Master Thesis

Efficient evaluation of PBel access control policies

Author(s):
Werthmüller, Tanja

Publication Date:
2013

Permanent Link:
https://doi.org/10.3929/ethz-a-009779094

Rights / License:
In Copyright - Non-Commercial Use Permitted
Efficient Evaluation of PBel Access Control Policies

Master Thesis
Tanja Werthmüller
April 18, 2013

Supervisor: Prof. Dr. D. Basin
Advisor: Dr. S. Marinovic
Department of Computer Science, ETH Zürich
Abstract

Formal access control policy languages have had a great impact in the field of access control. However, the evaluation processes for formal languages are not concerned with efficiency. Therefore, a programmer needs to write his own evaluation process, optimize it and manually prove its correctness with respect to formal semantics of the policy language, if he is concerned with efficiency.

In this thesis, we focus on the problem of automated generation of correct and efficient access control evaluation procedure. We first define an efficiency requirement that such a procedure should satisfy. Second, we present our solution to this problem based on Markov Decision Processes and prove formally its correctness. In contrast to previous work in this research area, our technique does not only consider sub-parts of the policy, but aims at optimizing the policy as a whole. We focus on the expressive access control policy language PBel, but the presented ideas can easily be generalized to other languages, similar to PBel.

Finally, we implement our solution to generate dynamic policy libraries, which we integrate into an Apache PBel evaluation module and test the resulting evaluation code for two sample policies. The test results show that our optimized evaluation process increases performance of the overall system compared to a naive evaluation process for formal policies.
Acknowledgments

This master thesis was the most exciting challenge I ever faced during my time at ETH, that I could not have mastered without the help and support of many people.

First of all, I would like to express my sincerest gratitude to my advisor Dr. Srdjan Marinovic for his excellent guidance, his unceasing support and his outstanding commitment.

I would also like to thank my supervisor Prof. Dr. David Basin for his constructive advice and his gentle support.

Further, I am grateful to Simon Stelling for his detailed proofreading and his unfailing patience.

Finally, I would like to thank my family and friends for all their encouragement.
# Contents

Acknowledgments iii

Contents v

1 Introduction 1

1.1 The Problem 2

1.2 Contributions 3

1.2.1 Theoretical Contribution 3

1.2.2 Practical Contribution 5

1.2.3 Summary of Contributions 5

1.3 Thesis Outline 6

2 Related Work 7

2.1 Evaluation Optimization for XACML 7

2.1.1 Statistics & Clustering Based Framework for Efficient XACML Policy Evaluation 8

2.1.2 Refactoring Access Control Policies for Performance Improvement 10

2.1.3 Automated XACML Policy Reconfiguration for Evaluation Optimisation 11

2.2 Firewall Filtering 11

2.3 MDP in Intrusion Detection 12

3 Preliminaries 13

3.1 PBel 13

3.1.1 PBel’s Decision Space 13

3.1.2 Core PBel Policy Language 15

3.1.3 Extensions to Core PBel 16

3.1.4 PBel Analysis Procedure 18

3.2 Markov Decision Process 19
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1 Finding Optimal Solutions for a Markov Decision Process</td>
</tr>
<tr>
<td>4 Translating a PBel Policy into an Efficient Evaluation Procedure</td>
</tr>
<tr>
<td>4.1 Mapping Access Control Policy Evaluation into the MDP Setting</td>
</tr>
<tr>
<td>4.2 Algorithm Overview</td>
</tr>
<tr>
<td>4.2.1 Constructing the MDP Structure for a PBel Policy</td>
</tr>
<tr>
<td>4.2.2 Calculating the Optimal Strategy</td>
</tr>
<tr>
<td>4.3 Dependency on PBel</td>
</tr>
<tr>
<td>4.4 Analysis of the Solution</td>
</tr>
<tr>
<td>4.4.1 Correctness of the MDP Construction</td>
</tr>
<tr>
<td>4.4.2 Correctness of the Solution</td>
</tr>
<tr>
<td>4.5 Complexity</td>
</tr>
<tr>
<td>5 Experimental Validation</td>
</tr>
<tr>
<td>5.1 Translating the Optimal Strategy into Evaluation Code</td>
</tr>
<tr>
<td>5.2 Experiment Setup</td>
</tr>
<tr>
<td>5.2.1 Apache 2.4 Web Server</td>
</tr>
<tr>
<td>5.2.2 Stress tests</td>
</tr>
<tr>
<td>5.3 Experiments</td>
</tr>
<tr>
<td>5.3.1 Overview</td>
</tr>
<tr>
<td>5.3.2 The Cambridge Policy</td>
</tr>
<tr>
<td>5.3.3 The Moodle Policy</td>
</tr>
<tr>
<td>5.3.4 Test Conclusions</td>
</tr>
<tr>
<td>6 Conclusion</td>
</tr>
<tr>
<td>A Translation of the Optimal Strategy to C-Code</td>
</tr>
<tr>
<td>B Experiment Mixture Description</td>
</tr>
<tr>
<td>Bibliography</td>
</tr>
</tbody>
</table>
Introduction

As systems get more complex, correctly enforcing and evaluating access control policies for systems’ resources becomes more and more difficult. Moreover, verifying or even understanding the specification of a complex policy can become unfeasible, especially when the enforcement is merged with the evaluation procedure.

Formal access control policy languages have had a great impact in the field of access control. They separate the evaluation from the enforcement. The evaluation is done by a Policy Decision Point (PDP) which interprets any formal policy specification according to its unambiguous semantics. Changes in the policy therefore do not require modifications of the evaluation code. This greatly reduces the problem of correct policy evaluation. Some formal access control policy languages also have a verification procedure, which allows one to compare the permissiveness of different policies.

However, the semantics and the PDP implementations of formal languages are not concerned with efficiency. To put it differently, the evaluation process naively implementing formal semantics of policies requires evaluation of every condition in the policy and therefore suffers from inefficient request evaluations. This means that a programmer concerned with efficiency has to write his own evaluation code, optimize it, and then manually prove that the new code is correct with respect to formal semantics, which is very error prone.

As an example consider the policy \textit{permit if } \((a \lor b)\), for some request conditions \(a\) and \(b\), which permits all requests satisfying conditions \(a\) or \(b\). This policy however does not require all the conditions to be evaluated in every case. If \(a\) evaluates to \textit{true} for a given request, then the value of \(b\) does not influence the resulting decision. Depending on the time needed to evaluate one specific condition, this consideration can have a huge impact on
evaluation performance. However, without manual optimization of the programmer, this knowledge remains unused.

In this thesis we focus on the formal policy language PBel\(^1\) [5]. We consider PBel to be a good choice to write access control policies because:

- It is a highly expressive formal language (e.g. it can express many other popular languages such as XACML and access control idioms such as role based access control, Chinese wall, and so forth).

- It has a verification procedure.

- It is very concise and suitable for writing complex policies through composition.

1.1 The Problem

The problem, and subject of this thesis, is that of automated generation of an **efficient** and **correct** evaluation procedure for a given PBel access control policy. A PBel policy specifies a mapping from an access request and a set of request conditions that the request satisfies into a policy decision. A PBel evaluation procedure takes a policy specification, checks which request conditions are satisfied by a given request, and then evaluates a given policy specification according to PBel’s formal semantics.

To more precisely define the efficiency requirement, we make the following assumptions. First, we assume that the evaluation of each request condition requires a fixed amount of time, which we call the cost of this condition. Second, we assume no knowledge of the likelihood that a condition will be satisfied for a given request. We assume therefore the probability of a given condition to be **true** or **false** for a given request to be 50%. Finally, we assume that condition evaluation costs are known during the generation of the evaluation procedure. The policy author is required to provide this meta-data as input to the optimization process described in this thesis.

Under these assumptions, we can formulate the efficiency requirement for a PBel evaluation procedure as:

Given a policy \(\Pi\), an evaluation procedure \(\text{dec}\), and the set of uniformly distributed requests \(R\) for \(\Pi\), there does not exist another evaluation procedure \(\text{dec}'\) such that:

\[
\sum_r \text{time}(\text{dec}', \Pi, r) < \sum_r \text{time}(\text{dec}, \Pi, r),
\]

where \(r \in R\) and \(\text{time}(\cdot; \Pi, r)\) denotes the amount of time needed by the procedure to evaluate \(r\) against \(\Pi\).

\(^1\)A detailed description of the PBel access control language is given in Chapter 3.
1.2 Contributions

1.2.1 Theoretical Contribution

We are the first to propose the problem of automated generation of an efficient evaluation procedure for a formal policy language. In this thesis we focus on the policy language PBel. As a solution to this problem we present an algorithm that translates a PBel policy in an automated fashion into an evaluation procedure. Further, we show that the generated procedure satisfies the stated efficiency requirement. The algorithm rests on a theory widely used in control theory called Markov Decision Process (MDP). The algorithm presented in this thesis constructs an MDP structure from a PBel policy specification and the given condition costs, then calculates an optimal evaluation strategy based on this MDP structure and finally generates an evaluation procedure from this optimal strategy. The operation range of the algorithm is depicted by steps (1) and (2) in Figure 1.1.

![Figure 1.1: Overview of the translation of a policy into evaluation code](image)

The core idea behind the MDP theory is to find an optimal strategy for an agent in a given, stochastic environment. This environment consists of states and actions. The agent can influence the state of the environment by selecting different actions. Each action leads to a (partly random) state transition. Every such state transition might award the agent with a certain reward. An environment is called fully observable if the agent has full information about state transition probabilities and rewards.

The agent’s goal is to maximize his total reward. If rewards are negative, the agent has an incentive to transfer the environment into a terminal state in few and cheap state transitions. A terminal state is an MDP state where no
successive actions are available, i.e., once the environment is in a terminal state, it will never transition to any other state. The MDP theory presents a method to calculate an optimal strategy for this problem which consists of a recommended successive action in each state and guarantees to maximize the expected total utility of the agent.

In the context of access control, the agent represents the PDP evaluating the policy for a given request. The states of the MDP environment correspond to the different states of the request evaluation of the PDP against the given policy. Terminal states represent a definite decision for the policy evaluation. Actions represent the evaluation of request conditions in the given states of the policy evaluation process.

Each state of the MDP structure identifies a specific state of the policy evaluation process, i.e., every request condition is either known to be true or false or has not yet been evaluated for the request under evaluation. Each state of the policy evaluation process also has a decision value which corresponds to the decision of the policy for requests satisfying the evaluation conditions of the state. This decision is either definite, if it can be made, or pending. The definite decision is a final access control decision which does not change no matter what the evaluation results for unevaluated conditions would be for the request characterized by that state. Otherwise, if the policy cannot be evaluated to make a definite decision, the decision of that state is pending, and more conditions need to be evaluated for such requests before a final decision may be reached.

Actions of the MDP represent the act of evaluation a next condition by the PDP. The evaluation of a request by the PDP then corresponds to a transition through the MDP structure. As the PDP’s goal is to reach a decision state spending as little time as possible in evaluation, and as each additional condition evaluation requires time, the PDP has an incentive to reach a decision state in as few and with as cheap condition evaluations as possible. This goal can be mapped precisely to the goal of the agent in the MDP context and therefore by calculating an optimal strategy for the MDP agent, we can also find an optimal strategy for the PDP.

The generated evaluation procedure, which is the evaluation of a request in the optimal way for the stated efficiency requirement, then corresponds to a transition through the MDP structure using only the actions recommended by the optimal strategy. This evaluation procedure is implemented by the PDP and results in an optimal evaluation order of request conditions.

The algorithm described in Chapter 4 implements the construction of the MDP structure, given by a PBel access control policy, and calculates the optimal strategy for this MDP structure.
1.2. Contributions

1.2.2 Practical Contribution

Given our theoretical investigation and results, we now want to show their practical relevance. In particular we want to validate the following hypothesis:

*Evaluation of a PBel policy specification, using an evaluation procedure which satisfies the stated efficiency requirement, can yield a performance gain for the overall application.*

We implement our solution to generate efficient evaluation code in the form of a dynamic C-library (depicted by step (3) in Figure 1.1), as well as a generic custom PDP Apache module which loads the evaluation code for the specified access control policy and runs it for the given access request. These components are general access control modules for Apache and do not only work for our theoretical contribution.

