according to standard performance metrics like the F1 score, true positive rate, false positive rate, and size [111]. They illustrate the four limitations of the state-of-the-art and demonstrate that Rhapsody and universal cross-validation overcome them.

Sparse logs. Rhapsody mines policies of higher quality than Xu and Stoller's approach [131, 132, 133]. When mining from *sparse logs*, Xu and Stoller's mined policies have true positive rates close to 0%. In contrast, Rhapsody's policies attain true positive rates above 80% in most of the cases.

Over-permissiveness. Subgroup-discovery algorithms like APRIORI-SD and APRIORI-C mine policies containing *overly-permissive rules*, which yield false positive rates above 10%. By changing these algorithms' input parameters, one can decrease the false positive rate to values close to 0% but at the cost of true positive rates below 40%. In contrast, Rhapsody's mined policies attain false positive rates close to 0% and true positive rates above 80% in most of the cases.

Rule size. Classification-tree learners [29, 117] and CN2 [37, 108] mine policies with *unnecessarily large rules.* For Amazon's logs, the policies we mined are at least 40% shorter and in some cases they are even 90% shorter.

Cross-validation. In 80% of the cases, universal cross-validation validates policies with F1 scores greater than or equal to those from policies validated by standard cross-validation. In most of the cases, the F1 scores improve by at least 30%. This holds for *any mining algorithm*.

4.1.5 Contributions

In summary, we make the following contributions.

Mining algorithm for sparse logs. We propose Rhapsody, a new ABAC mining algorithm that, in comparison with previous work, mines policies from sparse logs that are shorter and generalize better.

Quality measure for over-permissiveness. We introduce *reliability*, a new measure that quantifies how overly permissive a policy is. We show that reliability only gives high values to policies that are not overly permissive and show why other standard measures fail to measure over-permissiveness.

Succinctness and correctness. Rhapsody *guarantees* that if there is a succinct rule satisfied by a significant number of requests and with a reliability above a given threshold, then this rule will be mined (see Theorem 60).

Evaluation methodology. We propose *universal cross-validation*, a new validation method that, in comparison with cross-validation on logs, validates policies with substantially higher F1 scores.

Organization. The remainder of this chapter is organized as follows. In Section 4.2 we give background on the ABAC mining problem. In Section 4.3 we show how standard rule quality measures give high scores to overly permissive rules and we introduce a better measure, reliability. In Section 4.4 we present Rhapsody, a new ABAC mining algorithm. In Section 4.5 we show how cross-validation on logs validates overly permissive rules and present universal cross-validation. In Section 4.6 we experimentally compare Rhapsody with other ABAC mining algorithms and compare universal cross-validation with cross-validation. Finally, in Sections 4.7 and 4.8 we discuss related work and draw conclusions.

4.2 The ABAC mining problem

We begin by describing the setting for ABAC mining. Afterwards, we formalize ABAC's syntax and semantics and the ABAC mining problem.

4.2.1 Setting

We describe a setting that motivates ABAC mining. Recall from Chapter 2 that organizations define *(organizational) security policies* expressing which requests should be authorized. For enforcement purposes, the policy administrator must specify this as an ABAC policy in a machine-readable way that can be processed by the organization's IT systems. We call this the *implemented ABAC policy*.

Due to organizational changes or human errors made during the policy's implementation, mistakes may be introduced in the implemented access control policy. These mistakes can be classified into two types: incorrect authorizations (i.e., requests authorized by the implemented policy but not by the organizational security policy) or incorrect denials (i.e., requests authorized by the organizational security policy but not by the implemented policy).

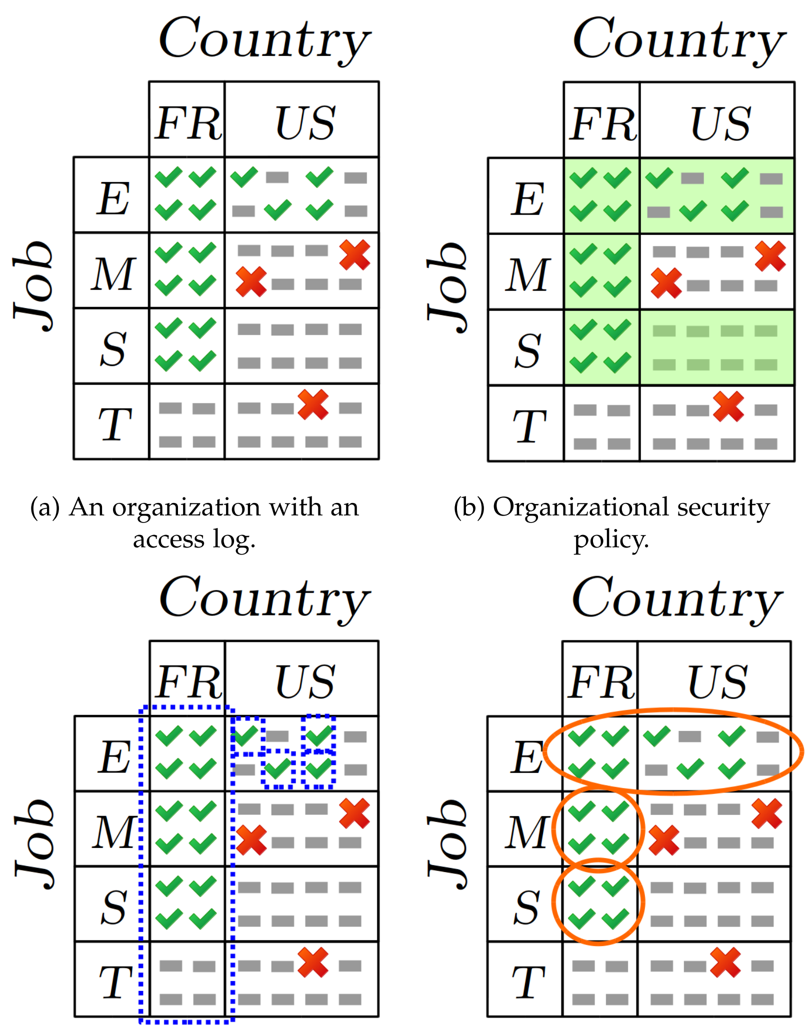
Incorrect authorizations are the hardest to detect and the most problematic because they may be used for nefarious purposes, e.g., to read confidential data, escalate privileges, and in general to attack systems. In contrast, incorrect denials are usually not so problematic from the security perspective; when users discover that valid requests are not authorized, they report them to the policy administrator, who must then manually add exceptions to the implemented policy. However, such manual changes are time-consuming and adding exceptions results in a convoluted policy.

4.2.2 Illustrative example

Figure 4.1 illustrates an example that helps to illustrate the concepts we introduce in this chapter. Figure 4.1a describes an organization with 48 users, one permission, and an access log. Each ✓, ✗, and ▬ denotes a request. Since there is only one permission, we can identify each request with the user who issues it. The ticks ✓ and crosses ✗ denote logged requests (corresponding to users) that have been authorized and denied, respectively. The rectangular marks ▬ denote users who have not requested the permission yet. Users have two attributes: *Country* and *Job*. *Country* can take the values "US" and "FR" whereas *Job* can take the values "E", "M", "S", and "T" (they stand for Engineer, Manager, Secretary, and Technician, respectively).

Figure 4.1b describes the same organization and the same log from Figure 4.1a, but with the organizational security policy, described by the shaded rectangles. This security policy encloses all requests that should be authorized. According to this policy, all engineers, secretaries, and French managers should be authorized.

Figure 4.1c describes the actual implemented ABAC policy, represented with dotted rectangles, which authorizes all French users and four US-based engineers. Observe that this policy incorrectly authorizes all French technicians and incorrectly denies requests from US-based secretaries and some US-based engineers. The four authorized US-based engineers represent users



(a) An organization with an access log.

(b) Organizational security policy.

(c) Implemented policy.

(d) Mined policy.

Figure 4.1: An illustrative example for ABAC mining. All four figures show the same organization and the same access log. Each ✓, ✗, and ▬ denotes a request (i.e., user). The ticks ✓ and crosses ✗ denote logged requests that have been authorized and denied, respectively. The tiny rectangles ▬ denote users who have not requested the permission yet. **©2018, IEEE. Reprinted with permission.**

who were initially denied authorization, reported these incorrect denials to the administrator, and were later added as exceptions to the policy.

Our goal is to propose an *ABAC mining algorithm* that, given a log of requests, mines ABAC rules that identify patterns among the authorized requests. Our algorithm, Rhapsody, would mine the rules given by the ovals in Figure 4.1d. Observe that Rhapsody's mined policy does not authorize US-based secretaries. We argue why this is the preferred policy in Section 4.5.1.

4.2.3 Objectives of ABAC mining

We now recall the objectives of policy mining, from the perspective of ABAC.

Generalization. The mining algorithm can simply *overfit* the log and mine a policy that authorizes precisely the authorized requests in the log and nothing else. This policy is not useful as it does not help to explain what is missing in the implemented policy. For this reason, a mining algorithm should return policies that *generalize well*; that is, the policies also authorize non-logged requests for which a significant number of similar requests have been authorized. A standard method for evaluating how well a mined policy generalizes is cross-validation [60], which we discuss in Section 4.5.

Precision. The algorithm should not mine policies authorizing sets of requests for which the log offers no evidence, like the rule authorizing all French technicians in Figure 4.1c. For the case of French technicians, the algorithm should conservatively risk an incorrect denial rather than an incorrect authorization, as the latter is harder to detect and more security critical. The absence of these requests in the log shows that these requests are infrequent, so the work required by the policy administrator would be low in case some of these requests are incorrectly denied.

Succinctness. The mined rules are used by policy administrators to correct the implemented policy. Therefore, the mined rules should be succinct, so that administrators can easily understand them. For example, if a rule states that "all Italian programmers are authorized" and we know that all programmers are Italian, then a better rule would be "all programmers are authorized."

4.2.4 ABAC syntax and semantics

ABAC policies are logical formulas expressing (binary) relationships between users and permissions. They were defined in Section 2.3.2 in a restricted fragment of many-sorted first-order logic.

Definition 51 Any unary function symbol f with domain **USERS** or **PERMS** is an *attribute*. If the domain is **USERS**, we call f a *user attribute*; otherwise, we call f a *permission attribute*. □

For the rest of this chapter, we fix a structure $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ and let $U = \mathbf{USERS}^{\mathfrak{S}}$ and $P = \mathbf{PERMS}^{\mathfrak{S}}$. Whenever we say "policy" or "rule, we mean an ABAC policy or an ABAC rule, respectively. Recall that, in the context of ABAC policies, we refer to atomic formulas as *atoms*. Recall also that a *rule* is any conjunction of atoms and that policies are disjunctions of rules.

Definition 52 Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a structure. We call a pair in $U \times P$ a *request*. For a user attribute f and $\mathsf{u} \in U$, the value $f^{\mathfrak{I}}(\mathsf{u})$ is called u*'s attribute value for* f. For a permission attribute f and $\mathsf{p} \in P$, the value $f^{\mathfrak{I}}(\mathsf{p})$ is called p*'s attribute value for* f. Let $\pi(u,p) = r_1(u,p) \vee \ldots \vee r_k(u,p)$ be a policy. A rule $r(u,p) = \alpha_1(u,p) \wedge \ldots \wedge \alpha_\ell(u,p)$ *authorizes* or *covers* a request (u,p) if $\mathbb{K}, \sigma_{(\mathsf{u},\mathsf{p})} \models r(u,p)$, where $\sigma_{(\mathsf{u},\mathsf{p})}$ is the substitution mapping u and p to u and p, respectively. A policy π *authorizes* a request (u,p) if $\mathbb{K}, \sigma_{(\mathsf{u},\mathsf{p})} \models \pi(u,p)$; that is, if some rule in π authorizes it. For a rule r and a set $S \subseteq U \times P$, we let $[\![r]\!]_S$ be the set of requests in S authorized by r. □

For convenience, we also sometimes treat a policy $\pi = r_1 \vee \ldots \vee r_k$ as a set $\pi = \{r_1, \ldots, r_k\}$ of rules. Similarly, we sometimes treat a rule $r = \alpha_1 \wedge \ldots \wedge \alpha_\ell$ as a set $r = \{\alpha_1, \ldots, \alpha_\ell\}$ of atoms.

4.2.5 ABAC mining

We now formally define the ABAC mining problem. We start by defining an ABAC instance as a structure describing the set of users, the set of permissions, and the set of requests that have been logged so far in an access control system.

Definition 53 (ABAC instance) An *ABAC instance* is a tuple (U, P, A, D) where U and P are sets representing all users and permissions in an organization, A and D are disjoint subsets of $U \times P$ and they denote the set of authorized and denied requests, respectively. The *log* of the instance is the set $A \cup D$. □

In *the ABAC mining problem*, we are given as input an ABAC instance (U, P, A, D) and the objective is to find a *precise* ABAC policy of *minimal size* that *generalizes well*. The policy's size is measured as described in Definition 15. To measure the precision and how well the policy generalizes, we use *universal cross-validation*, a new approach to cross-validation, introduced in Section 4.5. Algorithms intended to solve this problem are called *ABAC mining algorithms*.

4.3 Quantifying over-permissiveness

Most ABAC mining algorithms work by computing a set of candidate rules and selecting those that have a high score according to some rule quality measure. State-of-the-art quality measures depend only on the *confidence* of the rule (the ratio of authorized requests covered by the rule to the total requests covered by the rule). As we shall see, this is insufficient and these measures may give high scores to rules that we denote as *overly permissive*: rules that authorize significant sets of requests with insufficient evidence from the log. Since these rules are undesirable, we propose a new quality measure for rule selection called *reliability*. We prove that reliability gives a high value to a rule iff it has a high confidence and, in addition, it is *not* overly permissive (see Observation 1).

As a motivating example, consider the ABAC instance depicted in Figure 4.1a. From our discussion in Section 4.2.3, an ABAC mining algorithm should mine the rule $Job(u) = \text{“E”}$, but not the rule $Country(u) = \text{“FR”}$. The log shows that at least half of the engineers have requested access and been authorized. Since users with the same attribute values generally perform the same functions in an organization, they should also have the same permissions. Therefore, $Job(u) = \text{“E”}$ should be mined. In contrast, regarding the rule $Country(u) = \text{“FR”}$, although most of the French users have been authorized, none of the French technicians have been authorized. There is no evidence in the log yet to conclude that all French technicians should be authorized. Therefore, $Country(u) = \text{“FR”}$, should not be mined at this point.

Surprisingly, as we show next, current measures give a higher value to $Country(u) = \text{“FR”}$ than to $Job(u) = \text{“E”}$. Moreover, there are mining algorithms like APRIORI-SD [84] and APRIORI-C [80] that, when given as input the ABAC instance of Figure 4.1a, mine the rule $Country(u) = \text{“FR”}$. They mine this rule *even when all French technicians are marked as denied in the log.*

4.3.1 Over-permissiveness

We start with some definitions. Recall that, for an ABAC instance (U, P, A, D), a subset $S \subseteq U \times P$, and a rule r, the set $[\![r]\!]_S$ consists of all requests in S satisfying r.

Definition 54 (Confidence [4]) Let (U, P, A, D) be an ABAC instance. The *confidence* of a rule r is

$$Conf(r) = \frac{|[\![r]\!]_A|}{|[\![r]\!]_{U \times P}|}. \tag{4.1}$$

In the case where $|[\![r]\!]_{U \times P}| = 0$, we set $Conf(r) = 0$. □

Previous mining algorithms used confidence to measure a rule's quality [3, 4, 84, 80]. If a rule's confidence is high, then a large fraction of the requests covered by the rule has been authorized. According to these algorithms, a high confidence indicates that all the other requests covered by the rule should also be authorized.

In Figure 4.1a, the rules $Country(u) = ``\mathsf{FR}''$ and $Job(u) = ``\mathsf{E}''$ have confidence 0.75 and 0.67, respectively.

Definition 55 For a rule r, we call a *refinement of* r any rule of the form $r \wedge r'$, for some rule r'. □

A rule's refinement identifies subsets covered by the rule. We call two refinements $r \wedge r'$ and $r \wedge r''$ *semantically equivalent* if $[\![r \wedge r']\!]_{U\times P} = [\![r \wedge r'']\!]_{U\times P}$; otherwise, we call them *semantically different*. Since we assume only signatures with finitely many symbols and U and P can be assumed to be finite, a rule has only finitely many semantically different refinements.

Two refinements of the rule $Country(u) = ``\mathsf{FR}''$ are $Country(u) = ``\mathsf{FR}'' \wedge Job(u) = ``\mathsf{E}''$ and $Country(u) = ``\mathsf{FR}'' \wedge Job(u) = ``\mathsf{T}''$. These refinements have confidence 1 and 0, respectively.

Although $Country(u) = ``\mathsf{FR}''$ has a high confidence, a good ABAC mining algorithm should not mine this rule; it authorizes all French technicians, but none of them has even requested the permission. More precisely, $Country(u) = ``\mathsf{FR}''$ has the refinement $Country(u) = ``\mathsf{FR}'' \wedge Job(u) = ``\mathsf{T}''$ covering a significant set of requests, but with confidence 0. To describe these kind of rules, we introduce the following concept.

A rule is overly permissive *if one of its refinements covers a significant set of requests but has low confidence.*

An overly permissive rule goes against the principle of least privilege and should be replaced with rules that avoid the low-confidence refinement. In the case of Figure 4.1a, a policy containing the rule $Country(u) = ``\mathsf{FR}''$ should have instead the rules $Country(u) = ``\mathsf{FR}'' \wedge Job(u) = ``\mathsf{E}''$, $Country(u) = ``\mathsf{FR}'' \wedge Job(u) = ``\mathsf{M}''$, and $Country(u) = ``\mathsf{FR}'' \wedge Job(u) = ``\mathsf{S}''$.

To formally define over-permissiveness, we must agree on when a set of requests is significant and when a refinement has low confidence. These notions are not absolute and they depend on the ABAC instance. Hence, we let the policy administrator specify parameters T and K, which define when a set of requests is significant enough and when a refinement has low confidence.

Definition 56 Let $T \geq 1$ and $K \in [0,1]$. A rule is *overly permissive with respect to T and K* if there is a refinement $r \wedge r'$ of r with $|[\![r \wedge r']\!]_{U\times P}| \geq T$ and $Conf(r \wedge r') < K$. □

The values for T and K must be given as input to Rhapsody, so that Rhapsody can decide when a rule is overly permissive. We omit them when they are clear from the context.

In our experiments in Section 4.6 we found that, for an ABAC instance (U, P, A, D), a good value for K is around $|A| \,/\, |U \times P|$. Very high values are too harsh and many promising rules would be regarded as overly permissive. In a sparse log, there are many requests that have not been evaluated yet, so a refinement rarely has very high confidence. Analogously, very low values are too lenient and refinements with low confidence would not be regarded as overly permissive.

Regarding good values for T, one could argue that, in general, the best is 1, because this ensures that all refinements are considered. However, we are interested only in refinements that cover a significant number of requests. In ABAC instances with thousands of users and permissions and sparse logs, one can easily find refinements that cover 1 or 2 requests with confidence 0. Setting $T = 1$ could, therefore, unfairly classify promising rules as overly permissive. In general, the lower T is, the more likely it is for a rule to be overly permissive as more refinements are considered.

We now discuss suitable values for T and K for Figure 4.1a. The smallest refinement here has size 4, so we can let $T = 4$ to ensure that all significant refinements are considered when evaluating if a rule is overly permissive. For K, we can use $\frac{|A|}{|U \times P|} \approx 0.3$. If a refinement has a confidence below 0.3, then we cannot be convinced that all requests in that refinement should be authorized. We therefore fix $T = 4$ and $K = 0.3$ for this ABAC instance.

With the above choice of T and K, the rule $\mathit{Country}(u) = \text{"FR"}$ is overly permissive because one of its refinements, namely $\mathit{Country}(u) = \text{"FR"} \wedge \mathit{Job}(u) = \text{"T"}$, covers 4 requests but has confidence 0. In contrast, the rule $\mathit{Job}(u) = \text{"E"}$ is not overly permissive, since its two refinements have confidence above 0.3.

4.3.2 Reliability

Table 4.2 illustrates rule quality measures used by state-of-the-art mining algorithms. We can see that all of them depend just on the confidence of the rule, the total requests, and the total authorized requests. One can easily verify that the value of these measures is high whenever the confidence is high.

Despite extensive research on these measures, *none of them is able to quantify both confidence and over-permissiveness*. For example, in Figure 4.1a, all measures give a value to $\mathit{Country}(u) = \text{"FR"}$ that is higher than the value given to $\mathit{Job}(u) = \text{"E"}$. However, as we discussed above, $\mathit{Country}(u) = \text{"FR"}$ is overly permissive and $\mathit{Job}(u) = \text{"E"}$ is not. More generally, for any measure

Quality measure		Formula
Confidence	[4]	$Conf(r) = \frac{\lvert [\![r]\!]_A \rvert}{\lvert [\![r]\!]_{U\times P} \rvert}$
Likelihood ratio statistic	[37]	$2Supp(r)\,Conf(r)\log\left(\frac{Conf(r)}{\left(\frac{\lvert A\rvert}{\lvert U\times P\rvert}\right)}\right) +$ $2Supp(r)\,(1 - Conf(r))\log\left(\frac{1-Conf(r)}{\left(1-\frac{\lvert A\rvert}{\lvert U\times P\rvert}\right)}\right)$
Entropy	[119]	$-Conf(r)\log(Conf(r)) -$ $(1 - Conf(r))\log(1 - Conf(r))$
WRAcc	[91]	$\frac{Supp(r)}{\lvert U\times P\rvert}\left(Conf(r) - \frac{\lvert A\rvert}{\lvert U\times P\rvert}\right)$
Gini index	[29]	$-Conf(r)\,(1 - Conf(r))$

Table 4.2: Quality measures proposed for rule mining. Here, $Supp(r) = \lvert [\![r]\!]_{U\times P}\rvert$. All measures depend on $Conf(r)$, $Supp(r)$, $\lvert A\rvert$, and $\lvert U \times P\rvert$.

and any choice of T and K, there are scenarios where the measure gives high values to overly permissive rules.

To overcome these limitations, we propose *reliability*, a measure that quantifies not only the confidence of a rule, but also the confidence of all its significant refinements.

Definition 57 (Reliability) Let (U, P, A, D) be an ABAC instance. For $T \geq 1$, the *T-reliability of a rule r* is

$$Rel_T(r) = \min_{r' \in F_T(r)} Conf\left(r \,\wedge\, r'\right), \tag{4.2}$$

where $F_T(r) = \{r' : \lvert [\![r \,\wedge\, r']\!]_{U\times P}\rvert \geq T\}$. In the degenerate case of $F_T(r) = \emptyset$, define $Rel_T(r) = Conf(r)$. □

The parameter T corresponds to the same parameter T in the definition of over-permissiveness. The following observation, which is proven in Appendix B.2, gives the connection between reliability and over-permissiveness.

Observation 1 Let $T \geq 1$, $K \in [0, 1]$, and r be a rule. $Rel_T(r) \geq K$ iff $Conf(r) \geq K$ and r is not overly permissive with respect to T and K.

We compute the 4-reliability for $Country(u) = $ "FR" and $Job(u) = $ "E" for the ABAC instance of Figure 4.1a.

$$\begin{aligned}
&Rel_4(Country(u) = \text{“FR''})\\
&= \min \left\{ \begin{array}{l} Conf\big(Country(u) = \text{“FR''}\big)\,, \\ Conf\big(Country(u) = \text{“FR''} \;\wedge\; Job(u) = \text{“E''}\big)\,, \\ Conf\big(Country(u) = \text{“FR''} \;\wedge\; Job(u) = \text{“M''}\big)\,, \\ Conf\big(Country(u) = \text{“FR''} \;\wedge\; Job(u) = \text{“S''}\big)\,, \\ Conf\big(Country(u) = \text{“FR''} \;\wedge\; Job(u) = \text{“T''}\big) \end{array} \right\} \\
&= \min\{0.75, 1.0, 1.0, 1.0, 0.0\} = 0.0.
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&Rel_4(Job(u) = \text{“E''})\\
&= \min \left\{ \begin{array}{l} Conf\big(Job(u) = \text{“E''}\big)\,, \\ Conf\big(Job(u) = \text{“E''} \;\wedge\; Country(u) = \text{“FR''}\big)\,, \\ Conf\big(Job(u) = \text{“E''} \;\wedge\; Country(u) = \text{“US''}\big) \end{array} \right\} \\
&= \min\{0.67, 1.0, 0.5\} = 0.5.
\end{aligned} \tag{4.4}$$

For any measure in Table 4.2, $Country(u) = \text{“FR''}$ gets a higher score than $Job(u) = \text{“E''}$, despite $Country(u) = \text{“FR''}$ being overly permissive. However, $Rel_4(Country(u) = \text{“FR''}) < Rel_4(Job(u) = \text{“E''})$, as we have shown above.

The previous example and Observation 1 show that reliability achieves what other measures could not: it gives a high score to precisely those rules that have a high confidence and are not overly permissive.

4.4 Rhapsody

We now present Rhapsody, our ABAC mining algorithm. Rhapsody builds upon the algorithm APRIORI-SD [84]. We start with a brief overview of how APRIORI-SD can be used for ABAC mining.

4.4.1 APRIORI-SD

APRIORI-SD receives as input two parameters T' and K' and operates in three stages. We just summarize the main idea and refer the reader to the original paper [84].

1. Compute a set *FreqRules* of rules, such that $r \in FreqRules$ iff r covers at least T' requests.
2. Compute a subset $ConfRules \subseteq FreqRules$, such that $r \in ConfRules$ iff $Conf(r) \geq K'$.

3. Compute a subset $W \subseteq \mathit{ConfRules}$ by iteratively selecting from *ConfRules* the rule with highest weighted relative accuracy (WRAcc on Table 4.2), until W covers all authorized requests.

Although APRIORI-SD mines policies that generalize well, our experiments confirmed that it also mines overly permissive rules. APRIORI-SD uses the WRAcc measure to guide rule selection, which, as discussed in Section 4.5.2, may give high values to overly permissive rules. For example, when given $T' = 4$, $K' = 0.5$, and the ABAC instance of Figure 4.1a, APRIORI-SD outputs the overly-permissive rule $\mathit{Country}(u) = \text{“FR”}$.

4.4.2 Rhapsody algorithm

Rhapsody builds on APRIORI-SD by replacing its last two stages with two new stages that prevent the mining of overly permissive or unnecessarily large rules. Rhapsody takes as input an ABAC instance (U, P, A, D), $T \geq 1$, and $K \in [0, 1]$. Recall that T and K are the input parameters defining when a rule is overly permissive.

Rhapsody's three stages are as follows.

1. Compute the set *FreqRules*, such that $r \in \mathit{FreqRules}$ iff r covers at least T requests.
2. Compute the subset $\mathit{RelRules} \subseteq \mathit{FreqRules}$ of rules whose T-reliability is at least K and that do not cover a denied request.
3. Remove from *RelRules* all rules that have an equivalent shorter (i.e., smaller size) rule in *RelRules* and output the remaining rules.

We explain each stage in detail.

Stage 1 (Algorithm 1)

Compute the following:

- A set $\mathit{FreqRules} = \{r : |[\![r]\!]_{U \times P}| \geq T\}$.
- A function $n_{U \times P} : \mathit{FreqRules} \to \mathbb{N}$, such that $n_{U \times P}(r) = |[\![r]\!]_{U \times P}|$.
- A function $n_A : \mathit{FreqRules} \to \mathbb{N}$, such that $n_A(r) = |[\![r]\!]_A|$.

To compute *FreqRules* and $n_{U \times P}$, Rhapsody uses the APRIORI algorithm [3, 4] (not to be confused with APRIORI-SD). We give a brief overview of APRIORI and explain how Rhapsody uses it.

For $T' > 0$ and $\mathcal{F}$ a family of sets, we say that a set C is *T'-frequent in $\mathcal{F}$* if $|\{S \in \mathcal{F} : C \subseteq S\}| \geq T'$. APRIORI receives a family $\mathcal{F}$ of sets and a threshold T' and outputs

(i) all T'-frequent sets in $\mathcal{F}$ and

(ii) a function $n_{\mathcal{F}}$ mapping each T'-frequent set C in $\mathcal{F}$ to $|\{S \in \mathcal{F} : C \subseteq S\}|$.

Rhapsody computes *FreqRules* and $n_{U \times P}$ as follows. First, it computes for each request $(\mathsf{u}, \mathsf{p}) \in U \times P$ the set $\mathcal{A}(\mathsf{u}, \mathsf{p})$ of all atoms that (u, p) satisfies (Line 2). Then it invokes APRIORI on $\{\mathcal{A}(\mathsf{u}, \mathsf{p}) : (\mathsf{u}, \mathsf{p}) \in U \times P\}$ with the threshold T (Line 3). Afterwards, it uses APRIORI's output to compute the set *FreqRules* (Line 4). Finally, it computes n_A as follows. Initially, $n_A(r) = 0$, for all $r \in$ *FreqRules* (Lines 5–7). Then, for each $(\mathsf{u}, \mathsf{p}) \in A$ and each $\mathsf{r} \in$ *FreqRules* that covers (u, p), it increments $n_A(\mathsf{r})$ by 1 (Lines 8–12).

Algorithm 1 Rhapsody's first stage

```
 1: function Stage1($U$, $P$, $A$, $T$)
 2:     $\mathcal{F} \leftarrow \{\mathcal{A}(\mathsf{u},\mathsf{p}) : (\mathsf{u},\mathsf{p}) \in U \times P\}$
 3:     $FreqItemSets, n_{U \times P} \leftarrow$ APRIORI$(\mathcal{F}, T)$
 4:     $FreqRules \leftarrow$
            $\{\alpha_1 \wedge \ldots \wedge \alpha_k : \{\alpha_1, \ldots, \alpha_k\} \in FreqItemSets\}$
 5:     for $r \in FreqRules$ do
 6:         $n_A(r) \leftarrow 0$
 7:     end for
 8:     for $(\mathsf{u},\mathsf{p}) \in A$ do
 9:         for $r \in FreqRules$ s. t. $r$ covers $(\mathsf{u},\mathsf{p})$ do
10:             $n_A(r) \leftarrow n_A(r) + 1$
11:         end for
12:     end for
13:     return $FreqRules$, $n_{U \times P}$, $n_A$
14: end function
```

Stage 2 (Algorithm 2)

This stage computes the set *RelRules* of all rules in *FreqRules* whose T-reliability is at least K and that do not cover a denied request.

Definition 58 For two rules r_1 and r_2, r_2 *proves that* $Rel_T(r_1) < K$ if

(i) r_2 is a refinement of r_1,

(ii) $|[\![r_2]\!]_{U \times P}| \geq T$, and

(iii) $Conf(r_2) < K$.

□

Observation 2 For a rule r_1, if a rule proves that $Rel_T(r_1) < K$, then r_1's T-reliability is less than K.

In this stage, Rhapsody computes from *FreqRules* two subsets: *UnrelRules* and *CoversDenied*. A rule $r_1 \in \mathit{FreqRules}$ is added to *UnrelRules* if some rule in *FreqRules* proves that $\mathit{Rel}_T(r_1) < K$. The rule r_1 is added to *CoversDenied* if it covers a denied request. Afterwards, Rhapsody computes the set $\mathit{RelRules} = \mathit{FreqRules} \setminus (\mathit{UnrelRules} \cup \mathit{CoversDenied})$.

Algorithm 2 Rhapsody's second stage

```
1: function Stage2($\mathit{FreqRules}$, $n_{U\times P}$, $n_A$, $T$, $K$)
2:     $\mathit{UnrelRules} \leftarrow \emptyset$
3:     $\mathit{CoversDenied} \leftarrow \emptyset$
4:     for $r_1, r_2 \in \mathit{FreqRules}$ do
5:         if $r_2$ proves that $\mathit{Rel}_T(r_1) < K$ then
6:             $\mathit{UnrelRules} \leftarrow \mathit{UnrelRules} \cup \{r_1\}$
7:         end if
8:         if $r_1$ covers a denied request then
9:             $\mathit{CoversDenied} \leftarrow \mathit{CoversDenied} \cup \{r_1\}$
10:        end if
11:    end for
12:    $\mathit{RelRules} \leftarrow \mathit{FreqRules} \setminus (\mathit{UnrelRules} \cup \mathit{CoversDenied})$
13:    return $\mathit{RelRules}$
14: end function
```

Stage 3 (Algorithm 3)

This stage removes redundant rules from *RelRules* and outputs the result.

Definition 59 A rule r_1 is *equivalent* to a rule r_2 if $[\![r_1]\!]_{U\times P} = [\![r_2]\!]_{U\times P}$. □

Observation 3 Two rules $r_1, r_2 \in \mathit{RelRules}$ are equivalent iff $r_1 \wedge r_2 \in \mathit{FreqRules}$ and $n_{U\times P}(r_1) = n_{U\times P}(r_2) = n_{U\times P}(r_1 \wedge r_2)$.

In this final stage, Rhapsody computes a set *Subsumed* of redundant rules from the set *RelRules*. For this, each pair of rules $r_1, r_2 \in \mathit{RelRules}$ is analyzed. If r_2 is both shorter than and equivalent to r_1, then r_1 is inserted into *Subsumed*. Then Rhapsody computes $\mathit{ShortRules} = \mathit{RelRules} \setminus \mathit{Subsumed}$.

Rhapsody's parameters. Rhapsody receives as input two parameters T and K, which guide the selection of mined rules. The values for these parameters depend on the ABAC instance and affect how well the mined policy generalizes. To find the best values, we recommend evaluating Rhapsody as described in Section 4.5.3 with different values and then choosing those values for which Rhapsody mines the policy that generalizes best. In our ex-

Algorithm 3 Rhapsody's third stage

```
1: function $Stage3(RelRules, n_{U \times P}, n_A, FreqRules)$
2:     $Subsumed \leftarrow \emptyset$
3:     for $r_1, r_2 \in RelRules$ do
4:         if $r_1$ and $r_2$ are equivalent and
5:           $r_2$ is shorter than $r_1$ then
6:             $Subsumed \leftarrow Subsumed \cup \{r_1\}$
7:         end if
8:     end for
9:     $ShortRules \leftarrow RelRules \setminus Subsumed$
10:    return $ShortRules$
11: end function
```

periments, the best values for T and K are respectively around $0.01 * |U \times P|$ and $|A| \ / \ |U \times P|$.

Rhapsody's performance. The time complexity of Rhapsody's first stage is determined by APRIORI's time complexity, which is $O(|U \times P| * L)$ [4], where L a bound on the maximum number of semantically different rules that a request may satisfy. Observe that we assume only signatures with finitely many symbols and U and P are assumed to be finite, so L is always finite. Since Rhapsody's second and third stage's time complexity is quadratic in $|FreqRules| * L$. In the worst case, $|FreqRules|$ is $O\left(|U \times P| * L\right)$. We conclude then that Rhapsody's time complexity is $O(|U \times P|^2 * L^4)$.

Note that L grows exponentially in the number of attributes, so Rhapsody cannot mine logs with many attributes. Fortunately, in access control scenarios, the number of attributes is usually small, namely less than 20 [102]. Even for Amazon's logs, which are the largest logs in our case studies, the number of attributes is less than 15 [82, 93]. This allows Rhapsody to mine from any of the ABAC instances in our experiments within 24 hours. Moreover, there already exists *feature selection* techniques for mining access control policies [59], where one can extract a small subset of attributes that are relevant for deciding authorization before executing Rhapsody.

Finally, note that an ABAC mining algorithm does not need to mine policies in real time and organizations specify their access control policies only infrequently. Since the implemented policy is security critical, an offline algorithm providing guarantees is preferable to an online algorithm providing overly-permissive or unnecessarily long rules.

Policy simplification. After executing Rhapsody on (U, P, A, D), it is still possible to prune redundant rules from π by keeping only a small subset that covers all requests covered by π. In our experiments, we found that the best way to build this subset is using APRIORI-SD's last stage. This stage iteratively selects the rule with highest weighted relative accuracy (WRAcc in Table 4.2) from π until all selected rules cover A. Algorithm 4 gives the details of APRIORI-SD's last stage applied to our setting.

Algorithm 4 Policy simplification

```
1: function $simplify(\pi, U, P, A)$
2:     $uReqs \leftarrow U \times P$
3:     $uLog \leftarrow A$
4:     $simpPolicy \leftarrow \emptyset$
5:     while $uLog \neq \emptyset$ and $\pi \neq \emptyset$ do
6:         $r \leftarrow argmax_{r'}\{WRAcc(r') : r' \in \pi\}$
7:         $uReqs \leftarrow uReqs \setminus \{(\mathsf{u}, \mathsf{p}) : r \text{ covers } (\mathsf{u}, \mathsf{p})\}$
8:         $uLog \leftarrow uLog \setminus \{(\mathsf{u}, \mathsf{p}) : r \text{ covers } (\mathsf{u}, \mathsf{p})\}$
9:         $\pi \leftarrow \pi \setminus \{r\}$
10:        $simpPolicy \leftarrow simpPolicy \cup \{r\}$
11:    end while
12:    return $simpPolicy$
13: end function
```

Correctness. The following theorem, whose proof is given in Appendix B.2, establishes Rhapsody's main property: Rhapsody mines exactly those rules that do not cover a denied request, have a high reliability, and are shorter than any other equivalent rule.

Theorem 60 A rule r is output by Rhapsody iff

(i) $|[\![r]\!]_{U \times P}| \geq T$,

(ii) r covers no denied request,

(iii) $Rel_T(r) \geq K$, and

(iv) there is no rule r' that is both shorter than and equivalent to r, with $Rel_T(r') \geq K$.

4.5 Evaluating generalization

We discuss next two common methods for evaluating the precision and generalization of models mined from logs. We explain why they are inadequate and propose a better alternative: *universal cross-validation*.

4.5.1 Limitations of using an organizational security policy for evaluation

One evaluation method for ABAC mining algorithms uses only ABAC instances where the organizational security policy is already known. In this method, a mining algorithm is given the instance as input and the set of requests authorized by the mined policy is compared with the set of requests authorized by the security policy. For example, a policy mined from the instance in Figure 4.1a, would be compared with the security policy, which is described by the light-green shaded rectangles in Figure 4.1b. We explain next why this is *not an adequate approach* to evaluate mining algorithms.

In ABAC instances with sparse logs, there are rules covering a significant number of requests, but none of which occurs in the log. We call these rules *uncertain*. In Figure 4.1a, the rules $Country(u) = \text{“US''} \wedge Job(u) = \text{“S''}$ and $Country(u) = \text{“FR''} \wedge Job(u) = \text{“T''}$ are uncertain.

Observe that a mining algorithm cannot decide if an uncertain rule is part of the organizational security policy. Therefore, whenever a mining algorithm mines an uncertain rule, there is the risk that the rule is not part of the security policy, and hence, the rule can incorrectly authorize requests. As discussed in Section 4.2.3, mining algorithms should opt for incorrect denial in this case. Since the requests covered by the uncertain rule do not occur in the log, this means that these requests happen infrequently, so the work required by the policy administrator is low in case some of these requests are incorrectly denied.

Mining algorithms should mine a rule only if the log provides evidence for that rule. Therefore, if an algorithm mines an uncertain rule, then it should be penalized, *even when this rule happens to be part of the organizational security policy*, like the rule $Country(u) = \text{“US''} \wedge Job(u) = \text{“S''}$ in Figure 4.1b.

Summarizing, the approach of evaluating mined policies using the organizational security policy is inadequate as it may not penalize algorithms mining uncertain rules. In contrast, universal cross-validation, which we present in Section 4.5.3, uses metrics that penalize algorithms mining uncertain rules.

4.5.2 Limitations of using cross-validation on logs

A standard method for evaluating the precision and the generalization of models mined from logs is cross-validation [47, 135, 42]. In its simplest form, cross-validation splits the log into a *training* and a *testing log*. Only the training log is given to the algorithm. Once the algorithm finishes, the mined policy is evaluated on the testing log using performance metrics like the *true positive rate (TPR)* and *false positive rate (FPR)* [111]. The true positive rate is the fraction of authorized requests in the testing log that are correctly authorized by the mined policy. The false positive rate is the fraction of denied requests in the testing log that are incorrectly authorized by the mined policy. Figure 4.2 illustrates cross-validation on logs.

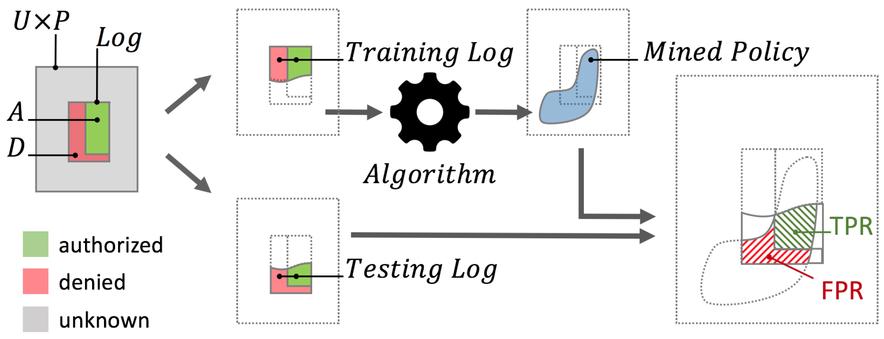


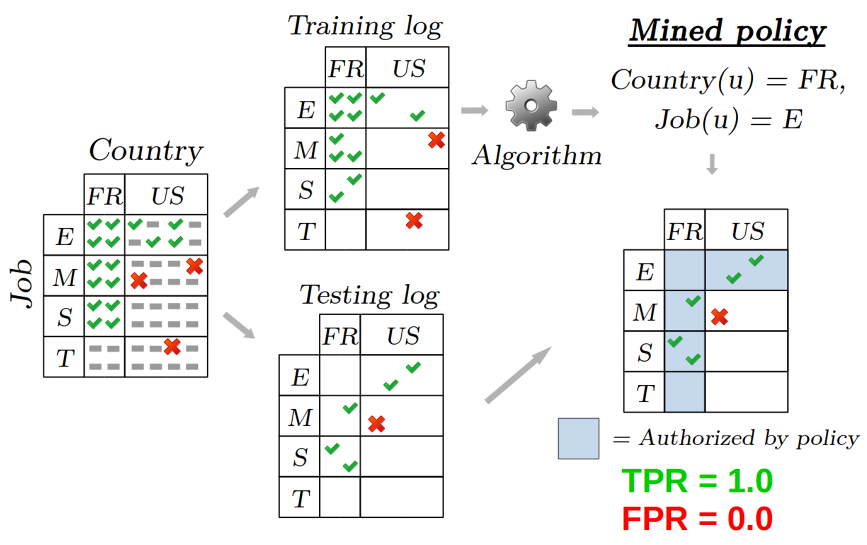
Figure 4.2: Cross-validation on logs. **©2018, IEEE. Reprinted with permission.**

If we use cross-validation to evaluate ABAC mining algorithms, then we would give *only the training log* as input to the mining algorithm, and then we would evaluate the mined policy *only on the requests in the testing log*. Mining algorithms would not receive in the input requests outside the log and mined policies would not be evaluated on requests not occurring in the log.

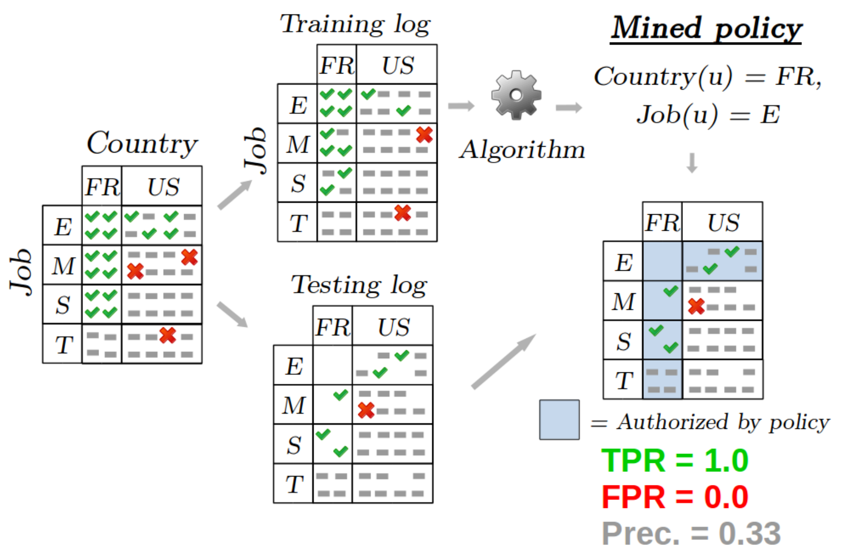
In the context of ABAC mining from sparse logs, cross-validation on logs validates overly permissive rules. To illustrate this, we use the ABAC instance from Figure 4.1a and the classification-tree learning algorithm (CTA) [29, 117], which is used to train classification trees from labeled data. It is easy to extract rules from a classification tree explaining how the tree classifies instances. Hence, CTA is suitable for ABAC mining. We illustrate next what happens if we use cross-validation on logs in this scenario.

Suppose that we split the log into a training and a testing log, as shown in Figure 4.2. If we use CTA on the training log, then we obtain the policy $\pi = \{Country(u) = \text{“FR”}, Job(u) = \text{“E”}\}$, which correctly authorizes and denies all requests in the testing log. Since no French technician has requested access yet, it seems like π grants and denies access perfectly. However, π contains the rule $Country(u) = \text{“FR”}$ and, as discussed in Section 4.5.2, this rule is questionable, as it authorizes all French technicians with no evidence from the log. Cross-validation does not evaluate π on those requests because the log does not contain requests from French technicians.

The problem of using cross-validation on logs is that it does not account for the *precision* of the mined policy. Precision is a standard machine-learning metric that measures the ratio of authorized requests covered by the policy to the total number of requests covered by the policy [111]. A mined policy



(a) In cross-validation on logs, algorithms are only given as input the log $A \cup D$ of an ABAC instance (U, P, A, D), and therefore they cannot evaluate the over-permissiveness of the mined rules. Moreover, the testing log does not include requests outside the original log. As a consequence, algorithms mining overly permissive rules, like $Country(u) =$ "FR", are *not* penalized.



(b) In universal cross-validation, algorithms are now given as input (U, P, A, D). Moreover, the testing log is expanded with requests not occurring in the original log and the precision of the mined policy is measured. With these changes, algorithms mining overly permissive rules, like $Country(u) =$ "FR", are penalized.

Figure 4.3: Cross-validation on logs and universal cross-validation. **©2018, IEEE. Reprinted with permission.**

may correctly authorize all requests in the log, but it may also have a low precision and authorize an unnecessarily large number of non-logged requests. A good ABAC mining algorithm should be able to anticipate from the log which requests are likely to occur in the future that can also be authorized, but without authorizing requests for which the log gives no evidence.

Although our critique addresses a simple form of cross-validation, it directly extends to more sophisticated forms of cross-validation, like K-fold cross-validation and leave-one-out cross-validation [60].

4.5.3 Universal cross-validation

Cross-validation on logs is problematic because it ignores all requests that are not in the log. Therefore, to validate an ABAC mining algorithm, we propose to include these ignored requests in the validation process. A good ABAC mining algorithm should then mine policies that strike a balance among the following properties:

- Attain a high true positive rate (TPR) by authorizing as many authorized requests in the testing log as possible.
- Attain a low false positive rate (FPR) by authorizing as few denied requests in the testing log as possible.
- Attain a high precision by authorizing as few requests outside the log as possible.

Based on these three properties, we present *universal cross-validation*, a new method for evaluating ABAC mining algorithms, illustrated in Figure 4.4. Assume given an ABAC mining algorithm and an ABAC instance (U, P, A, D). Sample uniformly at random two training sets $Tr(A)$ and $Tr(D)$ from A and D, respectively. Let $Ts(A) = A \setminus Tr(A)$ and $Ts(D) = D \setminus Tr(D)$. Give $(U, P, Tr(A), Tr(D))$ as input to the algorithm. Once the algorithm outputs a policy π, evaluate π's *F1 score* and π's *false positive rate* as follows:

$$TPR(\pi) = \frac{|[\![\pi]\!]_{U\times P} \cap Ts(A)|}{|Ts(A)|} \quad FPR(\pi) = \frac{|[\![\pi]\!]_{U\times P} \cap Ts(D)|}{|Ts(D)|} \tag{4.5}$$

$$Prec(\pi) = \frac{|[\![\pi]\!]_{U\times P} \cap Ts(A)|}{\left|[\![\pi]\!]_{(U\times P)} \cap \overline{Tr(A) \cup Tr(D)}\right|} \tag{4.6}$$

$$F1(\pi) = \frac{2 * TPR(\pi) * Prec(\pi)}{TPR(\pi) + Prec(\pi)} \tag{4.7}$$

Recall that $[\![\pi]\!]_{U\times P}$ is the set of requests in $U \times P$ authorized by π. The denominator in $Prec(\pi)$ is the number of requests authorized by π and outside the training sets. We use the F1 score and the FPR to measure how well a policy generalizes.

Observe that when computing the scores above, all requests have the same weight. However, the policy administrator can add more weight to requests for permissions that are more critical than others, so that policies incorrectly authorizing or denying critical requests are more heavily penalized.

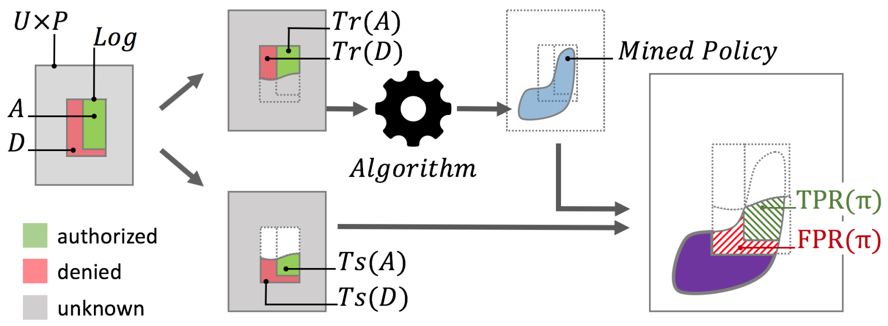


Figure 4.4: Universal cross-validation. **©2018, IEEE. Reprinted with permission.**

Figure 4.3 compares cross-validation on logs with universal cross-validation. First, the input to the algorithm is not only the log, but also the set of all possible requests. Second, universal cross-validation enlarges the set over which algorithms are evaluated. This set now includes a sample from all unknown requests (see the solid purple area in Figure 4.4).

4.6 Experiments

In this section, we experimentally compare Rhapsody with other ABAC mining algorithms and compare universal cross-validation with standard cross-validation. The results are the following.

Sparse logs. Previous ABAC mining algorithms generalize poorly when mining from sparse logs.

Over-permissiveness. Subgroup-discovery algorithms mine overly-permissive rules.

Rule size. Current machine-learning algorithms mine unnecessarily large rules.

Cross-validation. Cross-validation finds policies that generalize worse than those found using universal cross-validation.

Moreover, we show that Rhapsody is the only algorithm that is capable of mining *succinct* rules from *sparse logs*, without mining *overly permissive* rules.

4.6.1 ABAC instances

We use ABAC instances from four case studies for our evaluation. We summarize them briefly and refer to Appendix B.1 for details.

Amazon 1. We built four instances from an access log provided by Amazon in Kaggle, a platform for predictive modeling competitions [81, 82]. The log contains more than 12,000 users and 7,000 permissions and is very sparse. For any permission, less than 7% of all users have requested access.

Amazon 2. We built seven instances from another log provided by Amazon in the UCI machine learning repository [93]. The log contains more than 36,000 users and 27,000 permissions. For any permission, less than 10% of all users have requested access.

University. We used a log of students accessing a computer lab at ETH Zurich. The log contains more than 50,000 users. Less than 1% of the students have requested access and less than 5% of them were denied access. Experiments were approved by ETH's security administration.

Basic Organization. We generated five simple synthetic instances, where users and permissions contain only one attribute. All users, except those with a specific attribute value, are authorized to have any permission. The logs in these instances contain approximately 50% of all possible requests and less than 5% of the logged requests are denied.

4.6.2 Algorithms

We compare Rhapsody with the following algorithms.

1 Xu and Stoller's ABAC miner [132].

2 CN2 [37], an algorithm for learning classification rules. We used the implementation provided by the Orange data mining library [45].

3 The classification-tree learning algorithm (CTA) provided by the scikit-learn library [110].

4 APRIORI-SD [84], as described in Section 4.4.

We give an overview of these algorithms in Section 4.7.

4.6.3 Evaluation methodology

For each ABAC mining algorithm and each ABAC instance, we ran cross-validation on logs five times and then computed the average FPR, average F1 score, average TPR, and average size of the mined policies. Similarly, we ran universal cross-validation five times and then computed the average of the same metrics. In both types of cross-validation, we split the log into a training log and a testing log containing 80% and 20% of the requests, respectively.

Each mining algorithm has a set of parameters that affect the F1 score, FPR, TPR, and the size of the mined policy. We evaluated each algorithm with different values for these parameters. Among all the policies mined by the algorithm, we selected the one with the highest F1 score, subject to an FPR < 0.05.

APRIORI-SD and Rhapsody were evaluated on machines with a 2,8 GHz 8-core CPU and 32 GB of RAM. CTA and CN2 were evaluated on machines with a 3,8 GHz 8-core CPU and 32 GB of RAM. All algorithms were given a time limit of 24 hours for each instance. CN2 timed out when mining the instances from Amazon 2 and University. Xu and Stoller's miner timed out when mining the instances from University.

4.6.4 Results

Figure 4.5 compares the F1 score of policies selected using cross-validation on logs with the F1 score of policies selected using universal cross-validation. In most of the cases, the policy selected by universal cross-validation has an FPR equal to 0 (not shown in Figure). Figures 4.6a, 4.6b, and 4.7 compare, respectively, the F1 scores, sizes, and TPRs of the policies mined by each algorithm on the instances of each case study. The FPR was close to 0 for almost all mined policies. From these figures, we make four observations.

Cross-validation. In 80% of the cases, the policy selected by universal cross-validation has an F1 score greater than or equal to the policy selected by cross-validation on logs. In most of the cases, the improvement is at least 30%. Observe that the improvement holds for *any ABAC mining algorithm*.

Sparse logs. Xu and Stoller's ABAC miner generalizes poorly. In most of the cases, policies mined by their miner attain an F1 score equal to 0. This is because this algorithm was designed mainly for mining from *non-sparse* logs, where a large percentage of all access requests have already been decided. In contrast, Rhapsody's F1 score is among the highest, for almost all instances.

Over-permissiveness. APRIORI-SD mines overly permissive rules. Consider the results for the instances of the basic organization in Figure 4.7. The TPR of the policies mined by APRIORI-SD is below 0.4 whereas the TPR of the policies mined by Rhaposdy is close to 1. This is because APRIORI-SD uses the confidence measure to mine rules. In Section 4.3, we explained how this measure leads to mining policies with overly permissive rules, which

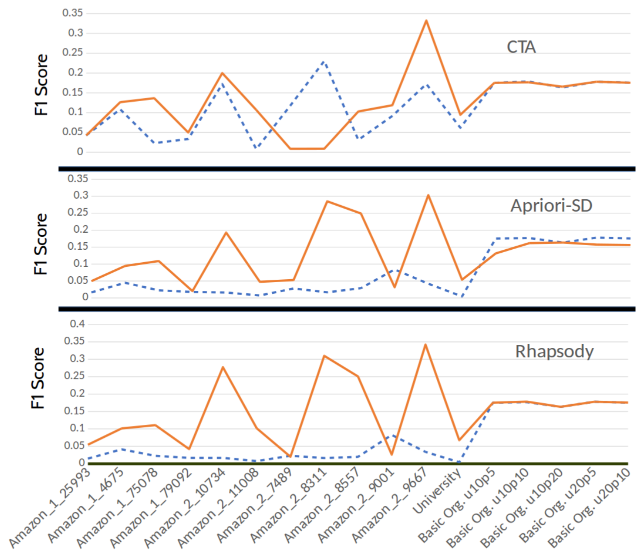


Figure 4.5: Comparison of F1 score of policies selected using cross-validation on logs versus F1 score of policies selected using universal cross-validation. Policies with *higher* F1 score are better as they are more accurate in deciding requests outside the log. **©2018, IEEE. Reprinted with permission.**

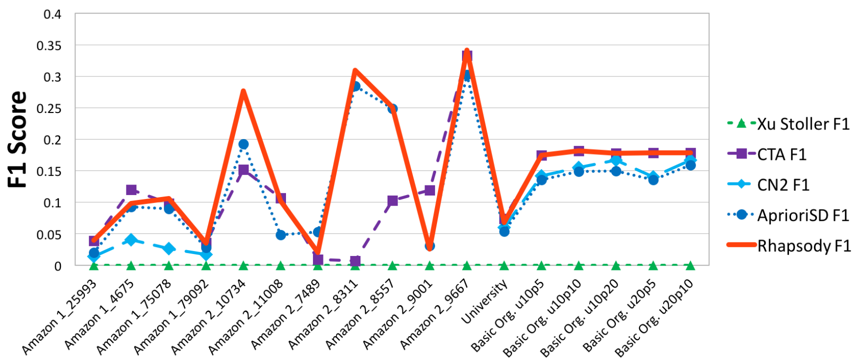
yielded FPRs above 0.1. The only policies mined by APRIORI-SD that attained FPR < 0.05, as we required in Section 4.6.3, attained a TPR < 0.4.

Rule size. CTA and CN2 mine unnecessarily large rules. In the Amazon 1 instances, the policies mined by CN2 have a size at least twice as large as those mined by RHAPSODY and those mined by CTA are 10 times larger than those mined by RHAPSODY. This cannot be fixed by expanding CTA or CN2 with RHAPSODY's third stage, which searches for each mined rule, the shortest equivalent rule. This is because these algorithms cannot compute for each rule the set of equivalent rules.

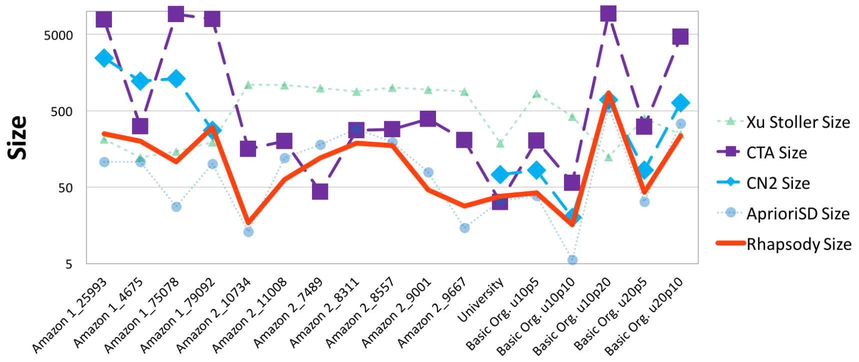
We conclude that RHAPSODY is the only ABAC mining algorithm capable of mining *succinct* rules from *sparse logs*, without mining *overly permissive rules*. Competing approaches mine rules that generalize poorly, mine unnecessarily large rules, or mine overly permissive rules.

4.7 Related work

Numerous algorithms have been proposed to mine policies from existing assignments of permission to users or from logs recording which users have required which permissions for their jobs. The approaches taken have been pri-



(a) Average F1 score of the policies mined by ABAC mining algorithms. Policies with a *higher* F1 score are better as they are more accurate in deciding requests outside the log.



(b) Average sizes of the policies mined by all ABAC mining algorithms. The sizes are logarithmically scaled. *Smaller* policies are better, as they are easier to maintain and audit.

Figure 4.6: F1 score and size of the policies mined by ABAC mining algorithms. ©2018, IEEE. **Reprinted with permission.**

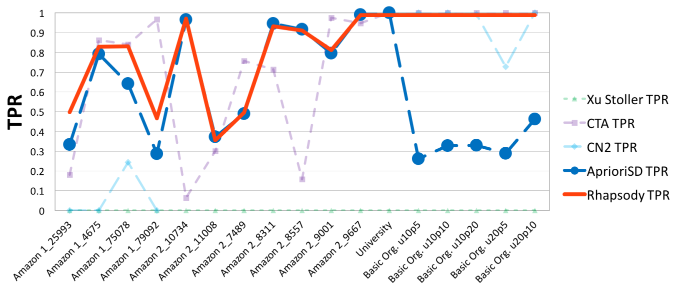


Figure 4.7: Average TPR of the policies mined by ABAC mining algorithms. Observe that the policies mined by APRIORI-SD have a TPR below 0.4 for the basic organization instances, whereas those mined by Rhapsody have a TPR close to 1. ©2018, IEEE. **Reprinted with permission.**

marily oriented towards role-based access control (RBAC), e.g., [101, 59, 103], and more recently towards ABAC [132, 133, 82]. Moreover, there are machine learning algorithms that learn models from sets of labeled requests, e.g. [29, 112, 37, 71, 59, 103]. The models learned by these algorithms generalize well and can be adapted to ABAC mining. We discuss them next.

Xu and Stoller's ABAC miner. Xu and Stoller have proposed an ABAC mining algorithm [132, 133] that mines ABAC policies from access control matrices and logs. Their algorithm considers only the case of logs that contain a large fraction of the authorized requests. In contrast, RHAPSODY can mine succinct policies that generalize well, even from sparse logs.

Rule learning. Rule learning addresses the following problem: Given a set of requests, where each request is labeled as positive or negative, find a set of rules that describe the requests labeled as positive. Several algorithms have been proposed for this, such as CN2 [37] and RIPPER [38]. They all work iteratively, where each iteration learns one rule at a time. In each iteration, the algorithm learns a rule r by computing a series of rules $r_0, r_1, \cdots,$ r_k, where $r_0 = \textbf{true}$, $r_{i+1} = r_i \land \alpha_i$, for $i \leq k$, and $r_k = r$. The atom α_i is chosen in a way that r_{i+1} maximizes a rule quality measure. After r_k is computed, all requests that satisfy r_k are removed and the algorithm starts learning another rule. This is repeated until all positive requests are covered or a given termination condition is satisfied.

The main limitation of these algorithms stems from their greedy behavior. There may be a high-quality rule where each of its atoms has a low quality, according to the quality measure. These rules will not be mined by the rule learning algorithm. RHAPSODY, in contrast, uses ideas from association rule mining [3, 4]. This guarantees that if a high-quality rule is very often satisfied, then RHAPSODY will find it, irrespective of the quality of its atoms. Hence, RHAPSODY can discover rules that are ignored by rule learning algorithms and can propose more accurate and more succinct rules.

Classification trees. A classification tree is a function encoding a partition of a set of labeled requests. Each partition has an associated label. To predict a value for a new request, the classification tree finds the partition where the new request belongs and uses the label associated to that partition as prediction. Algorithms for mining classification trees [29, 112] yield trees that generalize well. Moreover, one can easily extract rules from these trees. However, as our experiments demonstrate, these rules are unnecessarily long. RHAPSODY, in contrast, keeps track of all possible ways to specify a rule and at the end selects the most succinct one.

Random forests and neural networks. Classification trees are prone to overfitting because of their low bias and high variance [60]. For this reason, random forests are recommended. A random forest is a collection of classification trees trained over subsamples of the data. The classification decision of a random forests is obtained from the classification decisions of each tree, usually by a majority vote.

For this reason, we cannot apply the algorithms proposed by the winners of the Kaggle competition [83] for ABAC mining. Their models involve mixtures of 15 different models, including classification trees and logistic models. The competition only measured models according to how well the models generalized and not the simplicity of the rules proposed. The requirement of mining simple rules makes other techniques like neural networks unsuitable for ABAC mining.

Subgroup discovery. Subgroup discovery algorithms mine frequent and statistically significant rules from a set of labeled requests. These algorithms [11, 92, 84] require as input a threshold T for the number of requests that must satisfy a rule to be considered as frequent. Moreover, the rules must be statistically significant: the distribution of the requests' labels satisfying the rule must differ significantly from the distribution of all requests' labels. These algorithms compute *all* statistically significant rules that are satisfied by at least T requests.

All subgroup discovery algorithms use rule quality measures that depend just on the confidence of the rule. This results in these algorithms mining overly permissive rules. For example, the subgroup discovery algorithm APRIORI-SD uses the WRAcc measure for rule selection. As a result, it mines the overly permissive rule $Country(u) = ''\mathsf{FR}''$ when given as input the ABAC instance of Figure 4.1a. Our experiments with the basic organization (Section 4.6.4) also show that APRIORI-SD mines overly permissive rules. In contrast, Rhapsody uses our new measure, reliability, for rule selection. Reliability is guaranteed to select rules with high confidence that are not overly permissive (Theorem 1).

4.8 Conclusion

Mining ABAC policies from logs can identify future access requests that should be authorized. To get the maximum benefit from this, one should mine policies before a large fraction of all requests have been decided. When the log contains limited information, we observed two phenomena. First, cross-validation on logs is insufficient as it validates policies with overly permissive rules. Second, state-of-the-art algorithms mine policies with overly permissive rules or unnecessarily large rules.

We proposed universal cross-validation as a method for evaluating mined policies. This method penalizes policies with overly permissive rules but without causing mining algorithms to overfit logs. We also proposed a new measure, reliability, that quantifies better than standard measures how overly permissive a rule is. Based on reliability, we developed Rhapsody, which mines exactly all rules that cover a large number of requests, have a reliability above a given threshold, and have no shorter equivalent rule. When compared with other ABAC mining algorithms, Rhapsody mines policies that generalize better and have smaller size.

Chapter 5

Unicorn: Universal access-control policy mining

5.1 Introduction

5.1.1 The problem of designing policy miners

Numerous access control policy languages have been proposed over the last decades, e.g., RBAC (Role-Based Access Control) [52], ABAC (Attribute-Based Access Control) [68], XACML (eXtended Access-Control Markup Language) [64], and new proposals are continually being developed, e.g., [134, 104, 25, 35, 21]. To facilitate the policy specification and maintenance process, policy miners have been proposed, e.g., [59, 133, 96, 30, 61, 39, 103, 82].

Designing a policy miner is challenging and requires sophisticated combinatorial or machine-learning techniques. Moreover, policy miners are tailor-made for the specific policy language they were designed for and they are inflexible in that any modification to the miner's requirements necessitates the miner's redesign and reimplementation. For example, miners that mine RBAC policies from access control matrices [59] are substantially different from those that mine RBAC policies from access logs [103]. As evidence for the difficulty of this task, despite extensive work in policy mining, no policy miner exists for XACML [64], which is a well-known standard for access control specification. Therefore, each organization that wishes to benefit from policy mining faces the challenge of designing a policy miner that fits its own policy language and its own requirements. This problem, which we examine in Section 5.2, is summarized with the following question: *is there a more general and more practical method to design policy miners?*

5.1.2 Contribution

We propose a radical shift in the way policy miners are built. Rather than designing specialized mining algorithms, one per policy language, we propose Unicorn, *a universal method for building policy miners*. Using this method, the designers of policy miners no longer need to be experts in machine learning or combinatorial optimization to design effective policy miners. Our method gives a step-by-step procedure to build a policy miner from just the *policy language* and an *objective function* that measures how well a policy fits an assignment of permissions to users.

Let Γ be a policy language. We sketch in Figure 5.1 the workflow for designing a policy miner for Γ using Unicorn.

Policy language (Section 5.3) The designer specifies a *template formula for* Γ in a fragment $\mathcal{L}$ of first-order logic. Template formulas will be explained in Definition 62.

Objective function (Section 5.4) The designer specifies an objective function L that measures how well a policy fits a permission assignment.

Probability distribution (Section 5.4) From φ and L, we define a probability distribution $\mathbb{P}$ on permission assignments and policies. Recall that a permission assignment is a relation between the set of users and the set of permissions. The desired policy miner is a program that receives as input a permission assignment *Auth* and aims to compute the most likely policy conditioned on the given permission assignment; that is, the policy $\mathfrak{I}$ that maximizes $\mathbb{P}(\mathfrak{I} \mid Auth)$.

Approximation distribution (Section 5.5) Since this maximization is intractable, we propose the use of *mean-field variational inference* [26, 23] to derive an iterative procedure that computes a distribution q on policies that approximates $\mathbb{P}(\mathfrak{I} \mid Auth)$. Computing $\arg\max_{\mathfrak{I}} q(\mathfrak{I})$ is much easier than computing $\arg\max_{\mathfrak{I}} \mathbb{P}(\mathfrak{I} \mid Auth)$ as it only takes time linear in φ's size.

Policy miner template (Section 5.6) The desired policy miner is then a procedure that computes and maximizes q.