Stress testing the Apache web server, we validate the stated hypothesis using synthetic data and policies based on real-life use cases. In our tests we compare the average evaluation time of our generated evaluation code and a naive implementation of the policy for different kinds of requests. This comparison shows highest effectiveness of our approach if most requests only require evaluation of cheap request conditions. However, even for a large number of requests that require also evaluation of expensive conditions, a considerable performance increase can be observed.

1.2.3 Summary of Contributions

Our contributions can be summarized as:

1. We are the first to investigate the problem of efficient, yet provably correct evaluation of an expressive formal policy language. We map this problem to the setting of Markov Decision Processes (MDP) and design an algorithm that performs this translation. We focus on the PBel policy language, but the ideas presented in this thesis are more general and can be applied to other policy languages, similar to PBel. In particular other policy algebra languages ([7] [16] [17]) are suitable for our proposed solution.

2. We provide experimental evidence that show that optimization on the level of atomic policy reordering increases performance of policy evaluation.
1.3 Thesis Outline

The remainder of this thesis is organized in the following way. In Chapter 2 we discuss related work of this research area. In Chapter 3 we introduce the policy language PBel and Belnap’s four-valued logic on which it operates, as well as the basics of MDP theory. Then in Chapter 4 we describe our algorithm in detail and prove its correctness and show optimality. The description of the experiments we present in Chapter 5 and in Chapter 6 we summarize our contributions and discuss future work.
Chapter 2

Related Work

2.1 Evaluation Optimization for XACML

The problem of optimizing evaluation in access control has never been defined through an explicit efficiency requirement (as we have done). Furthermore, no research has ever considered optimization of evaluation for a formal expressive policy specification language. However, Marouf et al. [14], El Kateb et al. [10] and Miseldine [15] present optimization techniques for the evaluation of specific parts of policies specified in the **eXtensible Access Control Markup Language (XACML)** [1]. Their solutions only consider a subset of the XACML language.

Before we present their solutions, we first give a brief overview of XACML. XACML is a standardized access control language consisting of **rules**, **policies** and a **policy set**, where rules are grouped together into a policy, and different policies are combined in a policy set. The relationship between these elements is shown in Figure 2.1).

A rule’s **effect** specifies whether the decision returned by the rule is permit (P) or deny (D) if the rule is applicable and its condition is satisfies for the given request. If the rule is not applicable, the returned decision is NA and if an evaluation error occurs, the returned decision is IN.

The decisions of different rules in a policy are combined based on the **rule combining algorithm** of that policy. XACML defines the following rule combining algorithms: deny-overwrites, ordered-deny-overwrites, permit-overwrites, ordered-permit-overwrites and first-applicable. The decisions of different policies in a policy set are combined based on the **policy combining algorithm** specified by the policy set. The policy combining algorithms defined by

---

1Reproduced from https://www.ibm.com/developerworks/xml/library/x-xacml
XACML are the same as the rule combining algorithms and the additional algorithm called only-one-applicable.

As for PBel, access control policies specified in XACML can return one of four access decisions (P, D, NA, IN). However in contrast to PBel, XACML’s semantics are not formally specified.

As the proposed techniques all rely on reordering and grouping of rules, they all focus only on a subset of the rule combining algorithms (e.g., permit/deny-overwrite). We believe that our method is much more flexible because PBel can express XACML policies as well as policies of many other policy specification languages.

We now look at each solution in more detail.

### 2.1.1 Statistics & Clustering Based Framework for Efficient XACML Policy Evaluation

Marouf et al. [14] focus on policy evaluation optimization by reordering policies and policy rules according to access request statistics. They also introduce a clustering technique that categorizes policies with respect to target subjects.
They recognized the risk of a policy evaluation procedure becoming the bottleneck when policies are complex and requests are frequent and propose a very natural method of reordering rules based on request statistics.

Since unlimited reordering of XACML rules is only safe for a small set of rule combining algorithms, they only prove correctness of their approach in the cases of permit- and deny-overwrite.

The proposed policy evaluation engine (PDP) outperforms the Sun XACML PDP by a factor of two. The Sun XACML evaluation engine performs a brute-force search over the whole policy set and rule set to find those rules applicable to a given request. This is comparable to the evaluation method we used as baseline in our experimental validation (Section 5.3) reflecting formal semantics of the tested policy.

The optimized XACML evaluation engine on the other hand pre-processes the policy set by first categorizing the policies by subject and then dynamically calculating the best order in which the policies in the policy set and the rules in the applicable policies should be evaluated, based on the recent request history.

The request execution problem they want to solve optimally is to find an execution sequence (or ordering) that requires the minimum number of rule evaluations. Following this goal the ordering of rules, i.e., rule permutation, in a given policy \( P_i \) is chosen as the one permutation with the lowest expected cost. The expected cost of a given ordering depends on the cost and position of its rules. The cost of a rule depends on the number and complexity of its Boolean conditions as well as on the permit-, deny- and hit-ratio of the rule. Depending on the rule combining algorithm, the different ratios have different costs (e.g., for permit-overwrite a high permit-ratio is preferred). The authors distinguish between simple atomic Boolean conditions, which are assigned a constant cost, and standard XACML functions, which are assigned a cost corresponding to the estimated average execution time.

As the different weights depend on ratios that may change over time, the best permutations might have to be recalculated at a later time. The proposed approach bases its optimization on access request statistics and is therefore well-adapted to the specific environment it is deployed in. In contrast to our approach, however, it only applies local optimizations without considering the optimality of the whole policy.

The restriction of optimizing the overall policy on predefined policy levels (rule, policy, policy set) has the additional drawback that only a linear execution order can be specified. Our method, on the other hand, defines an execution tree depending on intermediate evaluation results and can therefore guarantee the optimal execution order with respect to the whole set of possible requests.
2. Related Work

2.1.2 Refactoring Access Control Policies for Performance Improvement

El Kateb et al. [10] propose an automated approach for refactoring a global access control policy into smaller policies (consisting of fewer rules) for performance improvement at request evaluation.

Their approach addresses the issue of a performance bottleneck in the case where the access control policy consists of a large number of rules which have to be processed in real time to find all the applicable rules to a given request. They propose an automated policy refactoring by means of splitting such a policy into its corresponding multiple policies with a smaller number of rules, each. Additionally to the idea of policy splitting, El Kateb et al. [10] propose a set of splitting criteria and examine their effectiveness in experiments.

At request evaluation the attributes of a request are compared to the attribute values in the target of a rule. Only rules for which the target attributes match the request attributes, i.e., rules which are applicable to the request, are relevant rules for this request. The proposed approach aims at reducing the evaluation time of a request by only evaluating a request against its relevant rules. The proposed splitting criteria as presented in the paper are shown in Table 2.1.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Splitting Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SC_1)</td>
<td>((\text{Subject}), (\text{Resource}), (\text{Action}))</td>
</tr>
<tr>
<td>(SC_2)</td>
<td>((\text{Subject}, \text{Action}), (\text{Subject}, \text{Resource}), (\text{Resource}, \text{Action}))</td>
</tr>
<tr>
<td>(SC_3)</td>
<td>((\text{Subject}, \text{Resource}, \text{Action}))</td>
</tr>
</tbody>
</table>

Table 2.1: Splitting Criteria according to attribute element combinations

For a given splitting criteria, e.g., \(SC_1 = (\text{Subject})\), the presented algorithm parses the global policy for all corresponding attribute values in the targets of policy rules. For each collected attribute value \(a\), a new policy \(P_a\) is generated only consisting of rules containing \(a\) as an attribute value in the rule target.

During request evaluation, for a given request, only the relevant policy is selected and therefore a much smaller set of rules needs to be checked for appliance to the request. Like in our solution, if the initial policy is changed, the refactoring needs to be repeated for the new policy.

A benefit of this automated approach to policy refactoring is the reduction of considerable amount of (tedious and error-prone) human efforts. This is an important gain which is achieved by our method as well.
2.2. Firewall Filtering

The approach has been tested for XACML policies evaluated on a Sun PDP as well as on an XEngine, showing high effectiveness of the splitting in reducing evaluation time on both PDPs. However, it only focuses on the first part of XACML request evaluation, which is the search for applicable rules. The evaluation of the rules and policies for a given request is not considered at all. Since this pre-processing stage of checking applicability of rules is very XACML-specific, the approach can not easily be generalized to other types of policy specification languages.

2.1.3 Automated XACML Policy Reconfiguration for Evaluation Optimisation

Miseldine [15] present a programmatic approach to reconfigure access control rules of XACML policies for optimized evaluation.

The problem handled in this paper is informally described as the ambiguity of how data is represented in XACML. As there are many ways to express the same access control policy in XACML they aim to find the optimal representation in terms of evaluation performance in an automated fashion. One goal thereby is, like in [10], to minimize the set of rules that need to be checked for applicability.

The two main optimization techniques applied in [15] are policy splitting, which groups rules by applicability similar to the method proposed in [10], and rule reordering based on a cost function derived from request heuristics.

2.2 Firewall Filtering

A related area where similar optimization techniques can be applied is firewall filtering. The method of rule reordering is very common in this area.

Hamed and Al-Shaer [12] present an approach based on statistical frequencies with which different policy rules are matched to a request. The ordering scheme splits the filtering policy into two sets, where the first set only contains a small number of very frequent (active) rules, ordered by their frequency. This reordering improves request evaluation performance, as it minimizes the search path for a matching rule for the most common request types.

The described approach is adaptive, i.e., the most frequently matched rules, which might change for a changing environment, are moved to the first set, as soon as they exceed a certain threshold. Similarly rules in the first set get moved to the second set as they are matched fewer times.
2. Related Work

In contrast to our method, the reordering proposed here only relies on request frequencies and does not take into account dependencies of different rules, which can shorten the search path for a matching rule as well.

2.3 MDP in Intrusion Detection


Because of system limitations like insufficient memory and battery power of sensors, existing IDS methods are unsuited for sensor networks. They both propose an IDS method based on an MDP, which observes the system, learns the behavior of an attacker and decides to protect some sensor nodes according to the expected future behavior of the attacker.

In contrast to our method, where MDP is applied to precompute an optimal encoding of the existing access control policy, the approaches presented here are adaptive and react on changing environments and attacker models. Precomputing an optimal solution, on the other hand, saves a great amount of online computing, which also is a preferable property in sensor networks.
Chapter 3

Preliminaries

3.1 PBel

PBel [5] is a formal access control policy language which is independent of specific application domains. Its main strength lies in its algebraic approach to policy specification. A PBel policy is written as an algebraic composition of primitive policies. Its composition operators are based on the operators of the 4-valued Belnap logic [4].

The decision of a PBel policy for a given access request is an element of Belnap’s 4-valued logic, which consists of the following truth values grant \( t \), deny \( f \), gap \( \perp \), and conflict \( \top \). The additional truth values \( \perp \) and \( \top \) indicate that a policy does not have enough information, or has conflicting information, to grant or deny a request, respectively.

The PBel language is expressive in that its core language operators can express any operator for decision composition. For example, it can express XACML operators, as well as any other 4-valued policy language. Furthermore, the language is independent of its application domain, meaning that the decision for an access request only depends on the request and on the conditions it satisfies. The evaluation of these conditions is carried out by a given access control model.

3.1.1 PBel’s Decision Space

PBel’s policy decision space is based on the Belnap truth space \( 4 = \{ t, f, \perp, \top \} \). Belnap’s truth space can be described with two different lattices:

- The knowledge order \( \leq_k \). Here, \( x \leq_k y \) for \( x, y \in 4 \) means that \( y \) contains at least as much information as \( x \).
3. Preliminaries

- The truth order $\leq_t$. Here, $x \leq_t y$ for $x, y \in 4$ means that $y$ is at least as permissive as $x$.

These two orderings can be composed into a single structure, called bilattice [11], which is defined as follows:

**Definition 3.1 (Bilattice)** A bilattice is a structure $(A, \leq_t, \leq_k, \neg)$, where $A$ is a non-empty set, and $\leq_t$ and $\leq_k$ are partial orders on $A$ such that $(A, \leq_t)$ and $(A, \leq_k)$ are complete lattices.

A complete lattice is a set for which a partial order is defined, such that for every subset there exists a greatest lower bound (meet) and a least upper bound (join).

The negation operator $\neg : A \to A$ only affects truth ordering and has no effect on knowledge ordering, i.e., the following conditions must hold for all $x, y \in A$:

$$
\begin{align*}
  x \leq_t y & \implies \neg y \leq_t \neg x \\
  x \leq_k y & \implies \neg x \leq_k \neg y \\
  \neg \neg x & = x
\end{align*}
$$

Figure 3.1\footnote{Reproduced from [5]} illustrates these two partial orderings.

![Belnap bilattice](image)

Figure 3.1: Belnap bilattice

For the truth ordering $(\leq_t)$ the meet and join operators are denoted by $\land$ and $\lor$, while for knowledge ordering $(\leq_k)$ they are denoted by $\otimes$ and $\oplus$. The definitions of these operators are given in Figure 3.3b, Figure 3.4a, Figure 3.4c and Figure 3.4b.
In [6], formal semantics of PBel are defined in the context of an access control model:

**Definition 3.2** A policy model $M$ is a mapping $M : \text{Req} \times \text{Atoms} \rightarrow \{\text{true}, \text{false}\}$, where Atoms is a set of all possible request conditions over which policies can be defined. Req is a set of constants denoting all possible access requests.

We will refer to a request condition, i.e., a member of Atoms, as **atom**. The set of all possible models is denoted by $\mathcal{M}$. A PBel policy specification is then defined as follows:

**Definition 3.3** A PBel policy specification $\Pi$ is a mapping from requests and the conditions they satisfy, i.e., a given model, into decisions in $\mathcal{M}$, i.e.,

$$\Pi : \text{Req} \times \mathcal{M} \rightarrow \mathcal{M}.$$

Finally, we formally state PBel's domain independence as follows:

**Theorem 3.4** ([5]) For any two models $M, M' \in \mathcal{M}$ and requests $r, r' \in \text{Req}$ it holds that if the two models map each atom to the same value for the given request, then the policy specification evaluates to the same decision for both model-request pairs, i.e.,

$$(\forall \text{atom} \in \text{Atoms} : M(r, \text{atom}) = M'(r', \text{atom})) \rightarrow (\Pi(r, M) = \Pi(r', M')).$$

### 3.1.2 Core PBel Policy Language

The operators implication ($\supset$), conjunction or logical meet ($\land$) and logical negation ($\neg$) together with the basic policy and the constant $\top$ build the functionally complete set of core PBel.

A **basic policy** of PBel consists of an **atom** ($\text{atom}$) and a result value $b \in \{t, f\}$. Atoms build the abstraction layer of PBel which allow domain independent policy design, as they encapsulate domain-specific aspects and only expose the resulting truth values.

The second core operator of PBel is the **implication** operator $\supset$. It extends the classical implication to $\mathcal{M}$.

The semantics of a PBel policy are defined in Figure 3.2. By $\llbracket p \rrbracket(r, M)$ we denote the evaluation of a policy $p$ for a request $r$ in a model $M$, where $p$ is in PBel syntax.

The syntax and semantics of PBel's core operators are shown in Figure 3.2. The truth tables of these five core operators are shown in Figure 3.3.
3. Preliminaries

\[ p, q ::= \text{Policy} \]

\[ b \text{ if } \text{atom} \quad \text{(Basic policy)} \]

\[ [b \text{ if } \text{atom}] (\text{req}, M) = \begin{cases} b & \text{if } M(\text{req,atom}) \\ \perp & \text{otherwise} \end{cases} \]

\[ \top \quad \text{(Conflict)} \]

\[ [\top] (\text{req}, M) = \top \]

\[ \neg p \quad \text{(Logical negation)} \]

\[ [\neg p] (\text{req}, M) = [\neg p] (\text{req}, M) \]

\[ p \land q \quad \text{(Logical meet)} \]

\[ [p \land q] (\text{req}, M) = [p] (\text{req}, M) \land [q] (\text{req}, M) \]

\[ p \supset q \quad \text{(Implication)} \]

\[ [p \supset q] (\text{req}, M) = [p] (\text{req}, M) \supset [q] (\text{req}, M) \]

Figure 3.2: Syntax and semantics of PBel core operators

(a) \( p \supset q \)

(b) \( p \land q \)

(c) \( \neg p \)

(d) \( v \text{ if } \text{atom} \)

(e) \( \top \)

Figure 3.3: Definition of core PBel operators

3.1.3 Extensions to Core PBel

Using the PBel core policy language, many extensions can be defined for convenience of policy writers. Here, we only describe the ones we use in our example policies in Chapter 5. Further extensions are introduced by Bruns et al. [6].

All these additional policy operators can be defined using only PBel core operators. For simplicity however, we reuse previously defined PBel extensions to keep the operator definitions as short as possible. Figure 3.4 describes most of the the operators used for our example policies.

A little more complex, but very useful are the four operators \( p[v \mapsto q] \) for each value \( v \in 4 \). They act as exception handlers and can be described as
3.1. PBel

The truth tables for these four operators are shown in Figure 3.5.

### Figure 3.4: Definition of additional PBel operators

- (a) $p \lor q = \neg (\neg p \land \neg q)$
- (b) $p \oplus q = (p \land \top) \lor (q \land \top) \lor (p \land q)$
- (c) $p \odot q = (p \land \bot) \lor (q \land \bot) \lor (p \land q)$
- (d) $\neg p = (\neg p \lor \bot) \lor (\neg (p \lor \bot))$

### Figure 3.5: Definition of exception handling PBel operators

follows for some model $M$ and some request $r$:

$$\llbracket p[r] \mapsto q \rrbracket (r, M) = \begin{cases} \llbracket q \rrbracket (r, M) & \text{if } \llbracket p \rrbracket (r, M) = v \\ \llbracket p \rrbracket (r, M) & \text{otherwise} \end{cases}$$
The last extension we present here is the following extension of basic policies. We allow atoms to be negated in basic policies, i.e., \((t \text{ if } \neg \text{atom})\).

### 3.1.4 PBel Analysis Procedure

PBel has a powerful analysis procedure, which can decide whether a policy is always evaluated to the same policy decision for a set of requests, that satisfy a given condition in all models. Such a condition could be, that only a subset of the policy’s atoms need to be satisfied in all models. In that case, the policy decision needs to be the same for all requests in the set, regardless of how the remaining atoms are evaluated in the given models and for the given requests. First we define an analysis query as:

**Definition 3.5** A query is defined as \(q := \phi \implies \psi\), where \(\phi\) is a propositional sentence over a set of atoms, and \(\psi\) is defined as the statement \(\Pi(r, M) = v\) for the policy \(\Pi\) and \(v \in \{\text{true}, \text{false}\}\).

To illustrate this, consider the following sample policy:

\[ \Pi_{\text{ex}} = (t \text{ if } a) \lor ((f \text{ if } b) \land (t \text{ if } c)) \]

For this policy, consider a query:

\[ q_{\text{ex}} := (a \land \neg b) \implies (\Pi_{\text{ex}}(r, M) = t). \]

This query states that we require a model and a request which evaluate atom \(a\) to \(\text{true}\) and atom \(b\) to \(\text{false}\) to also satisfy \(\Pi_{\text{ex}}(r, M) = t\). To verify that such a query is true for all models and all requests, we use a PBel analysis procedure.

PBel analysis procedure is formally defined as:

**Definition 3.6** A PBel analysis procedure \(P_{\text{PBel}} : Q \to \{\text{true}, \text{false}\}\) is a function which evaluates a query \(q \in Q\) for all models \(M \in \mathcal{M}\) and all requests \(r \in \text{Req}\).

\[ P_{\text{PBel}}(q) = \text{true} \iff \forall M \in \mathcal{M}, \forall r \in \text{Req} : M \models_r \phi \implies \Pi(r, M) = v, \]

where \(M \models_r \text{atom iff } M(r, \text{atom}) = \text{true}\). This is extended over a propositional sentence in the standard way, i.e., \(M \models_r (a \land b) \iff M \models_r a\) and \(M \models_r b\).

Evaluating the example query \(q_{\text{ex}}\) with the PBel analysis procedure validates the hypothesis that any model-request pair \((M, r)\) which evaluates \(a\) to \(\text{true}\) and \(b\) to \(\text{false}\), satisfies the policy \(\Pi_{\text{ex}}\), i.e., \(P_{\text{PBel}}(q_{\text{ex}}) = \text{true}\). In other words the mapping \(M(r, c)\) does not influence the policy decision, if the first two conditions are satisfied.

PBel’s analysis procedure also allows to define more complex queries (as shown in [6]), which we do not further explore in this work.
3.2 Markov Decision Process

A Markov Decision Process (MDP) consists of an agent, which interacts with its environment. The agent can execute actions to change the state of the environment. When interacting with the environment, the agent chooses an action \(a\) from the set of available actions \(A\), transferring the environment from the current state \(s\) to a successor state \(s'\).

In each state transition, the environment awards the agent with a numerical reward \(R(s,a,s')\). State transitions may be probabilistic in the sense that the successor state \(s'\) is determined not only by the current state \(s\) and the action \(a\) of the agent, but also by a random element. Such probabilistic state transitions need to fulfill the Markov property, which specifies that the probability of state \(s'\) being the successor state of state \(s\) after the agent executed action \(a\) only depends on \(s\) and \(a\) and not on the previous history of state transitions. Formally, an MDP is defined as follows:

**Definition 3.7 (MDP)** An MDP is a 5-tuple \((S,A,P,R,s_0)\), where \(S\) denotes a set of states and \(A\) denotes a set of actions. The state transition function

\[
P: S \times A \times S \rightarrow [0,1]
\]

defines the probability \(P(s,a,s')\) that the environment transitions to the state \(s' \in S\) if the agent performs the action \(a \in A\) while the environment is in the state \(s \in S\). \(P\) has the Markov property. The reward function

\[
R: S \times A \times S \rightarrow \mathbb{R}
\]

returns a numerical reward for any state transition. The state \(s_0 \in S\) denotes the starting state of the environment.

In the following, we use the short hand notation \(A(s)\) to refer to the set of actions available in state \(s \in S\), i.e.,

\[
A(s) = \{a \in A \mid \exists s' \in S: P(s,a,s') > 0\}.
\]

An environment is said to be fully observable if the agent always knows the current state of the environment and has complete knowledge of \(P\) and \(R\).

In the following, we will only consider MDPs for which the state set \(S\) and the action set \(A\) are finite and for which the state transition graph is acyclic.

The agents objective is to maximize its accumulated rewards, i.e., its overall utility. The utility \(U((s_0,a_0,s_1),(s_1,a_1,s_2),\ldots,(s_{n-1},a_{n-1},s_n))\) of a given

\footnote{All definitions given in this section are taken from Russell et al. [18] and Sutton and Barto [19].}
transition history \([s_0, a_0, s_1, (s_1, a_1, s_2), \ldots, (s_{n-1}, a_{n-1}, s_n)]\) is defined as the sum of the rewards received until reaching the state \(s_n\):

\[
U([s_0, a_0, s_1, (s_1, a_1, s_2), \ldots, (s_{n-1}, a_{n-1}, s_n)]) = \sum_{i=0}^{n-1} R(s_i, a_i, s_{i+1})
\]

Since \(S\) and \(A\) are finite and the transition graph is acyclic, the environment will reach a leaf state of the graph after a finite number of actions. In a leaf state, no further actions are possible.

We illustrate the application of an MDP using the example of a simple game we call **Chicken Tic-Tac-Toe**. Chicken Tic-Tac-Toe is a simplified version of the common board game Tic-Tac-Toe, where the opponent is a chicken instead of another human player. It is played on a board divided into four disjoint squares. Each player has grains of a player-specific color. In each round, one player puts a grain of his color into an empty square. The winner is the player that first gets two horizontally or vertically adjacent squares containing a grain of his color. The game starts when the chicken has placed its first grain. In the following we fix the first move of the chicken so that it always places its first grain in the upper left corner. The other three cases are symmetric. After the human player has placed his first grain, the chicken places its second grain uniformly at random in one of the two remaining squares.