Using Unicorn, the policy miner designer need not understand variational inference or the logic behind the computation of q. The designer is only required to specify a template formula for the policy language, specify the objective function, and implement Algorithm 5 using Lemma 79. Our examples show that specifying the template formula in $\mathcal{L}$ amounts to a straightforward encoding of the policy language's semantics in first-order logic and

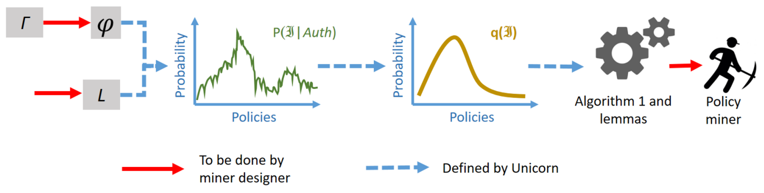


Figure 5.1: Workflow for designing a policy miner using Unicorn.

that the implementation of Algorithm 5 requires only basic knowledge of probability theory.

We emphasize that the Unicorn approach is substantially different from Rhapsody. Rhapsody is a policy miner whereas Unicorn is a universal method for building policy miners. In particular, Rhapsody receives as input an ABAC instance and outputs an ABAC policy. In Unicorn, the input includes *a template formula* that represents a policy language and the output is *a policy miner*. Unicorn requires a change in the way we model policy languages. In the previous chapters, policies were modeled using formulas and policy languages were sets of policies. In this chapter, we model policy languages with *formulas* and policies in those languages with *structures*. We explain this in detail in Section 5.3.

5.1.3 Applications and evaluation

In Section 5.7, we show how Unicorn facilitates the design of miners by building miners for different policy languages like RBAC, ABAC, and RBAC with user attributes. Furthermore, in Sections 5.8 and 5.9, we build policy miners for RBAC with spatio-temporal constraints and an expressive fragment of XACML. For these two languages, no miner existed before.

In Section 5.10, we conduct an extensive experimental evaluation using datasets from all publicly available real-world case studies on policy mining. We compare the miners we built with state-of-the-art miners on real-world as well as synthetic datasets. The true positive rates of the policies mined by our miners are within 5% of the true positive rates of the policies mined by the state of the art. For policy languages like XACML or RBAC with spatio-temporal constraints, the true positive rates are above 75% in all cases and above 80% in most of them. The false positive rates are always below 5%. In the case of mining ABAC policies, we mine policies with a substantially smaller size and higher precision than those mined by the state of the art. This demonstrates that, using Unicorn, we can build a wide variety of policy miners, including new ones, that are competitive with or even better than the state of the art.

We examine related work, draw conclusions, and discuss future work in Sections 5.11 and 5.12.

5.2 The problem of designing policy miners

Status quo: specialized solutions

Numerous policy languages exist for specifying access control policies, which fulfill different organizational requirements. Moreover, new languages are continually being proposed. Some of them allow the formulation of new concepts, like extensions of RBAC that can express temporal and spatial constraints [113, 34, 90, 125, 31, 6, 41]. Other languages facilitate policy specification in more specialized settings like distributed systems [64, 126] or social networks [56].

Policy mining researchers have proposed miners for a variety of policy languages. Moreover, for some policy languages, multiple miners have been developed that optimize different objectives. For example, initial RBAC miners mined policies with a minimal number of roles [127, 95, 116, 129, 136]. Subsequent RBAC miners mined policies that are as consistent as possible with the user-attribute information [103, 57, 130].

The development of policy miners is non-trivial and generally requires advanced combinatorial and machine-learning algorithms. Recent ABAC miners have also used association rule mining [39] and classification trees [33]. The most effective RBAC miners use deterministic annealing [57] and latent Dirichlet allocation [103].

The proposed miners are so specialized that it is usually unclear how to apply them to other policy languages or even to extensions of the languages for which they were conceived. For example, different extensions of RBAC that allow the specification of spatio-temporal constraints have been proposed over the last two decades, e.g., [113, 34, 90, 125, 31, 6, 41]. However, not a single miner has been proposed for these extensions. Miners have only recently emerged that can mine RBAC policies with constraints, albeit only temporal ones [97, 98, 121]. As a result, if an organization wants to use a specialized policy language, it must invent its own policy miner, which is challenging and time-consuming.

Alternative: A universal method for building policy miners

To facilitate the development of policy miners, we propose a new method, *universal access control policy mining* (Unicorn). With this method, organizations no longer need to spend substantial effort designing specialized policy miners for their unique and specific policy languages; they only need to perform the following tasks (see also Figure 5.1).

Task 1 Define a *template formula* φ for the organization's policy language. We explain later in Section 5.3 what a template formula is.

Task 2 Specify an objective function.

Task 3 Implement the policy miner as indicated by the algorithm template in Section 5.6.

We formalize these tasks in the next sections.

5.3 A universal policy language

Let Γ be a policy language for which we want to design a policy miner. The first step is to specify a template formula φ_Γ for Γ. This is a first-order formula that fulfills some conditions that we explain later in Definition 62. We propose a fragment $\mathcal{L}$ of first-order logic that is powerful enough to contain template formulas for policy languages like RBAC, ABAC, and an expressive fragment of XACML. We show afterwards, using RBAC as an example, how a policy language Γ can be identified with a template formula $\varphi_\Gamma \in \mathcal{L}$ and how policies in Γ can be identified with *interpretation functions* (Definition 17) that interpret φ_Γ's symbols. We thereby reduce the problem of designing a policy miner to designing an algorithm that searches a particular interpretation function.

5.3.1 Motivating example

The approach taken by Unicorn requires a change in the way we model policies and policy languages. So far, policies have been modeled as formulas built with symbols from some signature and we have modeled policy languages as sets of policies. Unicorn requires policies to be modeled as structures and policy languages as template formulas. We illustrate how to model a simplified version of RBAC as a template formula. We only provide here an intuition and leave a formal presentation for Section 5.3.4. We formally define template formulas later in Definition 62.

RBAC policies were defined in Definition 26. Consider the set of RBAC policies with at most N roles. Assume given a signature Σ' with two relation symbols UA and PA of types $\textsf{\textbf{USERS}} \times \textsf{\textbf{ROLES}}$ and $\textsf{\textbf{ROLES}} \times \textsf{\textbf{PERMS}}$, respectively. Observe that for any Σ'-structure $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$, the tuple

$$\pi_{\mathbb{K}} = \left(\textsf{\textbf{USERS}}^{\mathfrak{S}}, \textsf{\textbf{ROLES}}^{\mathfrak{S}}, \textsf{\textbf{PERMS}}^{\mathfrak{S}}, UA^{\mathfrak{S}}, PA^{\mathfrak{S}}\right) \tag{5.1}$$

is an RBAC policy, as defined in Section 2.3.1.

Consider now the following formula:

$$\varphi_N^{RBAC}(u,p) \equiv \bigvee_{i \leq N} \left(UA\left(u, \mathsf{r}_i\right) \ \wedge \ PA\left(\mathsf{r}_i, p\right)\right). \tag{5.2}$$

We show later that φ_N^{RBAC} is a template formula for the set of RBAC policies with at most N roles. For the moment, observe that for any Σ'-structure $\mathbb{K}$, any user u in $\mathbb{K}$, and any p in $\mathbb{K}$, we have the following: $\pi_{\mathbb{K}}$ assigns p to u iff $\mathbb{K}, \sigma_{\mathsf{u},\mathsf{p}} \vDash \varphi_N^{RBAC}(u,p)$, where $\sigma_{\mathsf{u},\mathsf{p}}$ is the substitution mapping u to u and p to p. That is, $\varphi_N^{RBAC}(u,p)$ describes when an RBAC policy assigns a permission to a user; it holds when the permission represented by p is assigned to a role that is assigned to the user represented by u.

We see then that every Σ'-structure defines an RBAC policy and φ_N^{RBAC} describes the semantics for RBAC policies with at most N roles. Therefore, we identify φ_N^{RBAC} with the language of RBAC policies with at most N roles and any Σ'-structure with an RBAC policy.

Unicorn rephrases the policy mining problem as follows: given a template formula for a policy language and an objective function, find a structure $(\mathfrak{S}, \mathfrak{I})$ that minimizes the objective function. In all cases we consider here, $\mathfrak{S}$ defines the carrier sets for the sorts and, as we have seen in the previous chapters, such sets denote sets like the sets of users, permissions, and integers. Such sets are already known by the organization and need not be mined. Therefore, the policy mining problem's goal is to mine an *interpretation function* $\mathfrak{I}$ that minimizes a given objective function.

5.3.2 Language definition

We now define the fragment $\mathcal{L}$ of first-order logic that we use to specify template formulas.

In this chapter, for any signature, we require the organization to specify, for every relation and function symbol, whether it is *rigid* or *flexible*. Rigid symbols are those for which the organization already knows the interpretation function. Flexible symbols are those for which an interpretation function must be found using mining. For example, a function that maps each user to a unique identifier should be modeled with a rigid function symbol, as the organization is not interested in mining new identifiers. In contrast, when mining RBAC policies, for example, one should define a flexible relation symbol to denote the assignment of roles to users, as the organization does not know this assignment and wants to compute one using mining.

Definition 61 The *universal policy language* $\mathcal{L}$ is the set of all quantifier-free first-order formulas having at most two free variables u and p of sorts **USERS** and **PERMS**, respectively. $\square$

In this chapter, we restrict ourselves to *finite* structures. A structure $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ is finite if, for every sort S, $S^{\mathfrak{S}}$ is finite. As a result, for every function or relation symbol W, $W^{\mathfrak{I}}$ is finite. Observe that no structure used in the previous chapters is finite. For example, in any of those structures, $\mathsf{INT}^{\mathfrak{I}}$ is always the set of integers, which is not a finite set. However, notice that organizations never work with the whole set of integers; they work with a finite range like the set of integers between -2^{63} and $2^{63} - 1$. Therefore, we can assume in this chapter that $\mathsf{INT}^{\mathfrak{I}}$ is a finite subset of integers. We make similar assumptions for all other sorts.

Let $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$ be a structure. Recall that $\mathfrak{I}$ is called an *interpretation function* and it maps function and relation symbols to function and relations, respectively. We can see $\mathfrak{I}$ as the union of two interpretation functions $\mathfrak{I}_r$ and $\mathfrak{I}_f$, where $\mathfrak{I}_r$ takes as input rigid symbols and $\mathfrak{I}_f$ takes as input flexible symbols. Recall also that the goal of policy mining for the organization is to find an interpretation function $\mathfrak{I}_f$ for the flexible symbols that minimizes an objective function. It does not need to search for $\mathfrak{S}$ as these function defines the carrier sets for sorts like USERS and PERMS, which the organization already knows. It does not need to search for $\mathfrak{I}_r$ either. Hence, we assume $\mathfrak{S}$ and $\mathfrak{I}_r$ fixed and known to the organization. We also let $U = \mathsf{USERS}^{\mathfrak{S}}$ and $P = \mathsf{PERMS}^{\mathfrak{S}}$. As a consequence, we will not distinguish between S and $S^{\mathfrak{S}}$, for any sort S. We underline rigid symbols and, since $\mathfrak{I}_r$ is fixed, we do not distinguish between $\underline{W}$ and $\underline{W}^{\mathfrak{I}_r}$.

Recall from Definition 24 that the function $\mathfrak{I}$ gives rise to a function that maps each formula $\varphi(u,p) \in \mathcal{L}$ to a relation $\varphi^{\mathfrak{I}} \subseteq \mathsf{USERS}^{\mathfrak{S}} \times \mathsf{PERMS}^{\mathfrak{S}}$.

5.3.3 Template formulas

We can now formalize template formulas. We assume that the semantics of any policy language Γ defines a function $assign_\Gamma$ satisfying the following:

- $assign_\Gamma$ maps every policy π that can be specified with Γ to a relation $assign_\Gamma(\pi) \subseteq U \times P$.
- For $(\mathsf{u}, \mathsf{p}) \in U \times P$, $(\mathsf{u}, \mathsf{p}) \in assign_\Gamma(\pi)$ iff p is assigned to u by π.

For example, in the case of RBAC, $assign_{RBAC}$ maps an RBAC policy (U, P, Ro, Ua, Pa) to the relation $Ua \circ Pa$.

Definition 62 Let Γ be a policy language. A formula $\varphi(u,p) \in \mathcal{L}$ is a *template formula for* Γ if there exists a function $\mathcal{M}$ fulfilling the following:

- $\mathcal{M}$ is a surjective function from the set of all interpretation functions to the set of all policies that can be specified with Γ.
- For any interpretation function $\mathfrak{I}$ and any $(\mathsf{u}, \mathsf{p}) \in U \times P$, $(\mathsf{u}, \mathsf{p}) \in \varphi^{\mathfrak{I}}$ iff $(\mathsf{u}, \mathsf{p}) \in assign_\Gamma(\mathcal{M}(\mathfrak{I}))$.

□

The first task to build a policy miner using Unicorn is to define a template formula for the organization's policy language.

The mapping $\mathcal{M}$ shows that there is a correspondence between interpretations and policies. Although not injective, $\mathcal{M}$ guarantees that each policy is represented by at least one interpretation. Therefore, we can search for an interpretation instead of a policy. For this reason, for the rest of this chapter, we identify every formula in $\mathcal{L}$ with a *policy language* and sometimes refer to interpretation functions as *policies*. Remember that our modeling of policies with interpretation functions in this chapter contrasts with our modeling of policies with *formulas* in Chapters 3 and 4.

5.3.4 An example: RBAC

We now present a template formula $\varphi_N^{RBAC}(u,p) \in \mathcal{L}$ for the language Γ_N of all RBAC policies with at most N roles. Recall that RBAC policies were defined in Section 2.3.1.

Template formula definition: Consider a signature with a sort **ROLES** denoting roles and with two (flexible) binary relation symbols UA and PA of types **USERS** $\times$ **ROLES** and **ROLES** $\times$ **PERMS**, respectively. Define the following formula:

$$\varphi_N^{RBAC}(u,p) \equiv \bigvee_{i \leq N} \left(UA(u,\underline{\mathsf{r}}_i) \ \wedge \ PA(\underline{\mathsf{r}}_i,p)\right). \tag{5.3}$$

Here, $\underline{\mathsf{r}}_i$, for $1 \leq i \leq N$, is a rigid constant symbol of sort **ROLES** (recall that we underline rigid symbols and denote constant symbols with serif letters). One could also use flexible constant symbols for roles, but, as we see later, the policy miner's complexity increases with the number of flexible symbols.

Correctness proof: We now define a mapping $\mathcal{M}$ that proves that φ_N^{RBAC} is a template formula for Γ_N. For any interpretation function $\mathfrak{I}$, let

$$\mathcal{M}\left(\mathfrak{I}\right) = \left(U, P, \{\underline{\mathsf{r}}_1, \ldots, \underline{\mathsf{r}}_N\}, UA^{\mathfrak{I}}, PA^{\mathfrak{I}}\right). \tag{5.4}$$

Observe that $\mathcal{M}\left(\mathfrak{I}\right)$ is an RBAC policy. Moreover, for $(\mathsf{u},\mathsf{p}) \in U \times P$, $(\mathsf{u},\mathsf{p}) \in \left(\varphi_N^{RBAC}\right)^{\mathfrak{I}}$ iff $(\mathsf{u},\mathsf{p}) \in assign_{RBAC}\left(\pi_{\mathfrak{I}}\right)$ (Recall that $assign_{RBAC}\left(\pi_{\mathfrak{I}}\right) = UA^{\mathfrak{I}} \circ PA^{\mathfrak{I}}$). It is also easy to prove that $\mathfrak{I}$ is surjective on the set of all RBAC policies with at most N roles. Hence, we can identify φ_N^{RBAC} with the language of all RBAC policies with at most N roles.

Example 63 To facilitate the understanding of how $\mathcal{M}$ works we show an RBAC policy π and an interpretation function $\mathfrak{I}$ such that $\mathcal{M}\left(\mathfrak{I}\right) = \pi$.

Let $N = 2$ and assume that $U = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Carlos}, \mathsf{David}\}$ and that $P = \{\mathsf{readfile}, \mathsf{writefile}, \mathsf{createfile}\}$. Let r_1 and r_2 denote two roles. Consider the RBAC policy defined by Tables 5.1 and 5.2.

	r_1	r_2
Alice	×	
Bob	×	
Carlos		×
David		×

Table 5.1: User-assignment relation

	read	write	create
r_1	×	×	
r_2			×

Table 5.2: Permission-assignment relation

We can define an interpretation function $\mathfrak{I}$ such that $\mathcal{M}(\mathfrak{I})$ corresponds to the RBAC policy above. $\mathfrak{I}$ interprets the relation symbols UA and PA in the formula $\varphi_N^{RBAC}(u,p)$ as follows. For $\mathsf{u} \in U$ and $i \leq 2$, $UA^{\mathfrak{I}}(\mathsf{u}, \underline{\mathsf{r}}_i)$ iff $(\mathsf{u}, \mathsf{r}_i)$ is marked with an $\times$ in Table 5.1. Similarly, for $\mathsf{p} \in P$ and $i \leq 2$, $PA^{\mathfrak{I}}(\underline{\mathsf{r}}_i, \mathsf{p})$ iff $(\mathsf{r}_i, \mathsf{p})$ is marked with an $\times$ in Table 5.2.

□

5.3.5 Another example: simple ABAC

We now present a template formula $\varphi_N^{ABAC}(u,p) \in \mathcal{L}$ for the language Γ_N of all *simple* ABAC policies with exactly N rules. A simple ABAC policy is an ABAC policy, as in Definition 29, where all atomic formulas are only of the form $f(u) = \mathsf{c}$ or $f(p) = \mathsf{c}$. Here, f is a function symbol and c is a constant symbol. Afterwards, we show how to change $\varphi_N^{ABAC}(u,p)$ into a template formula the language of all simple ABAC policies with *at most* N rules.

For simplicity, we assume that there are no permission attributes (i.e., the object associated to the permission has no attributes) and that the user attributes are $\underline{UAtt}_1$, $\underline{UAtt}_2$, . . ., $\underline{UAtt}_M$. The presentation can easily be adapted to the general case where there are also permission attributes. Observe that we use rigid function symbols to denote the user attributes, as the organization knows how to interpret these symbols and is not interested in mining one.

Consider the following formula $\varphi_N^{ABAC}(u,p)$:

$$\bigvee_{i \leq N} \bigwedge_{j \leq M} \left(\mathsf{b}_{ij} = \underline{1} \rightarrow \underline{UAtt_j}(u) = \mathsf{c}_{ij} \right). \tag{5.5}$$

Here, b_{ij}, for $i \leq N$ and $j \leq M$, is a flexible constant symbol of sort **BOOL** and c_{ij} is a flexible constant symbol of sort **STR**. The rigid constant symbol

$\underline{1}$: **INT** represents the value 1. For an interpretation function $\mathfrak{I}$, if $\mathsf{b}_{ij}^{\mathfrak{I}} = 1$, then the i-th rule requires the value of $\underline{UAtt_j}$ to be equal to c_{ij}.

Let $\mathcal{M}$ be the mapping that maps an interpretation function $\mathfrak{I}$ to the following simple ABAC policy:

$$\mathcal{M}(\mathfrak{I}) \equiv \bigvee_{i \leq N} \bigwedge_{\substack{j \leq M \\ \mathsf{b}_{ij}=1}} \underline{UAtt_j}(u) = \mathsf{c}_{ij}^{\mathfrak{I}}. \tag{5.6}$$

It is straightforward to show that $\mathcal{M}$ fulfills all conditions of Definition 62. Therefore, $\varphi_N^{ABAC}(u,p)$ is a template formula for the language of simple ABAC policies with exactly N rules.

Example 64 To better understand how $\mathcal{M}$ works we show a simple ABAC policy π and an interpretation function $\mathfrak{I}$ such that $\mathcal{M}(\mathfrak{I}) = \pi$.

We assume $M = 2$ and $N = 3$. That is, there are 2 user attributes and we consider only simple ABAC policies with exactly three rules. Consider the following simple ABAC policy:

$$\pi = \begin{pmatrix} \underline{UAtt_1}(u) = \underline{1} \vee \\ \underline{UAtt_2}(u) = \underline{2} \vee \\ \underline{UAtt_1}(u) = \underline{11} \wedge \underline{UAtt_2}(u) = \underline{22} \end{pmatrix}. \tag{5.7}$$

We now present an interpretation function $\mathfrak{I}$ such that $\mathcal{M}(\mathfrak{I}) = \pi$. This function $\mathfrak{I}$ interprets the symbols in Formula 5.5 as follows:

$\mathsf{b}_{11}^{\mathfrak{I}} = 1$	$\mathsf{b}_{12}^{\mathfrak{I}} = 0$	$\mathsf{c}_{11}^{\mathfrak{I}} = 1$	$\mathsf{c}_{12}^{\mathfrak{I}} = 42$
$\mathsf{b}_{21}^{\mathfrak{I}} = 0$	$\mathsf{b}_{22}^{\mathfrak{I}} = 1$	$\mathsf{c}_{21}^{\mathfrak{I}} = 42$	$\mathsf{c}_{22}^{\mathfrak{I}} = 2$
$\mathsf{b}_{31}^{\mathfrak{I}} = 1$	$\mathsf{b}_{32}^{\mathfrak{I}} = 1$	$\mathsf{c}_{31}^{\mathfrak{I}} = 11$	$\mathsf{c}_{32}^{\mathfrak{I}} = 22$

Observe that $\mathsf{c}_{21}^{\mathfrak{I}}$ and $\mathsf{c}_{12}^{\mathfrak{I}}$ can take arbitrary values. Observe also that we do not need to specify the interpretation for any $\underline{UAtt_i}$, for $i \leq N$, as they are rigid symbols and, hence, we assume that the interpretation of those symbols has already been fixed by the organization. □

Finally, we redefine $\varphi_N^{ABAC}(u,p)$ so that it becomes a template formula for the language of simple ABAC policies with *at most* N rules. Let

$$\varphi_N^{ABAC}(u,p) \equiv \bigvee_{i \leq N} \psi_i, \tag{5.8}$$

where $\psi_i \equiv \mathsf{h}_i = \underline{1} \wedge \eta_i$ and $\eta_i \equiv \bigwedge_{j \leq M} \left(\mathsf{b}_{ij} = \underline{1} \to \underline{UAtt_j}(u) = \mathsf{c}_{ij}\right)$. Observe that η_i, for $i \leq N$, is the same formula for modeling rules in Formula 5.5. The only difference between this new definition of $\varphi_N^{ABAC}(u,p)$ and the one in Formula 5.5 is that we introduced the constants $\mathsf{h}_1, \mathsf{h}_2, \ldots, \mathsf{h}_N$. Also, for an interpretation function $\mathfrak{I}$ and $i \leq N$, if $\mathsf{h}_i^{\mathfrak{I}} = 0$, then the value of $\eta_i^{\mathfrak{I}}$ does not affect the value of $\psi^{\mathfrak{I}}$. Hence, the value of $\mathsf{h}_i^{\mathfrak{I}}$ defines if the rule represented by η_i is part of the policy or not. In particular, setting $\mathsf{h}_i^{\mathfrak{I}} = 1$, for all $i \leq N$, yields a policy with N rules and setting $\mathsf{h}_i^{\mathfrak{I}} = 0$, for all $i \leq N$, yields a policy with no rules.

5.3.6 A remark on signatures

Observe that the designer of a policy miner only needs to specify a template formula $\varphi \in \mathcal{L}$ for a policy language. He does not need to specify a signature as this can be deduced from the symbols occurring in φ. Therefore, from now on, whenever we speak in the context of a formula $\varphi \in \mathcal{L}$, we assume that the underlying signature consists only of those symbols occurring in φ.

5.4 A probability distribution on permission assignments and policies

Let $\varphi(u, p) \in \mathcal{L}$ be a policy language. To design a policy miner using Unicorn one must also specify an *objective function* L. This is a function takes two inputs: a permission assignment $Auth \subseteq U \times P$, which is a relation between U and P indicating what permissions each user has, and a policy $\mathfrak{I}$. An objective function outputs a value in $\mathbb{R}^+$ measuring how well $\varphi^{\mathfrak{I}}$ fits $Auth$ and other policy requirements. We remark that the objective function must be defined by the policy miner designer.

For illustration, we define the following objective function:

$$L(Auth, \mathfrak{I}; \varphi) = \sum_{(\mathsf{u},\mathsf{p}) \in U \times P} \left| Auth(\mathsf{u}, \mathsf{p}) - \varphi^{\mathfrak{I}}(\mathsf{u}, \mathsf{p}) \right|. \tag{5.9}$$

Here, we give a value of 1 to the Boolean value `true` and 0 to the Boolean value `false`. Observe that $L(Auth, \mathfrak{I}; \varphi)$ is the size of the symmetric difference of the relations $Auth$ and $\varphi^{\mathfrak{I}}$. Hence, lower values for $L(Auth, \mathfrak{I}; \varphi)$ are better. In Section 5.7, we give other examples of objective functions.

The policy miners built with Unicorn are *probabilistic*. They operate by receiving as input a permission assignment $Auth$ and then computing a probability distribution over the set of all policies in a fixed policy language Γ. We use a Bayesian instead of a frequentist interpretation of the concept of probability. The probability of a policy $\mathfrak{I}$ does not measure how often $\mathfrak{I}$ is the outcome of an experiment, but rather how strong we believe $\mathfrak{I}$ to be the policy that decided the requests in $Auth$.

We now define a probability distribution $\mathbb{P}$ on permission assignments and policies. We first give an intuition on $\mathbb{P}$'s definition and then formally define $\mathbb{P}$. For a permission assignment *Auth* and a policy $\mathfrak{I}$, we can see $\mathbb{P}\left(Auth, \mathfrak{I}\right)$ as a quantity telling us how much we believe $\mathfrak{I}$ to be the organization's policy and how much we believe *Auth* to be the organization's permission assignment. $\mathbb{P}$'s definition must satisfy the chain rule of probability theory:

$$\mathbb{P}\left(Auth, \mathfrak{I}\right) = \mathbb{P}\left(\mathfrak{I} \mid Auth\right) \mathbb{P}\left(Auth\right). \tag{5.10}$$

The value $\mathbb{P}\left(\mathfrak{I} \mid Auth\right)$ indicates how much we believe $\mathfrak{I}$ is the organization's policy after we see that *Auth* is the organization's permission assignment. Policy miners receive as input a permission assignment *Auth* and then compute a policy $\mathfrak{I}^*$ that maximizes $\mathbb{P}\left(\mathfrak{I} \mid Auth\right)$. We later see that how $\mathbb{P}\left(Auth\right)$ is defined is irrelevant. So we focus only on defining $\mathbb{P}\left(\mathfrak{I} \mid Auth\right)$.

The conditional probability $\mathbb{P}\left(\mathfrak{I} \mid Auth\right)$ is defined as the "most general" distribution that fulfills the following requirement: *for any policy $\mathfrak{I}$, the lower $L\left(Auth, \mathfrak{I}; \varphi\right)$ is, the more likely $\mathfrak{I}$ is.* Following the principle of maximum entropy [85], the most general distribution that achieves this is the following:

$$\mathbb{P}(\mathfrak{I} \mid Auth) = \frac{\exp\left(-\beta L(Auth, \mathfrak{I}; \varphi)\right)}{\sum_{\mathfrak{I}'} \exp\left(-\beta L(Auth, \mathfrak{I}'; \varphi)\right)}, \tag{5.11}$$

where $\mathfrak{I}'$ ranges over all policies. Recall that we consider only finite structures. Hence, all our carrier sets are finite, so there are only finitely many policies.

The value $\beta > 0$ is a parameter that the policy miner changes during the search for the most likely policy. The search uses deterministic annealing, an optimization procedure inspired by simulated annealing [**?**, 114, **?**]. In our case, it initially sets β to a very low value, so that all policies are equally likely. Then it gradually decreases β while, at the same time, searching for the most likely policy. As β decreases, those policies that minimize $L(Auth, \mathfrak{I}; \varphi)$ become more likely. In this way, deterministic annealing has experimentally been shown to escape low-quality local maxima. When $\beta \to \infty$, only those policies that minimize $L(Auth, \mathfrak{I}; \varphi)$ have a positive probability and the search converges to a local maximum.

We now formally define the probability distribution given in Equation 5.11.

Definition 65 For a formula $\varphi \in \mathcal{L}$, we define the probability space $\mathfrak{P}_\varphi = \left(\Omega, 2^\Omega, \mathbb{P}\right)$ as follows.

- Ω contains all pairs $(Auth, \mathfrak{I})$, where $Auth \subseteq U \times P$ and $\mathfrak{I}$ is an interpretation function.
- 2^Ω is the set of all subsets of Ω. We assume that all carrier sets for all sorts are finite. Therefore, Ω and 2^Ω are finite.

- For $(Auth, \mathfrak{I}) \in \Omega$, let $\mathbb{P}\left(Auth\right)$ be any real-valued function on permission assignments such that $\mathbb{P}\left(Auth\right) \geq 0$ and $\sum_{Auth'} \mathbb{P}\left(Auth'\right) = 1$. Let also

$$\mathbb{P}(\mathfrak{I} \mid Auth) = \frac{\exp\left(-\beta L(Auth, \mathfrak{I}; \varphi)\right)}{\sum_{\mathfrak{I}'} \exp\left(-\beta L(Auth', \mathfrak{I}'; \varphi)\right)}. \tag{5.12}$$

$$\mathbb{P}\left(Auth, \mathfrak{I}\right) = \mathbb{P}(\mathfrak{I} \mid Auth)\mathbb{P}\left(Auth\right). \tag{5.13}$$

Finally, for $O \in 2^{\Omega}$, let $\mathbb{P}(O) = \sum_{(Auth,\mathfrak{I}) \in O} \mathbb{P}\left(Auth, \mathfrak{I}\right)$.

□

We also define the shorthand $\mathbb{P}\left(\mathfrak{I}\right) = \mathbb{P}\left(\{\mathfrak{I}\}\right) = \sum_{Auth} \mathbb{P}\left(Auth, \mathfrak{I}\right)$.

The theorem below proves that $\mathbb{P}\left(\mathfrak{I} \mid Auth\right)$ is the "most general" distribution that fulfills the requirement mentioned above. More precisely, $\mathbb{P}\left(\mathfrak{I} \mid Auth\right)$ is the maximum-entropy probability distribution with a bounded expected value for $L(Auth, \mathfrak{I}; \varphi)$ and where the probability of $\mathfrak{I}$ increases whenever $L(Auth, \mathfrak{I}; \varphi)$ decreases [74, 124].

Theorem 66 $\mathbb{P}\left(\mathfrak{I} \mid Auth\right)$ is the probability distribution P on policies that maximizes P's entropy and is subject to the following constraints.

- $\sum_{\mathfrak{I}} P\left(\mathfrak{I}\right) L(Auth, \mathfrak{I}; \varphi) \leq \ell$, for some fixed bound ℓ and a fixed $Auth$.
- If $\beta > 0$, then $P\left(\mathfrak{I}\right) > P\left(\mathfrak{I}'\right)$, for any two policies $\mathfrak{I}$ and $\mathfrak{I}'$ with $L(Auth, \mathfrak{I}; \varphi) < L(Auth, \mathfrak{I}'; \varphi)$.

Proof. It suffices to drop the second constraint and then use Lagrange multipliers to find an optimal distribution P^*. One can verify that $P^*\left(\mathfrak{I}\right) = \mathbb{P}\left(\mathfrak{I} \mid Auth\right)$ and that $P^*\left(\mathfrak{I}\right)$ satisfies the second constraint. □

Example 67 We illustrate the probability distribution defined above for the language of all RBAC policies with at most N roles, defined in Section 5.3.4. For simplicity, we fix $N = 2$ and $\beta = 1$ in this example. Assume that $U = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Carlos}, \mathsf{David}\}$ and that $P = \{\mathsf{read}, \mathsf{write}, \mathsf{create}\}$. Assume given two permission assignments $Auth_1$ and $Auth_2$ and two policies $\mathfrak{I}_1$ and $\mathfrak{I}_2$ as shown in Tables 5.9–5.8. Table 5.12 computes the conditional probabilities $\mathbb{P}\left(\mathfrak{I}_j, Auth_i\right)$ with $i, j \in \{1, 2\}$. Observe the following relations:

- $L\left(Auth_1, \mathfrak{I}_1; \varphi_N^{RBAC}\right) < L\left(Auth_1, \mathfrak{I}_2; \varphi_N^{RBAC}\right)$. This is because the permission assignment induced by the policy $\mathfrak{I}_1$ resembles more $Auth_1$ than the permission assigned induced by $\mathfrak{I}_2$.
- $L\left(Auth_2, \mathfrak{I}_2; \varphi_N^{RBAC}\right) < L\left(Auth_2, \mathfrak{I}_1; \varphi_N^{RBAC}\right)$. The reason for this is analogous to the previous one.
- $\mathbb{P}\left(\mathfrak{I}_1 \mid Auth_1\right) > \mathbb{P}\left(\mathfrak{I}_2 \mid Auth_1\right)$. This is because $L\left(Auth_1, \mathfrak{I}_1; \varphi_N^{RBAC}\right) < L\left(Auth_1, \mathfrak{I}_2; \varphi_N^{RBAC}\right)$ and the second property stated in Theorem 66.

	$\underline{r}_1^{\mathfrak{I}_1}$	$\underline{r}_2^{\mathfrak{I}_1}$
Alice	×	
Bob	×	
Carlos		×
David		×

Table 5.3: $UA^{\mathfrak{I}_1}$

	$\underline{r}_1^{\mathfrak{I}_2}$	$\underline{r}_2^{\mathfrak{I}_2}$
Alice	×	
Bob	×	
Carlos		×
David		×

Table 5.4: $UA^{\mathfrak{I}_2}$

	read	write	create
$\underline{r}_1^{\mathfrak{I}_1}$	×	×	
$\underline{r}_2^{\mathfrak{I}_1}$			×

Table 5.5: $PA^{\mathfrak{I}_1}$

	read	write	create
$\underline{r}_1^{\mathfrak{I}_2}$		×	
$\underline{r}_2^{\mathfrak{I}_2}$	×		×

Table 5.6: $PA^{\mathfrak{I}_2}$

	read	write	create
Alice	×	×	
Bob	×	×	
Carlos			×
David			×

Table 5.7: $\left(\varphi_N^{RBAC}\right)^{\mathfrak{I}_1}$

	read	write	create
Alice		×	
Bob		×	
Carlos	×		×
David	×		×

Table 5.8: $\left(\varphi_N^{RBAC}\right)^{\mathfrak{I}_2}$

	read	write	create
Alice	×	×	
Bob	×	×	
Carlos			×
David	×		×

Table 5.9: $Auth_1$

	read	write	create
Alice		×	
Bob		×	
Carlos	×		
David	×		

Table 5.10: $Auth_2$

$$\begin{array}{ll} L\left(Auth_1, \mathfrak{I}_1; \varphi_N^{RBAC}\right) = 1 & L\left(Auth_2, \mathfrak{I}_2; \varphi_N^{RBAC}\right) = 2 \\ L\left(Auth_1, \mathfrak{I}_2; \varphi_N^{RBAC}\right) = 3 & L\left(Auth_2, \mathfrak{I}_1; \varphi_N^{RBAC}\right) = 4 \end{array}$$

Table 5.11

$$\begin{array}{ll} \mathbb{P}\left(\mathfrak{I}_1 \mid Auth_1\right) = \exp\left(-1\right)/Z_1 & \mathbb{P}\left(\mathfrak{I}_2 \mid Auth_2\right) = \exp\left(-2\right)/Z_2 \\ \mathbb{P}\left(\mathfrak{I}_2 \mid Auth_1\right) = \exp\left(-3\right)/Z_1 & \mathbb{P}\left(\mathfrak{I}_1 \mid Auth_2\right) = \exp\left(-4\right)/Z_2 \end{array}$$

Table 5.12: Probabilities for the policies and permission assignments depicted in Tables 5.9–5.8. Here, $Z_i = \sum_{\mathfrak{I}} \exp\left(-L\left(Auth_i, \mathfrak{I}; \varphi_N^{RBAC}\right)\right)$. Recall that we assume $\beta = 1$.

- $\mathbb{P}(\mathfrak{I}_2 \mid \mathit{Auth}_2) > \mathbb{P}(\mathfrak{I}_1 \mid \mathit{Auth}_2)$.

$\square$

5.5 Approximating distributions with mean-field variational inference

The policy miner that is built with Unicorn is an algorithm that receives as input a permission assignment *Auth* and computes a policy $\mathfrak{I}$ that approximately maximizes $\mathbb{P}(\mathfrak{I} \mid \mathit{Auth})$; that is, the most likely policy, conditioned on the given permission assignment. However, computing this maximizer is intractable. Hence, we use *mean-field variational inference* [23], a technique that defines an iterative procedure to approximate $\mathbb{P}(\cdot \mid \mathit{Auth})$ with a distribution $q(\cdot)$. It turns out that maximizing $q(\cdot)$ is much easier than maximizing $\mathbb{P}(\cdot \mid \mathit{Auth})$. The policy miner is then an algorithm implementing the computation of q and its maximization.

This section has two parts. First, we introduce some random variables that help to measure the probability that a policy authorizes a particular request $(\mathsf{u}, \mathsf{p}) \in U \times P$. Afterwards, we present the approximating distribution q.

5.5.1 Random variables

Recall that the sample space Ω of the distribution $\mathbb{P}$ from Definition 65 is the set of all pairs $(\mathit{Auth}, \mathfrak{I})$ with *Auth* a permission assignment and $\mathfrak{I}$ a policy.

Definition 68 Let $\mathfrak{X}$ be a random variable mapping $(\mathit{Auth}, \mathfrak{I}) \in \Omega$ to $\mathfrak{I}$. $\square$

Following standard probability theory, we can understand $\mathfrak{X}$ as an "unknown policy" and, for a policy $\mathfrak{I}$, the probability statement $\mathbb{P}(\mathfrak{X} = \mathfrak{I})$ measures how much we believe that $\mathfrak{X}$ is actually $\mathfrak{I}$. Recall that, by definition,

$$\begin{aligned}\mathbb{P}(\mathfrak{X} = \mathfrak{I}) &= \mathbb{P}\left(\left\{(\mathit{Auth}', \mathfrak{I}') \in \Omega \mid \mathfrak{X}(\mathit{Auth}', \mathfrak{I}') = \mathfrak{I}\right\}\right) \qquad (5.14)\\ &= \sum_{\mathit{Auth}} \mathbb{P}(\mathit{Auth}, \mathfrak{I}) = \mathbb{P}(\mathfrak{I}).\end{aligned}$$

Definition 69 Let $\varphi \in \mathcal{L}$, and let W be a flexible relation symbol occurring in φ of type $\mathbf{S}_1 \times \ldots \times \mathbf{S}_k$ and let f be a flexible function symbol occurring in φ of type $\mathbf{S}_1 \times \ldots \times \mathbf{S}_k \to \mathbf{S}$. Let $(\mathsf{a}_1, \ldots, \mathsf{a}_k) \in \mathbf{S}_1^{\mathfrak{S}} \times \ldots \times \mathbf{S}_k^{\mathfrak{S}}$. Recall that $\mathfrak{S}$ maps sorts to carrier sets. We define the random variable $W^{\mathfrak{X}}(\mathsf{a}_1, \ldots, \mathsf{a}_k) : \Omega \to \{0, 1\}$ that maps $(\mathit{Auth}, \mathfrak{I}) \in \Omega$ to $W^{\mathfrak{I}}(\mathsf{a}_1, \ldots, \mathsf{a}_k) \in \{0, 1\}$. Similarly, we define the random variable $f^{\mathfrak{X}}(\mathsf{a}_1, \ldots, \mathsf{a}_k) : \Omega \to \mathbf{S}^{\mathfrak{S}}$ that maps $(\mathit{Auth}, \mathfrak{I}) \in \Omega$ to $f^{\mathfrak{I}}(\mathsf{a}_1, \ldots, \mathsf{a}_k) \in \mathbf{S}^{\mathfrak{S}}$. We call these random variables *random facts of* φ. $\square$

Example 70 Let us examine some random facts of the formula φ_N^{RBAC} from Example 67. One such random fact is $UA^{\mathfrak{X}}(\mathsf{Alice}, \underline{\mathsf{r}}_1)$, which can take the values 0 and 1, so $UA^{\mathfrak{X}}(\mathsf{Alice}, \underline{\mathsf{r}}_1)$ is a Bernoulli random variable. Its probability distribution is defined in the standard way by the following:

$$\begin{aligned}&\mathbb{P}\left(UA^{\mathfrak{X}}(\mathsf{Alice}, \underline{\mathsf{r}}_1) = 1\right) \\ &\quad = \mathbb{P}\left(\left\{(Auth, \mathfrak{I}) \in \Omega \mid UA^{\mathfrak{I}}(\mathsf{Alice}, \underline{\mathsf{r}}_1) = 1\right\}\right).\end{aligned} \tag{5.15}$$

More generally, the set of all random facts for the formula φ_N^{RBAC} is the following:

$$\begin{aligned}\mathfrak{F}\left(\varphi_N^{RBAC}\right) = &\left\{UA^{\mathfrak{X}}(\mathsf{u}, \underline{\mathsf{r}}_i) \mid \mathsf{u} \in U, i \leq N\right\} \cup \\ &\left\{PA^{\mathfrak{X}}(\underline{\mathsf{r}}_i, \mathsf{p}) \mid \mathsf{p} \in P, i \leq N\right\}.\end{aligned} \tag{5.16}$$

If we set $N = 2$ and replace each random fact with a Boolean value, as indicated by Tables 5.3 and 5.4, then we obtain an RBAC policy. □

Just like a statement of the form $\mathbb{P}(\mathfrak{X} = \mathfrak{I})$ quantifies how much we believe that $\mathfrak{X} = \mathfrak{I}$, a statement of the form $\mathbb{P}\left(UA^{\mathfrak{X}}(\mathsf{Alice}, \underline{\mathsf{r}}_1) = 1\right)$ quantifies how much we believe that role $\underline{\mathsf{r}}_1$ is assigned to Alice.

Example 71 The set of all random facts for the formula φ_N^{ABAC} presented in Section 5.3.5 is the following:

$$\mathfrak{F}\left(\varphi_N^{ABAC}\right) = \left\{\mathsf{b}_{ij}^{\mathfrak{X}} \mid i \leq N, j \leq M\right\} \cup \left\{\mathsf{c}_{ij}^{\mathfrak{X}} \mid i \leq N, j \leq M\right\}. \tag{5.17}$$

If we set $N = 3$ and replace each random fact $\mathfrak{f}$ with a value in $Range(\mathfrak{f})$, as indicated by the interpretation function in Section 5.3.5, then we obtain a simple ABAC policy. □

Observation 4 Since we assume carrier sets to be finite, a random fact always has a discrete distribution. In particular, random facts built from flexible relation symbols have Bernoulli distributions as they can only take Boolean values. □

We usually denote random facts with Fraktur letters $\mathfrak{f}, \mathfrak{g}, \ldots$ For a random fact $\mathfrak{f}$ of the form $W^{\mathfrak{X}}(\mathsf{a}_1, \ldots, \mathsf{a}_k)$, we denote by $\mathfrak{f}^{\mathfrak{I}}$ the Boolean value $W^{\mathfrak{I}}(\mathsf{a}_1, \ldots, \mathsf{a}_k)$. Similarly, when $\mathfrak{f}$ is of the form $f^{\mathfrak{X}}(\mathsf{a}_1, \ldots, \mathsf{a}_k)$, we denote by $\mathfrak{f}^{\mathfrak{I}}$ the value $f^{\mathfrak{I}}(\mathsf{a}_1, \ldots, \mathsf{a}_k)$. Finally, we denote with $Range(\mathfrak{f})$ the range of a random fact $\mathfrak{f}$.

For a policy language $\varphi \in \mathcal{L}$, we denote by $\mathfrak{F}(\varphi)$ the set of all random facts of φ. Recall that we assume all our carrier sets to be finite, so $\mathfrak{F}(\varphi)$ is finite.

Observe that, for any formula $\varphi \in \mathcal{L}$, replacing each random fact $\mathfrak{f}$ in $\mathfrak{F}(\varphi)$ with a value in $Range(f)$ yields a policy. Indeed, we prove below in Lemma 72 that fixing the values of all random facts determine a policy and, conversely, each policy determines the values of all random facts.

The observation above plays an important role in Unicorn. The policy miner for φ built with Unicorn, when given a permission assignment as input, works by computing the set $\mathfrak{F}(\varphi)$ and then computing values that are approximately most likely for all random facts. Such values determine a policy. The lemma below shows that we can cast the problem of maximizing $\mathbb{P}(\mathfrak{I} \mid Auth)$ in a way that we will later see to be suitable for the application of mean-field approximation.

Lemma 72 For a policy language $\varphi \in \mathcal{L}$,

$$\begin{aligned}\mathbb{P}(\mathfrak{I} \mid Auth) &= \mathbb{P}(\mathfrak{X} = \mathfrak{I} \mid Auth) \\ &= \mathbb{P}\left(\left(\mathfrak{f}^{\mathfrak{X}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)} = \left(\mathfrak{f}^{\mathfrak{I}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)} \middle| Auth\right).\end{aligned} \tag{5.18}$$

Proof. The first equality follows from $\mathfrak{X}$'s definition, so we prove the second equality only. Let $\mathcal{F}$ be the random vector $\left(\mathfrak{f}^{\mathfrak{X}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}$. That is, $\mathcal{F}$ is a random variable that maps $(Auth', \mathfrak{I}') \in \Omega$ to $\left(\mathfrak{f}^{\mathfrak{I}'}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}$. It suffices to show that the events $\mathfrak{X} = \mathfrak{I}$ and $\mathcal{F} = \left(\mathfrak{f}^{\mathfrak{I}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}$ are the same. To achieve this, note that these events can be written, respectively, as follows:

$$\left\{(Auth', \mathfrak{I}') \in \Omega \mid Auth' = Auth,\ \mathfrak{X}(Auth', \mathfrak{I}') = \mathfrak{I}\right\}. \tag{5.19}$$

$$\left\{(Auth', \mathfrak{I}') \in \Omega \mid Auth' = Auth,\ \mathcal{F}(Auth', \mathfrak{I}') = \left(\mathfrak{f}^{\mathfrak{I}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}\right\}. \tag{5.20}$$

It suffices to show then that, for $(Auth', \mathfrak{I}') \in \Omega$, $\mathfrak{X}(Auth', \mathfrak{I}') = \mathfrak{I}$ iff $\mathcal{F}(Auth', \mathfrak{I}') = \left(\mathfrak{f}^{\mathfrak{I}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}$. To achieve this, observe that $\mathfrak{I}$ completely determines $\left(\mathfrak{f}^{\mathfrak{I}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}$. Moreover, the values $\left(\mathfrak{f}^{\mathfrak{I}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}$ also turn out to completely determine $\mathfrak{I}$. We conclude then that the two sets and, therefore, the two events are the same. □

We denote by $h_{\mathcal{F}}(\cdot)$ the probability density function $\mathbb{P}\left(\left(\mathfrak{f}^{\mathfrak{X}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)} = \cdot \middle| Auth\right)$. To avoid awkward notation, we write $h_{\mathcal{F}}(\mathfrak{I})$ instead of $h_{\mathcal{F}}\left(\left(\mathfrak{f}^{\mathfrak{I}}\right)_{\mathfrak{f}\in\mathfrak{F}(\varphi)}\right)$.

We conclude this section by defining some other useful random variables.

Definition 73 For $(\mathsf{u}, \mathsf{p}) \in U \times P$, $\varphi \in \mathcal{L}$, and the random variable $\mathfrak{X}$ from Definition 68, we define the random variable $\varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p}) : \Omega \to \{0, 1\}$ as the function mapping $(Auth, \mathfrak{I})$ to $\varphi^{\mathfrak{I}}(\mathsf{u}, \mathsf{p})$. □

Definition 74 For $\varphi \in \mathcal{L}$, $Auth \subseteq U \times P$, and the random variable $\mathfrak{X}$ from Definition 68, we define the random variable $L\left(Auth, \mathfrak{X}; \varphi\right)$ as

$$\sum_{(\mathsf{u},\mathsf{p}) \in U \times P} \left| Auth(\mathsf{u},\mathsf{p}) - \varphi^{\mathfrak{X}}(\mathsf{u},\mathsf{p}) \right| . \tag{5.21}$$

□

5.5.2 Approximating the conditional distribution

Mean-field variational inference approximates the probability distribution $h_{\mathcal{F}}$ with a distribution q defined as follows:

$$q\left(\mathfrak{I}\right) := \prod_{\mathfrak{f} \in \mathfrak{F}(\varphi)} q_{\mathfrak{f}}\left(\mathfrak{f}^{\mathfrak{I}} \mid \theta_{\mathfrak{f}}\right), \tag{5.22}$$

where $q_{\mathfrak{f}}\left(\cdot \mid \theta_{\mathfrak{f}}\right)$ is a probability density function for $\mathfrak{f}$, which can be represented with a function $\theta_{\mathfrak{f}} : Range\left(\mathfrak{f}\right) \to [0,1]$ such that $\sum_{\mathsf{b} \in Range(\mathfrak{f})} \theta_{\mathfrak{f}}\left(\mathsf{b}\right) = 1$. For $\mathsf{b} \in Range\left(\mathfrak{f}\right)$, the value $\theta_{\mathfrak{f}}\left(\mathsf{b}\right)$ denotes the probability, according to $q_{\mathfrak{f}}\left(\cdot \mid \theta_{\mathfrak{f}}\right)$, that $\mathfrak{f} = \mathsf{b}$.

Observe that $q\left(\mathfrak{I}\right)$'s factorization implies that the set of random facts is mutually independent. This is not true in general, as $h_{\mathcal{F}}$ may not be necessarily factorized like q. This independence assumption is imposed by mean-field theory to facilitate computations. Our experimental results in Section 5.10 show that, despite this approximation, we still mine high quality policies.

In Appendix [?], we show that the set of parameters $\left\{\widehat{\theta_{\mathfrak{f}}} \mid \mathfrak{f} \in \mathfrak{F}\left(\varphi\right)\right\}$ that makes q best approximate $h_{\mathcal{F}}$ is defined by the following equation:

$$\widehat{\theta_{\mathfrak{f}}}\left(\mathsf{b}\right) = \frac{\exp\left(-\beta \mathbb{E}_{\mathfrak{f} \mapsto \mathsf{b}}[L\left(Auth, \mathfrak{X}; \varphi\right)]\right)}{\sum_{\mathsf{b}' \in Range(\mathfrak{f})} \exp\left(-\beta \mathbb{E}_{\mathfrak{f} \mapsto \mathsf{b}'}[L\left(Auth, \mathfrak{X}; \varphi\right)]\right)}, \tag{5.23}$$

where $\mathsf{b} \in Range\left(\mathfrak{f}\right)$ and $\mathbb{E}_{\mathfrak{f} \mapsto \mathsf{b}}[L\left(Auth, \mathfrak{X}; \varphi\right)]$ is the expectation of $L\left(Auth, \mathfrak{X}; \varphi\right)$ after replacing every occurrence of the random fact $\mathfrak{f}$ with b. This expectation is computed using the distribution q. Therefore,

$$\begin{aligned}
&\mathbb{E}_{\mathfrak{f} \mapsto \mathsf{b}}[L\left(Auth, \mathfrak{X}; \varphi\right)] \\
&\quad = \sum_{\mathfrak{I}} q\left(\mathfrak{I}\right)\left(L\left(Auth, \mathfrak{I}; \varphi\right)\{\mathfrak{f} \mapsto \mathsf{b}\}\right) \\
&\quad = \sum_{\mathfrak{I}} \prod_{\substack{\mathfrak{g} \in \mathfrak{F}(\varphi) \\ \mathfrak{g} \neq \mathfrak{f}}} q_{\mathfrak{g}}\left(\mathfrak{g}^{\mathfrak{I}} \mid \widehat{\theta_{\mathfrak{g}}}\right)\left(L\left(Auth, \mathfrak{I}; \varphi\right)\{\mathfrak{f} \mapsto \mathsf{b}\}\right).
\end{aligned} \tag{5.24}$$

Here, $L\left(Auth, \mathfrak{I}; \varphi\right)\{\mathfrak{f} \mapsto \mathsf{b}\}$ is obtained from $L\left(Auth, \mathfrak{I}; \varphi\right)$ by replacing $\mathfrak{f}$ with b. For a formal derivation of Equation 5.23, we refer to [23].

Using Lemma 72 and the distribution q, we can approximate $\arg\max_{\mathfrak{I}} \mathbb{P}\left(\mathfrak{I} \mid Auth\right)$ by maximizing q.

Observation 5 Let $\mathfrak{I}^* = \arg\max_{\mathfrak{I}} \mathbb{P}(\mathfrak{I} \mid Auth)$. Then,

$$\mathbb{P}(\mathfrak{I}^* \mid Auth) = \mathbb{P}(\mathfrak{X} = \mathfrak{I}^* \mid Auth) = h_{\mathcal{F}}(\mathfrak{I}^*) \approx q(\mathfrak{I}^*) . \tag{5.25}$$

The desired miner is then an algorithm that computes and maximizes q.

5.6 Building the policy miner

To compute q, the desired policy miner uses Equation 5.23 to compute $\widehat{\theta}_{\mathfrak{f}}$, for each $\mathfrak{f} \in \mathfrak{F}(\varphi)$. Observe, however, that this is a recursive equation as the computation of the expectations on the right hand side requires $\left\{\widehat{\theta}_{\mathfrak{f}} \mid \mathfrak{f} \in \mathfrak{F}(\varphi)\right\}$. This recursive dependency is handled by iteratively computing, for each $\mathfrak{f} \in \mathfrak{F}(\varphi)$, a function $\tilde{\theta}_{\mathfrak{f}}$ that approximates $\widehat{\theta}_{\mathfrak{f}}$ [23].

Algorithm 5 gives the pseudocode for this approximation, which is the essence of the desired policy miner. We give next an overview.

1 **Initialization (lines 2–4).** Each parameter $\tilde{\theta}_{\mathfrak{f}}$ is randomly set to an arbitrary function, respecting the constraint that $\sum_{\mathsf{b}} \tilde{\theta}_{\mathfrak{f}}(\mathsf{b}) = 1$.

2 **Update loop (lines 5–12).** We perform a sequence of iterations that update $\{\tilde{\theta}_{\mathfrak{f}} \mid \mathfrak{f} \in \mathfrak{F}(\varphi)\}$ and β, the hyper-parameter controlling $\mathbb{P}$'s entropy. The number of iterations is fixed before execution.

   a) **Parameter update (line 6–10).** At each iteration, we compute a random ordering $RS(\mathfrak{F}(\varphi))$ of all random facts. Then, for each $\mathfrak{f}$ in that order, $\tilde{\theta}_{\mathfrak{f}}$ is updated to the right-hand side of Equation 5.23 (lines 7–8), but using $\{q_{\mathfrak{f}}(\cdot \mid \tilde{\theta}_{\mathfrak{f}}) \mid \mathfrak{f} \in \mathfrak{F}(\varphi)\}$ instead of $\left\{q_{\mathfrak{f}}\left(\cdot \mid \widehat{\theta}_{\mathfrak{f}}\right) \mid \mathfrak{f} \in \mathfrak{F}(\varphi)\right\}$ to compute the expectations.

   b) **Hyper-parameter update (line 11).** After each iteration, we increase β by a factor of α, defined before execution. This avoids that a $\tilde{\theta}_{\mathfrak{f}}$ is trapped in a local minimum in the early iterations and facilitates its convergence in later iterations [114, 67].

3 **Policy computation (line 13).** Finally, we compute $\mathfrak{I}^* = \arg\max_{\mathfrak{I}} q(\mathfrak{I})$. By looking at Equation 5.22, we see that to maximize $q(\cdot)$, it suffices to maximize $q_{\mathfrak{f}}(\cdot \mid \theta_{\mathfrak{f}})$, for every $\mathfrak{f} \in \mathfrak{F}(\varphi)$. Hence, we let $\mathfrak{I}^*$ be the policy that satisfies $\mathfrak{f}^{\mathfrak{I}^*} = \arg\max_{\mathsf{b} \in Range(\mathfrak{f})} q_{\mathfrak{f}}(\mathsf{b} \mid \theta_{\mathfrak{f}})$.

Observe that the policy miner requires values for the hyper-parameters α, β, and T as input. Adequate values can be computed using machine-learning methods like grid search [118], which we briefly recall in Appendix C.1.

Algorithm 5

```
1:  function PolicyMiner($L$, $Auth$, $\varphi$, $\alpha$, $\beta$, $T$)
2:      for $\mathfrak{f} \in \mathfrak{F}(\varphi)$ do
3:          Initialize $\tilde{\theta}_{\mathfrak{f}} : Range(\mathfrak{f}) \to \{0,1\}$ with an arbitrary distribution.
4:      end for
5:      for $i = 1 \ldots T$ do
6:          for $\mathfrak{f} \in RS(\mathfrak{F}(\varphi))$ do
7:              for $\mathsf{b} \in Range(\mathfrak{f})$ do
8:                  $\tilde{\theta}_{\mathfrak{f}}(\mathsf{b}) \leftarrow \frac{\exp(-\beta \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[L(Auth, \mathfrak{X}; \varphi)])}{\sum_{\mathsf{b}' \in Range(\mathfrak{f})} \exp(-\beta \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[L(Auth, \mathfrak{X}; \varphi)])}$.
9:              end for
10:         end for
11:         $\beta \leftarrow \alpha \times \beta$.
12:     end for
13:     Define $\mathfrak{I}^*$ by letting $\mathfrak{f}^{\mathfrak{I}^*} = \arg\max_{\mathsf{b} \in Range(\mathfrak{f})} q_{\mathfrak{f}}(\mathsf{b} \mid \theta_{\mathfrak{f}})$, for $\mathfrak{f} \in \mathfrak{F}(\varphi)$.
14:     return $\mathfrak{I}^*$
15: end function
```

5.6.1 Simplifying the computation of expectations

One need not be knowledgeable about variational inference to implement Algorithm 5 in a standard programming language. The only part requiring knowledge in probability thereby is the computation of the expectation in line 8. We now define the notion of *diverse* random variables and show that expectations of some diverse random variables can easily be computed recursively using some basic equalities.

Definition 75 A random variable X is *diverse* if (i) it can be constructed from constant values and random facts using only arithmetic and Boolean operations and (ii) any random fact is used in the construction at most once. $\square$

Example 76 Let $(\mathsf{u}, \mathsf{p}) \in U \times P$ and let V, W, and Y be flexible relation symbols. Then the random variable $V^{\mathfrak{X}}(\mathsf{u}, \mathsf{p}) + W^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})$ is diverse, but the random variable $V^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\, W^{\mathfrak{X}}(\mathsf{u}, \mathsf{p}) + W^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\, Y^{\mathfrak{X}}(\mathsf{u}, \mathsf{p}) + V^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\, Y^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})$ is not, since each random fact there occurs more than once. $\square$

Corollary 77 Let $\varphi \in \mathcal{L}$ and $(\mathsf{u}, \mathsf{p}) \in U \times P$, then $\varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})$ is diverse iff every atomic formula that occurs in φ occurs exactly once.

This corollary is a direct consequence of Definition 75. Observe that, for $\varphi \in \mathcal{L}$, one can check in time linear in φ's length that every atomic formula occurring in φ occurs exactly once.

Example 78 Recall the formulas φ_N^{RBAC} and φ_N^{ABAC} defined in Sections 5.3.4 and 5.3.5. Observe that each atomic formula in any of these formulas occurs exactly once. Hence, for $(\mathsf{u},\mathsf{p}) \in U \times P$, the random variables $\left(\varphi_N^{RBAC}\right)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$ and $\left(\varphi_N^{ABAC}\right)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$ are diverse. □

The following lemma, proved in Appendix C.3, shows how to recursively compute $\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[L\left(Auth, \mathfrak{X}; \varphi\right)]$ when $\varphi^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$ is diverse.

Lemma 79 Let $\mathfrak{f}$ and $\mathfrak{g}$ be facts, φ be a formula in $\mathcal{L}$, $(\mathsf{u},\mathsf{p}) \in U \times P$, and $\{\psi_i\}_i \subseteq \mathcal{L}$. Assume that $\varphi^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$, $(\neg\varphi)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$, and $\left(\bigwedge_i \psi_i\right)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$ are diverse. Then the following equalities hold.

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\mathfrak{g}] = \begin{cases} \mathsf{b} & \text{if } \mathfrak{f} = \mathfrak{g} \text{ and} \\ \sum_{\mathsf{b}\in Range(\mathfrak{g})} \theta_{\mathfrak{g}}\left(\mathsf{b}\right)\mathsf{b} & \text{otherwise.} \end{cases} \tag{5.26}$$

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[(\neg\varphi)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})\right] = 1 - \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\varphi^{\mathfrak{X}}(\mathsf{u},\mathsf{p})\right]. \tag{5.27}$$

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\left(\bigwedge_i \psi_i\right)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})\right] = \prod_i \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\psi_i^{\mathfrak{X}}(\mathsf{u},\mathsf{p})\right]. \tag{5.28}$$

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[L\left(Auth, \mathfrak{X}; \varphi\right)] = \sum_{(\mathsf{u},\mathsf{p})\in U\times P} \left| Auth(\mathsf{u},\mathsf{p}) - \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\varphi^{\mathfrak{X}}(\mathsf{u},\mathsf{p})\right] \right|. \tag{5.29}$$

Recall that $\wedge$ and $\neg$ form a complete set of Boolean operators. So one can also use this lemma to compute expectations of diverse random variables of the form $(\varphi \to \psi)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$ and $(\varphi \vee \psi)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$.

5.7 Mining policies

We explain next how to use Unicorn to build policy miners for a wide variety of policy languages.

5.7.1 RBAC policies

We already explained how formula $\varphi_N^{RBAC} \in \mathcal{L}$ is a template formula for the language of all RBAC policies with at most N roles. To implement Algorithm 5, we only need a procedure to compute $\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[L\left(Auth, \mathfrak{X}; \varphi_N^{RBAC}\right)\right]$. Since, as noted in Example 78, $\left(\varphi_N^{RBAC}\right)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})$ is diverse, we can apply Lemma 79 to show that

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[L\left(Auth, \mathfrak{X}; \varphi_N^{RBAC}\right)\right] = \sum_{(\mathsf{u},\mathsf{p})\in U\times P} \left| Auth(\mathsf{u},\mathsf{p}) - \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\left(\varphi_N^{RBAC}\right)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})\right] \right|. \tag{5.30}$$

$$\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[\left(\varphi_N^{RBAC}\right)^{\mathfrak{X}}(\mathsf{u},\mathsf{p})\right] = \tag{5.31}$$
$$1-\prod_{i\leq N}\left(1-\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[UA^{\mathfrak{X}}(\mathsf{u},\underline{\mathsf{r}}_i)\right]\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[PA^{\mathfrak{X}}(\underline{\mathsf{r}}_i,\mathsf{p})\right]\right),$$

where,

$$\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[UA^{\mathfrak{X}}(\mathsf{u},\underline{\mathsf{r}}_i)\right] = \begin{cases}\mathsf{b} & \text{if } UA^{\mathfrak{X}}(\mathsf{u},\underline{\mathsf{r}}_i)=\mathfrak{f}\\ \sum_{\mathsf{b}\in Range\left(UA^{\mathfrak{X}}(\mathsf{u},\underline{\mathsf{r}}_i)\right)}\theta_{UA^{\mathfrak{X}}(\mathsf{u},\underline{\mathsf{r}}_i)}(\mathsf{b})\,\mathsf{b} & \text{otherwise.}\end{cases}$$

$\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[PA^{\mathfrak{X}}(\underline{\mathsf{r}}_i,\mathsf{p})\right]$ is computed analogously. We can now implement an RBAC miner by implementing Algorithm 5 in a standard programming language and using the results above to compute the needed expectations.

5.7.2 Simple RBAC policies

The objective function used above has a limitation. When the number of role constants N used by $\varphi_N^{RBAC}(u,p)$ is very large, we might obtain a policy $\tilde{\mathfrak{I}}$ that assigns each role to exactly one user. The role assigned to a user would be assigned all permissions that the user needs. As a result, $L(Auth,\tilde{\mathfrak{I}};\varphi_N^{RBAC})=0$, but $\tilde{\mathfrak{I}}$ is not a desirable policy. We can avoid mining such policies by introducing in the objective function a *regularization term* that measures the complexity of the mined policy $\mathfrak{I}$. A candidate regularization term is:

$$\|\mathfrak{I}\| = \sum_{i\leq N}\left(\sum_{\mathsf{u}\in U}UA^{\mathfrak{I}}(\mathsf{u},\underline{\mathsf{r}}_i)+\sum_{\mathsf{p}\in P}PA^{\mathfrak{I}}(\underline{\mathsf{r}}_i,\mathsf{p})\right). \tag{5.32}$$

Observe that $\|\mathfrak{I}\|$ measures the sizes of the relations $UA^{\mathfrak{I}}$ and $PA^{\mathfrak{I}}$, for $i\leq N$, thereby providing a measure of $\mathfrak{I}$'s complexity. We now define the following loss function:

$$L^r_{RBAC}(Auth,\mathfrak{I}) = \lambda\,\|\mathfrak{I}\| + L(Auth,\mathfrak{I};\varphi). \tag{5.33}$$

Here, $\lambda>0$ is a trade-off hyper-parameter, which again must be fixed before executing the policy miner and can be estimated using grid search. Note that L^r_{RBAC} now penalizes not only policies that substantially disagree with *Auth*, but also policies that are too complex.

The computation of $\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[L^r_{RBAC}(Auth,\mathfrak{X})\right]$ now also requires the computation of $\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}[\|\mathfrak{X}\|]$, where $\|\mathfrak{X}\|$ is the random variable obtained by replacing each occurrence of $\mathfrak{I}$ in $\|\mathfrak{I}\|$ with $\mathfrak{X}$. Fortunately, one can see that $\|\mathfrak{X}\|$ is diverse. Hence, we can use the linearity of expectation and Lemma 79 to compute all needed expectations.

5.7.3 Mining policies from unbalanced permission assignments

During our experiments in Section 5.10, we found permission assignments where the set of authorized requests was so small in comparison to the whole set of requests, that policy miners produced policies that did not authorize any request.

To force miners to give more importance to the authorized requests, we use the objective function $\ell\left(Auth, \mathfrak{I}; \varphi\right)$:

$$\lambda_1 \sum_{(\mathsf{u},\mathsf{p}) \in Auth} \left(1 - \varphi^{\mathfrak{I}}(\mathsf{u},\mathsf{p})\right) + \lambda_2 \sum_{(\mathsf{u},\mathsf{p}) \notin Auth} \varphi^{\mathfrak{I}}(\mathsf{u},\mathsf{p}). \tag{5.34}$$

Here, λ_1 and λ_2 are hyper-parameters assigning weight importance to the authorized and denied requests, respectively. By setting λ_1 substantially higher than λ_2, the mined policies were more accurate when deciding requests in *Auth*. However, one must take care not to set λ_1 too high, as this would incentivize policy miners to mine overly permissive policies. To find suitable values for hyper-parameters, we refer to Appendix C.1.

5.7.4 ABAC policies

When mining ABAC policies, we are not only given a permission assigment $Auth \subseteq U \times P$, but also *attribute assignment relations* $UAtt \subseteq U \times AttVals$ and $PAtt \subseteq P \times AttVals$ that describe what *attribute values* each user and each permission has. Here, *AttVals* denotes the set of possible attribute values. We refer to previous work for a discussion on how to obtain these attribute assignment relations [133, 39].

The objective in mining ABAC policies is to find a set of rules that assigns permissions to users based on the users' and the permissions' attribute values. We explain next how to build a policy miner for ABAC using UNICORN. Let *Rules* and *AttVals* be sorts for rules and attribute values, respectively. Let *RUA* and *RPA* be flexible binary relation symbols of type $Rules \times AttVals$. For $M, N \in \mathbb{N}$, the formula $\varphi^{ABAC}_{M,N}(u,p)$ below is a template formula for ABAC:

$$\bigvee_{i \leq N} \bigwedge_{j \leq M} \left(\begin{array}{l} \left(RUA\left(\underline{\mathsf{s}}_i, \underline{\mathsf{a}}_j\right) \to \underline{UAtt}(u, \underline{\mathsf{a}}_j)\right) \wedge \\ \left(RPA\left(\underline{\mathsf{s}}_i, \underline{\mathsf{a}}_j\right) \to \underline{PAtt}(p, \underline{\mathsf{a}}_j)\right) \end{array} \right). \tag{5.35}$$

In this formula, $\underline{\mathsf{s}}_i$, for $i \leq N$, is a rigid constant symbol of sort *Rules* denoting a rule. The symbol $\underline{\mathsf{a}}_j$, for $j \leq M$, is a rigid constant denoting an attribute value. The formula $RUA\left(\underline{\mathsf{s}}_i, \underline{\mathsf{a}}_j\right)$ describes whether rule $\underline{\mathsf{s}}_i$ requires the user to have the attribute value $\underline{\mathsf{a}}_j$. The formula $RPA\left(\underline{\mathsf{s}}_i, \underline{\mathsf{a}}_j\right)$ describes an analogous requirement. We use two rigid relation symbols $\underline{UAtt}$ and $\underline{PAtt}$ to represent the attribute assignment relations. The formulas $\underline{UAtt}\left(u, \underline{\mathsf{a}}_j\right)$ and $\underline{PAtt}\left(p, \underline{\mathsf{a}}_j\right)$ describe whether u and p, respectively, are assigned the attribute value $\underline{\mathsf{a}}_j$. Intuitively, the formula $\varphi^{ABAC}_{M,N}(u,p)$ is satisfied by $(\mathsf{u},\mathsf{p}) \in U \times P$ if, for some rule $\underline{\mathsf{s}}_i$, (u,p) possesses all user and permission attribute values required by $\underline{\mathsf{s}}_i$ under RUA and RPA.