We can now model this game as an MDP:

- The agent of this MDP is the playing human, which has to decide where to place his grains.
- The environment consists of the game board with its already placed grains, as well as the chicken.
- Each MDP state \(s\) corresponds to one specific state of the board. There are exactly eight relevant states in this MDP, depicted in Figure 3.6.
- An action \(a\) of the agent in a given state represents the act of placing a grain in one of the empty squares.
- Figure 3.6 also describes the state transitions (arrows between states) and the corresponding probabilities.
- The reward function \(R\) is defined as follows:

\[
R(s, a, s') = \begin{cases} 
-1 & \text{if } s' \text{ is a loosing state, i.e., the chicken wins (red border in the Figure)} \\
1 & \text{if a tie was reached in } s', \text{ i.e., no player wins (green border in the Figure)} \\
0 & \text{otherwise (no border in the Figure)} 
\end{cases}
\]
3.2. Markov Decision Process

Figure 3.6: An MDP model for the chicken tic-tac-toe game

3.2.1 Finding Optimal Solutions for a Markov Decision Process

For a general MDP, a solution is given by a deterministic strategy $\pi(s)$ that defines the behavior of the agent, i.e., an action $a \in A$ to be executed, for every state $s \in S$ it might reach. Because of the stochastic nature of the environment, a strategy $\pi$ might result in a different environment history each time it is executed.

One possible solution for the game modeled in Figure 3.6 is the strategy that chooses action 2 in the start state (a) and actions 3 and 2 in the states (c) and (f), respectively. Following this strategy might lead to one of the two state sequences $[(a), (b)]$ or $[(a), (c), (h)]$ every time the game is played.

The expected utility $u_\pi(s)$ of a state $s$ for an agent following a given strategy $\pi$ is defined as the accumulated rewards the agent can expect to collect in the future when starting in state $s$ and following $\pi$. It can be calculated as follows:

$$u_\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot (R(s, \pi(s), s') + u_\pi(s'))$$

Note that for any leaf state $\hat{s}$ we have, $u_\pi(\hat{s}) = 0$.

The quality of two strategies $\pi, \pi'$ is compared via the expected utilities of all states for an agent following the specified strategy. We say that $\pi \geq \pi'$ if for all states $s \in S$, we have

$$u_\pi(s) \geq u_{\pi'}(s).$$

In every MDP, there is at least one strategy that produces a better or equal expected utility than every other strategy. This strategy is called an optimal strategy $\pi^*$.
Clearly, choosing action 3 in the start state (a) of the example will not maximize the expected utility of the agent, as both successor states are loosing states. The expected utility of the starting state for any strategy that first chooses action 3 is -1. On the other hand, every possible strategy in this example that avoids this mistake is an optimal strategy. The expected utility of the starting state for these strategies is $u_{\pi^*}(s_0) = 0$.

In this thesis we will use an algorithm called Value Iteration for finding the optimal strategy to a given MDP. Other methods and algorithms are described in [18].

Value iteration is an algorithm that calculates the optimal strategy $\pi^*$ by iterative refinement. It initializes the variables $\text{action}(s)$ and $\text{util}(s)$ arbitrarily for all states $s \in \mathcal{S}$. It then updates the variables as follows, until convergence is reached:

$$
\text{action}(s) \leftarrow \arg \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} P(s,a,s') \cdot (R(s,a,s') + \text{util}(s'))
$$

$$
\text{util}(s) \leftarrow \sum_{s' \in \mathcal{S}} P(s,\text{action}(s),s') \cdot (R(s,\text{action}(s),s') + \text{util}(s'))
$$

Let $\text{action}_i(s)$ and $\text{util}_i(s)$ denote the values of $\text{action}(s)$ and $\text{util}(s)$ after $i$ iterations, respectively. Then the variables $\text{action}_i(s)$ and $\text{util}_i(s)$ are updated using $\text{action}_{i-1}(s')$ and $\text{util}_{i-1}(s')$ of the successor states $s'$ of $s$.

Since for any leaf state $s_l$ and any strategy $\pi$ we have $u_{\pi}(s_l) = 0$, the values of $\text{action}(s_l)$ and $\text{util}(s_l)$ converge after only one iteration, i.e., $\text{util}_1(s_l) = 0$ and $\text{action}_1(s_l)$ is the empty action.

Remember that we only consider MDPs with finite acyclic transition graphs. Given such an MDP, the values of $\text{action}_k(s)$ and $\text{util}_k(s)$ converge to $\pi^*(s)$ and $u_{\pi^*}(s)$ after a finite number $k$ of iterations for any state $s \in \mathcal{S}$, respectively.

Further, the distance from $s$ to its farthest successive leaf state is an upper bound for the number $k$ of iterations required.
Translating a PBel Policy into an Efficient Evaluation Procedure

This chapter describes our theoretical contribution. It consists of three parts:

**Mapping Access Control Policy Evaluation into the MDP Setting** shows how the theory of MDP can be applied to access control problems. It illustrates what roles the different elements of access control play in the MDP context. Further, it gives a formal definition of the MDP elements used in this chapter and their relation to the defined access control elements.

**Algorithm Overview** describes in more detail the different stages of the translation algorithm. It shows how, from a PBel policy specification, an MDP structure is generated and how an optimal strategy is calculated which is then used by the efficient evaluation procedure. Finally we discuss the dependency of the presented method on the PBel policy language.

**Analysis of the Solution** reasons about correctness of the construction algorithm and optimality of the resulting evaluation procedure. It also discusses the analysis of the asymptotic space complexity of our construction algorithm.

### 4.1 Mapping Access Control Policy Evaluation into the MDP Setting

The goal of this Section is to map the process of access control policy evaluation to a corresponding MDP structure given by the 5-tuple \((S, A, P, R, s_0)\). In order to define an MDP structure and its containing elements in this context, we need the formal definition of an access control policy.
4. Translating a PBel Policy into an Efficient Evaluation Procedure

Definition 4.1 An access control policy $\mathcal{P} = \langle \Pi, \mathcal{A}_\Pi \rangle$ consists of a PBel access control policy specification $\Pi$ and the set of atoms $\mathcal{A}_\Pi$ used in $\Pi$.

Recall that by atom we denote a request condition specified by the policy.

As an example, consider the PBel policy specification

$$\Pi = (\text{t if } a) \lor (\text{t if } b)[\bot \rightarrow f].$$

Here, $\mathcal{A}_\Pi = \{a, b\}$.

In the process of policy evaluation, the atoms defined in $\mathcal{P}$ may assume different values. In each step of the evaluation process, one unevaluated atom is chosen to be evaluated. The evaluation determines the atom’s value which is either true or false.

Every atom evaluation changes the overall state of the policy evaluation process. We describe the state $s$ of the policy evaluation process using the function $\rho_s: \mathcal{A}_\Pi \rightarrow \{\text{true}, \text{false}, \text{pending}\}$. For every atom $\overline{a}$ already evaluated in $s$, $\rho_s(\overline{a})$ returns its value. For an unevaluated atom $\overline{a}_u$, $\rho_s(\overline{a}_u) = \text{pending}$. The set of all possible states $s$ of the policy evaluation process we denote by $S$.

In the above example policy, there are 9 different states of the policy evaluation process, which are listed here:

$$S = \{(\text{pending, pending}), (\text{true, pending}), (\text{false, pending}), (\text{pending, true}), (\text{pending, false}), (\text{true, true}), (\text{true, false}), (\text{false, true}), (\text{false, false})\}$$

In every state of the policy evaluation process, we can check the policy for decidability by using the PBel analysis procedure $P_{\text{PBel}}$ for the following type of queries:

$$q_v = \text{prop}(\rho_\mathcal{P}) \implies \Pi(r, M) = v,$$

for every $v \in 4$, where $\text{prop}(\rho_\mathcal{P})$ is the propositional sentence asserting that every atom $\overline{a}$ evaluated in $s$ evaluates to $\rho_\mathcal{P}(\overline{a})$ in every model and for every request for which the second condition of the query is evaluated.

$$\text{prop}(\rho_\mathcal{P}) = \left( \bigwedge_{a_i \in \mathcal{A}_\Pi; \rho_\mathcal{P}(a_i) = \text{true}} a_i \right) \land \left( \bigwedge_{b_i \in \mathcal{A}_\Pi; \rho_\mathcal{P}(b_i) = \text{false}} \neg b_i \right)$$

In other words, $\text{prop}(\rho_\mathcal{P})$ is a filter to ensure that only requests which could have led to the current state $s$ of the policy evaluation process, are considered in the analysis of the policy.

We define an evaluation function $\text{eval}$ that evaluates the policy $\Pi$ in a given state $s$ of the policy evaluation process. The evaluation function evaluates the
4.1. Mapping Access Control Policy Evaluation into the MDP Setting

analysis procedure \( P_{PBel} \) for all different queries \( q_v \), where \( v \in 4 \). It returns the decision value of \( \Pi \), if \( \Pi \) is decidable for the given atom evaluation values defined by \( \rho_s \), or \textit{pending} otherwise.

**Definition 4.2** For a given PBel policy specification \( \Pi \) and a given atom-value mapping \( \rho_s \), the evaluate function is defined as

\[
eval(\Pi, \rho_s) = \begin{cases} 
  t & \text{if } P_{PBel}(q_t) \\
  f & \text{if } P_{PBel}(q_f) \\
  \bot & \text{if } P_{PBel}(q_\bot) \\
  \top & \text{if } P_{PBel}(q_\top) \\
  \text{pending} & \text{otherwise}
\end{cases}
\]

This allows us to formally define an evaluation process state as follows:

**Definition 4.3** A state \( \bar{s} \in S \) of the policy evaluation process is a 3-tuple \( \langle \rho_s, \text{dec}, \mathcal{P} \rangle \). Here, \( \mathcal{P} \) denotes the policy which is evaluated in this policy evaluation process. The mapping function \( \rho_s : A_\Pi \rightarrow \{ \text{true}, \text{false}, \text{pending} \} \) returns for every atom in \( A_\Pi \) of \( \mathcal{P} \) a value in \( \{ \text{true}, \text{false}, \text{pending} \} \) in the current state \( \bar{s} \) of the policy evaluation process. The decision of an evaluation process state \( \bar{s} \) is given by its decision value \( \text{dec}_s = \eval(\Pi, \rho_s) \).

Once the policy evaluation process reaches a state \( \bar{s} \) which has a decision different from \textit{pending}, the policy evaluation terminates, as the policy is decidable for the set of evaluated atoms.

We can now map states of the policy evaluation process to MDP states as follows.

**Definition 4.4** For every policy evaluation process state \( \bar{s} \in S \), there is a unique corresponding MDP state \( s \). The set of MDP states \( S \) defining an MDP is then the set of MDP states corresponding to all possible states \( \bar{s} \in S \) of the policy evaluation process.

If \( \bar{s} \) has a decision value \textit{pending}, \( s \) is called an undecided state. Otherwise, \( s \) is called a decided state. In the MDP the starting state \( s_0 \) corresponds to the state \( \bar{s}_0 \) of the policy evaluation process, where no atom was evaluated, i.e., \( \rho_{\bar{s}_0}(\bar{a}) = \text{pending} \) for every atom \( \bar{a} \in A_\Pi \).

The evaluation steps, which choose an atom of value \textit{pending} in the current states of the policy evaluation process and evaluate the atom, can be mapped to the corresponding MDP actions. The set of successor actions \( A(s_u) \) for a given undecided MDP state \( s_u \in \mathcal{S} \) is the set of all actions that choose atoms for which \( \rho_{\bar{s}_u} = \text{pending} \). For a decided MDP state \( s_d \in \mathcal{S} \) the set of available actions \( A(s_d) \) is the empty set \( \emptyset \). Action \( a \in A(s) \) corresponds to choosing atom \( \bar{a} \in A_\Pi \) for evaluation in the policy evaluation process state \( \bar{s} \in S \).
An atom $\bar{a} \in A_{\Pi}$ has a given probability of evaluating to true, and the counter probability of evaluating to false. We assume these probabilities to be 50%. Based on these probabilities we can define the state transition from one MDP state $s \in S$ to another MDP state $s' \in S$ using the action $a \in A(s)$ as follows:

$$P(s, a, s') = \begin{cases} 
0.5 & \text{if } (s \prec_a s') \wedge (\rho_\bar{a}(\bar{a}) = \text{pending}) \wedge (\text{dec}_s = \text{pending}) \\
0 & \text{otherwise}
\end{cases}$$

For two MDP states $s, s' \in S$ and some action $a \in A$ we say that $s \preceq_a s'$ iff

$$\forall \bar{a}' \neq \bar{a} \in A_{\Pi}: (\rho_\bar{a}(\bar{a}') = \rho_{\bar{a}'}(\bar{a}')) \wedge (\rho_\bar{a}(\bar{a}) = \text{pending})$$

and $s \prec_a s'$ iff additionally

$$\rho_{\bar{a}'}(\bar{a}') \neq \text{pending}.$$ 

The transitive closure over $\preceq_a$ and $\prec_a$ is denoted by $\preceq_{\text{trans}}$ and $\prec_{\text{trans}}$, respectively.