Observe that a policy miner does not need to find an interpretation for the symbols $\underline{UAtt}$ and $\underline{PAtt}$ because *the organization already has interpretations for those symbols*. When mining ABAC policies, the organization already knows what attribute values each user and each permission has and wants to mine from them an ABAC policy. The miner only needs to specify which attribute values must be required by each rule. This is why we specify the attribute assignment relations with rigid symbols.

We use $L(Auth, \mathfrak{I}; \varphi^{ABAC}_{M,N})$ as the objective function. Observe that every atomic formula occurs at most once in $\varphi^{ABAC}_{M,N}$, so, by Corollary 77, we can use Lemma 79 to compute all relevant expectations.

Finally, we can also add a regularization term to $L(Auth, \mathfrak{I}; \varphi^{ABAC}_{M,N})$ to avoid mining policies with too many rules or unnecessarily large rules. One such regularization term is

$$\|\mathfrak{I}\| = \sum_{i \leq N} \sum_{j \leq M} RUA^{\mathfrak{I}}\left(\underline{\mathsf{s}}_i, \underline{\mathsf{a}}_j\right) + RPA^{\mathfrak{I}}\left(\underline{\mathsf{s}}_i, \underline{\mathsf{a}}_j\right). \tag{5.36}$$

The expression $\|\mathfrak{I}\|$ counts the number of attribute values required by each rule, which is a common way to measure an ABAC policy's complexity [133, 39]. If we instead use the objective function $\lambda \|\mathfrak{I}\| + L(Auth, \mathfrak{I}; \varphi^{ABAC}_{M,N})$, then the objective function penalizes not only policies that differ substantially from *Auth*, but also policies that are too complex. Observe that $\|\mathfrak{X}\|$ is diverse. Hence, we can use the linearity of expectation and Lemma 79 to compute all expectations needed to implement Algorithm 5.

5.7.5 ABAC policies from logs

Some policy miners, like Rhapsody, are geared towards mining policies from logs of access requests [103, 132, 39]. We now present an objective function that can be used to mine ABAC policies from access logs, instead of permission assignments. We let $\varphi \equiv \varphi^{ABAC}_{M,N}$ for the rest of this subsection.

A log G is a disjoint union of two subsets A and D of $U \times P$, denoting the set of requests that have been *authorized* and *denied*, respectively.

In the case of ABAC, a policy mined from a log should aim to fulfill three requirements. As discussed in Section 2.5, the policy should be *succinct*, *generalize well*, and be *precise* [39]. Therefore, we define an objective function

$L'_{ABAC}(G, \mathfrak{I})$ as the sum

$$L'_{ABAC}(G, \mathfrak{I}) = \lambda_0 \|\mathfrak{I}\| + L_1(G, \mathfrak{I}) + L_2(G, \mathfrak{I}). \tag{5.37}$$

The term $\|\mathfrak{I}\|$ is as defined in Section 5.7.4 and aims to make the policy succinct by penalizing complex policies. The term $L_1(G, \mathfrak{I})$ aims to make the mined policy generalize well and is defined as follows:

$$\begin{aligned} L_1(G, \mathfrak{I}) = \lambda_{1,1} \sum_{(\mathsf{u},\mathsf{p}) \in A} \left(1 - \varphi^{\mathfrak{I}}(\mathsf{u}, \mathsf{p})\right) + \\ \lambda_{1,2} \sum_{(\mathsf{u},\mathsf{p}) \in D} \varphi^{\mathfrak{I}}(\mathsf{u}, \mathsf{p}). \end{aligned} \tag{5.38}$$

Finally, the function $L_2(G, \mathfrak{I})$ aims to make the mined policy precise by penalizing policies that authorize too many requests that are not in the log.

$$L_2(G, \mathfrak{I}) = \lambda_2 \sum_{(\mathsf{u},\mathsf{p}) \in U \times P \setminus G} \varphi^{\mathfrak{I}}(\mathsf{u}, \mathsf{p}). \tag{5.39}$$

One can show that $\varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})$ is diverse, for any $(\mathsf{u}, \mathsf{p}) \in U \times P$. Therefore, we can compute $\mathbb{E}_{\mathsf{f} \mapsto \mathsf{b}}\left[L'_{ABAC}(G, \mathfrak{X})\right]$ using only the linearity of expectation and Lemma 79.

5.7.6 Business-meaningful RBAC policies

Frank et. al. [57] developed a probabilistic policy miner for RBAC policies that incorporated business information. Aside from a permission assignment, the miner takes as input an *attribute-assignment relation* $AA \subseteq U \times AVal$, where $AVal$ denotes all possible combination of attribute values. It is assumed that each user is assigned exactly one combination of attribute values.

This miner grants similar sets of roles to users that have similar attribute values. For this, it uses the following formula $\Delta(\mathsf{u}, \mathsf{u}', \mathfrak{I})$ that measures the disagreement between the roles that a policy $\mathfrak{I}$ assigns to two users u and u':

$$\begin{aligned} &\Delta(\mathsf{u}, \mathsf{u}', \mathfrak{I}) = \\ &\sum_{i \leq N} UA^{\mathfrak{I}}(\mathsf{u}, \underline{\mathsf{r}}_i) \left(1 - 2UA^{\mathfrak{I}}(\mathsf{u}, \underline{\mathsf{r}}_i) UA^{\mathfrak{I}}(\mathsf{u}', \underline{\mathsf{r}}_i)\right). \end{aligned} \tag{5.40}$$

The formula $\|\mathfrak{I}\|$ below shows how Frank et al.'s miner measures an RBAC policy's complexity. The complexity increases whenever two users with the same combination of attribute values get assigned significantly different sets of roles.

$$\|\mathfrak{I}\| = \frac{1}{N} \sum_{\mathsf{u},\mathsf{u}' \in U} \sum_{\underline{\mathsf{a}} \in AVal} \underline{AA}(\mathsf{u}, \underline{\mathsf{a}}) \underline{AA}(\mathsf{u}', \underline{\mathsf{a}}) \Delta(\mathsf{u}, \mathsf{u}', \mathfrak{I}). \tag{5.41}$$

Here, N denotes the total of users. Note that $\underline{AA}$ is a rigid relation symbol representing AA. Its interpretation is therefore fixed and not computed by the policy miner.

To mine business-meaningful RBAC policies, we use the objective function $\lambda \|\mathfrak{I}\| + L\left(Auth, \mathfrak{I}; \varphi_N^{RBAC}\right)$, where $\lambda > 0$ is a trade-off hyper-parameter. Observe that this objective function penalizes the following types of policies.

- Policies that assign significantly different sets of roles to users with the same attribute values.
- Policies whose assignment of permissions to users substantially differs from the assignment given by *Auth*.

The random variable $\|\mathfrak{X}\|$ is, however, not diverse. This is because, for $i \leq N$, the random fact $UA^{\mathfrak{X}}(\mathsf{u}, \underline{\mathsf{r}}_i)$ occurs more than once in $\Delta(\mathsf{u}, \mathsf{u}', \mathfrak{X})$. Nonetheless, observe that

$$\begin{aligned}&\Delta(\mathsf{u}, \mathsf{u}', \mathfrak{X}) = \\ &\sum_{i \leq N} UA^{\mathfrak{X}}(\mathsf{u}, \underline{\mathsf{r}}_i) - 2\left(UA^{\mathfrak{X}}(\mathsf{u}, \underline{\mathsf{r}}_i)\right)^2 UA^{\mathfrak{X}}(\mathsf{u}', \underline{\mathsf{r}}_i).\end{aligned} \tag{5.42}$$

One can then compute $\mathbb{E}_{\mathfrak{f} \to \mathsf{b}}[\Delta(\mathsf{u}, \mathsf{u}', \mathfrak{X})]$ by using the linearity of expectation and the fact that $\mathbb{E}\left[X^n\right] = \left(\mathbb{E}\left[X\right]\right)^n$, for $n \in \mathbb{N}$ and X a Bernoulli random variable. Hence,

$$\begin{aligned}&\mathbb{E}_{\mathfrak{f} \to \mathsf{b}}\left[\Delta(\mathsf{u}, \mathsf{u}', \mathfrak{X})\right] = \\ &\sum_{i \leq N} \left(\begin{array}{l} \mathbb{E}_{\mathfrak{f} \to \mathsf{b}}\left[UA^{\mathfrak{X}}(\mathsf{u}, \underline{\mathsf{r}}_i)\right] - \\ 2\left(\mathbb{E}_{\mathfrak{f} \to \mathsf{b}}\left[UA^{\mathfrak{X}}(\mathsf{u}, \underline{\mathsf{r}}_i)\right]\right)^2 \mathbb{E}_{\mathfrak{f} \to \mathsf{b}}\left[UA^{\mathfrak{X}}(\mathsf{u}', \underline{\mathsf{r}}_i)\right] \end{array} \right).\end{aligned} \tag{5.43}$$

One can check that this observation and Lemma 79 suffice to compute the expectations necessary for Algorithm 5.

5.7.7 Disjunctive policy languages

We call a policy language Γ *disjunctive* if (i) there is a template formula $\varphi \in \mathcal{L}$ for Γ of the form $\varphi_1 \vee \ldots \vee \varphi_n$ and (ii) every atomic formula in φ occurs exactly once. RBAC and ABAC are examples of disjunctive policy languages.

Let L be an objective function. If Γ is a disjunctive policy language, then we recommend that, instead of executing $\textsc{MinePolicy}\,(L, Auth, \varphi, \alpha, \beta, T)$, we mine N policies $\mathfrak{I}_1, \ldots, \mathfrak{I}_N$ as follows. For $i \leq N$, $\mathfrak{I}_i$ is the output of $\textsc{MinePolicy}\,(L_i, Auth_i, \varphi_i, \alpha, \beta_i, T)$ where $L_i = L(Auth_i, \mathfrak{I}; \varphi_i)$, $Auth_i$ is the result of removing from *Auth* all requests authorized by any $\mathfrak{I}_j$, with $j < i$, and $\beta_i = \eta^i \beta_0$, with $\eta > 1$ and $\beta_0 \in \mathbb{R}^+$. The final mined policy is $\bigcup_{j \leq N} \mathfrak{I}_j$. This approach works faster while still yielding policies that generalized well.

5.8 Spatio-temporal RBAC policies

We now use UNICORN to build the first policy miner for RBAC extensions with spatio-temporal constraints [6, 40, 79, 22, 18, 19]. In policies in these extensions, users are assigned permissions not only according to their roles, but also based on constraints depending on the current time and the user's and the permission's locations. The syntax for specifying these constraints allows for policies like "A user is assigned a role the third Monday of every month, except in December, and from 8:00 AM through 17:00 PM" or "A role is granted permission to access an object within a radius of three miles from the main building."

We present a template formula $\varphi^{st}(t,u,p) \in \mathcal{L}$ for a policy language that we call *spatio-temporal RBAC*. This is an extension of RBAC enhanced with a syntax for spatial constraints based on [18, 19] and a syntax for temporal constraints based on temporal RBAC [20].

$$\varphi^{st}(t,u,p) = \bigvee_{i \leq N} \left(\psi_{UA}(t,u,\underline{\mathrm{r}}_i) \;\wedge\; \psi_{PA}(t,\underline{\mathrm{r}}_i,p)\right). \tag{5.44}$$

Here, we assume the existence of a sort **TIME** and that t is a variable of this sort representing the time when u exercises p. We also assume the existence of a sort **SPACE** that we use later to specify spatial constraints. The formulas $\psi_{UA}(t,u,\underline{\mathrm{r}}_i)$ and $\psi_{PA}(t,\underline{\mathrm{r}}_i,p)$ describe when a user is assigned the role $\underline{\mathrm{r}}_i$ and when a permission is assigned to the role $\underline{\mathrm{r}}_i$, respectively. We use the rigid constants $\underline{\mathrm{r}}_1,\ldots,\underline{\mathrm{r}}_N$ to denote roles.

The grammar below defines the syntax of ψ_{UA} and ψ_{PA}.

$$\begin{aligned}
\langle \texttt{cstr\_list} \rangle &::= \langle \texttt{cstr} \rangle \;(\;\wedge\;\langle \texttt{cstr} \rangle\;)* \\
\langle \texttt{cstr} \rangle &::= \langle \texttt{sp\_cstr} \rangle \mid \langle \texttt{tmp\_cstr} \rangle \\
\langle \texttt{sp\_cstr} \rangle &::= \langle \texttt{loc} \rangle \;(\;\vee\;\langle \texttt{loc} \rangle\;)* \\
\langle \texttt{loc} \rangle &::= (\neg?)\,\underline{isWithin}\,(\underline{Loc}\,(o)\,,\mathsf{d},\mathsf{b}) \\
\langle \texttt{tmp\_cstr} \rangle &::= \langle \texttt{tmp\_itv} \rangle \;(\;\vee\;\langle \texttt{tmp\_itv} \rangle\;)* \\
\langle \texttt{tmp\_itv} \rangle &::= \psi_{cal}\,(t)
\end{aligned}$$

An expression in Γ is a conjunction of *constraints*, each of which is either a disjunction of *temporal constraints* or a disjunction of *spatial constraints*.

5.8.1 Modeling spatial constraints

A *spatial constraint* is a (possibly negated) formula of the form $\underline{isWithin}\,(\underline{Loc}\,(o)\,,\mathsf{d},\mathsf{b})$, where o is a variable of sort **USERS** or **PERMS**, $\underline{Loc}\,(o)$ denotes o's location, d is a flexible constant symbol of a sort whose carrier set is $\mathbb{N}^{\leq M} = \{0,1,\ldots,\mathsf{M}\}$ (where M is a value fixed in advance), and b is a flexible constant symbol of a sort describing the organization's physical facilities. For example, $\underline{isWithin}\left(\underline{Loc}\left(u\right), \mathsf{4}, \mathsf{MainBuilding}\right)$ holds when the user represented by u is within 4 space units of the main building.

Intuitively, the formula $\underline{isWithin}\left(\underline{Loc}\left(o\right), \mathsf{d}, \mathsf{b}\right)$ evaluates whether the entity represented by o is located within d spatial units from b. Observe that a policy miner does not need to compute interpretations for rigid function symbols like $\underline{Loc}$ or rigid relation symbols like $\underline{isWithin}$, since they already have a fixed interpretation.

5.8.2 Modeling temporal constraints

A *temporal constraint* is a formula $\psi_{cal}\left(t\right)$ that represents a *periodic expression* [20], which describes a set of time intervals. We give here a simplified overview and refer to the literature for a more technical presentation.

Definition 80 A *periodic expression* is a tuple

$$\left(yearSet, monthSet, daySet, hourSet, hourDuration\right) \in \left(2^{\mathbb{N}}\right)^4 \times \mathbb{N}. \tag{5.45}$$

A *time instant* is a tuple $\left(y, m, d, h\right) \in \mathbb{N}^4$. A time instant $\left(y, m, d, h\right)$ *satisfies* a periodic expression $\left(yearSet, monthSet, daySet, hourSet, hourDuration\right)$ if $y \in yearSet$, $m \in monthSet$, $d \in daySet$, and there is $h' \in hourSet$ such that $h' \leq h \leq h' + hourDuration$. □

Previous works on analyzing temporal RBAC with SMT solvers [75] show that temporal constraints can be expressed as formulas in $\mathcal{L}$. Furthermore, one can see that any expression in Γ and, therefore, φ^{st} are in $\mathcal{L}$.

As objective function, we use $\lambda \left\|\mathfrak{I}\right\| + L\left(Auth, \mathfrak{I}; \varphi^{st}\right)$. Here, $\left\|\mathfrak{I}\right\|$ counts the number of spatial constraints plus the sum of the *weighted structural complexities* of all temporal constraints [121]. One can show that $\left\|\mathfrak{X}\right\|$ is diverse and that every atomic formula in φ^{st} occurs exactly once. Hence, one can compute all necessary expectations using the linearity of expectation and Lemma 79.

5.9 XACML policies

Although XACML is the de facto standard for access control specification, no algorithm has previously been proposed for mining XACML policies. We now illustrate how, using Unicorn, we build the first XACML policy miner.

5.9.1 Background

XACML syntax. To simplify the presentation, we use a reduced version of XACML, given as a BNF grammar below. However, our approach extends to the core XACML. Moreover, our reduced XACML is still powerful enough to express Continue [87, 55], a benchmark XACML policy used for policy analysis.

$$\begin{array}{rcl}
\texttt{<Dec>} & ::= & \mathit{allow} \mid \mathit{deny} \\
\texttt{<Rule>} & ::= & (\langle\texttt{<Dec>}\rangle, \alpha) \\
\texttt{<Comb>} & ::= & \mathit{FirstApp} \mid \mathit{AllowOv} \mid \mathit{DenyOv} \\
\texttt{<Pol>} & ::= & (\texttt{<Comb>}, (\texttt{<Pol>}^* \mid \texttt{<Rule>}^*))
\end{array}$$

Fix a set *AVals* of attribute values. An *XACML rule* is a pair (δ, α), where $\delta \in \{\mathit{allow}, \mathit{deny}\}$ is the *rule's decision* and α is a subset of *AVals*. An XACML *policy* is a pair $(\kappa, \bar{\pi})$, where $\kappa \in \{\mathit{FirstApp}, \mathit{AllowOv}, \mathit{DenyOv}\}$ is a *combination algorithm* and $\bar{\pi}$ is either a list of policies or a list of XACML rules. *FirstApp*, *AllowOv*, *DenyOv* denote XACML's standard policy combination algorithms. We explain later how they work. For a policy π, we denote its combination algorithm by $\mathit{Comb}(\pi)$ and, for a rule r, we denote its decision by $\mathit{Dec}(r)$.

XACML semantics. We now recall XACML's semantics. A *request* is a subset of *AVals* denoting the attribute values that a subject s, an action a, and an object o satisfy when s attempts to execute a on o. We denote by 2^{AVals} the set of requests. A request *satisfies* a rule (δ, α) if the request contains all attributes in α. In this case, if $\delta = \mathit{allow}$, then we say that the rule *authorizes the request*; otherwise, we say that the rule *denies the request*.

A policy π of the form $(\mathit{AllowOv}, (\pi'_1, \ldots, \pi'_\ell))$ authorizes a request z if there is an $i \leq \ell$ such that π'_i authorizes z. The policy π *denies* z if no π'_i, for $i \leq \ell$, authorizes z, but some π'_j, for $j \leq \ell$, denies it.

A policy π of the form $(\mathit{DenyOv}, (\pi'_1, \ldots, \pi'_\ell))$ denies a request if some π'_i denies it. The policy authorizes the request if no π'_i denies it, but some π'_i authorizes it.

A policy π of the form $(\mathit{FirstApp}, (\pi'_1, \ldots, \pi'_\ell))$ authorizes a request if there is an $i \leq \ell$ such that π'_i authorizes it and π'_j, for $j < i$, neither authorizes it nor denies it. The policy denies a request if there is $i \leq \ell$ such that π'_i denies it and π'_j, for $j < i$, neither authorizes it nor denies it.

5.9.2 Auxiliary definitions

We provide some definitions that will facilitate some proofs later. For a policy $\pi = (\kappa, (\pi'_1, \ldots, \pi'_\ell))$, we call π'_i a *child* of π. A policy is a *descendant* of π if it is a child of π or is a descendant of a child of π.

A policy π has *breadth* $N \in \mathbb{N}$ if $\ell \leq N$ and each of π's children is either a rule or has breadth N. A policy π has *depth* is $M \in \mathbb{N}$ (i) if $M = 1$ and each of its children is a rule, or (ii) if $M > 1$ and some child of π has depth $M - 1$ and the rest have depth at most $M - 1$.

Two formulas $\psi_1, \psi_2 \in \mathcal{L}$ are *mutually exclusive* if there is no $\mathfrak{I}$ and no $\mathsf{z} \in 2^{AVals}$ such that both $\psi_1^{\mathfrak{I}}(\mathsf{z})$ and $\psi_2^{\mathfrak{I}}(\mathsf{z})$ hold. When ψ_1 and ψ_2 are mutually exclusive, we write $\psi_1 \oplus \psi_2$ instead of $\psi_1 \vee \psi_2$.

5.9.3 A template formula for XACML

For $M, N \in \mathbb{N}$, we present a template formula for the language of all XACML policies of depth and breadth at most M and N, respectively.

Let $\mathcal{S}$ be the set of all N-ary sequences of length at most M and let $\varepsilon \in \mathcal{S}$ be the empty sequence. For $j \in \{0, \ldots, N-1\}$, we denote by $\sigma \triangleright j$ the result of appending j to σ and by $j \triangleleft \sigma$ the result of prepending j to σ.

Let **REQS** be a sort representing all requests, **AVALS** be a sort representing all attribute values, and **POLS** a sort representing policies and rules. For each $\sigma \in \mathcal{S}$, define a rigid constant $\underline{\mathsf{y}}_\sigma$ symbol of sort **POLS** such that $\underline{\mathsf{y}}_\sigma \neq \underline{\mathsf{y}}_{\sigma'}$, whenever $\sigma \neq \sigma'$.

The set of rigid constants $\{\underline{\mathsf{y}}_\sigma \mid \sigma \in \mathcal{S}\}$ are intended to represent a tree of XACML policies and rules. The constant $\underline{\mathsf{y}}_\varepsilon$ is the root policy. For $\sigma \in \mathcal{S}$ with length less than M and $j \in \{0, \ldots, N-1\}$, the constant $\underline{\mathsf{y}}_{\sigma \triangleright j}$ represents one of $\underline{\mathsf{y}}_\sigma$'s children.

Let z be a variable of sort **REQS**. The formula $\varphi_{M,N}^{\text{XACML}}\left(\underline{\mathsf{y}}_\varepsilon, z\right)$ below is a template formula for the XACML fragment introduced above. We explain its main parts.

- We define signature symbols that represent all terminal symbols in the BNF grammar above. For example, we define two rigid constant symbols $\underline{\texttt{XAllows}}$ and $\underline{\texttt{XDenies}}$ that represent the decisions `allows` and `denies`. We define two flexible function symbols *XDec* and *XComb*. For a rigid constant $\underline{\mathsf{y}}_\sigma$, $XDec\left(\underline{\mathsf{y}}_\sigma\right)$ denotes the decision of the rule represented by $\underline{\mathsf{y}}_\sigma$. Similarly, $XComb\left(\underline{\mathsf{y}}_\sigma\right)$ denotes the combination algorithm of the policy represented by $\underline{\mathsf{y}}_\sigma$.
- The formula $\texttt{allows}\left(\underline{\mathsf{y}}_\sigma, z\right)$ holds if $\underline{\mathsf{y}}_\sigma$ authorizes the request represented by z. The formula $\texttt{denies}\left(\underline{\mathsf{y}}_\sigma, z\right)$ holds if $\underline{\mathsf{y}}_\sigma$ denies the request represented by z and is defined analogously. Observe that $\texttt{allows}\left(\underline{\mathsf{y}}_\sigma, z\right)$ and $\texttt{denies}\left(\underline{\mathsf{y}}_\sigma, z\right)$ denote formulas. Hence, `allows` and `denies` are not symbols in the signature we use to specify $\varphi_{M,N}^{\text{XACML}}$.
- *XActive* is a flexible relation symbol and $XActive\left(\underline{\mathsf{y}}_\sigma\right)$ holds if $\underline{\mathsf{y}}_\sigma$ is a descendant of $\underline{\mathsf{y}}_\varepsilon$.

- The formula $\mathtt{NA}\left(\underline{\mathsf{y}}_{\sigma}, z\right)$ holds if $\underline{\mathsf{y}}_{\sigma}$ neither authorizes nor denies the request represented by z. It can be expressed in $\mathcal{L}$ as follows:

$$\mathtt{NA}\left(\underline{\mathsf{y}}_{\sigma}, z\right) \equiv \neg XActive\left(\underline{\mathsf{y}}_{\sigma}\right) \vee \bigwedge_{j \leq N} \mathtt{NA}\left(\underline{\mathsf{y}}_{\sigma \triangleright j}, z\right). \tag{5.46}$$

- The formula $z \vDash \underline{\mathsf{y}}_{\sigma}$ holds if all attributes required by $\underline{\mathsf{y}}_{\sigma}$ are contained by the request represented by z. This formula can be expressed in $\mathcal{L}$ as follows:

$$\bigwedge_{\mathsf{a} \in AVals} \left(XReqsAVal\left(\underline{\mathsf{y}}_{\sigma}, \underline{\mathsf{a}}\right) \rightarrow \underline{hasAttVal}\left(z, \underline{\mathsf{a}}\right)\right), \tag{5.47}$$

where $XReqsAVal$ is a flexible relation symbol and $\underline{hasAttVal}$ and $\underline{\mathsf{a}}$, for $\mathsf{a} \in AVals$, are rigid symbols. For a policy $\mathfrak{I}$, $XReqsAVal^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}, \underline{\mathsf{a}}\right)$ holds if $\underline{\mathsf{y}}_{\sigma}$ is a rule and requires attribute a to be satisfied. The formula $\underline{hasAttVal}\left(z, \underline{\mathsf{a}}\right)$ checks if the request contains attribute a.

$$\varphi^{\mathrm{XACML}}_{M,N}\left(\underline{\mathsf{y}}_{\varepsilon}, z\right) \equiv \mathtt{allows}\left(\underline{\mathsf{y}}_{\varepsilon}, z\right) \tag{5.48}$$

$$\mathtt{allows}\left(\underline{\mathsf{y}}_{\sigma}, z\right) \equiv \left(XIsRule\left(\underline{\mathsf{y}}_{\sigma}\right) \rightarrow \mathtt{allowsRule}\left(\underline{\mathsf{y}}_{\sigma}, z\right)\right) \wedge \left(\neg XIsRule\left(\underline{\mathsf{y}}_{\sigma}\right) \rightarrow \mathtt{allowsPol}\left(\underline{\mathsf{y}}_{\sigma}, z\right)\right) \tag{5.49}$$

$$\mathtt{allowsRule}\left(\underline{\mathsf{y}}_{\sigma}, z\right) \equiv XActive\left(\underline{\mathsf{y}}_{\sigma}\right) \wedge XDec\left(\underline{\mathsf{y}}_{\sigma}\right) = \underline{allow} \wedge z \vDash \underline{\mathsf{y}}_{\sigma} \tag{5.50}$$

$$\mathtt{allowsPol}\left(\underline{\mathsf{y}}_{\sigma}, z\right) \equiv XActive\left(\underline{\mathsf{y}}_{\sigma}\right) \wedge \tag{5.51}$$

$$\left[\begin{array}{l} \left(\begin{array}{l} XComb\left(\underline{\mathsf{y}}_{\sigma}\right) = \underline{AllowOv} \wedge \\ \bigvee_{j \leq N} \mathtt{allows}\left(\underline{\mathsf{y}}_{\sigma \triangleright j}, z\right) \end{array}\right) \oplus \\ \left(\begin{array}{l} XComb\left(\underline{\mathsf{y}}_{\sigma}\right) = \underline{FirstApp} \wedge \\ \bigoplus_{j \leq N} \left(\begin{array}{l} \bigwedge_{i<j} \mathtt{NA}\left(\underline{\mathsf{y}}_{\sigma \triangleright i}, z\right) \wedge \\ \mathtt{allows}\left(\underline{\mathsf{y}}_{\sigma \triangleright j}, z\right) \end{array}\right) \end{array}\right) \oplus \\ \left(\begin{array}{l} XComb\left(\underline{\mathsf{y}}_{\sigma}\right) = \underline{DenyOv} \wedge \\ \bigoplus_{j \leq N} \left(\begin{array}{l} \bigwedge_{i<j} \mathtt{NA}\left(\underline{\mathsf{y}}_{\sigma \triangleright i}, z\right) \wedge \\ \mathtt{allows}\left(\underline{\mathsf{y}}_{\sigma \triangleright j}, z\right) \wedge \bigwedge_{i<k} \neg\mathtt{denies}\left(\underline{\mathsf{y}}_{\sigma \triangleright k}, z\right) \end{array}\right) \end{array}\right) \end{array}\right] \tag{5.52}$$

Lemma 81 Formula $\varphi^{\mathrm{XACML}}_{M,N}$ is a template formula for the language of all XACML policies of depth and breadth at most M and N, respectively.

Proof. We define a mapping $\mathcal{M}$ from interpretation functions to XACML policies using an auxiliary mapping $\mathcal{M}'$. For a sequence $\sigma \in \mathcal{S}$, we inductively define $\mathcal{M}'(\mathfrak{I}, \sigma)$ as follows:

- If (i) σ has length M or (ii) $XActive^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma \triangleright j}\right) = 0$, for all $j \leq N$, then

$$\mathcal{M}'(\mathfrak{I}, \sigma) = \left(XDec^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}\right), \left\{a \in AVals \mid XReqsAVal^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}, \underline{\mathsf{a}}\right)\right\}\right). \quad (5.53)$$

- Otherwise,

$$\mathcal{M}'(\mathfrak{I}, \sigma) = \left(XComb^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}\right), \left(\mathcal{M}'(\mathfrak{I}, \sigma \triangleright j)_{j \leq N}\right)\right). \quad (5.54)$$

For an interpretation function $\mathfrak{I}$, we define $\mathcal{M}(\mathfrak{I}) = \mathcal{M}'(\mathfrak{I}, \varepsilon)$. We show that $\mathcal{M}$ is surjective. Let π be a XACML policy. For $\sigma \in \mathcal{S}$ and π' a descendant of π, we inductively define the following policy:

$$\pi'[\sigma] \equiv \begin{cases} \pi' & \text{if } \sigma = \varepsilon \text{ and} \\ \pi'_i[\sigma'] & \text{if } \sigma = i \triangleleft \sigma' \text{ and } \pi' = \left(\kappa, \left(\pi'_1, \ldots, \pi'_k\right)\right). \end{cases} \quad (5.55)$$

We now present an interpretation function $\mathfrak{I}$ such that $\mathcal{M}(\mathfrak{I}) = \pi$. Let $\sigma \in \mathcal{S}$ and $\perp$ be any arbitrary value. Then

$$XActive^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}\right) \begin{cases} 1 & \text{if there is a descendant } \pi' \text{ of } \pi \text{ with } \pi[\sigma] = \pi' \text{ and} \\ 0 & \text{otherwise.} \end{cases} \quad (5.56)$$

$$XComb^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}\right) \begin{cases} Comb(\pi[\sigma]) & \text{if } \pi[\sigma] \text{ is a policy and} \\ \perp & \text{otherwise.} \end{cases} \quad (5.57)$$

$$XDec^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}\right) \begin{cases} Dec(\pi[\sigma]) & \text{if } \pi[\sigma] \text{ is a rule and} \\ \perp & \text{otherwise.} \end{cases} \quad (5.58)$$

$$XIsRule^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}\right) \begin{cases} 1 & \text{if } \pi[\sigma] \text{ is a rule and} \\ 0 & \text{otherwise.} \end{cases} \quad (5.59)$$

$$XReqsAVal^{\mathfrak{I}}\left(\underline{\mathsf{y}}_{\sigma}, \underline{\mathsf{a}}\right) \begin{cases} 1 & \text{if } \pi[\sigma] \text{ is a rule of the form } (d, \alpha), \\ & \alpha \subseteq AVals, \text{ and } a \in \alpha \\ 0 & \text{otherwise.} \end{cases} \quad (5.60)$$

It is straightforward to verify that $\mathcal{M}(\mathfrak{I}) = \pi$. $\square$

Having a template formula for this XACML fragment, we now define an objective function. An example of an objective function is $\lambda \|\mathfrak{I}\| + L\left(Auth, \mathfrak{I}; \varphi_{M,N}^{\text{XACML}}\right)$, where $\lambda > 0$ is a hyper-parameter and $\|\mathfrak{I}\|$ defines $\mathfrak{I}$'s complexity. We inductively define the complexity *compl* (π) of a XACML policy π as follows.

- If π is a rule of the form (δ, α), then *compl* $(\pi) = |\alpha|$.
- If π is a policy of the form $(\kappa, (\pi_1, \ldots, \pi_k))$, then *compl* $(\pi) = |\alpha|$.

Finally, we define $\|\mathfrak{I}\|$ as *compl* $(\mathcal{M}(\mathfrak{I}))$.

5.9.4 Computing expectations

For a formula $\varphi \in \mathcal{L}$ and a request $\mathsf{z} \in 2^{AVals}$, we define the random variable $\varphi^{\mathfrak{X}}(\mathsf{z})$ in a way similar to the one given in Section 5.5.1. We now give some auxiliary definitions that help to compute $\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[\left(\varphi_{M,N}^{\text{XACML}}\right)^{\mathfrak{X}}(\mathsf{z})\right]$.