In the policy evaluation process every atom $\bar{a} \in A_{\Pi}$ has an atom-specific cost $c(\bar{a}) \in \mathbb{R}$, which represents the estimated time spent evaluating $\bar{a}$. Using this cost we can define the reward function $R(s, a, s')$ of a given state transition as the negative cost of evaluating the atom $\bar{a}$ chosen by the action $a \in A(s)$, i.e.,

$$R(s, a, s') = -c(\bar{a}).$$

The MDP structure corresponding to the evaluation of the example PBel policy

$$\Pi = ((\text{if } a) \lor (\text{if } b))[\bot \rightarrow \text{f}]$$

is shown in Figure 4.1. We used the symbol ‘?’ in the MDP state assignments as a short hand for the value pending and symbols ‘t’ and ‘f’ as a short hand for values true and false, respectively. Decision states are marked by double borders, where a green border represents a decision state $s_d$ for which the decision value $\text{dec}_s$ of the corresponding policy evaluation process state $\overline{s_d}$ is t and a red border represents a decision state for which $\text{dec}_s = f$.

Note that the mapping described in this section only produces an MDP structure, with an acyclic state transition graph. This property is guaranteed because no MDP state $s \in S$ can be its own transitive successor state, i.e., there is no state $s$ such that $s \prec_{\text{trans}} s$. This follows immediately from the definition of $\prec_a$ (for any $a \in A$).
4.2 Algorithm Overview

The translation of a PBel access control policy specification $\Pi$ into an evaluation procedure, consists of the following key steps:

1. Construct the MDP structure according to the mapping described in Section 4.1, using an implementation of the $eval(\Pi, \rho_s)$ evaluation function to precompute a decision for every state $s$ of the policy evaluation process corresponding to a reachable MDP state $s$.

The procedure that builds this structure consists of two main parts:

- the initialization of the start state $s_0$ of the MDP structure.
- the recursive function taking an unfinished MDP structure and creating another possibly unfinished MDP structure.

2. Calculate the optimal strategy by performing value iteration on the constructed MDP structure. This method is implemented in the procedure $find \text{ } optimal \text{ } strategy$ which calculates the best action $\pi^*(s) \in A(s)$ in each MDP state $s \in S$ based on the expected utility $u_{\pi^*}(s)$.

Finally, evaluating a specific access request corresponds to traversing the MDP structure, always choosing the actions recommended by the optimal strategy.
4.2.1 Constructing the MDP Structure for a PBel Policy

The construction of the MDP structure starts by creating the MDP start state $s_0$ which is identified by its assignment where $\rho_{s_0}(\bar{a}) = \text{pending}$ for every $\bar{a} \in A_{II}$. We call the mapping of all atoms $\bar{a} \in A_{II}$ to the values $\rho_s(\bar{a})$ in an MDP state $s$ corresponding to the state $\bar{s}$ of the policy evaluation process the state assignment of $s$.

When a new MDP state $s_{new}$ is created, the policy II is evaluated for the new state assignment of $s_{new}$, i.e., $\text{eval}(\Pi, \rho_{s_{new}})$ is called. The evaluation result can fall into one of the following two cases:

- If the evaluation returns a decision value $v \in 4$, this means that the policy is decidable for the set of evaluated atoms in $s_{new}$ and no further atom evaluation can change this decision. In this case, $s_{new}$ is a decided state.
- If $\text{pending}$ is returned by the evaluation function, this means that more atoms need to be evaluated for a decision to be possible and therefore $s_{new}$ is an undecided state.

In a next step, the MDP starting state $s_0$ is used as the base MDP structure from which the rest of the MDP structure is constructed. The recursive construction function $\text{constructMDP}$ takes an unfinished MDP structure $T_i$ as input and outputs the finished MDP structure $T_o$ as sketched below.

\[
\text{constructMDP}(T_i) :
\]
\[
T_o \leftarrow \text{generate}(T_i)
\]
\[
\text{if } T_o = T_i
\]
\[
\text{return } T_i
\]
\[
\text{else}
\]
\[
\text{return } \text{constructMDP}(T_o)
\]

The $\text{generate}(T_i)$ function takes an unfinished MDP structure as input and adds all new states to the MDP, which are direct successor states of undecided leaf states in $T_i$. If $T_i$ is the empty MDP structure, the new MDP structure returned by $\text{generate}(T_i)$ is the MDP structure only containing the starting state $s_0$. Otherwise, the generation adds all MDP states $s'$ to $T_i$, for which $s \prec_a s'$ for some undecided state $s$ and some action $a \in A$ such that $\rho_{\bar{s}}(\bar{a}) = \text{pending}$, if $s'$ is not already contained in $T_i$.

The procedure implementing this recursive construction process, after the generation of the MDP starting state $s_0$ is sketched in Algorithm 4.1.

This procedure takes a parent MDP state as input and outputs all child actions for this MDP state, if any. The initialization of an action is done by the action constructor $\text{MDPAction}$, which is shown in Algorithm 4.2.
4.2. Algorithm Overview

**Algorithm 4.1** create\_MDP\_successive\_actions(state)

```python
action_set = new Set()

if (state.decision == pending):
    #create actions for each unevaluated atom in the state assignment.
    for (atom in state.assignment):
        if (atom.assignment == pending):
            action_id = next_id(state.mdp_actions)
            suc_states = create\_successive\_MDP\_states(atom, state.assignment)
            action = MDPAction(atom, action_id, suc_states)
            action_set = add\_action\_to\_set(action_set, action)

return action_set
```

**Algorithm 4.2** MDPAction(atom, id, suc\_states)

```python
action = new Action()

#core
action.atom = atom
action.id = id
action.visited = false

#true state
action.true_state = suc\_states[0]
action.t.prob = 0.5

#false state
action.false_state = suc\_states[1]
action.f.prob = 0.5

return action
```
The MDP state given as input to the procedure can be categorized in the following way:

- In a particular undecided state every atom in the state assignment which was not evaluated so far, could be chosen to be evaluated next. Therefore every undecided state \( s_u \) has one successor action for every atom \( \bar{a} \) for which \( \rho_{s_u}(\bar{a}) = \text{pending} \).

- If the MDP state is a decided state, no child actions are created and therefore the MDP state is a leaf node of the MDP structure. In this case the recursive procedure returns immediately.

Each action \( a \) has two successor states, one where the action-specific atom \( \bar{a} \) is evaluated to \( \text{true} \), i.e., \( \rho(\bar{a}) = \text{true} \), and one where it is evaluated to \( \text{false} \). If there already exists an MDP state with that particular state assignment, the action links to the existing MDP state. Otherwise, a new MDP state is created. The procedure responsible for creating these new MDP states is sketched in Algorithm 4.3. The probability for each successor MDP state is encoded in the action and is set to 0.5. The probability of the true state and the false state in an action is the probability of the action-specific atom to evaluate to \( \text{true} \) or \( \text{false} \), respectively.

**Algorithm 4.3 create_successive_MDP_states(atom, prev_assignment)**

```plaintext
#fill in new assignments
(true_a, false_a) = find_new_assignments(atom, prev_assignment)

#fetch state for true_assignment, if it exists and create it otherwise
if (state_exists(true_a)):
    true_state = get_state(true_a)
else:
    true_state = MDPState(true_a)
t_actions = create_MDP_successive_actions(true_state)
true_state = add_actions_to_state(true_state, t_actions)

#fetch state for false_assignment, if it exists and create it otherwise
if (state_exists(false_a)):
    false_state = get_state(false_a)
else:
    false_state = MDPState(false_a)
f_actions = create_MDP_successive_actions(false_state)
false_state = add_actions_to_state(false_state, f_actions)

return [true_state, false_state]
```
4.2. Algorithm Overview

Only when the action is created and its successor states have been linked to it, it is added to the list of child actions of the parent state. The initialization of a state node is done by the state constructor `MDPState`, which is shown in Algorithm 4.4.

**Algorithm 4.4 MDPState(assignment)**

```
state = new State()
state.assignment = assignment
state.expected_utility = 0

# calculate if the state is a decision state
decision_set = evaluate(policy, assignment)
if (len(decision_set) == 1):
    state.decision = decision_set[0]
else:
    state.decision = pending

return state
```

This recursive process terminates when every state of the MDP structure which has no child actions is a decided state and therefore a leaf state of the MDP structure and when every non-leaf state \( s \) has as many child actions as its state assignment has atoms \( \pi \) for which \( \rho_\pi(\pi) = pending \).

We prove the correctness of this construction in subsection 4.4.1.

### 4.2.2 Calculating the Optimal Strategy

After construction of the MDP structure, a procedure implementing value iteration called `find_optimal_strategy` calculates the best successor action for each state. Remember that the quality of a strategy \( \pi \) for a given MDP compared to another strategy \( \pi' \) is defined via the expected utilities of all states for an agent following \( \pi \) or \( \pi' \), respectively.

The expected utility \( u_\pi(s) \) of a given MDP state \( s \) can be interpreted as the negative expected time left in the policy evaluation process until a decision can be made, if atom \( \overline{\pi}(s) \) is evaluated next. The optimization algorithm has to find the strategy \( \pi^* \) which minimizes this time for each state.

The optimization algorithm is implemented as a recursive procedure, which calculates the expected utility of a state when choosing a particular successor action. The action resulting in the highest expected utility is then used by the optimal strategy \( \pi^* \) for the MDP state. In the implementation, we use a
field *best_action* in every MDPState object to store the optimal strategy $\pi^*$. The procedure is sketched in Algorithm 4.5.

**Algorithm 4.5 find_optimal_strategy(state)**

1. if (state.decision == pending):
   1. if (state.visited == false):
      1. state.visited = true

      #calculate expected utility when choosing action
      for (action in state.mdp_actions):
         utilities[action.id] = calculate_utility(action)

      #pick action of highest expected utility as best action
      max_utility = max(utility)
      state.best_action = get_action_from_id(index_of(utilities, max_utility))

   else:
      #decided state
      max_utility = 0

      return max_utility

The expected utility of each state when choosing a particular action $a$ is calculated as the weighted average of $a$’s successor states’ expected utilities minus the cost of evaluating the atom $\bar{a}$. These costs are given to the algorithm as policy meta-data by the policy author. The calculations of the optimal strategy are shown in Algorithm 4.6.

**Algorithm 4.6 calculate_utility(action)**

1. #calculate utilities of successive states
   t_utility = find_optimal_strategy(action.true_state)
   f_utility = find_optimal_strategy(action.false_state)

1. #calculate weighted average of successive state utilities
   expected_utility = action.t_prob*t_utility + action.f_prob*f_utility

1. #subtract action cost from expected_utility
   expected_utility = expected_utility - action.atom.cost

   return expected_utility

In this recursive process, the algorithm again traverses the whole MDP struc-
4.3. Dependency on PBel

ture and then propagates the corresponding utilities from the leaf nodes up to the MDP starting state. The utilities of decided states are 0, as a decision was already reached. The utility of an undecided state is the negative average cost of all optimal actions in the sub-structure of this MDP state.

Recall our example MDP structure in Figure 4.1 for the PBel policy

$$\Pi = (\text{t if } a) \vee (\text{t if } b) | \bot \rightarrow f$$

If we assume atom costs $c(a) = 1$ and $c(b) = 2$ we can calculate the optimal strategy $\pi^*$ as described in Algorithm 4.5:

- The algorithm starts by checking, if the starting state ‘??’ is a decided state. As it is not, the algorithm calculates the expected utility for each successor action and then picks the one as best action which maximizes this utility.

- The utilities are calculated by again applying find_optimal_strategy to every successor state of ‘??’ for the given action, i.e., for action $a$ find_optimal_strategy('t?') and find_optimal_strategy('f?') are called. As ‘t?’ is a decided state, its expected utility, which is 0, is returned immediately. The calculations for the expected utility of ‘f?’ proceed in the same way.

- Given that the state ‘f?’ only has one successor action $b$ and both successor states are decided states, the expected utility returned by find_optimal_strategy('f?') is -2.

- Then the expected utility for state ‘??’ when choosing action $a$ is -2, which is returned by calculate_utility(a). For action $b$ this expected utility is calculated analogously. The call calculate_utility(b) returns -2.5 in state ‘??’.

This results in an optimal strategy $\pi^*$, where $\pi^*(??) = a$, $\pi^*(f?) = b$, and $\pi^*(f?) = a$ are defined by the best_action of every state (see Figure 4.2).

4.3 Dependency on PBel

The translation presented in Section 4.1 and the following correctness proofs in subsection 4.4.1 focus on access control policies specified in the PBel access control policy language. However, the idea of mapping an access control policy evaluation process to an MDP structure and calculating an optimal strategy for the generated MDP structure is not restricted to policies specified in PBel. Our solution could be applied to any access control policy written in a formal access control language which is independent of its application domain and provides a procedure for determining whether a policy
4. Translating a PBel Policy into an Efficient Evaluation Procedure

Figure 4.2: The optimal strategy in the MDP structure for the PBel policy $\Pi = ((t \text{ if } a) \lor (t \text{ if } b))[\bot \rightarrow f]$

can reach a decision in all models and for all requests, which satisfy some subset of conditions.

4.4 Analysis of the Solution

In this section we prove the correctness of the MDP construction (subsection 4.4.1) and of the computed solution strategy (subsection 4.4.2).

The goal is to show, that given any PBel policy specification $\Pi$ our algorithm computes an MDP strategy $\pi^*$, which always leads to the same decision as the original policy. Further, we show that under the given assumptions the calculated strategy satisfies the efficiency requirement.

4.4.1 Correctness of the MDP Construction

Soundness

To prove soundness we show that every decision made by the evaluation function in a given MDP state $s$, for a request corresponding to $\rho_s$, is also the decision made by the PBel policy in the corresponding policy for that request:
Theorem 4.5 For every state \( s \) in the MDP structure generated from \( \Pi \), for all models \( M \in \mathcal{M} \) and all requests \( r \in \text{Req} \),

\[
\text{if } \text{dec}_s = v \text{ and } M \models_r \text{prop}(\rho_s) \text{ then } \Pi(r, M) = v
\]

where \( v \in 4 \) and \( s \) is the state of the policy evaluation process corresponding to \( s \).

Proof. The decision value in any state \( \overline{s} \) of the policy decision process is calculated by the evaluation function \( \text{eval} \), which uses the PBel analysis procedure to prove that the property defined by \( q_0 := \text{prop}(\rho) \implies \Pi(r, M) = v \) (for any \( v \in 4 \)) is satisfied in \( \overline{s} \) for every model \( M \) and every requests \( r \).

Now, we show that every decision present in the MDP structure cannot be changed by evaluating more atoms in the given decided state. More precisely, we prove the following property:

In every decided state \( s_d \) of the MDP structure for which there are unevaluated atoms in the corresponding state \( s \) of the policy evaluation process, i.e., \( \rho_{s_d}(\overline{a}) = \text{pending} \), every action \( a \in A \) choosing such an unevaluated atom \( \overline{a} \in A_{s_d} \) for evaluation can only produce state transitions \( (s_d, a, s_x) \) where \( s_x \) is a decided state. Moreover, the decision value \( \text{dec}_{s_d} \) of the corresponding state \( s_x \) of the policy evaluation process is equal to the decision value \( \text{dec}_{s_d} \) of \( s_d \).

Recall the definition of a decided state:

If a state \( \overline{s} \) of the policy evaluation process has a decision value \( \text{dec}_s = \text{pending} \), the MDP state \( s \) corresponding to \( \overline{s} \) is called an undecided state. Otherwise, \( s \) is called a decided state.

The decision value of \( \overline{s} \) is given by \( \text{dec}_s = \text{eval}(\Pi, \rho_{s_d}) \). If \( \text{dec}_{s_d} \neq \text{pending} \), i.e., if \( \text{eval}(\Pi, \rho_{s_d}) = v \) for some \( v \in 4 \), then the query

\[
q_0 := \text{prop}(\rho_{s_d}) \implies \Pi(r, M) = v
\]

is satisfied in every model \( M \in \mathcal{M} \) and for every request \( r \in \text{Req} \) by the definitions of \( \text{eval} \) and the PBel analysis procedure \( P_{\text{PBel}} \), i.e., the following statement holds:

\[
\exists v \in 4, \forall M \in \mathcal{M}, \forall r \in \text{Req} : M \models_r \text{prop}(\rho_{s_d}) \rightarrow \Pi(r, M) = v.
\]

In other words, \( \Pi(r, M) = v \) in all models \( M \) and for all requests \( r \) which satisfy \( M \models_r \text{prop}(\rho_{s_d}) \). These are all \( M \) and \( r \), for which

\[
\forall \overline{a} \in A_{s_d} : \rho_{s_d}(\overline{a}) \in \{\text{true, false}\} \rightarrow M(r, \overline{a}) = \rho_{s_d}(\overline{a}). \tag{4.6}
\]

In the following we prove that for all \( s_x \in \mathcal{S} \) it holds that

\[
(\exists a \in A : (\rho_{s_d}(\overline{a}) = \text{pending}) \land (s_d \prec_a s_x)) \rightarrow (\text{eval}(\Pi, \rho_{s_d}) = \text{eval}(\Pi, \rho_{s_d})) \tag{4.7}
\]
Let $D$ be the set of all the pairs $(M, r)$ which satisfy Equation 4.6, i.e., for which $M \models r \text{ prop}(\rho_{\overline{r}})$ holds. Let $X$ be the set of all the pairs $(M, r)$ which satisfy Equation 4.8, i.e., for which $M \models r \text{ prop}(\rho_{s \overline{x}})$ holds.

\[
\forall \overline{a} \in A_{\Pi} : M(r, \overline{a}) = \rho_{\overline{r}}(\overline{a}) \quad (4.8)
\]

Then, proving Equation 4.7 can be reduced to proving that $D$ is a superset of $X$ as then Equation 4.9 follows immediately.

\[
(\forall (M, r) \in D : \Pi(r, M) = v) \rightarrow (\forall (M, r) \in X : \Pi(r, M) = v) \quad (4.9)
\]

Let the set $A_{\overline{x}}$ be the set of all atoms $\overline{a} \in A_{\Pi}$ for which $\rho_{\overline{x}}(\overline{a}) \in \{true, false\}$. Using this we can rewrite Equation 4.6 and Equation 4.8 respectively to

\[
\forall \overline{a} \in A_{\Pi} : M(r, \overline{a}) = \rho_{\overline{r}}(\overline{a}), \quad (4.10)
\]
\[
\forall \overline{a} \in A_{\Pi} : M(r, \overline{a}) = \rho_{\overline{x}}(\overline{a}). \quad (4.11)
\]

From the definition of $\prec$ it follows that

\[
\forall s, s' \in S, \forall \overline{a} \in A_{\overline{x}} : (\exists a \in A(s) : s \prec_a s') \rightarrow (\rho_{\overline{x}}(\overline{a}) = \rho_{\overline{x}}(\overline{a})).
\]

Finally we can decompose Equation 4.11 into the two sub statements shown in Equation 4.12 and Equation 4.13, where every pair $(M, r)$ needs to satisfy both equations, to satisfy Equation 4.11.

\[
\forall \overline{a} \in A_{\Pi} : M(r, \overline{a}) = \rho_{\overline{r}}(\overline{a}), \quad (4.12)
\]
\[
\forall \overline{a} \in A_{\Pi} \setminus (A_{\Pi} \cap A_{\overline{x}}) : M(r, \overline{a}) = \rho_{\overline{x}}(\overline{a}). \quad (4.13)
\]

As Equation 4.12 is equivalent to Equation 4.10, every pair $(M, r)$ which satisfies Equation 4.12, also satisfies Equation 4.10 and therefore the set $D$ of all pairs $(M, r)$ satisfying Equation 4.10 and therefore satisfying Equation 4.6, is a superset of the set $X$ of all pairs $(M, r)$ satisfying Equation 4.12 and Equation 4.13 and therefore Equation 4.11 and therefore Equation 4.8.

As $D$ and $X$ are the set of all pairs $(M, r)$ which satisfy $M \models r \text{ prop}(\rho_{\overline{r}})$ and $M \models r \text{ prop}(\rho_{s \overline{x}})$ respectively, we can conclude that $\text{eval}(\Pi, \rho_{\overline{r}}) = \text{eval}(\Pi, \rho_{s \overline{x}})$ for any successor MDP state $s_x$ of a decided state $s_d$. \(\square\)

Completeness

To prove completeness, we show that for every request $r \in \text{Req}$ and every model $M \in \mathcal{M}$ there exists a decision state $s_d$ in the MDP structure which has the same decision value as $\Pi(r, M)$ and which can be reached by evaluating the request conditions of $\Pi$ for $M$ and $r$. 

36
4.4. Analysis of the Solution

**Theorem 4.14** Given a model \( M \in \mathcal{M} \), a request \( r \in \text{Req} \), a policy \( \Pi \) and some value \( v \in 4 \),

\[
\text{if } \Pi(r, M) = v, \text{ then } \exists s \in \mathcal{S} : M \models_r \text{prop}(\rho_s) \wedge \text{dec}_s = v.
\]

**Proof.** We show that for every model \( M \in \mathcal{M} \) and every request \( r \in \text{Req} \) for a given policy \( \Pi \) one of the following two properties holds:

1. There exists an MDP state \( s \) which has a complete state assignment corresponding to \( r \) in \( M \), i.e., \( \forall \bar{a} \in \mathcal{A}_1 : \rho_s(\bar{a}) = M(r, \bar{a}) \), and \( s \) is a decided state such that \( \text{dec}_s = \Pi(r, M) \).

2. There exists a decided state \( s_d \) which does not have a complete state assignment, i.e., \( \exists \bar{a} \in \mathcal{A}_1 : \rho_{s_d}(\bar{a}) = \text{pending} \), such that \( M \models_r \text{prop}(\rho_{s_d}) \) and \( \text{dec}_{s_d} = v \). Furthermore this MDP state satisfies \( s_d \preceq_{\text{trans}} s \), where \( s \) is the MDP state which has a complete state assignment corresponding to \( r \) in \( M \).

If there exists an MDP state \( s \) which has a complete state assignment corresponding to \( r \) in \( M \), then this state is a decision state. We prove this in Lemma 4.15. As we have shown for Theorem 4.5, every decided state represents the correct decision of the policy, and therefore \( \text{dec}_s = \Pi(r, M) \) holds.

If such a state \( s \), which has a complete state assignment corresponding to \( r \) in \( M \), is not present in the MDP structure, then there must exist a decided state \( s_d \) such that \( s_d \preceq_{\text{trans}} s \).

We show this by contradiction. Assume there exists no decided state \( s_d \) in the MDP structure, for which \( s_d \preceq_{\text{trans}} s \). Then there must exist an undecided state \( s_u \) such that \( s_u \preceq_{\text{trans}} s \), for which \( \mathcal{A}(s_u) = \emptyset \), because otherwise \( s \) would have been created by the definition of the MDP construction. However, since \( s_u \preceq_{\text{trans}} s \) holds, \( s_u \) has to have atoms \( \bar{a} \) in its state assignment for which \( \rho_{s_u}(\bar{a}) = \text{pending} \). Following the construction function \text{constructMDP}, every leaf state \( s_l \) of an unfinished MDP structure which is an undecided state and has atoms \( \bar{a} \) in its state assignment for which \( \rho_{s_l}(\bar{a}) = \text{pending} \), is extended, such that all successor MDP states \( s' \in \mathcal{S} : s_l \prec_{a} s' \) get created for all unevaluated atoms \( \bar{a} \) in \( s_l \)'s state assignment.