Lemma 82 Let $\mathsf{z} \in 2^{AVals}$ and ψ_1, ψ_2 be mutually exclusive formulas, then

$$\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[(\psi_1 \oplus \psi_2)^{\mathfrak{X}}(\mathsf{z})\right] = \mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[\psi_1^{\mathfrak{X}}(\mathsf{z})\right] + \mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[\psi_2^{\mathfrak{X}}(\mathsf{z})\right]. \tag{5.61}$$

Proof. Note that $(\psi_1 \oplus \psi_2)^{\mathfrak{I}}(\mathsf{z}) = 1$ iff either $\psi_1^{\mathfrak{I}}(\mathsf{z}) = 1$ or $\psi_2^{\mathfrak{I}}(\mathsf{z}) = 1$.

$$\begin{aligned}
\mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[(\psi_1 \oplus \psi_2)^{\mathfrak{X}}(\mathsf{z})\right] &= \sum_{\mathfrak{I}} q(\mathfrak{I})\,(\psi_1 \oplus \psi_2)^{\mathfrak{I}}(\mathsf{z}) && (5.62)\\
&= \sum_{\mathfrak{I}:(\psi_1\oplus\psi_2)^{\mathfrak{I}}(\mathsf{z})=1} q(\mathfrak{I}) && (5.63)\\
&= \sum_{\mathfrak{I}:\psi_1^{\mathfrak{I}}(\mathsf{z})=1} q(\mathfrak{I}) + \sum_{\mathfrak{I}:\psi_2^{\mathfrak{I}}(\mathsf{z})=1} q(\mathfrak{I}) && (5.64)\\
&= \sum_{\mathfrak{I}} q(\mathfrak{I})\,\psi_1^{\mathfrak{I}}(\mathsf{z}) + \sum_{\mathfrak{I}} q(\mathfrak{I})\,\psi_2^{\mathfrak{I}}(\mathsf{z}) && (5.65)\\
&= \mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[\psi_1^{\mathfrak{X}}(\mathsf{z})\right] + \mathbb{E}_{\mathfrak{f}\mapsto \mathsf{b}}\left[\psi_2^{\mathfrak{X}}(\mathsf{z})\right]. && (5.66)
\end{aligned}$$

$\square$

Definition 83 A set $\Phi \subseteq \mathcal{L}$ of formulas is *unrelated* if for every $\varphi \in \Phi$ and every atomic formula α occurring in φ, there is no $\psi \in \Phi \setminus \{\varphi\}$ such that α occurs in φ. $\square$

Lemma 84 If $\mathsf{z} \in 2^{AVals}$ and Φ is a set of unrelated formulas, then $\left\{\varphi^{\mathfrak{X}}(\mathsf{z}) \mid \varphi \in \Phi\right\}$, under the distribution q, is mutually independent.

Proof. For simplicity, we assume that $\Phi = \{\varphi_1, \varphi_2\}$. The proof for the general case is analogous.

Observe that, since Φ is unrelated, an interpretation function $\mathfrak{I}$ can be regarded as the union of two interpretation functions $\mathfrak{I}_1$ and $\mathfrak{I}_2$ where $\mathfrak{I}_1$ interprets the atomic formulas occurring in φ_1 and $\mathfrak{I}_2$ interprets those in φ_2. Consequently, the distribution $q(\mathfrak{I})$ can be factorized as $q_1(\mathfrak{I}_1)\, q_2(\mathfrak{I}_2)$, where q_i, for $i \leq 2$, is the marginal mean-field approximating joint distribution of the random facts of φ_i.

For an event A, let $\mathbb{P}_q(A)$ denote the probability of A under the distribution q. To prove the lemma, it suffices to show, for $\mathsf{b}_1, \mathsf{b}_2 \in \{0,1\}$, that $\mathbb{P}_q\left(\varphi_1^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_1, \varphi_2^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_2\right) = \mathbb{P}_q\left(\varphi_1^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_1\right)\mathbb{P}_q\left(\varphi_2^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_2\right)$, which implies that Φ is mutually independent.

$$\mathbb{P}_q\left(\varphi_1^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_1, \varphi_2^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_2\right) \tag{5.67}$$

$$= \sum_{\substack{\mathfrak{I}:\varphi_1^{\mathfrak{I}}(\mathsf{z})=\mathsf{b}_1\\ \varphi_2^{\mathfrak{I}}(\mathsf{z})=\mathsf{b}_2}} q(\mathfrak{I}) \tag{5.68}$$

$$= \sum_{\substack{\mathfrak{I}:\varphi_1^{\mathfrak{I}}(\mathsf{z})=\mathsf{b}_1\\ \varphi_2^{\mathfrak{I}}(\mathsf{z})=\mathsf{b}_2}} q_1(\mathfrak{I}_1)\, q_2(\mathfrak{I}_2) \tag{5.69}$$

$$= \sum_{\mathfrak{I}_1:\varphi_1^{\mathfrak{I}_1}(z)=\mathsf{b}_1} \sum_{\mathfrak{I}_2:\varphi_2^{\mathfrak{I}_2}(\mathsf{z})=\mathsf{b}_2} q_1(\mathfrak{I}_1)\, q_2(\mathfrak{I}_2) \tag{5.70}$$

$$= \left(\sum_{\mathfrak{I}_1:\varphi_1^{\mathfrak{I}_1}(\mathsf{z})=\mathsf{b}_1} q_1(\mathfrak{I}_1)\right)\left(\sum_{\mathfrak{I}_2:\varphi_2^{\mathfrak{I}_2}(\mathsf{z})=\mathsf{b}_2} q_2(\mathfrak{I}_2)\right) \tag{5.71}$$

$$= \left(\sum_{\mathfrak{I}_1:\varphi_1^{\mathfrak{I}_1}(\mathsf{z})=\mathsf{b}_1} \sum_{\mathfrak{I}_2} q_1(\mathfrak{I}_1)\, q_2(\mathfrak{I}_2)\right)\left(\sum_{\mathfrak{I}_1} \sum_{\mathfrak{I}_2:\varphi_2^{\mathfrak{I}_2}(z)=\mathsf{b}_2} q_1(\mathfrak{I}_1)\, q_2(\mathfrak{I}_2)\right) \tag{5.72}$$

$$= \left(\sum_{\mathfrak{I}:\varphi_1^{\mathfrak{I}}(\mathsf{z})=\mathsf{b}_1} q(\mathfrak{I})\right)\left(\sum_{\mathfrak{I}:\varphi_2^{\mathfrak{I}}(\mathsf{z})=\mathsf{b}_2} q(\mathfrak{I})\right) \tag{5.73}$$

$$= \mathbb{P}_q\left(\varphi_1^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_1\right)\mathbb{P}_q\left(\varphi_2^{\mathfrak{X}}(\mathsf{z}) = \mathsf{b}_2\right). \tag{5.74}$$

□

Lemma 85 We can compute $\mathbb{E}_{\mathsf{f}\to\mathsf{b}}\left[\left(\varphi_{M,N}^{\mathrm{XACML}}\right)^{\mathfrak{X}}(\mathsf{z})\right]$ using only the equations given in Lemmas 79 and 82.

Proof. Observe that every atomic formula in `allowsRule` $\left(\underline{\mathsf{y}}_{\sigma}, z\right)$ occurs exactly once, so `allowsRule`$^{\mathfrak{X}}$ $\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)$ is diverse. Hence, by Corollary 77, we can use Lemma 79 to compute $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\mathtt{allowsRule}^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right]$.

The formula $\mathtt{allowsPol}\left(\underline{\mathsf{y}}_{\sigma}, z\right)$ has the form

$$XActive\left(\underline{\mathsf{y}}_{\sigma}\right) \wedge \left(\psi_1\left(\underline{\mathsf{y}}_{\sigma}, z\right) \oplus \psi_2\left(\underline{\mathsf{y}}_{\sigma}, z\right) \oplus \psi_3\left(\underline{\mathsf{y}}_{\sigma}, z\right)\right). \tag{5.75}$$

Each formula $\psi_i\left(\underline{\mathsf{y}}_{\sigma}, z\right)$ is built from a set of unrelated formulas. Hence, by Lemma 84, we can use Lemma 79 to compute $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\psi_i\left(\underline{\mathsf{y}}_{\sigma}, z\right)\right]$. Finally, observe that the set

$$\left\{XActive\left(\underline{\mathsf{y}}_{\sigma}\right), \psi_1\left(\underline{\mathsf{y}}_{\sigma}, z\right), \psi_2\left(\underline{\mathsf{y}}_{\sigma}, z\right), \psi_3\left(\underline{\mathsf{y}}_{\sigma}, z\right)\right\} \tag{5.76}$$

is unrelated. Hence, by Lemma 84, the corresponding set of random variables is independent. Using Lemmas 82 and 79, we can show that

$$\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\mathtt{allowsPol}^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right] = \tag{5.77}$$

$$\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[XActive^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right] \times \left(\begin{array}{l} \mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\psi_1^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right] + \\ \mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\psi_2^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right] + \\ \mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\psi_3^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right]. \end{array}\right) \tag{5.78}$$

Therefore, $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\mathtt{allowsPol}^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right]$ can be computed using only Lemmas 82 and 79.

Finally, recall that $\varphi_{M,N}^{\mathrm{XACML}}\left(\underline{\mathsf{y}}_{\varepsilon}, z\right) = \mathtt{allows}\left(\underline{\mathsf{y}}_{\varepsilon}, z\right)$. Observe now that $\mathtt{allows}\left(\underline{\mathsf{y}}_{\varepsilon}, z\right)$ is built from the following unrelated set:

$$\left\{XIsRule\left(\underline{\mathsf{y}}_{\sigma}\right), \mathtt{allowsRule}\left(\underline{\mathsf{y}}_{\sigma}, z\right), \mathtt{allowsPol}\left(\underline{\mathsf{y}}_{\sigma}, z\right)\right\}. \tag{5.79}$$

By Lemma 84, the corresponding set of random variables is independent. Hence, we can use Lemma 79 to reduce the computation of $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\left(\varphi_{M,N}^{\mathrm{XACML}}\right)^{\mathfrak{X}}(\mathsf{z})\right]$ to the computation of $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[XIsRule^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}\right)\right]$, $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\mathtt{allowsRule}^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right]$, and $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\mathtt{allowsPol}^{\mathfrak{X}}\left(\underline{\mathsf{y}}_{\sigma}, \mathsf{z}\right)\right]$. However, as observed above, all these expectations can be computed using Lemmas 79 and 82. Hence, we can compute $\mathbb{E}_{\mathfrak{f}\to\mathsf{b}}\left[\left(\varphi_{M,N}^{\mathrm{XACML}}\right)^{\mathfrak{X}}(\mathsf{z})\right]$ using only those two lemmas. $\square$

Having proven the previous lemmas, we can now implement Algorithm 5 to produce a policy miner for XACML policies.

5.10 Experiments

In this section, we experimentally validate two hypotheses. First, by using Unicorn, we can build policy miners for a wide variety of policy languages. Second, the policies mined by these miners have as low complexity and high generalization ability as those mined by the state of the art.

5.10.1 Datasets

Our experiments are divided in the following categories.

Mine RBAC policies from access control matrices We use three access control matrices from three real organizations, named "healthcare", "firewall", and "americas" [49]. For healthcare, there are 46 users and 46 permissions, for firewall, there are 720 users and 587 permissions, and for americas, there are more than 10,000 users and around 3,500 permissions. We refer to these access control matrices as RBAC1, RBAC2, and RBAC3.

Mine ABAC policies from logs We use four logs of access requests provided by Amazon for a Kaggle competition in 2013 [82], where participants had to develop mining algorithms that predicted from the logs which permissions must be assigned to which users. We refer to these logs as ABAC1, ABAC2, ABAC3, and ABAC4.

Mine business-meaningful RBAC policies from access control matrices We use the access control matrix provided by Amazon for the IEEE MLSP 2012 competition [72], available at the UCI machine learning repository [93]. It assigns three types of permissions, named "HOST", "PERM_GROUP", and "SYSTEM_GROUP". The number of permissions for each type are approximately 1,700, 6,000, and 20,000, respectively. For each type of permission, we created an access control matrix containing more than 5,000 users. We explain in detail how we create these matrices in Appendix C.5.1. We refer to these matrices as BM-RBAC1, BM-RBAC2, and BM-RBAC3.

Mine XACML policies from access control matrices We use Continue [87, 55], the most complex set of XACML policies to our knowledge in the literature. We use seven of the largest policies in the set. For each of them, we compute the set of all possible requests and decide which of them are authorized by the policy. We then mine a policy from this set of decided requests. For the simplest policy, there are around 60 requests and for the most complex policy, there are more than 30,000 requests. We call these seven sets of requests XACML1, XACML2, ..., XACML7.

Mine spatio-temporal RBAC policies from logs There are no publicly available datasets for mining spatio-temporal RBAC policies. Based on policies provided as examples in recent works [18, 19], we created a synthetic policy and a synthetic log by creating 1,000 access requests uniformly at random and evaluating them against the policy. We refer to this log as STARBAC. The synthetic policy is in Appendix C.5.2.

5.10.2 Methodology

For RBAC and ABAC, we mine two policies in the corresponding policy language's syntax. The first one using a miner built according to Unicorn and the second one using a state-of-the-art miner. For RBAC, we use the miner presented in [57] and, for ABAC, we use the miner from [39]. For the XACML and spatio-temporal RBAC categories, there are no other known miners, so we only mine one policy in these categories using our miner. For business meaningful RBAC, we contacted the authors of miners for this RBAC extension [57, 103], but implementations of their algorithms were not available.

The objective function we use to mine ABAC policies from logs is the one defined in Section 5.7.5. For all other policy languages, we use $\lambda_1 \|\mathfrak{I}\| + \ell\,(Auth, \mathfrak{I}; \varphi)$, where λ_1 is a hyper-parameter, $\|\mathfrak{I}\|$ was the complexity measure defined for each policy language in Section 5.7, φ is the template formula for the corresponding policy language, and $\ell\,(Auth, \mathfrak{I}; \varphi)$ was defined in Section 5.7.3. The values for the hyper-parameters were computed using grid search (Appendix C.1).

To evaluate miners for RBAC, BM-RBAC, and XACML, we use 5-fold cross-validation [47, 135, 42]. We refer to Appendix 4.1.4 for an overview of cross-validation. To measure the mined policy's generalizability, we measure its *true positive rate* (TPR) and its *false positive rate* (FPR) [111]. To measure a mined policy's complexity, we use $\|\mathfrak{I}\|$. To evaluate miners for ABAC and STARBAC, which receive a log instead of an access control matrix as input, we use universal cross-validation (Chapter 4). We measure the mined policy's TPR, FPR, precision, and complexity.

All policy miners, except the one for BM-RBAC, were developed in Python 3.6 and were executed on machines with 2,8 GHz 8-core CPUs and 32 GB of RAM. The miner for BM-RBAC was developed in Pytorch version 0.4 [109] and executed on an NVIDIA GTX Titan X GPU with 12 GB of RAM. For all policy languages except STARBAC, our experiments finished within 4 hours. For STARBAC, they took 7 hours. We remark that organizations do not need to mine policies on a regular basis, so policies need not be mined in real time [39].

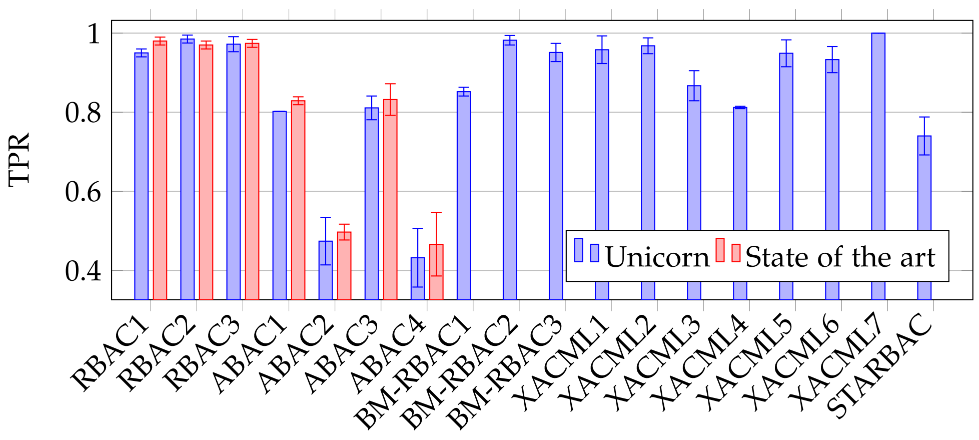


Figure 5.2: Comparison of the TPRs between policies mined using Unicorn and policies mined by a state-of-the-art policy miner across different policy languages. Policies with *higher* TPRs are better at granting permissions to the correct users.

5.10.3 Results

Figures 5.2–5.4 compare, respectively, the TPRs, complexities, and precisions of the policies we mined with those mined by an available state-of-the-art policy miner across the different datasets with respect to the different policy languages. We make the following observations.

- We mine policies whose TPR is within 5% of the state-of-the-art policies' TPR. For the XACML and STARBAC scenarios, where no other miners exist, we mine policies with a TPR above 80% in most cases.
- In most cases, we mine policies with a complexity lower than the complexity of policies mined by the state of the art.
- When mining from logs, we mine policies that have a similar or greater precision than those mined by the state of the art, sometimes substantially greater.
- In all cases, we mine policies with an FPR $\leq$ 5% (not shown in the figures).

5.10.4 Discussion

Our experimental results show that, with the exception of ABAC, all policies we mined attain a TPR of at least 80% in most of the cases. The low TPR in ABAC is due to the fact that the logs contain only 7% of all possible requests [39]. But even in that case, the ABAC miner we built attains a TPR that is within 5% of the TPR attained by the state of the art [39]. Moreover,

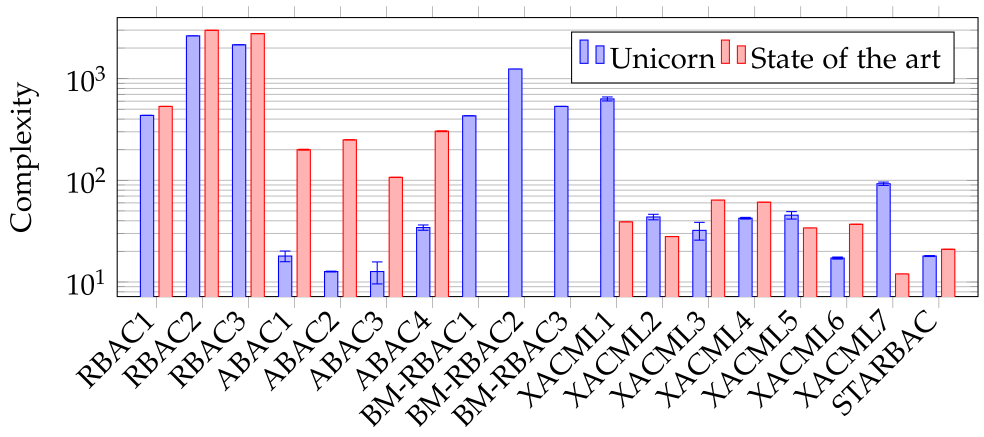


Figure 5.3: Comparison of the complexities between policies mined using Unicorn and policies mined by a state-of-the-art policy miner across different policy languages. Policies with *lower* complexities are better as they are easier to interpret by humans. For XACML and STARBAC, there is no known miner, but we compared the mined policy's complexity with that of the original policy.

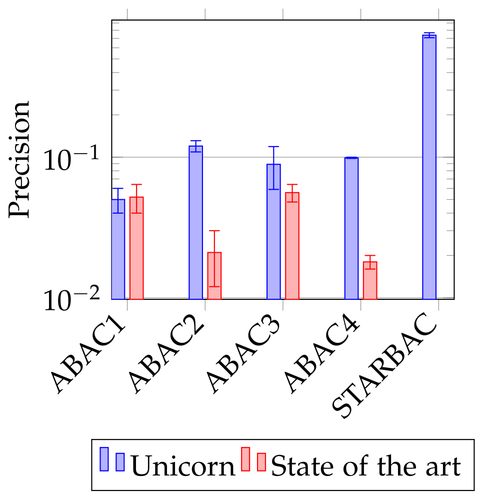


Figure 5.4: Comparison of the precision between policies mined using Unicorn and policies mined by a state-of-the-art policy miner across different policy languages. Policies with *higher* precision are better as they avoid incorrect authorizations. We only compare the precision of mined policies when mining from logs, as discussed in Chapter 4.

our ABAC miner mines policies with substantially smaller size and higher precision. These results provide strongly support our hypothesis that by using Unicorn we can build competitive policy miners for a wide variety of policy languages.

5.11 Related work

We recount the areas where policy mining research has focused in the last years and discuss the techniques that have been proposed to mine policies in different policy languages.

5.11.1 Policy mining

RBAC mining

Early research on policy mining focused on RBAC [49, 88, 128]. The approaches developed used combinatorial algorithms to find, for an assignment of permissions to users, an approximately minimal set of role assignments, e.g., [127, 95, 116, 129, 136]. A major step forward was the development of probabilistic models to model the assignment of permissions to roles and the assignment of roles to users. These works employ machine-learning techniques like latent Dirichlet allocation [103] and deterministic annealing [57, 123] to find the model parameters that maximize the likelihood of the given assignment of permission to users. More recent works mine RBAC policies with time constraints [97, 98] and role hierarchies [121, 65], using combinatorial techniques that are specific to the RBAC extension.

Despite the plethora of RBAC miners, there are still many RBAC extensions for which no miner has been developed. A recent survey in role mining [99], covering over a dozen RBAC miners, reports not a single RBAC miner that can mine spatio-temporal constraints, even though there have been several spatio-temporal extensions of RBAC since 2000, e.g., [113, 34, 90, 125, 31, 6, 41], and additional extensions are under way [18, 19]. Unicorn offers a practical solution for mining RBAC policies for these extensions. As we illustrated in Section 5.8, we can now mine spatio-temporal RBAC policies.

Other miners

Miners have recently been proposed for other policy languages like ABAC [132, 39] and ReBAC (Relationship-Based Access Control) [30]. These algorithms use dedicated combinatorial and machine-learning methods, but mine policies tailored to the given policy language. Unicorn has the advantage of being applicable to a much broader class of policy languages.

5.11.2 Interpretable machine learning

Machine-learning algorithms have been proposed for classification that train an *interpretable* model [84, 80, 120, 36, 9] consisting of a set of human-readable rules that describe how an instance is classified. Such algorithms are attractive for policy mining, as policies must not only correctly grant and deny access, they should also be easy to understand.

The main limitation of the rules mined by these models is that they often do not comply with the underlying policy language's syntax. State-of-the-art algorithms in this field [120, 36, 9] produce rules that are simply conjunctions of constraints on the instances' features. This is insufficient for many policy languages, like XACML, where policies can consist of nested subpolicies that are composed with XACML's policy combination algorithms [64]. Moreover, the rules produced by these algorithms are often unnecessarily long and complex [39].

The main advantage of Unicorn is that it can mine policies that not only correctly grant and deny access in most cases, but are also compliant with a given policy language's syntax, like XACML's (Appendix 5.9). Moreover, as illustrated in Sections 5.7.2 and 5.7.5, one can tailor the objective function so that the policy miner searches for a simple policy.

5.12 Conclusion

The difficulty of specifying and maintaining access control policies has spawned a large and growing number of policy languages with associated policy miners. However, developing such miners is challenging and substantially more difficult than creating a new policy language. This problem is exacerbated by the fact that existing mining algorithms are inflexible in that they cannot be easily modified to mine policies for other policy languages with different features. In this paper, we demonstrated that it is in fact possible to create a universal method for building policy miners that works very well for a wide variety of policy languages.

We validated the effectiveness of Unicorn experimentally, including a comparison against state-of-the-art policy miners for different policy languages. In all cases, the miners built using Unicorn are competitive with the state of the art.

As future work, we plan to automate completely the workflow in Figure 5.1. We envision a *universal policy mining algorithm* based on Algorithm 5 that, given as input the policy language, the permission assignment, and the objective function, automatically computes the probabilistic model and the most likely policy constrained by the given permission assignment. As a result, the designer of a policy miner would not need to do Tasks 1 or 3.

Chapter 6

Conclusion

Organizations define access control policies in order to prevent users from abusing their privileges and performing nefarious acts. In spite of a plethora of mechanisms for specifying and maintaining access control infrastructures, privilege abuse continues to be one of the main causes for data breaches and organizations are exhorted to ensure that only the necessary permissions are granted to users [50, 51]. In this thesis, we propose three solutions to prevent privilege abuse.

First, we propose FORBAC, a new extension for RBAC that strikes a balance between expressiveness in policy specification and complexity in policy analysis. FORBAC offers a new framework for policy specification and analysis by which policy administrators can analyze RBAC policies. By using policy analysis, the policy administrator can ensure that users have the permissions they need and that only those authorized users get sensitive permissions. We designed FORBAC so that it could express the complex policies that current organizations need to specify, while at the same time keeping the complexity of policy analysis within NP. We also argued why this is a natural complexity class for policy analysis.

Second, we propose Rhapsody, an algorithm for mining ABAC policies from sparse logs. Rhapsody is a policy miner that analyzes how users exercise permissions in the organization and uses that information to mine an adequate policy. In the context of preventing privilege abuse, policy mining helps to compute a policy that better reflects the users' needs within the organization. Rhapsody's main advantage over other ABAC miners is that it guarantees to mine exactly all significant, reliable, and succinct rules from any given sparse log. Such guarantees ensure that mined policies prevent privilege abuse by assigning permissions to users only if there is substantial evidence in the log.

Finally, we propose Unicorn, a universal method for building policy miners that depends only on the policy language and an objective function evaluating policy quality. Unicorn's main advantages are that it streamlines the process of designing policy miners and removes the need for knowledge of machine-learning. Using Unicorn, we developed competitive policy miners for a wide variety of policy languages. In the case of ABAC, the policy miner built by Unicorn outperformed Rhapsody.

The research on these works have shown some challenges for future work.

Including constraints and role hierarchies in FORBAC. FORBAC cannot specify separation-of-duty or cardinality constraints [52]. More generally, there is no standard way to specify such concepts in the presence of role templates. For example, it is unclear how a role instance's attribute values from one role template R_1 affect another role instance's attribute values from another role template R_2 that is subsumed by R_1 in a role hierarchy. Cardinality constraints can also be difficult to specify. For example, just limiting the number of role instances may not be enough, as role instances with exactly the same attribute values or where one instance's attribute values are just a subset of the other's should not count as different instances. A case study on organizations using role templates could help to understand the requirements for such concepts.

Mining RBAC policies with role hierarchies. We do not know yet of any formalization of RBAC with role hierarchies in first-order logic that can be a template formula for that policy language. Finding such a formula or extending Unicorn's approach so that it can mine RBAC policies with role hierarchies would help to provide simpler RBAC policies.

A universal policy miner. Is it possible to extend Unicorn so that Tasks 1 and 3 are done automatically? Ideally, this universal policy miner would receive as input a policy language, specified as a BNF grammar or some other standard language specification, and an objective function and it would directly output a policy miner. As a result, the policy administrator would be relieved of the work of finding a template formula for a policy language and then implementing the pseudo-code of Algorithm 5.

Appendix A

Appendix for FORBAC

A.1 Complexity results for F

Theorem 86 For a policy $T \equiv \varphi_1(u,p) \lor \ldots \lor \varphi_\ell(u,p)$ in F, deciding whether the formula $\forall u : \textsf{USERS} \,.\, \forall p : \textsf{PERMS} \,.\, (\varphi_1(u,p) \lor \ldots \lor \varphi_\ell(u,p))$ is valid is NP-hard.

Proof. We reduce this problem to the validity problem for propositional Boolean formulas in disjunctive normal form. Let $\psi = \bigvee_{i \le M} \bigwedge_{j \le N} \ell_{ij}$ be a propositional Boolean formula in disjunctive normal form. For every propositional variable y occurring in ψ, pick a binary relation symbol Q_y of type $\textsf{USERS} \times \textsf{PERMS}$. For $i \le M$ and $j \le N$, let $L_{ij}(u,p)$ be the following formula:

$$L_{ij}(u,p) = \begin{cases} Q_y(u,p) & \text{if } \ell_{ij} = y \\ \neg Q_y(u,p), & \text{if } \ell_{ij} = \neg y. \end{cases} \tag{A.1}$$

It is easy to check that ψ is valid iff the following formula is valid.

$$\forall u : \textsf{USERS} \,.\, \forall p : \textsf{PERMS} \,.\, \left(\bigvee_{i \le M} \bigwedge_{j \le N} L_{ij}(u,p) \right). \tag{A.2}$$

□

We now explain how to translate a policy T in F to a Margrave [106] policy T'. Define a Margrave formula $\mathit{Permit}(x)$ with a free variable x and one Margrave formula $Q'(x)$ for every first-order formula of the form $Q(u,p)$ occurring in T. For every formula $\bigwedge_{i \le M} P_i(u,p) \land \bigwedge_{j \le N} \neg Q_j(u,p)$ in T, define the following Margrave rule in T':

$$\mathit{Permit}(x) :- \begin{array}{l} P'_1(x), P'_2(x), \ldots, P'_M(x), \\ \neg Q'_1(x), \neg Q'_2(x), \ldots, \neg Q'_N(x). \end{array} \tag{A.3}$$

The rule says that those access requests that satisfy $P'_1, P'_2, \ldots, P'_M$ and not $Q'_1, Q'_2, \ldots, Q'_N$ are authorized. It is easy to check that an access request is authorized by T iff the access request is also authorized by T'. The same technique can be used to translate policies in F into policies in other popular and recent languages [10, 18, 19].

A.2 Complexity results for FORBAC

A.2.1 Complexity of deciding authorization in FORBAC

Theorem 87 Given a FORBAC-policy $(\Sigma, \mathcal{UA}, \mathcal{PA})$, a Σ-FORBAC-structure $\mathbb{K} = (\mathfrak{S}, \mathfrak{I})$, a user $\mathsf{u} \in \textbf{USERS}^{\mathfrak{S}}$, and a permission $\mathsf{p} \in \textbf{PERMS}^{\mathfrak{S}}$, deciding whether u is authorized for p takes time

$$O\left(\left|\textbf{ROLES}^{\mathfrak{S}}\right| \cdot |\mathbb{K}|^2 \cdot (|\mathcal{UA}| + |\mathcal{PA}|)\right),$$

where $\textbf{ROLES}^{\mathfrak{S}}$ is the set of role instances in $\mathbb{K}$, $|\mathbb{K}|$ is the length of $\mathbb{K}$ encoded as a string, $|\mathcal{UA}| = \sum_{\mathbf{R} \in RT(\Sigma)} |\mathcal{UA}_{\mathbf{R}}|$, and $|\mathcal{PA}| = \sum_{\mathbf{R} \in RT(\Sigma)} |\mathcal{PA}_{\mathbf{R}}|$..

Proof. The next algorithm checks if u is authorized for p:

1 For every $\mathsf{r} \in \textbf{ROLES}^{\mathfrak{S}}$, do the following.

a) Compute $[\![\mathcal{UA}_{\mathbf{R}}(u, r) \ \wedge \ \mathcal{PA}_{\mathbf{R}}(r, p)]\!]$, a propositional Boolean formula obtained from $\mathcal{UA}_{\mathbf{R}}(u, r) \ \wedge \ \mathcal{PA}_{\mathbf{R}}(r, p)$ by replacing every atomic formula φ with $\top$ or $\bot$, depending on whether $\mathbb{K}, \sigma_{\mathsf{u},\mathsf{p}} \vDash \varphi$ or not. Here, $\sigma_{\mathsf{u},\mathsf{p}}$ is the substitution mapping u and p to u and p, respectively.

b) If $[\![\mathcal{UA}_{\mathbf{R}}(u, r) \ \wedge \ \mathcal{PA}_{\mathbf{R}}(u, r)]\!]$ evaluates to true, then output that u is authorized for p; otherwise, try another $\mathsf{r} \in \textbf{ROLES}^{\mathfrak{S}}$.

2 If this fails with all $\mathsf{r} \in \textbf{ROLES}^{\mathfrak{S}}$, then output that u is not authorized for p.

Note that Steps 1b and 2 take $O(|\mathcal{UA}| + |\mathcal{PA}|)$ and constant time, respectively. It suffices to show then that Step 1a takes $O\left(|\mathbb{K}|^2 \cdot (|\mathcal{UA}| + |\mathcal{PA}|)\right)$ time. We can achieve this time by encoding $\mathbb{K}$ in a way that (i) checking the value of $f^{\mathfrak{I}}(\mathsf{e})$ for f a single-valued attribute and $\mathsf{e} \in \{\mathsf{u}, \mathsf{r}, \mathsf{p}\}$ takes $O(|\mathbb{K}|)$-time and (ii) checking whether $\mathsf{e} \in F^{\mathfrak{I}}(\mathsf{v})$ holds, for F a set-valued attribute, $\mathsf{e} \in \{\mathsf{u}, \mathsf{r}, \mathsf{p}\}$, and v an integer or a string, takes $O(|\mathbb{K}|)$-time. As a result, checking whether $\mathbb{K}, \sigma_{\mathsf{u},\mathsf{p}} \vDash \varphi$, for an atomic formula φ, takes $O\left(|\mathbb{K}|^2\right)$-time. □

A.2.2 Complexity of deciding satisfiability of an existential FORBAC-formula

We now prove Theorem 50: Deciding satisfiability of an existential FORBAC-formula is NP-complete.

We prove NP-hardness by reduction to the satisfiability problem for propositional Boolean formulas. Let ψ be a Boolean propositional formula. Define a FORBAC-signature that contains a single-valued attribute f_p of type $\textsf{\textbf{USERS}} \to \textsf{\textbf{INT}}$, for each proposition p in ψ. Let $\exists u.\, \psi'(u)$ be the existential FORBAC-formula where $\psi'(u)$ is obtained from ψ by replacing every proposition p in ψ with $f_p(u) = 1$. Note that $\exists u.\, \psi'(u)$ is satisfiable iff ψ is satisfiable. Hence, satisfiability of existential FORBAC-formulas is NP-hard.

We now prove that satisfiability of existential FORBAC-formulas is in NP. Let Σ be a FORBAC-signature and

$$\Phi := \exists x_1 : \textsf{\textbf{S}}_1\, \exists x_2 : \textsf{\textbf{S}}_2 \ldots \exists x_k : \textsf{\textbf{S}}_k\, .\, \varphi(x_1, x_2, \ldots, x_k) \tag{A.4}$$

be an existential FORBAC-formula over Σ. We assume that all functions map to integers. The proof for the general case is analogous. A *certificate for* Φ is a function $\mathcal{C}$ mapping every single-valued term in Φ to an integer and every set-valued term in Φ to a set of integers. A *term* is any single or set-valued term. For a term t occurring in $\mathcal{C}$, let $[\![t]\!]_{\mathcal{C}} := \mathcal{C}(t)$. *$\mathcal{C}$ satisfies a formula ψ* if the expression that results from replacing every term t in ψ with $[\![t]\!]_{\mathcal{C}}$ evaluates to true. Note that Φ is satisfiable iff there is a certificate that satisfies Φ.

Notation used in the proof. We define the *size* of a certificate $\mathcal{C}$ for Φ. The *size* of an integer is the length of its natural representation as a string. The *size* of a set of integers is the sum of the sizes of its elements. The *size* of a certificate $\mathcal{C}$ for Φ is the sum of all the sizes of $[\![t]\!]_{\mathcal{C}}$, where t ranges over terms occurring in Φ.

From now on, we write t and t' to denote single-valued terms, c and c' to denote integer constants, T and T' to denote set-valued terms, C and C' to denote sets of integer constants, $f(e)$ to denote any single-valued term with f a single-valued attribute and e a variable, and $F(e)$ to denote any set-valued term with F a set-valued attribute and e a variable.