By the definition of \( \preceq_{\text{trans}} \), if \( M \models_r s \) and \( s_d \preceq_{\text{trans}} s \) then \( M \models_r s_d \) holds. Furthermore, using again the property stated in Theorem 4.5 it follows that \( \text{dec}_{s_d} = \Pi(r, M) \) is satisfied. \( \square \)

**Lemma 4.15** Every leaf state \( s_l \) which represents a state \( s_l \) of the policy evaluation process where all atoms have been evaluated, is a decided state.

**Proof.** We show that for every request \( r \in \text{Req} \) in every model \( M \in \mathcal{M} \) the following implication holds for some \( v \in 4 \):

\[
M \models_r \text{prop}(\rho_{s_l}) \rightarrow \Pi(M, r) = v.
\]
If all atoms have been evaluated in a given state \( \overline{s} \) of the policy evaluation process, every pair \((M, r)\) of requests \( r \in \text{Req} \) and models \( M \in \mathcal{M} \) for which \( M \models_r \text{prop}(\rho_{\overline{s}}) \) holds, satisfies the following condition:

\[
\forall \overline{\pi} \in A_{\Pi} : M(r, \overline{\pi}) = \rho_{\overline{s}}(\overline{\pi})
\]

From this, it follows for every two models \( M, M' \in \mathcal{M} \) and for every two requests \( r, r' \in \text{Req} \) the following holds:

\[
(M \models_r \text{prop}(\rho_{\overline{s}}) \land M' \models_{r'} \text{prop}(\rho_{\overline{s}})) \rightarrow (\forall \overline{\pi} \in A_{\Pi} : M(r, \overline{\pi}) = M'(r', \overline{\pi}))
\]

In other words, every model-request pair \((M, r)\) which satisfies \( M \models_r \text{prop}(\rho_{\overline{s}}) \) maps a given atom \( \overline{s} \) to the same value \( x \in \{ \text{true}, \text{false} \} \).

By Theorem 3.4, the policy will decide the same value \( v \in 4 \) for every \((M, r)\) which satisfies \( M \models_r \text{prop}(\rho_{\overline{s}}) \), i.e., \( \Pi(r, M) = v \), and therefore the implication in Equation 4.16 holds. □

Termination

In order to prove termination of the construction algorithm, we show that the following properties hold in the MDP structure:

- We show that there are only a finite number of distinct states in the MDP structure and
- We show that no state is created more than once.

**Lemma 4.17** The number of distinct states in the MDP structure is finite.

**Proof.** The set of atoms \( A_{\Pi} \) in a given policy \( \Pi \) is finite. As every state \( s \) of the MDP structure corresponds to a unique state \( \overline{s} \) of the policy evaluation process, which defines for every atom \( \overline{s} \in A_{\Pi} \) an evaluation value in \( \{ \text{true}, \text{false} \} \) or the value pending for unevaluated atoms, the number of different states of the policy evaluation process, and therefore also in the MDP structure, is at most \( 3^{|A_{\Pi}|} \), which is finite. □

**Lemma 4.18** No state is created more than once.

**Proof.** This lemma is trivially true, as in the construction of the MDP structure we have a filter which only adds a given MDP successor state \( s' \in \mathcal{S} \) of some undecided state \( s \in \mathcal{S} \), if it is not already contained in the MDP structure. □
4.4. Analysis of the Solution

4.4.2 Correctness of the Solution

Optimality

Recall our efficiency requirement for a PBel evaluation procedure:

Given a policy \( \Pi \), an evaluation procedure \( \text{dec} \), and the set of uniformly distributed requests \( R \) for \( \Pi \), there does not exist another evaluation procedure \( \text{dec}' \) such that:

\[
\sum_r \text{time}(\text{dec}', \Pi, r) < \sum_r \text{time}(\text{dec}, \Pi, r),
\]

where \( r \in R \) and \( \text{time}(\cdot, \Pi, r) \) denotes the amount of time needed by the procedure to evaluate \( r \) against \( \Pi \).

To show optimality of our solution with respect to our efficiency requirement, we show that every decision is made as early as possible and that the strategy \( \pi^* \) found by the function \( \text{find\_optimal\_strategy} \) achieves our original goal, namely to minimize the expected time spent in the evaluation of an access request.

This follows from the following two properties of the solution:

1. If a decision is possible for a state \( s \) of the policy evaluation process, then \( \text{eval}(\Pi, \rho_s) \neq \text{pending} \) and therefore the MDP state \( s \) corresponding to \( s \) is a decided state.

   This property immediately follows from the correctness proofs of the MDP construction (see subsection 4.4.1).

2. The function \( \text{find\_optimal\_strategy} \) calculates a solution strategy \( \pi^* \) which minimizes the expected time spent in the evaluation for every possible access request.

Recall the mapping of access control into the MDP setting used in this thesis:

- Each state \( s \) of the policy evaluation process, described by its \( \rho_s \) function, corresponds to an MDP state \( s \).

- The evaluation steps, which choose an atom with decision value \( \text{pending} \) in a given state \( s \) of the policy evaluation process and evaluate the atom, are mapped to the corresponding MDP actions.

- An atom \( \overline{a} \) has a probability of 50% to evaluate to \( \text{true} \) and \( \text{false} \), respectively. A state transition \( (s, a, s') \) in an MDP state \( s \) is given by the probability \( P(s, a, s') \) that \( s' \) is the successor state of \( s \), if action \( a \) is executed.
4. Translating a PBel Policy into an Efficient Evaluation Procedure

- In the policy evaluation process every atom \( \overline{a} \in A_\Pi \) has a atom-specific cost \( c(\overline{a}) \in \mathbb{R} \), which represents the estimated time spent evaluating \( \overline{a} \). The reward \( R(s, a, s') \) of a given MDP state transition is defined as the negative cost of evaluating atom \( \overline{a} \) chosen by the action \( a \in \mathcal{A}(s) \), i.e., \( R(s, a, s') = -c(\overline{a}) \).

The function \textit{find\_optimal\_strategy} implements the value iteration algorithm, which guarantees optimality of the calculated strategy \( \pi^* \), such that in every state \( s \) in the state transition graph of the MDP, the action \( a \) which maximizes the expected utility of \( s \) over all possible actions in \( \mathcal{A}(s) \) is chosen. As the expected utility in every state of our MDP corresponds to the negative time spent in the evaluation of a request before reaching a decision, a strategy which maximizes this expected utility, minimizes the time spent in the evaluation and thus achieves our original goal.

4.5 Complexity

The generated evaluation procedure, based on the calculated optimal strategy of the MDP, satisfies our efficiency requirement. Note, that this evaluation procedure at most evaluates all atoms when evaluating a policy, and never yields higher cost than a naive implementation of the policy.

However, the construction of the underlying MDP structure suffers from state-space explosion. The worst-case space complexity of this construction is in \( O(3^n) \), where \( n \) is the number of atoms specified in the PBel policy. This means that our approach can only support policies which specify up to tens of atoms, as for larger numbers of atoms the problem becomes intractable. Note that this limitation only restricts the number of atoms and does not depend on the number of rules or operators used in the policy.

The problem of state-space explosion is a known issue for MDPs and is similar to the state-space explosion problem in model-checking. There are partial solutions proposed in the MDP community such as factorization of MDPs [9] [13], where the most relevant features of the problem are determined and the reduced problem, focusing on those features, is solved. Another proposed method focuses the computation of the solution to MDP states which are likely to be final states [8]. Both methods achieve a better time and space complexity than standard MDP solutions, however they also sacrifice the optimal property guarantee. Nevertheless, their heuristics show good results and in many cases the deviation from the optimal strategy is negligible.

We do not explore these solutions in more detail, and leave further analysis of the problem as well as possible adaptations of our method to known solutions as Future Work.
Chapter 5

Experimental Validation

This chapter describes our practical contribution, which is the automated generation of an efficient evaluation procedure for a given PBel policy. We then focus on experimentally validating our solution. We show empirically, that:

1. Taking into account evaluation costs of atoms when specifying evaluation order, in practice, can yield performance gains compared to evaluating all atoms, as strict conformance to formal semantics mandates.

2. Our solution can generate evaluation code that is as efficient as a heavily optimized manual translation.

The chapter is divided into three main parts:

**Translating the Optimal Strategy into Evaluation Code** shows how the optimal strategy described in subsection 4.2.2 can be translated into executable C-based evaluation code.

**Experiment Setup** describes what system was used and which basic assumptions were made to test our solution.

**Experiments** presents the two policies for which our solution was tested and the results of these tests (subsection 5.3.2 and subsection 5.3.3).

### 5.1 Translating the Optimal Strategy into Evaluation Code

In Chapter 4, we have focused on the theoretical underpinning of our algorithmic solution for automated generation of an efficient and correct evaluation procedure for a given PBel policy. In order to leverage our proposal
in practice, the optimal strategy constructed in subsection 4.2.2 needs to be translated into executable code. In this section we discuss one such translation mechanism that generates C-code, which can be adapted to other languages as well. We use the code generated by this method in our PDP implementation in the experiments.

To translate our optimal strategy, we use the best action stored in each state as a splitting point. The translation is done recursively on the MDP structure and can be implemented as follows.

For every best action:

1. Create a variable $atom_i$ that stores the evaluation result of the action-specific atom.

2. Build an if-then-else statement, checking the truth value of $atom_i$ and finally

3. Use the true- and the false-child-state of the action to generate the bodies of the if and else branch, respectively.

A decision state is transformed into a return decision statement, while undecided states just forward execution to their best action.

This translation allows each atom to be evaluated as late as possible in the evaluation order, respecting the goal of maximizing the evaluations final reward, which is the negative of the time spent in evaluation.

As in every state only one of the possibly $n$ successor actions is considered, the majority of states does not have to be processed by the translation. Every action is translated to an if- and an else-code block, i.e., for every action there are exactly two branching possibilities.

The total number of atom evaluations in a the evaluation procedure can never exceed the number of atoms in the policy.

The python\(^1\) code performing this translation is shown in Appendix A.

### 5.2 Experiment Setup

All our tests are carried out on a desktop PC running Ubuntu 12.04 with a 2.53-GHz Intel Core 2 Duo CPU and 4GB of RAM.

We use our generated code to protect web resources served by an Apache 2.4 web server instance, for which we implement a Policy Decision Point (PDP).

\(^1\)Python is a high-level scripting language. For more details on python see the according web-page www.python.org
For every experiment, a test-run consist of a set of requests for a specific URL, i.e., a protected resource. A URL can either point to a file or a program. The web server processes requests for this URL and finally either allows or denies the request according to the decision determined by the generated evaluation code.

In more detail, a request is received by the Apache core module, where it is preprocessed and forwarded to our Apache PDP module. This module extracts the relevant information from the request and calls the policy library evaluation function for a given policy. Note that our PDP module is generic and as input it takes a library consisting of the generated evaluation code. The decision returned by the evaluation function is forwarded to the Apache core module, which takes the according actions to enforce the decision. The general setup of such a test-run is shown in Figure 5.1.

![Figure 5.1: General Setup of a Test-Run on a Server](image)

In order to store and query group membership of 10'000 users we use a database. Each user is a member of three randomly chosen groups. For authentication, we use the Apache password file. To be able to simulate requests from authorized and unauthorized users, only 1000 users have an entry in the password file.

In order to stress test our policy evaluation, we use JMeter, an Apache stress testing tool, which simulates multiple concurrent users requesting the protected resource in a short time period.

### 5.2.1 Apache 2.4 Web Server

The Apache web server is one of the most popular web servers in deployment. It was originally developed for Unix-based operating systems, but has since been ported to other platforms as well. Apache is developed and maintained by an open source community[^2].