Goal of the proof. To show that the set of satisfiable existential FORBAC-formulas is in NP, it suffices to show that for a satisfiable existential FORBAC-formula Φ, there exists a certificate $\overline{\mathcal{C}}$ that satisfies Φ and is *small*. A certificate is small if its size is at most $n^2 \log(d + n)$, where d is the size of the largest integer constant occurring in Φ and n is the length of Φ. Observe that $d \leq n$.

If Φ is satisfiable, then there exists a certificate $\mathcal{C}$ that satisfies Φ; however $\mathcal{C}$ does not need to have a size polynomial in the length of Φ for two reasons. First, the size of $[\![f(e)]\!]_{\mathcal{C}}$ might be large. Second, the set $[\![F(e)]\!]_{\mathcal{C}}$ might have a large number of elements or some element in $[\![F(e)]\!]_{\mathcal{C}}$ might have a large size. The rest of the proof shows how to build a small certificate $\overline{\mathcal{C}}$ from $\mathcal{C}$.

Auxiliary set. We start by building an auxiliary set σ having the following property: any certificate that satisfies all formulas in σ also satisfies Φ. This set is built as follows. For every atomic formula ψ occurring in Φ, put ψ in σ, if $\mathcal{C}$ satisfies ψ, and put $\neg\psi$ in σ, otherwise.

We now reduce the variety of types of formulas in σ by rewriting some of them as follows:

1 $f(e) \leq t$: Replace $f(e) \leq t$ with $f(e) = t$, if $[\![f(e)]\!]_{\mathcal{C}} = [\![t]\!]_{\mathcal{C}}$, and with $f(e) < t$, otherwise.

2 $t \leq f(e)$: Proceed analogously.

3 $f(e) = t$: Replace every occurrence of $f(e)$ in σ with t. After $\overline{\mathcal{C}}$ is built, define $[\![f(e)]\!]_{\overline{\mathcal{C}}}$ as $[\![t]\!]_{\overline{\mathcal{C}}}$.

4 $f(e) \neq t$: If $[\![f(e)]\!]_{\mathcal{C}} < c$, then replace every occurrence of this formula with $f(e) < t$. Otherwise, replace it with $f(e) > t$.

5 $\mathsf{c} \sim \mathsf{c}'$ with $\sim \in \{=, \leq, >, \neq\}$: Remove $\mathsf{c} \sim \mathsf{c}'$ from σ.

6 $\mathsf{C}_1 \sim \mathsf{C}_2$, with $\sim \in \{=, \subseteq, \neq, \not\subseteq\}$: Remove the formula from σ.

7 $\mathsf{c} \in \mathsf{C}$ or $\mathsf{c} \notin \mathsf{C}$: Remove the formula from σ.

8 $F(e) = T$: Replace every occurrence of $F(e)$ in σ with T. After $\overline{\mathcal{C}}$ is built, define $[\![F(e)]\!]_{\overline{\mathcal{C}}}$ as $[\![T]\!]_{\overline{\mathcal{C}}}$.

9 $F(e) \neq T$: Replace $F(e) \neq T$ in σ with $F(e) \not\subseteq T$ if $[\![F(e)]\!]_{\mathcal{C}} \not\subseteq [\![T]\!]_{\mathcal{C}}$ and $T \not\subseteq F(e)$ otherwise.

10 $\mathsf{c} \in F(e)$: Replace this formula in σ with $\{\mathsf{c}\} \subseteq F(e)$.

11 $\mathsf{c} \notin F(e)$: Replace this formula in σ with $F(e) \not\subseteq \mathbb{Z} \setminus \{\mathsf{c}\}$.

12 $f(e) \notin \mathsf{C}$: Replace $f(e) \notin \mathsf{C}$ in σ with $f(e) \in \mathbb{Z} \setminus \mathsf{C}$.

13 $T \not\subseteq T'$: Take a new unary function symbol f and a variable e and replace $T \not\subseteq T'$ with the two formulas $f(e) \in T$ and $f(e) \notin T'$. Afterwards, define $[\![f(e)]\!]_{\mathcal{C}}$ as a number that belongs to $[\![T]\!]_{\mathcal{C}}$ but not to $[\![T']\!]_{\mathcal{C}}$.

After this, every formula in σ has one of the following forms: $\mathsf{c} < f(e)$, $f(e) < \mathsf{c}$, $f(e) < f'(e')$, $\mathsf{C} \subseteq F(e)$, $F(e) \subseteq \mathsf{C}$, $F(e) \subseteq F'(e')$, $f(e) \in F(e)$, $f(e) \notin F(e)$, and $f(e) \in \mathsf{C}$.

We sometimes write $[\psi] \in \sigma$ instead of $\psi \in \sigma$ to prevent awkward notation like $f(e) \in F(e) \in \sigma$. For two set-valued terms T and T', we say that $T \subseteq_\sigma T'$ if $[T \subseteq T'] \in \sigma$. We denote with $\subseteq_\sigma^*$ the reflexive-transitive closure of $\subseteq_\sigma$.

Note that any certificate that satisfies all formulas in σ will also satisfy Φ. In particular, $\mathcal{C}$ satisfies all formulas in σ.

Sketch of the proof. Let $f_1(e_1), \ldots, f_m(e_m)$ be all the non-constant single-valued terms that occur in Φ. We suppose that they are enumerated in a way that $[\![f_j(e_j)]\!]_\mathcal{C} \leq [\![f_{j+1}(e_{j+1})]\!]_\mathcal{C}$, for $j < m$. We build from $\mathcal{C}$ a sequence of certificates $\mathcal{C} = \mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_m$ that satisfies the following invariant: for $\mathcal{C}_j$ and $i < j$, $[\![f_i(e_i)]\!]_{\mathcal{C}_j} \leq [\![f_j(e_j)]\!]_{\mathcal{C}_j}$ and $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \leq j + d$, where d is the largest constant that occurs in Φ. Therefore, the size of $[\![f_i(e_i)]\!]_{\mathcal{C}_j}$ is at most $\log(j + d) \leq \log(n + d)$, which is polynomial in the length of Φ. After that, we build from $\mathcal{C}_m$ a certificate $\overline{\mathcal{C}}$ that satisfies Φ and where for every set-valued term $F(e)$ the size of $[\![F(e)]\!]_\mathcal{C}$ is also polynomial in the length of Φ.

Construction of $\mathcal{C}_1$, $\mathcal{C}_2, \ldots,$ $\mathcal{C}_m$. For j such that $0 < j \leq m$, suppose that $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{j-1}$ are already built. We explain how to build $\mathcal{C}_j$. For any set-valued term $F(e)$, let $[\![F(e)]\!]_{\mathcal{C}_j} := [\![F(e)]\!]_\mathcal{C}$. For $i < j$, let $[\![f_i(e_i)]\!]_{\mathcal{C}_j} := [\![f_i(e_i)]\!]_{\mathcal{C}_{j-1}}$ and for $i > j$, let $[\![f_i(e_i)]\!]_{\mathcal{C}_j} := [\![f_i(e_i)]\!]_\mathcal{C}$. Next, we compute a value for $[\![f_j(e_j)]\!]_{\mathcal{C}_j}$ small enough that fits all the constraints in σ that $f_j(e_j)$ must fulfill. First, we compute the set A_j of feasible values for $[\![f_j(e_j)]\!]_{\mathcal{C}_j}$. Let A_j be the set of integers a such that

1 $[\![f_{j-1}(e_{j-1})]\!]_{\mathcal{C}_{j-1}} \leq \mathsf{a}$, if $j > 1$.

2 $\mathsf{c} < \mathsf{a}$, for any integer constant c such that $[\mathsf{c} < f_j(e_j)] \in \sigma$.

3 $\mathsf{a} \in \mathsf{C}$, for any set of constants C and any set-valued term $F(e)$ such that $[f_j(e_j) \in F(e)] \in \sigma$ and $F(e) \subseteq_\sigma^* \mathsf{C}$.

4 $\mathsf{a} \notin \mathsf{C}$, for any set of constants C and any set-valued term $F(e)$ such that $[f_j(e_j) \notin F(e)] \in \sigma$ and $\mathsf{C} \subseteq_\sigma^* F(e)$.

5 $\mathsf{a} \neq [\![f_i(e_i)]\!]_{\mathcal{C}_{j-1}}$, for any single-valued term $f_i(e_i)$ with $i < j$ and any set-valued term T such that

   a) $[f_i(e_i) \in T'] \in \sigma$, $[f_j(e_j) \notin T] \in \sigma$ and $T' \subseteq_\sigma^* T$, or

   b) $[f_i(e_i) \notin T] \in \sigma$, $[f_j(e_j) \in T'] \in \sigma$ and $T' \subseteq_\sigma^* T$.

Note that A_j is an intersection of sets of integer intervals, one set for every item above. Furthermore, an extreme point in any of those intervals is either $[\![f_i(e_i)]\!]_{\mathcal{C}_{j-1}}$, for some $i < j$, or an integer constant occurring in σ. Therefore, an extreme point in any interval of A_j is at most one plus the maximum among $[\![f_1(e_1)]\!]_{\mathcal{C}_{j-1}}, \ldots, [\![f_{j-1}(e_{j_1})]\!]_{\mathcal{C}_{j-1}}$ and d, the largest integer constant occurring in σ. By our invariant, $[\![f_i(e_i)]\!]_{\mathcal{C}_{j-1}} \leq d + j - 1$, for any $i < j$. Therefore, an extreme point in any interval of A_j is at most $d + j$.

Let $f_j(e_j)$ be the minimum value of A_j. Note that (i) A_j is not empty, as $[\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}} \in A_j$ and (ii), for $j > 1$, A_j has a minimum, as it is bounded below by $[\![f_{j-1}(e_{j-1})]\!]_{\mathcal{C}_{j-1}}$. A_j might be unbounded below when $j = 1$. In that case, let $[\![f_1(e_1)]\!]_{\mathcal{C}_1}$ be an extreme point of A_1 and, if A_1 is the entire set of integers, then let it be 0.

Recall that $[\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}} \in A_j$. Therefore, $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \leq [\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}} \leq j + d$. Moreover, by Item 1, $[\![f_i\ (e_i)]\!]_{\mathcal{C}_j} \leq [\![f_j\ (e_j)]\!]_{\mathcal{C}_j}$, for any $i < j$.

Correctness of construction of $\mathcal{C}_m$. Let σ_S be the subset of σ that contains all atomic formulas where a set-valued term of the form $F(e)$ occurs.

Lemma 88 If $\mathcal{C}$ satisfies all formulas in σ, then $\mathcal{C}_m$ satisfies all formulas in $\sigma \setminus \sigma_S$.

Proof. Suppose that $\mathcal{C}$ satisfies all formulas in σ. It suffices to show by induction on $j \leq m$ that $\mathcal{C}_j$ satisfies all formulas in σ. The base case is obvious, since $\mathcal{C}_0 = \mathcal{C}$ by definition, so suppose that $\mathcal{C}_j$ satisfies all formulas in σ. We show that $\mathcal{C}_{j+1}$ satisfies all formulas in σ. $\mathcal{C}_j$ and $\mathcal{C}_{j+1}$ differ only on the interpretation of $f_j(e_j)$, so it suffices to show that $\mathcal{C}_{j+1}$ satisfies any atomic formula $\psi \in \sigma$ that involves $f_j(e_j)$. We evaluate the possible cases for ψ:

- $\mathsf{c} < f_j(e_j)$. Since $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \in A_j$, we have by Item 2 of the definition of A_j that $\mathsf{c} < [\![f_j(e_j)]\!]_{\mathcal{C}_j}$.
- $f_j(e_j) < \mathsf{c}$. Recall that $[\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}} \in A_j$. Therefore, $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \leq [\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}}$. Also, note that $[\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}} < \mathsf{c}$ as, by the induction hypothesis, $\mathcal{C}_{j-1}$ satisfies all formulas in σ. Therefore, $[\![f_j(e_j)]\!]_{\mathcal{C}_j} < \mathsf{c}$.
- $f_j(e_j) < f'(e')$. First, recall that $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \leq [\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}}$. Second, note that $[\![f_j(e_j)]\!]_{\mathcal{C}_{j-1}} < [\![f'(e')]\!]_{\mathcal{C}_{j-1}}$ as, by the induction hypothesis, $\mathcal{C}_{j-1}$ satisfies all formulas in σ. Finally, observe that $f'(e')$ must be different from $f_j\left(e_j\right)$, so $[\![f'(e')]\!]_{\mathcal{C}_{j-1}} = [\![f'(e')]\!]_{\mathcal{C}_j}$. From these three observations, we conclude that $[\![f_j(e_j)]\!]_{\mathcal{C}_j} < [\![f'(e')]\!]_{\mathcal{C}_j}$.
- $f'(e') < f_j(e_j)$. By assumption, $[\![f'(e')]\!]_{\mathcal{C}} \leq [\![f_j(e_j)]\!]_{\mathcal{C}}$. Since we assume that $[\![f_\ell(e_\ell)]\!]_{\mathcal{C}} \leq [\![f_k(e_k)]\!]_{\mathcal{C}}$, for any ℓ and k such that $\ell < k$, we must have that, for some $i < j$, $f'(e') \equiv f_i(e_i)$. By our invariant, we have that $[\![f_i(e_i)]\!]_{\mathcal{C}_j} < [\![f_j(e_j)]\!]_{\mathcal{C}_j}$. Therefore, $[\![f'(e')]\!]_{\mathcal{C}_j} < [\![f_j(e_j)]\!]_{\mathcal{C}_j}$.
- $f_j(e_j) \in F(e)$ and $f_j(e_j) \notin F(e)$. Formulas of this form do not belong to $\sigma \setminus \sigma_S$.

- $f(e) \in \mathsf{C}$. Since $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \in A_j$, we have, by Item 3 of the definition of A_j, that $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \in \mathsf{C}$.

□

Once $\mathcal{C}_m$ is built, $[\![f_i(e_i)]\!]_{\mathcal{C}_m} \leq d + m \leq d + n$, for $i \leq m$. Recall that d is the largest constant occurring in Φ. Therefore, the size of $[\![f_i(e_i)]\!]_{\mathcal{C}_m}$, for $i \leq m$, is at most $\log(d + n)$.

Construction of $\overline{\mathcal{C}}$. We now address the problem that, for a set-valued term of the form $F(e)$, the size of $[\![F(e)]\!]_{\mathcal{C}_m}$ might be very large. We define a final certificate $\overline{\mathcal{C}}$ where $[\![f(e)]\!]_{\overline{\mathcal{C}}} := [\![f(e)]\!]_{\mathcal{C}_m}$ and $[\![F(e)]\!]_{\overline{\mathcal{C}}}$ is the smallest set that satisfies the following:

1 It contains the union of all sets of constants C such that $\mathsf{C} \subseteq^*_\sigma F(e)$.

2 It contains every $[\![f(e)]\!]_{\overline{\mathcal{C}}}$ such that there is a formula of the form $f(e) \in F'(e')$ in σ and $F'(e') \subseteq^*_\sigma F(e)$.

Note that there is at least one set that satisfies this: $[\![F(e)]\!]_{\mathcal{C}_m}$. Therefore, $[\![F(e)]\!]_{\overline{\mathcal{C}}} \subseteq [\![F(e)]\!]_{\mathcal{C}_m}$.

Correctness of construction of $\overline{\mathcal{C}}$.

Lemma 89 If $\mathcal{C}_m$ satisfies all formulas in $\sigma \setminus \sigma_S$, then $\overline{\mathcal{C}}$ satisfies all formulas in σ.

Proof. Suppose that $\mathcal{C}_m$ satisfies all formulas in $\sigma \setminus \sigma_S$. Note that $\mathcal{C}_m$ and $\overline{\mathcal{C}}$ agree on the interpretation of single-valued terms. Therefore, it suffices to show that $\overline{\mathcal{C}}$ satisfies all atomic formulas $\psi \in \sigma$ that involve set-valued terms. We proceed by evaluating the possible cases for ψ:

- $\mathsf{C} \subseteq F(e)$. By construction, $\mathsf{C} \subseteq [\![F(e)]\!]_{\overline{\mathcal{C}}}$.
- $F(e) \subseteq \mathsf{C}$. Recall that $[\![F(e)]\!]_{\overline{\mathcal{C}}} \subseteq [\![F(e)]\!]_{\mathcal{C}_m}$ and that $\mathcal{C}_m$ interprets set-valued terms in the same way $\mathcal{C}$ does. Therefore $[\![F(e)]\!]_{\overline{\mathcal{C}}} \subseteq [\![F(e)]\!]_{\mathcal{C}}$. Since $\mathcal{C}$ satisfies all formulas in σ, we have $[\![F(e)]\!]_{\mathcal{C}} \subseteq \mathsf{C}$. Therefore $[\![F(e)]\!]_{\overline{\mathcal{C}}} \subseteq \mathsf{C}$.
- $F(e) \subseteq F'(e')$. The same argument holds. Recall that $[\![F(e)]\!]_{\overline{\mathcal{C}}} \subseteq [\![F(e)]\!]_{\mathcal{C}_m}$ and that $\mathcal{C}_m$ interprets set-valued terms in the same way $\mathcal{C}$ does. Therefore $[\![F(e)]\!]_{\overline{\mathcal{C}}} \subseteq [\![F(e)]\!]_{\mathcal{C}}$. Since $\mathcal{C}$ satisfies all formulas in σ, we have $[\![F(e)]\!]_{\mathcal{C}} \subseteq [\![F'(e')]\!]_{\mathcal{C}}$. Therefore $[\![F(e)]\!]_{\overline{\mathcal{C}}} \subseteq [\![F'(e')]\!]_{\mathcal{C}}$.
- $f(e) \in F(e)$. By construction, $[\![f(e)]\!]_{\overline{\mathcal{C}}} \in [\![F(e)]\!]_{\overline{\mathcal{C}}}$.
- $f_j(e_j) \notin F(e)$, with $j \leq m$. By the definition of $[\![F(e)]\!]_{\overline{\mathcal{C}}}$, it suffices to show that

1 $[\![f_j(e_j)]\!]_{\overline{\mathcal{C}}} \notin \mathsf{C}$, for any set of integer constants C such that $\mathsf{C} \subseteq^*_\sigma F(e)$. Recall that $[\![f_j(e_j)]\!]_{\overline{\mathcal{C}}} = [\![f_j(e_j)]\!]_{\mathcal{C}_m} = [\![f_j(e_j)]\!]_{\mathcal{C}_j} \in A_j$, which implies, by Item 4 of the definition of A_j, that $[\![f_j(e_j)]\!]_{\overline{\mathcal{C}}} \notin \mathsf{C}$.

2 $[\![f_j(e_j)]\!]_{\overline{\mathcal{C}}} \neq [\![f_\ell(e_\ell)]\!]_{\overline{\mathcal{C}}}$ for any single-valued term $f_\ell(e_\ell)$ such that $[f_\ell(e_\ell) \in F'(e')] \in \sigma$ and $F'(e') \subseteq^*_\sigma F(e)$. We consider two subcases. If $\ell < j$, then since $[\![f_j(e_j)]\!]_{\overline{\mathcal{C}}} = [\![f_j(e_j)]\!]_{\mathcal{C}_j} \in A_j$, $[f_j(e_j) \notin F(e)] \in \sigma$, $[f_\ell(e_\ell) \in F'(e')] \in \sigma$ and $F'(e') \subseteq^*_\sigma F(e)$, by Item 5a of the definition of A_j, we have that $[\![f_j(e_j)]\!]_{\mathcal{C}_j} \neq [\![f_\ell(e_\ell)]\!]_{\mathcal{C}_j}$. If $\ell > j$, then since $[\![f_\ell(e_\ell)]\!]_{\overline{\mathcal{C}}} = [\![f_\ell(e_\ell)]\!]_{\mathcal{C}_\ell} \in A_\ell$, $[f_j(e_j) \notin F(e)] \in \sigma$, $[f_\ell(e_\ell) \in F'(e')] \in \sigma$ and $F'(e') \subseteq^*_\sigma F(e)$, by Item 5b of the definition of A_ℓ, we have that $[\![f_\ell(e_\ell)]\!]_{\mathcal{C}_\ell} \neq [\![f_j(e_j)]\!]_{\mathcal{C}_\ell}$.

We conclude then that $[\![f_j(e_j)]\!]_{\overline{\mathcal{C}}} \notin [\![F(e)]\!]_{\overline{\mathcal{C}}}$.

□

This concludes the proof of Theorem 50. We showed that the problem of deciding satisfiability of existential FORBAC-formulas is NP-hard by reduction of the Boolean satisfiability problem. Then we showed that this problem is in NP. For this, we assumed given an existential FORBAC-formula Φ and a certificate $\mathcal{C}$ for Φ and we explained how to compute a certificate $\overline{\mathcal{C}}$ whose size is polynomial in the length of Φ. $\overline{\mathcal{C}}$ was built from a sequence $\mathcal{C}_1, \ldots, \mathcal{C}_m$ of auxiliary certificates, which were in turn built from $\mathcal{C}$. Lemmas 88 and 89 ensure that $\overline{\mathcal{C}}$ is a certificate for Φ. Hence, deciding satisfiability of existential FORBAC-formulas is NP-complete.

Appendix B

Appendix for Rhapsody

B.1 ABAC instances used for experiments

Amazon 1. These are instances built from two access logs provided by Amazon in Kaggle, a platform for predictive modelling competitions [81, 82]. One log is for training and contains access requests made by Amazon's employees over two years [82]. Each entry in this log describes an employee's request to a resource and whether the request was authorized or not. The request contains all the employee's attribute values and the resource identifier. The second log is for evaluation. It contains access requests only, but it does not specify which requests are authorized. Participants in the Kaggle competition had to decide for the evaluation log which requests to authorize. The logs contain more than 12,000 users and 7,000 resources.

From the Amazon logs one can build an ABAC instance (U, P, A, D), where U and P are the set of users and the set of resources occurring in the logs, respectively, and $A \cup D$ are the requests occurring in the training log. However, such an instance is too large to fit in main memory and some of the implementations of competing ABAC mining algorithms (Rhapsody included) cannot handle such large amounts of data. To deal with this, we observe that the only permission attribute is the resource's identifier, so any ABAC rule authorizes requests for at most one resource. Therefore, any ABAC policy for (U, P, A, D) can be partitioned into several policies, each authorizing requests for only one resource. Hence, rather than mining over (U, P, A, D), we can mine over instances of the form $(U, \{\mathsf{p}\}, A_{\mathsf{p}}, D_{\mathsf{p}})$, where p is a single resource and $A_{\mathsf{p}} \cup D_{\mathsf{p}}$ are all requests for p occurring in the training log. These instances are much smaller and are easily handled by all competing ABAC mining algorithms.

For our experiments, we selected the five instances $(U, \{\mathsf{p}\}, A_{\mathsf{p}}, D_{\mathsf{p}})$ with the highest value for $|A_{\mathsf{p}} \cup D_{\mathsf{p}}|$. In all cases, $|A_{\mathsf{p}} \cup D_{\mathsf{p}}| \ / \ |U| < 0.07$. Hence, the log contains, for each resource, less than 7% of all possible requests.

Table B.1: Properties of the basic organization policies

	Instance				
	1	2	3	4	5
Num. jobs	10	10	10	20	20
Num. categories	5	10	20	5	10

Amazon 2. These are instances built from access data provided by Amazon in the UCI machine learning repository [93]. The data contains more than 36,000 users and 27,000 permissions. We took the eight most requested permissions and for each of them, we created an ABAC instance $(U, \{\mathsf{p}\}, A, D)$ where U is the set of all users in the access data, p is the permission, A is the set of users who requested p and were authorized, and D is the set of users who requested p and were denied.

Basic Organization. These are synthetic instances with only one user attribute value, *Job*, identifying the job the user performs and only one permission attribute value, *Category*, identifying the category where the permission belongs. We use natural numbers to identify jobs and categories. There is only one permission for each category and there are 100 users for each job. For each category, we assume that all jobs, except one, are authorized to request permissions for that category. For each category c, we denote by p_c the permission from that category and by j_c the job that is *not* authorized to request p_c. We let J and C denote the total number of jobs and categories, respectively. The values for J and C in each instance are described in Table B.1.

We now describe the log for each instance. For each category c and for each job $\mathsf{j} \neq \mathsf{j}_c$, the fraction of users with job j that have requested p_c is c/C. We use this to simulate a non-uniform distribution of the categories of the permissions requested by users. In addition, for each category c, there is only one user with job j_c that has requested access to p_c. This yields fewer denied requests than authorized requests in the log.

B.2 Proofs

Observation 1

Proof. We assume that $|[\![r]\!]_{U\times P}| \geq T$. For the case when $|[\![r]\!]_{U\times P}| \leq T$, the proof follows directly from Definitions 56 and 57.

($\Rightarrow$) Assume that $Rel_T(r) \geq K$ and let $\top$ be a trivially true rule. Observe that $\top \in F_T(r)$, the set of refinements of r that cover at least T requests. Therefore $Conf(r) = Conf(r \wedge \top) \geq Rel_T(r) \geq K$. Let now r' be a rule such that $|[\![r \wedge r']\!]_{U\times P}| \geq T$. Then $r' \in F_T(r)$, which implies that $Conf(r \wedge r') \geq Rel_T(r) \geq K$. By Definition 56, r is not overly permissive with respect to T and K.

($\Leftarrow$) Assume now that $Conf(r) \geq K$ and that r is not overly permissive with respect to T and K. Let $r' \in F_T(r)$. Then $r \wedge r'$ is a refinement of r with $|[\![r \wedge r']\!]_{U\times P}| \geq T$. Since r is not overly permissive, $Conf(r \wedge r') \geq K$. This means that $K \leq \min_{r' \in F_T(r)} Conf(r \wedge r') = Rel_T(r)$. □

Observation 2

Proof. Suppose that r_2 proves that $Rel_T(r_1) < K$. Then, $r_2 \in F_T(r_1)$, as defined in Definition 57. Therefore $Rel_T(r_1) < K$. □

Observation 3

Proof. ($\Rightarrow$) Let r_1 and r_2 be two equivalent rules. Then $[\![r_1]\!]_{U\times P} = [\![r_2]\!]_{U\times P}$. This implies that $n_{U\times P}(r_1) = |[\![r_1]\!]_{U\times P}| = |[\![r_2]\!]_{U\times P}| = n_{U\times P}(r_2)$. Moreover, $n_{U\times P}(r_1 \wedge r_2) = |[\![r_1 \wedge r_2]\!]_{U\times P}| = |[\![r_1]\!]_{U\times P} \cap [\![r_2]\!]_{U\times P}| = |[\![r_i]\!]_{U\times P}| = n_{U\times P}(r_i)$, for $i \in \{1,2\}$.

($\Leftarrow$) Suppose that $n_{U\times P}(r_1) = n_{U\times P}(r_2) = n_{U\times P}(r_1 \wedge r_2)$. Then, for any $i \in \{1,2\}$, $|[\![r_i]\!]_{U\times P}| = n_{U\times P}(r_i) = n_{U\times P}(r_1 \wedge r_2) = |[\![r_1]\!]_{U\times P} \cap [\![r_2]\!]_{U\times P}|$. Since $[\![r_1]\!]_{U\times P}$ and $[\![r_2]\!]_{U\times P}$ are finite, it follows that $[\![r_1]\!]_{U\times P} \subseteq [\![r_2]\!]_{U\times P}$ and $[\![r_2]\!]_{U\times P} \subseteq [\![r_1]\!]_{U\times P}$, which means that $[\![r_1]\!]_{U\times P} = [\![r_2]\!]_{U\times P}$. Therefore r_1 and r_2 are equivalent. □

Theorem 1

Proof. Let π be the policy output by Rhapsody. Observe that $\pi = FreqRules \setminus (UnrelRules \cup CoversDenied \cup Subsumed)$.

($\Rightarrow$) Let $r \in \pi$. Therefore, $r \in FreqRules$, $r \notin UnrelRules$, $r \notin CoversDenied$, and $r \notin Subsumed$.

(i) Since $r \in FreqRules$, $|[\![r]\!]_{U\times P}| \geq T$.

(ii) Since $r \notin CoversDenied$, r does not cover a denied request.

(iii) Since $r \notin UnrelRules$, there is no $r' \in FreqRules$ proving that $Rel_T(r) < K$. That is, $Conf(r \wedge r'') \geq K$, for any r'' such that $|[\![r \wedge r'']\!]_{U\times P}| \geq T$. Thus, by the definition of Rel_T, we have that $Rel_T(r) \geq K$.

(iv) Since $r \notin \mathit{Subsumed}$, there is no $r' \in \mathit{RelRules}$ both shorter than and equivalent to r. Hence there is no rule r' that is both shorter than and equivalent to r, with $\mathit{Rel}_T(r') \geq K$.

$(\Leftarrow)$ Let r be a rule such that $|[\![r]\!]_{U \times P}| \geq T$, $\mathit{Rel}_T(r) \geq K$, r covers no denied request, and for which there is no rule r' that is both shorter and equivalent to r, with $\mathit{Rel}_T(r') \geq K$. It suffices to show that, after Rhapsody finishes, $r \in \mathit{FreqRules}$, $r \notin \mathit{UnrelRules}$, $r \notin \mathit{CoversDenied}$, and $r \notin \mathit{Subsumed}$.

(i) If $|[\![r]\!]_{U \times P}| \geq T$, then the set consisting of all atoms occurring in r is found by APRIORI. As a consequence, $r \in \mathit{FreqRules}$.

(ii) Since r covers no denied request, then $r \notin \mathit{CoversDenied}$.

(iii) If $\mathit{Rel}_T(r) \geq K$, then, by the definition of Rel_T, no rule $r' \in \mathit{FreqRules}$ proves that $\mathit{Rel}_T(r) < K$. Therefore $r \notin \mathit{UnrelRules}$.

(iv) If there is no rule r' with $\mathit{Rel}_T(r') \geq K$ that is both shorter than and equivalent to r, then $r \notin \mathit{Subsumed}$.

Since $r \in \mathit{FreqRules}$, $r \notin \mathit{UnrelRules}$, $r \notin \mathit{CoversDenied}$, and $r \notin \mathit{Subsumed}$, we conclude that $r \in \pi$. $\square$

Appendix C

Appendix for Unicorn

C.1 Grid search for policy mining

Policy miners sometimes require input values for *hyper-parameters*. To determine the values that make the miners compute the best policies, we use *grid search* [118], which we briefly recall next. Grid search uses cross-validation, which we recall in Section 4.5.

Let $\alpha_1, \ldots, \alpha_N$ be the hyper-parameters of a policy miner. In grid search, for $i \leq N$, one defines a set D_i of *candidate values*. Then for each $(a_1, \ldots, a_N) \in D_1 \times \ldots \times D_N$, one executes cross-validation using $(a_1, \ldots, a_N)$ as values for the hyper-parameters. Let TPR $(a_1, \ldots, a_N)$ and FPR $(a_1, \ldots, a_N)$ be the TPR and FPR of the policy mined during that execution of cross-validation, respectively. Finally, one chooses the tuple $(a_1^*, \ldots, a_N^*)$ that maximizes TPR $(a_1, \ldots, a_N)$ subject to FPR $(a_1^*, \ldots, a_N^*) \leq c$. The threshold c for the FPR is arbitrary (we used 0.05) and can be adjusted. It defines a maximum bound of false positives that can be tolerated from a mined policy.

C.2 Optimal parameters for the mean-field approximating distribution

Variational inference establishes that the set of parameters $\left\{\widehat{\theta}_{\mathfrak{f}} \mid \mathfrak{f} \in \mathfrak{F}(\varphi)\right\}$ that makes q best approximate $h_{\mathcal{F}}$ is defined by the following equation:

$$\widehat{\theta}_{\mathfrak{f}}(\mathsf{b}) = \frac{\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log \mathbb{P}(Auth, \mathfrak{I})]\right)}{\sum_{\mathsf{b}' \in Range(\mathfrak{f})} \exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[\log \mathbb{P}(Auth, \mathfrak{I})]\right)}. \tag{C.1}$$

The derivation of this equation can be found in [23]. From here, we derive the following:

$$\widehat{\theta}_{\mathfrak{f}}(\mathsf{b}) = \frac{\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log\mathbb{P}(Auth, \mathfrak{I})]\right)}{\sum_{\mathsf{b}'\in Range(\mathfrak{f})}\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[\log\mathbb{P}(Auth, \mathfrak{I})]\right)} \tag{C.2}$$

$$= \frac{\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log\mathbb{P}(\mathfrak{I}\mid Auth) + \log\mathbb{P}(Auth)]\right)}{\sum_{\mathsf{b}'\in Range(\mathfrak{f})}\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[\log\mathbb{P}(\mathfrak{I}\mid Auth) + \log\mathbb{P}(Auth)]\right)} \tag{C.3}$$

$$= \frac{\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log\mathbb{P}(\mathfrak{I}\mid Auth)]\right)\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log\mathbb{P}(Auth)]\right)}{\sum_{\mathsf{b}'}\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[\log\mathbb{P}(\mathfrak{I}\mid Auth)]\right)\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[\log\mathbb{P}(Auth)]\right)}. \tag{C.4}$$

Observe that $\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log\mathbb{P}(Auth)] = \log\mathbb{P}(Auth)$ as $\log\mathbb{P}(Auth)$ does not involve $\mathfrak{f}$. Therefore,

$$\widehat{\theta}_{\mathfrak{f}}(\mathsf{b}) = \frac{\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log\mathbb{P}(\mathfrak{I}\mid Auth)]\right)\exp\left(\log\mathbb{P}(Auth)\right)}{\sum_{\mathsf{b}'\in Range(\mathfrak{f})}\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[\log\mathbb{P}(\mathfrak{I}\mid Auth)]\right)\exp\left(\log\mathbb{P}(Auth)\right)} \tag{C.5}$$

$$= \frac{\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\log\mathbb{P}(\mathfrak{I}\mid Auth)]\right)}{\sum_{\mathsf{b}'\in Range(\mathfrak{f})}\exp\left(\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[\log\mathbb{P}(\mathfrak{I}\mid Auth)]\right)} \tag{C.6}$$

$$= \frac{\exp\left(-\beta\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[L(Auth, \mathfrak{X}; \varphi)]\right)}{\sum_{\mathsf{b}'\in Range(\mathfrak{f})}\exp\left(-\beta\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}'}[L(Auth, \mathfrak{X}; \varphi)]\right)}. \tag{C.7}$$

C.3 Simplifying the computation of expectations

We prove here Lemma 79. We start with some auxiliary lemmas and definitions.

Lemma 2 Let $\mathfrak{f}$ and $\mathfrak{g}$ be facts, φ be a formula in $\mathcal{L}$, $(\mathsf{u}, \mathsf{p}) \in U \times P$, and $\{\psi_i\}_i \subseteq \mathcal{L}$ such that $\{\psi_i^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\}_i$ is a set of mutually independent random variables under the distribution q.

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}[\mathfrak{g}] = \begin{cases} \mathsf{b} & \text{if } \mathfrak{f} = \mathfrak{g} \\ \theta_{\mathfrak{g}} & \text{otherwise.} \end{cases} \tag{C.8}$$

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[(\neg\varphi)^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\right] = 1 - \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\right]. \tag{C.9}$$

$$\mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\left(\bigwedge_i \psi_i\right)^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\right] = \prod_i \mathbb{E}_{\mathfrak{f}\mapsto\mathsf{b}}\left[\psi_i^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\right]. \tag{C.10}$$

Proof. Observe that, for a Bernoulli random variable X, $\mathbb{E}[X] = \mathbb{P}(X = 1)$. Recall also that $\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y]$, whenever X and Y are mutually independent. With these observations and using standard probability laws, one can derive the equations above. □

Lemma 3 Let $\varphi \in \mathcal{L}$ and let $(\mathsf{u}, \mathsf{p}) \in U \times P$. If $\varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})$ is diverse, then $\mathbb{E}_{\mathfrak{f} \mapsto \mathsf{b}}\left[\varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\right]$ can be computed using only the equations from Lemma 79.

This lemma is proved by induction on φ and by recalling that any two different random facts are independent under the distribution q, which follows from the way that the distribution q is factorized.

Corollary 4

$$\mathbb{E}_{\mathfrak{f} \mapsto \mathsf{b}}[L(Auth, \mathfrak{X}; \varphi)] = \sum_{(\mathsf{u},\mathsf{p}) \in U \times P} \left| Auth(\mathsf{u}, \mathsf{p}) - \mathbb{E}_{\mathfrak{f} \mapsto \mathsf{b}}\left[\varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\right] \right|. \tag{C.11}$$

Proof. $L(Auth, \mathfrak{X}; \varphi)$ can be rewritten as follows:

$$\sum_{(\mathsf{u},\mathsf{p}) \in Auth} \left(1 - \varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p})\right) + \sum_{(\mathsf{u},\mathsf{p}) \in U \times P \setminus Auth} \varphi^{\mathfrak{X}}(\mathsf{u}, \mathsf{p}). \tag{C.12}$$

The result follows from the linearity of expectation. □

Lemma 79 follows from Lemma 3 and Corollary 4.

C.4 Modeling RBAC temporal constraints

C.4.1 Periodic expressions

We recall here the definition of temporal constraints and then explain how we model them in our language $\mathcal{L}$. We start by defining a periodic expression. In order to prove some results, we use a formal definition instead of the original definition [20].

Definition 5 A *calendar sequence* is a tuple $C = (C_0, \ldots, C_n)$ of strings. Each C_i, for $i \leq n$, is called a *calendar*. □

Each string in a calendar sequence denotes a time unit. A standard example of a calendar sequence is $C_s = (\text{"Months"}, \text{"Days"}, \text{"Hours"}, \text{"Hours"})$. The string "Hours" intentionally occurs twice.

Definition 6 Let $C = (C_i)_{i \leq n}$ be a calendar sequence. A *C-periodic expression* is a tuple

$$(O_1, O_2, \ldots, O_{n-1}, \mathsf{w}) \in \left(2^{\mathbb{N}}\right)^{n-1} \times \mathbb{N}. \tag{C.13}$$

The set of all C-periodic expressions is denoted by $PEx(C)$. □

Intuitively, for $i < n$, the set O_i represents a set of time units in C_i and r is a time length, measured with the unit C_n. We give an example.

Example 7 Let $C_s = \left(\text{“Months”}, \text{“Days”}, \text{“Hours”}, \text{“Hours”}\right)$. The first eight hours of the first and fifth day of each even month can be represented with the following C_s-periodic expression:

$$(\{2,4,\ldots,12\}, \{1,5\}, \{1\}, 8).$$

Here, the first set in the tuple denotes the even months, the second set denotes the first and fifth day, the third set denotes the first hour, and the last number denotes the 8-hour length. □

Definition 8 Let $C = (C_i)_{i\leq n}$ be a calendar sequence. A *C-time instant* t is a tuple in $\mathbb{N}^{n-1}$. A C-time instant $\mathsf{t} = (\mathsf{t}_i)_{i<n}$ *satisfies* a C-periodic expression $(O_1, \ldots, O_{n-1}, \mathsf{w})$ if all of the following hold:

- For $i < n-1$, $\mathsf{t}_i \in O_i$.
- There is $\mathsf{t}'_{n-1} \in O_{n-1}$ such that $\mathsf{t}'_{n-1} \leq \mathsf{t}_{n-1} < \mathsf{t}'_{n-1} \oplus \mathsf{w}$. Here, $\mathsf{t}'_{n-1} \oplus \mathsf{w}$ is the result of transforming t'_{n-1} and w to a common time unit and then adding them.

□

Example 9 Let P be the C_s-periodic expression from Example 7. Then the C_s-time instant $(4,1,2)$ (i.e., the second hour of the first day of April) satisfies P, whereas the C_s-time instant $(5,1,2)$ (i.e., the second hour of the first day of May) does not. □

C.4.2 Formalizing periodic expressions in $\mathcal{L}$

Let C be a calendar sequence. We now show how to formalize periodic expressions in $PEx(C)$ in our language $\mathcal{L}$. For illustration, we show how to do this when $C = C_s$, but the general case is analogous. We start by defining a signature for defining temporal constraints. We first formally introduce the components of this signature and then give some intuition.

Definition 10 Let Σ_t be a signature containing the following:

- A sort **INSTANTS** for denoting C_s-time instants.
- Three sorts **MONTHS**, **DAYS**, **HOURS** for denoting months, days, and hours, respectively.
- Three flexible unary relation symbols:
  - PM : **MONTHS**.
  - PD : **DAYS**.
  - PH : **HOURS**.

- Three rigid unary function symbols:
  - $\underline{monthOf}$: $\textsf{\textbf{INSTANTS}} \to \textsf{\textbf{MONTHS}}$.
  - $\underline{dayOf}$: $\textsf{\textbf{INSTANTS}} \to \textsf{\textbf{DAYS}}$.
  - $\underline{hourOf}$: $\textsf{\textbf{INSTANTS}} \to \textsf{\textbf{HOURS}}$.
- Rigid binary relation symbols $\leq$: $\textsf{\textbf{HOURS}} \times \textsf{\textbf{HOURS}}$ and $<$: $\textsf{\textbf{HOURS}} \times \textsf{\textbf{HOURS}}$.
- Rigid constant symbols $\underline{1}, \underline{2}, \ldots, \underline{12}$ of type $\textsf{\textbf{MONTHS}}$.
- Rigid constant symbols $\underline{1}, \underline{2}, \ldots, \underline{31}$ of type $\textsf{\textbf{DAYS}}$.
- Rigid constant symbols $\underline{1}, \underline{2}, \ldots, \underline{24}$ of type $\textsf{\textbf{HOURS}}$.

$\square$

For an interpretation function $\mathfrak{I}$, the sets $PM^{\mathfrak{I}}$, $PD^{\mathfrak{I}}$, and $PH^{\mathfrak{I}}$ define the first three entries of a C_s-periodic expression. The functions $\underline{monthOf}^{\mathfrak{I}}$, $\underline{dayOf}^{\mathfrak{I}}$, and $\underline{hourOf}^{\mathfrak{I}}$ compute, respectively, the month, day, and hour of a time instant. Recall that $\underline{monthOf}$, $\underline{dayOf}$, and $\underline{hourOf}$ are rigid symbols, so $\underline{monthOf}^{\mathfrak{I}}$, $\underline{dayOf}^{\mathfrak{I}}$, and $\underline{hourOf}^{\mathfrak{I}}$ do not depend on $\mathfrak{I}$.

Let $\textsf{\textbf{INSTANTS}}$ be a sort denoting C_s-time instants, t be a variable of type $\textsf{\textbf{INSTANTS}}$, and $\varphi(t)$ be the following formula:

$$\begin{aligned}
&\left(\bigvee_{1 \leq \mathsf{m} \leq 12} \left(PM(\underline{\mathsf{m}}) \ \wedge\ \underline{monthOf}(t) = \underline{\mathsf{m}}\right)\right) \ \wedge \\
&\left(\bigvee_{1 \leq \mathsf{d} \leq 31} \left(PD(\underline{\mathsf{d}}) \ \wedge\ \underline{dayOf}(t) = \underline{\mathsf{d}}\right)\right) \ \wedge \\
&\left(\bigvee_{1 \leq \mathsf{h} \leq 24} \left(PH(\underline{\mathsf{h}}) \ \wedge\ \underline{\mathsf{h}} \leq \underline{hourOf}(t) < \underline{\mathsf{h}} + \mathsf{w}\right)\right).
\end{aligned} \tag{C.14}$$

Definition 11 We define $\mathcal{M}$ as the following mapping from interpretation functions to $PEx(\mathcal{C}_s)$ defined as follows. For an interpretation function $\mathfrak{I}$, $\mathcal{M}(\mathfrak{I}) := (M_{\mathfrak{I}}, D_{\mathfrak{I}}, H_{\mathfrak{I}}, 1)$, where

- $M_{\mathfrak{I}} := \left\{\mathsf{m} \mid \mathsf{m} \in PM^{\mathfrak{I}}\right\}$,
- $D_{\mathfrak{I}} := \left\{\mathsf{d} \mid \mathsf{d} \in PD^{\mathfrak{I}}\right\}$,
- $H_{\mathfrak{I}} := \left\{\mathsf{h} \mid \mathsf{h}' \leq \mathsf{h} < \mathsf{h}' + \mathsf{w}^{\mathfrak{I}},\ \text{for some } \mathsf{h}' \in PH^{\mathfrak{I}}\right\}$,

$\square$

We use $\mathcal{M}$ to prove Theorem 12 below, which claims that φ is a template formula for the set of periodic expressions. Note that $\mathcal{M}$ is not surjective on $PEx(\mathcal{C}_s)$. However, any periodic expression $(O_1, O_2, O_3, \mathsf{w})$ is equivalent to an expression of the form $(O_1, O_2, O_3^{\mathsf{w}}, 1)$, where $O_3^{\mathsf{w}} := \{\mathsf{o} + \mathsf{w}' \mid \mathsf{o} \in \mathsf{O}_3, \mathsf{w}' < \mathsf{w}\}$. Therefore, although $\mathcal{M}$ is not surjective, it is expressive enough to capture all periodic expressions up to equivalence.

Theorem 12 For every interpretation function $\mathfrak{I}$, a C_s-time instant $\mathsf{t} = (\mathsf{m}, \mathsf{d}, \mathsf{h})$ satisfies $\mathcal{M}(\mathfrak{I}) = (M_{\mathfrak{I}}, D_{\mathfrak{I}}, H_{\mathfrak{I}}, 1)$ iff $\mathsf{t} \in \varphi^{\mathfrak{I}}$.

Proof.

$$\begin{aligned}
&\mathsf{t} \text{ satisfies } \mathcal{M}(\mathfrak{I}) && \text{(C.15)}\\
&\Leftrightarrow \mathsf{m} \in M_{\mathfrak{I}}, \mathsf{d} \in D_{\mathfrak{I}}, \mathsf{h} \in H_{\mathfrak{I}} && \text{(C.16)}\\
&\Leftrightarrow \mathsf{m} \in PM^{\mathfrak{I}}, \mathsf{d} \in PD^{\mathfrak{I}}, \text{ there is } \mathsf{h}' \in PH^{\mathfrak{I}} \text{ such that } \mathsf{h}' \leq \mathsf{h} < \mathsf{h}' + \mathsf{w}^{\mathfrak{I}} && \text{(C.17)}\\
&\Leftrightarrow \begin{pmatrix} \mathsf{m} \in PM^{\mathfrak{I}},\ \underline{monthOf}^{\mathfrak{I}}(\mathsf{t}) = \mathsf{m}, \\ \mathsf{d} \in PD^{\mathfrak{I}},\ \underline{dayOf}^{\mathfrak{I}}(\mathsf{t}) = \mathsf{d}, \\ \text{there is } \mathsf{h}' \in PH^{\mathfrak{I}} \text{ s.t. } \mathsf{h}' \leq \mathsf{h} < \mathsf{h}' + \mathsf{w}^{\mathfrak{I}},\ \underline{hourOf}^{\mathfrak{I}}(\mathsf{t}) = \mathsf{h} \end{pmatrix} && \text{(C.18)}\\
&\Leftrightarrow \mathsf{t} \in \varphi^{\mathfrak{I}}. && \text{(C.19)}
\end{aligned}$$

□

C.5 Datasets and synthetic policies used for experiments

C.5.1 Datasets for BM-RBAC

We use the access control matrix provided by Amazon for the IEEE MLSP 2012 competition [72]. They assign three types of permissions, named "HOST", "PERM_GROUP", and "SYSTEM_GROUP". For each type of permission, we created an access control matrix by collecting all users and all permissions belonging to that type. There are approximately 30,000 users, 1,700 permissions of type "HOST", 6,000 of type "PERM_GROUP", and 20,000 of type "SYSTEM_GROUP".

The resulting access control matrices are far too large to be handled efficiently by the policy miner we developed. To address this, during 5-fold cross-validation (see Section 4.1.4 for an overview), we worked instead with an access control submatrix induced by a sample of 30% of all users. Each fold used a different sample of users. To see why this is enough, we remark that, in RBAC policies, the number R of roles is usually much smaller than the number N of users. Moreover, the number K of possible subsets of permissions that users are assigned by RBAC policies is small in comparison to the whole set of possible subsets of permissions. If N is much larger than

K, then, by the pidgeonhole principle, many users have the same subset of permissions. Therefore, it is not necessary to use all N users to mine an adequate RBAC policy, as only a fraction of them has all the necessary information. The high TPR (above 80%) of the policy that we mined supports the fact that using a submatrix is still enough to mine policies that generalize well.

C.5.2 Synthetic policy for spatio-temporal RBAC

We present here the synthetic spatio-temporal RBAC policy that we used for our experiments. We assume the existence of five rectangular buildings, described in Table C.1. The left column indicates the building's name and the right column describes the two-dimensional coordinates of the building's corners. There are five roles, which we describe next. We regard a permission as an *action* executed on an *object*.

Name	Corners
Main building	$(1,3),(1,4),(4,4),(4,3)$
Library	$(1,1),(1,2),(2,2),(2,1)$
Station	$(8,1),(8,9),(9,9),(9,1)$
Laboratory	$(2,6),(2,8),(4,8),(4,6)$
Computer room	$(6,6),(6,7),(7,7),(7,6)$

Table C.1

The first role assigns a permission to a user if all of the following hold:

- The user is at most 1 meter away from the computer room.
- The object is in the computer room or in the laboratory.
- The current day is an odd day of the month.
- The current time is between 8AM and 5PM.

The second role assigns a permission to a user if all of the following hold:

- The user is outside the library.
- The object is at most 1 meter away from the library.
- Either
  - the current day is before the 10th day of the month and the current time is between 2PM and 8PM or
  - the current day is after the 15th day of the month and the current time is between 8AM and 12PM.

The third role assigns a permission to a user if all of the following hold:

- The user is at most 3 meters away from the main building.
- The object is at most 3 meters away from the main building.

The fourth role assigns a permission to a user if all of the following hold:

- The user is inside the library.
- The object is outside the library.
- The current day is before the 15th day of the month.
- The current time is between 12AM and 12PM.

The fifth role assigns a permission to a user if all of the following hold:

- The user is inside the main building, at most 1 meter away from the library, inside the laboratory, at most 2 meters away from the computer room, or inside the station.
- The object satisfies the same spatial constraint.
- The current day is before the 15th day of the month.
- The current time is between 12AM and 12PM.

Bibliography

[1] Business: The Economy, how Leeson broke the bank. http://news.bbc.co.uk/2/hi/business/375259.stm.

[2] Ali E Abdallah and Etienne J Khayat. A formal model for parameterized role-based access control. In *Formal Aspects in Security and Trust*, pages 233–246. Springer, 2005.

[3] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[4] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th International Conference on Very Large Databases, VLDB*, volume 1215, pages 487–499, 1994.

[5] Gail-Joon Ahn and Ravi Sandhu. The rsl99 language for role-based separation of duty constraints. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 43–54. ACM, 1999.

[6] Subhendu Aich, Shamik Sural, and Arun K Majumdar. STARBAC: Spatiotemporal role based access control. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1567–1582. Springer, 2007.

[7] Mohammad A Al-Kahtani and Ravi Sandhu. A model for attribute-based user-role assignment. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 353–362. IEEE, 2002.

[8] Mohammad A Al-Kahtani and Ravi Sandhu. Induced role hierarchies with attribute-based rbac. In *Proceedings of the eighth ACM Symposium on access control models and technologies*, pages 142–148. ACM, 2003.

[9] Elaine Angelino, Nicholas Larus-Stone, Daniel Alabi, Margo Seltzer, and Cynthia Rudin. Learning certifiably optimal rule lists. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 35–44. ACM, 2017.

[10] Konstantine Arkoudas, Ritu Chadha, and Jason Chiang. Sophisticated access control via smt and logical frameworks. *ACM Transactions on Information and System Security (TISSEC)*, 16(4):17, 2014.

[11] Martin Atzmueller and Frank Puppe. SD-map–a fast algorithm for exhaustive subgroup discovery. In *Knowledge Discovery in Databases: PKDD 2006*, pages 6–17. Springer, 2006.

[12] Ezedin Barka, Ravi Sandhu, et al. A role-based delegation model and some extensions. In *Proceedings of the 23rd National Information Systems Security Conference*, volume 4, pages 49–58. Citeseer, 2000.

[13] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[14] Jon Barwise. *Handbook of mathematical logic*, volume 90. Elsevier, 1982.

[15] David Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009.

[16] Moritz Y Becker, Cédric Fournet, and Andrew D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.

[17] Moritz Y Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 139–154. IEEE, 2004.

[18] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. GemRBAC-DSL: a high-level specification language for role-based access control policies. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, pages 179–190. ACM, 2016.

[19] Ameni Ben Fadhel, Domenico Bianculli, Lionel Briand, and Benjamin Hourte. A model-driven approach to representing and checking RBAC contextual policies. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 243–253. ACM, 2016.

[20] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, 2001.

[21] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. ABAC with group attributes and attribute hierarchies utilizing the policy machine. In *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control*, pages 17–28. ACM, 2017.

[22] Rafae Bhatti, Arif Ghafoor, Elisa Bertino, and James BD Joshi. X-gtrbac: an xml-based policy specification framework and architecture for enterprise-wide access control. *ACM Transactions on Information and System Security (TISSEC)*, 8(2):187–227, 2005.

[23] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[24] Matt Bishop. *Computer security: art and science*. Addison-Wesley Professional, 2003.

[25] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. Label-based access control: an ABAC model with enumerated authorization policy. In *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*, pages 1–12. ACM, 2016.

[26] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[27] Piero Bonatti, Clemente Galdi, and Davide Torres. ERBAC: event-driven RBAC. In *Proceedings of the 18th ACM symposium on Access control models and technologies*, pages 125–136. ACM, 2013.

[28] Egon Börger, Erich Grädel, and Yuri Gurevich. *The classical decision problem*. Springer Science & Business Media, 2001.

[29] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[30] Thang Bui, Scott D Stoller, and Jiajie Li. Mining relationship-based access control policies. *arXiv preprint arXiv:1708.04749*, 2017.

[31] Suroop Mohan Chandran and James BD Joshi. LoT-RBAC: a location and time-based RBAC model. In *International Conference on Web Information Systems Engineering*, pages 361–375. Springer, 2005.

[32] Suresh N Chari and Ian M Molloy. Generation of attribute based access control policy from existing authorization system, September 2 2014. US Patent App. 14/474,747.

[33] Suresh N Chari and Ian M Molloy. Generation of attribute based access control policy from existing authorization system, February 16 2016. US Patent 9,264,451.

[34] Liang Chen and Jason Crampton. On spatio-temporal constraints and inheritance in role-based access control. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 205–216. ACM, 2008.

[35] Yuan Cheng, Khalid Bijon, and Ravi Sandhu. Extended ReBAC administrative models with cascading revocation and provenance support. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, SACMAT '16, pages 161–170, New York, NY, USA, 2016. ACM.

[36] Peter Clark and Robin Boswell. Rule induction with CN2: Some recent improvements. In *European Working Session on Learning*, pages 151–163. Springer, 1991.

[37] Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine learning*, 3(4):261–283, 1989.

[38] William W Cohen. Fast effective rule induction. In *Proceedings of the twelfth international conference on machine learning*, pages 115–123, 1995.

[39] Carlos Cotrini, Thilo Weghorn, and David Basin. Mining ABAC rules from sparse logs. In *European Symposium on Security and Privacy (EuroS&P).* IEEE, 2018. **©2018, IEEE. Reprinted with permission.**

[40] Carlos Cotrini, Thilo Weghorn, David Basin, and Manuel Clavel. Analyzing first-order role based access control. In *Computer Security Foundations Symposium (CSF).*, pages 3–17. IEEE, 2015. **©2015, IEEE. Reprinted with permission.**

[41] Xiutao Cui, Yuliang Chen, and Junzhong Gu. Ex-RBAC: an extended role based access control model for location-aware mobile collaboration system. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*, pages 36–36. IEEE, 2007.

[42] Massimiliano de Leoni and Wil MP van der Aalst. Data-aware process mining: discovering decisions in processes using alignments. In *Proceedings of the 28th annual ACM Symposium on Applied Computing*, pages 1454–1461. ACM, 2013.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[44] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[45] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.

[46] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[47] AG D'yakonov. Solution methods for classification problems with categorical attributes. *Computational Mathematics and Modeling*, 26(3):408–428, 2015.

[48] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[49] Alina Ene, William Horne, Nikola Milosavljevic, Prasad Rao, Robert Schreiber, and Robert E Tarjan. Fast exact and heuristic methods for role minimization problems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 1–10. ACM, 2008.

[50] Verizon Enterprise. Data breach investigation report. http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017/.

[51] Verizon Enterprise. Data breach investigation report. https://www.verizonenterprise.com/resources/reports/rp_DBIR_2018_Report_en_xg.pdf.

[52] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[53] Anna Lisa Ferrara, P Madhusudan, and Gennaro Parlato. Policy analysis for self-administrated role-based access control. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 432–447. Springer, 2013.

[54] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 196–205, New York, NY, USA, 2005. ACM.

[55] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205. ACM, 2005.

[56] Philip WL Fong. Relationship-based access control: protection model and policy language. In *Proceedings of the first ACM conference on Data and application security and privacy*, pages 191–202. ACM, 2011.

[57] Mario Frank, Joachim M Buhman, and David Basin. Role mining with probabilistic models. *ACM Transactions on Information and System Security (TISSEC)*, 15(4):15, 2013.

[58] Mario Frank, Joachim M Buhmann, and David Basin. On the definition of role mining. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, pages 35–44. ACM, 2010.

[59] Mario Frank, Andreas P Streich, David Basin, and Joachim M Buhmann. A probabilistic approach to hybrid role mining. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 101–111. ACM, 2009.

[60] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer Series in Statistics, Berlin, 2001.

[61] Mayank Gautam, Sadhana Jha, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. Poster: Constrained policy mining in attribute based access control. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pages 121–123. ACM, 2017.

[62] Mei Ge and Sylvia L Osborn. A design for parameterized roles. In *Research Directions in Data and Applications Security XVIII*, pages 251–264. Springer, 2004.

[63] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the second ACM workshop on Role-based access control*, pages 153–159. ACM, 1997.

[64] Simon Godik, Anne Anderson, Bill Parducci, Polar Humenn, and Sekhar Vajjhala. OASIS eXtensible Access Control 2 Markup Language (XACML) 3. Technical report, Technical Report, OASIS, 2002.

[65] Qi Guo, Jaideep Vaidya, and Vijayalakshmi Atluri. The role hierarchy mining problem: Discovery of optimal role hierarchies. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 237–246. IEEE, 2008.

[66] Joseph Y Halpern and Vicky Weissman. Using first-order logic to reason about policies. *ACM Transactions on Information and System Security (TISSEC)*, 11(4):21, 2008.

[67] Thomas Hofmann and Joachim M Buhmann. Pairwise data clustering by deterministic annealing. *Ieee transactions on pattern analysis and machine intelligence*, 19(1):1–14, 1997.

[68] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (ABAC) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[69] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *IEEE Computer*, 48(2):85–88, 2015.

[70] Jingwei Huang, David M Nicol, Rakesh Bobba, and Jun Ho Huh. A framework integrating attribute-based policies into role-based access control. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, pages 187–196. ACM, 2012.

[71] Jens Hühn and Eyke Hüllermeier. FURIA: an algorithm for unordered fuzzy rule induction. *Data Mining and Knowledge Discovery*, 19(3):293–319, 2009.

[72] IEEE. 2012 IEEE International workshop on machine learning for signal processing. Amazon data science competition, 2012. `http://mlsp2012.conwiz.dk/index.php?id=43`.

[73] Neil Immerman. *Descriptive complexity*. Springer, 1999.

[74] Edwin T Jaynes. Information theory and statistical mechanics. *Physical review*, 106(4):620, 1957.

[75] Sadhana Jha, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. Security analysis of temporal rbac under an administrative model. *Computers & Security*, 46:154–172, 2014.

[76] Xin Jin, Ram Krishnan, and Ravi Sandhu. A role-based administration model for attributes. In *Proceedings of the First International Workshop on Secure and Resilient Architectures and Systems*, pages 7–12. ACM, 2012.

[77] Xin Jin, Ram Krishnan, and Ravi Sandhu. A unified attribute-based access control model covering dac, mac and rbac. In *Data and Applications Security and Privacy XXVI*, pages 41–55. Springer, 2012.

[78] Xin Jin, Ravi Sandhu, and Ram Krishnan. Rabac: role-centric attribute-based access control. In *Computer Network Security*, pages 84–96. Springer, 2012.

[79] James BD Joshi. Access-control language for multidomain environments. *Internet Computing, IEEE*, 8(6):40–50, 2004.

[80] Viktor Jovanoski and Nada Lavrač. Classification rule learning with Apriori-C. In *Portuguese Conference on Artificial Intelligence*, pages 44–51. Springer, 2001.

[81] Kaggle: the home of data science. http://www.kaggle.com.

[82] Amazon.com – Employee access challenge. http://www.kaggle.com/c/amazon-employee-access-challenge.

[83] Amazon.com – Employee access challenge. Winners' solution and final results. https://www.kaggle.com/c/amazon-employee-access-challenge/forums/t/5283/winning-solution-code-and-methodology.

[84] Branko Kavšek and Nada Lavrač. Apriori-SD: Adapting association rule learning to subgroup discovery. *Applied Artificial Intelligence*, 20(7):543–583, 2006.

[85] HK Kesavan and JN Kapur. Maximum entropy and minimum cross-entropy principles: Need for a broader perspective. In *Maximum Entropy and Bayesian Methods*, pages 419–432. Springer, 1990.

[86] Gregg Kreizman, Ant Allan, Felix Gaehtgens, Brian Iverson, and Anmol Singh. Identity and access management scenario 2020: Powering digital business, 2015. https://www.gartner.com/doc/3174723/identity-access-management-scenario-.

[87] Shriram Krishnamurthi. The continue server (or, how i administered padl 2002 and 2003). In *Practical aspects of declarative languages*, pages 2–16. Springer, 2003.

[88] Martin Kuhlmann, Dalia Shohat, and Gerhard Schimpf. Role mining-revealing business roles for security administration using data mining technology. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 179–186. ACM, 2003.

[89] D Richard Kuhn, Edward J Coyne, and Timothy R Weil. Adding attributes to role-based access control. *Computer*, 43(6):79–81, 2010.

[90] Mahendra Kumar and Richard E Newman. STRBAC–an approach towards spatio-temporal role-based access control. In *Communication, Network, and Information Security*, pages 150–155, 2006.

[91] Nada Lavrač, Peter Flach, and Blaz Zupan. *Rule evaluation measures: A unifying view.* Springer, 1999.

[92] Nada Lavrač, Branko Kavšek, Peter Flach, and Ljupčo Todorovski. Subgroup discovery with CN2-SD. *The Journal of Machine Learning Research*, 5:153–188, 2004.

[93] M. Lichman. UCI machine learning repository. amazon access samples data set, 2013.

[94] Torsten Lodderstedt, David Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML 2002 The Unified Modeling Language*, pages 426–441. Springer, 2002.

[95] Haibing Lu, Jaideep Vaidya, and Vijayalakshmi Atluri. Optimal boolean matrix decomposition: Application to role engineering. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 297–306. IEEE, 2008.

[96] Barsha Mitra, Shamik Sural, Vijayalakshmi Atluri, and Jaideep Vaidya. Towards mining of temporal roles. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 65–80. Springer, 2013.

[97] Barsha Mitra, Shamik Sural, Vijayalakshmi Atluri, and Jaideep Vaidya. The generalized temporal role mining problem. *Journal of Computer Security*, 23(1):31–58, 2015.

[98] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. Mining temporal roles using many-valued concepts. *Computers & Security*, 60:79–94, 2016.

[99] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. A survey of role mining. *ACM Computing Surveys (CSUR)*, 48(4):50, 2016.

[100] Decebal Mocanu, Fatih Turkmen, and Antonio Liotta. Towards ABAC Policy Mining from Logs with Deep Learning. In *Proceedings of the 18th International Multiconference*, Intelligent Systems, 2015.

[101] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin Calo, and Jorge Lobo. Mining roles with semantic meanings. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, pages 21–30. ACM, 2008.

[102] Ian Molloy, Jorge Lobo, and Suresh Chari. Adversaries' holy grail: access control analytics. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 54–61. ACM, 2011.

[103] Ian Molloy, Youngja Park, and Suresh Chari. Generative models for access control policies: applications to role mining over logs with attribution. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, pages 45–56. ACM, 2012.

[104] Subhojeet Mukherjee, Indrakshi Ray, Indrajit Ray, Hossein Shirazi, Toan Ong, and Michael G. Kahn. Attribute based access control for healthcare resources. In *Proceedings of the 2Nd ACM Workshop on Attribute-Based Access Control*, ABAC '17, pages 29–40, New York, NY, USA, 2017. ACM.

[105] National Cybersecurity Center of Excellence. Attribute-based Access Control, 2017. https://nccoe.nist.gov/projects/building-blocks/attribute-based-access-control.

[106] Tim Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, Shriram Krishnamurthi, and Varun Singh. The margrave tool. http://www.margrave-tool.org/v3/.

[107] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[108] Orange: Rule Induction with CN2. http://orange.readthedocs.io/en/latest/reference/rst/Orange.classification.rules.html.

[109] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[110] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[111] David Martin Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation, 2011.

[112] J Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

[113] Indrakshi Ray and Manachai Toahchoodee. A spatio-temporal role-based access control model. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 211–226. Springer, 2007.

[114] Kenneth Rose, Eitan Gurewitz, and Geoffrey C Fox. Vector quantization by deterministic annealing. *IEEE Transactions on Information theory*, 38(4):1249–1257, 1992.

[115] Ravi Sandhu. Role hierarchies and constraints for lattice-based access controls. In *European Symposium on Research in Computer Security*, pages 65–79. Springer, 1996.

[116] Jürgen Schlegelmilch and Ulrike Steffens. Role mining with ORCA. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 168–176. ACM, 2005.

[117] sklearn.tree.DecisionTreeClassifier. http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier.

[118] Scikit-learn. Tuning the hyper-parameters of an estimator, 2007–2017.

[119] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[120] Dan Steinberg and Phillip Colla. Cart: classification and regression trees. *The top ten algorithms in data mining*, 9:179, 2009.

[121] Scott D Stoller and Thang Bui. Mining hierarchical temporal roles with multiple metrics. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 79–95. Springer, 2016.

[122] Scott D Stoller, Ping Yang, Mikhail I Gofman, and CR Ramakrishnan. Symbolic reachability analysis for parameterized administrative role-based access control. *Computers & Security*, 30(2):148–164, 2011.

[123] Andreas P Streich, Mario Frank, David Basin, and Joachim M Buhmann. Multi-assignment clustering for boolean data. In *Proceedings of the 26th annual international conference on machine learning*, pages 969–976. ACM, 2009.

[124] Y Tikochinsky, NZ Tishby, and Raphael David Levine. Alternative approach to maximum-entropy inference. *Physical Review A*, 30(5):2638, 1984.

[125] Manachai Toahchoodee, Indrakshi Ray, Kyriakos Anastasakis, Geri Georg, and Behzad Bordbar. Ensuring spatio-temporal access control for real-world applications. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 13–22. ACM, 2009.

[126] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin. *Decentralized Composite Access Control*. Springer, 2014.

[127] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 175–184. ACM, 2007.

[128] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: A formal perspective. *ACM Transactions on Information and System Security (TISSEC)*, 13(3):27, 2010.

[129] Jaideep Vaidya, Vijayalakshmi Atluri, and Janice Warner. Roleminer: mining roles using subset enumeration. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 144–153. ACM, 2006.

[130] Zhongyuan Xu and Scott D Stoller. Algorithms for mining meaningful roles. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 57–66. ACM, 2012.

[131] Zhongyuan Xu and Scott D Stoller. Mining attribute-based access control policies from RBAC policies. In *Emerging Technologies for a Smarter World (CEWIT), 2013 10th International Conference and Expo on*, pages 1–6. IEEE, 2013.

[132] Zhongyuan Xu and Scott D Stoller. Mining attribute-based access control policies from logs. In *Data and Applications Security and Privacy XXVIII*, pages 276–291. Springer, 2014.

[133] Zhongyuan Xu and Scott D Stoller. Mining attribute-based access control policies. *Dependable and Secure Computing, IEEE Transactions on*, 12(5):533–545, 2015.

[134] Kan Yang, Zhen Liu, Xiaohua Jia, and Xuemin Sherman Shen. Time-domain attribute-based access control for cloud-based video content sharing: A cryptographic approach. *IEEE Transactions on Multimedia*, 18(5):940–950, 2016.

[135] Qiang Yang, Haining Henry Zhang, and Tianyi Li. Mining web logs for prediction models in WWW caching and prefetching. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 473–478. ACM, 2001.

[136] Dana Zhang, Kotagiri Ramamohanarao, and Tim Ebringer. Role engineering using graph optimisation. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 139–144. ACM, 2007.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.