The Apache web server has a modular architecture. The basic functionality is provided by the core Apache modules which include basic authentication schemes and support for different server-side programming languages. Additional functionality can be included by the server administrator by selecting the respective modules to be compiled into the web server. There are a variety of different Apache modules available from the Apache community, which cover most functionalities required by a normal web server. If there is a need for other functionality, a developer can build a custom Apache module and hook it into the server. A hook tells the server that the defined module needs to have access to selected requests to handle them. If a certain module only wants to handle requests of a given type, e.g., requests to a specific URL, this can be specified using specific server directives\(^3\). We used this mechanism to protect certain resources with our Apache PDP module.

The PDP module defines an Apache hook of medium priority in the request handling process. This hook tells the server to call the custom request handler for every request received. The hook priority defines the order in which different modules are called. We chose medium priority to still allow Apache internal authentication mechanisms to execute before our module is called, but to be called before any request is served.

The request handler extracts the following information from a HTTP request:

- requester host name/IP
- target host name/IP (if multiple virtual hosts exist on the same server)
- username/password
- action
- target file name

If the evaluation returns the decision value \( t \), the request is allowed and the request is passed on to the next processing module. Otherwise the request is denied and request processing is interrupted with an unauthorized error.

### 5.2.2 Stress tests

In order to measure performance, we use the Apache stress testing tool JMeter. JMeter is a pure Java application that simulates different loads on the server\(^4\). One can specify different test plans by defining test variables in JMeter.

---

\(^3\)For more details on Apache modules see http://httpd.apache.org/docs/2.4/developer/modguide.html

\(^4\)http://jmeter.apache.org/
5.3 Experiments

The variables we used are:

- The number of concurrent users simulated, the so called threads, which we varied from 50 to 500,
- The time in seconds until the last thread is started, the so called ramp-up period, which we fixed to 1 second, and
- The number of requests issued by each user, the so called loop count, which we fixed to 10 to simulate a web page containing multiple elements like images and news boxes that all need to be requested separately.

A test case defines a set of requests sent to the server. Depending on the number of threads specified, JMeter simulates a different number of users concurrently sending requests from this set. Tests for a given number of threads are repeated 10 times for averaging purposes. For every test run, a log file containing the response time for every request is produced, from which the average response time is then calculated.

5.3 Experiments

5.3.1 Overview

To test the effectiveness of our method in different scenarios, we designed two access control policies based on real-life policies.

A scenario is a fixed set of requests, representing a number of concurrent users, with fixed ratio of expensive and inexpensive requests. An expensive request requires evaluation of at least one atom with high cost, while an inexpensive request only requires atom evaluations of low cost.

To evaluate each scenario we implemented the following policy evaluation procedures:

unoptimized (uop): This evaluation procedure is the naive policy translation, where no optimization is computed and every atom is evaluated for every request.

optimized hard coded (oph): This evaluation procedure is a manual translation of the policy specification into evaluation code, following our best understanding of the policy and our perceived optimal strategy. This manual translation required many cumbersome review iterations and revisions to straighten out small mistakes and find the optimal ordering considering all border cases.
optimized generated (opg): This evaluation procedure is the generated evaluation code produced by our implementation of the algorithm, as described in Section 5.1.

In the experiments, abstract the exact evaluation time using so-called cost levels. We consider different types of atoms and assume that they each have an associated cost level between 1 and 20. Here, a cost level of 1 indicates that the atom is evaluated in a negligible amount of time, whereas a cost level of 20 signifies that evaluation takes much longer and does not scale for many concurrent requests.

The cost levels we used in the two test cases are:

**Inexpensive:**
- String comparison which is assigned the cost level 1.
- Check the authenticity of the requesting user using a server-specific password file. The evaluation of this atom type always requires opening and scanning a possibly large file and potentially performing multiple string comparisons, which is more expensive than simple string comparison.

  This atom type is assigned the cost level 5.

  Depending on the authentication mechanism used by a specific server, this atom type could also require connecting to a separate authentication server. This would increase the cost of the atom type even further, as network traffic would be involved. In our tests however, we implemented the first variant.

**Expensive:**
- Connect to a local database and send basic queries. Without further optimization this mechanism scales very poorly if many concurrent database connections are established.

  This atom type is assigned the cost level 10.
- Connect to a local database and call expensive stored procedures. This is the most expensive atom type used.

  This atom type is assigned the cost level 20.

In practice, additional costs, e.g., involving network traffic, remote databases or complex computations, may arise. We did not consider them because we already have high cost atoms.

The costs used are only estimated and do not reflect the exact time spent in evaluation of the different atom types. Heuristics can be estimated easily on a productive server by recording evaluation times of atoms and requests. In
our case, as we are using a test server, we did not have real life data and therefore the costs assigned are based on our experimental timings.

For every individual real-world deployment, these cost levels and assigned cost values will vary.

5.3.2 The Cambridge Policy

The Cambridge policy is based on the sample policy for the University of Cambridge Computer Laboratory\(^5\). A user requesting the protected web page has to satisfy the following policy:

1. He is an administrator of the web page, in that case access is granted as soon as he logs in, or

2. He is connecting from within the University of Cambridge, i.e., the host is whitelisted, or he is able to log into the authentication service. In this case, access is granted only if the third property is satisfied as well.

3. He is a student of the course or a helper of the course or at least belongs to the Computer Laboratory group. Excluded from the Computer Laboratory are blacklisted machines.

This policy can be specified in PBel as follows:

1. College wide ($\Pi_1$):

   $$(t \text{ if } \text{from}(\text{Req}, \text{whitelist}')) \lor (t \text{ if } \text{validUser(Req)})$$

2. Lab wide ($\Pi_2$):

   $$(t \text{ if } \text{inGroup}(\text{Req}, \text{lab}')) \oplus (f \text{ from}(\text{Req}, \text{blacklist}'))$$

3. Course1-specific ($\Pi_3$):

   $$(t \text{ if } \text{inGroup}(\text{Req}, \text{course1}')) \lor (t \text{ if } \text{inGroup}(\text{Req}, \text{helper -- course1}'))$$

4. Admin ($\Pi_4$):

   $$(t \text{ if } \text{is\_admin(Req)}) \land (t \text{ if } \text{validUser(Req)})$$

5. Composition:

   $$(\Pi_4 \lor (\Pi_1[t \mapsto (\Pi_2 \oplus \Pi_3)])[T \mapsto false] \{\bot \mapsto false\})$$

\(^5\text{http://www.cl.cam.ac.uk/local/web/htaccess.html}\)
The Cambridge policy uses seven different atoms:

- `from(Req, 'whitelist')`
- `from(Req, 'blacklist')`
- `validUser(Req)`
- `inGroup(Req, 'lab')`
- `inGroup(Req, 'course1')`
- `inGroup(Req, 'helper-course1')`
- `is_admin(Req)`

These atoms can be split into four atom types with different mechanisms and costs. They are again divided into the two groups *inexpensive* and *expensive*.

**Inexpensive:**

- `from` takes the IP address of the calling host and checks if it appears in a white- or blacklist. As our lists are very short, this can be seen as a simple string comparison. It is assigned cost 1 in the policy metadata.

- `validUser` checks if the username given in the request is listed in the password file of the server and compares the hash of the request password with the hash in the file. It is assigned cost 5 in the policy metadata.

- `is_admin` is another string comparison. The evaluation of this atom type assumes that every administrator has a username starting with `admin`. This is of course a very naive check for administrator rights, but is sufficient for our test cases. As for the atom type `from` the `is_admin` atom type is assigned cost 1 in the policy metadata.

**Expensive:**

- `inGroup` connects to a local database and checks the membership table for a `user:group` pair containing the given user and the required group. It is assigned cost 10 in the policy metadata.

**Test Scenarios for Cambridge Policy**

We define the different request types as follows:

**fail1:** The user is not an administrator, the requesting host is not whitelisted and the user is unauthorized. Requests of this type are denied and have an average cost level of 7.

**fail2:** The user is not an administrator, the requesting host is not blacklisted, the user is unauthorized and he is not in course1, not in helper_course1
5.3. Experiments

and not in lab. Requests of this type are denied and have an average cost level of 37.

**pass1:** The user is an administrator and is valid (i.e., authorized). Requests of this type are allowed and have an average cost level of 6.

**pass2:** The user is not an administrator, the requesting host is not blacklisted, the user is valid and he is in one of the three groups (lab, course1 or helper_course1). Requests of this type are allowed and have an average cost level of 17.

These request types can be divided into two groups:

- The **inexpensive** request types are fail1 and pass1, which both only require string comparisons and file reads to decide with the optimized evaluation procedures.

- The **expensive** request types are fail2 and pass2, which both require at least one database query to decide, independent of the evaluation procedure.

We examine scenarios with mixed request types, consisting of different percentages of expensive and inexpensive requests. This will show us the impact of different evaluation procedures on average response times for different usage types defined through mixtures of request types.

The distributions are shown in Table 5.1. The represented usage types can be described as follows:

- **pure inexpensive:** 0% of the requests fall into the expensive test cases fail2 and pass2 (i.e., 100% fail1 and pass1).

- **mixed 1:** 51.75% of the requests fall into the expensive test cases.

- **mixed 2:** 68.75% of the requests fall into the expensive test cases.

- **mixed 3:** 81.5% of the requests fall into the expensive test cases.

- **pure expensive:** 100% of the requests fall into the expensive test cases.

For the **inexpensive usage type** (0% expensive requests), the test results in Figure 5.2 show a considerable performance improvement in the optimized versions, hard coded (oph) and generated (opg), compared to the unoptimized version (uop). This behavior coincides with the expected test results, as in the unoptimized version all atoms, also the database queries, are evaluated while in the optimized versions the decision is returned as soon as possible, which is before the database connection is established.

---

*Possible scenarios for the mixture usage types are given in the Appendix.*
5. Experimental Validation

<table>
<thead>
<tr>
<th></th>
<th>fail1</th>
<th>fail2</th>
<th>pass1</th>
<th>pass2</th>
<th>inexpensive</th>
<th>expensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>pure inexpensive</td>
<td>50%</td>
<td>0%</td>
<td>50%</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>mixed 1</td>
<td>45.5%</td>
<td>5%</td>
<td>2.75%</td>
<td>46.75%</td>
<td>48.25%</td>
<td>51.75%</td>
</tr>
<tr>
<td>mixed 2</td>
<td>27.5%</td>
<td>5%</td>
<td>3.75%</td>
<td>63.75%</td>
<td>31.25%</td>
<td>68.75%</td>
</tr>
<tr>
<td>mixed 3</td>
<td>14%</td>
<td>5%</td>
<td>4.5%</td>
<td>76.5%</td>
<td>18.5%</td>
<td>81.5%</td>
</tr>
<tr>
<td>pure expensive</td>
<td>0%</td>
<td>50%</td>
<td>0%</td>
<td>50%</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 5.1: Distributions of request types per usage type

As database connections do not scale well for many concurrent connection requests, it is also not surprising that the optimized versions for the inexpensive request types scale much better to high request numbers than the unoptimized. In fact, in the two test cases fail1 and pass1 in the optimized versions the performance loss for many concurrent requests is minimal and barely notable. In the unoptimized version however, the response time increases linearly with respect to the number of concurrent requests.

For the mixture usage types (51.75%, 68.75% and 81.5% expensive requests), the test results in Figure 5.3 show the convergence of the average response times produced by the optimized evaluation procedures to those produced by the unoptimized evaluation procedure when the percentage of expensive requests is increased.

We can also see, that even thought a majority of the requests fall into the expensive test cases (fail2 and pass2) a considerable performance improvement from the unoptimized versions (uop) to the optimized versions (oph and opg) of the evaluation procedures can be observed in all three mixture usage types. This shows that even optimizing only a minority of the request evaluations can have a substantial impact on access control perfor-
Figure 5.3: Test Results for the mixture types for 51.75% (a), 68.75% (b) and 81.5% expensive requests (c) in Cambridge using the optimized hard coded (oph), the optimized generated (opg) and the unoptimized (uop) version of the evaluation procedure.

For the **expensive usage type** (100% expensive requests), the test results in Figure 5.4 show that the evaluation procedure does not influence the average response time for a request. As, even with the optimized evaluation procedures, every atom needs to be evaluated. For these two request types, no performance gain is achieved solely by reordering policy atom evaluations.

Another important observation is that in all test cases the optimized generated version (opg) performs as good as the manually optimized hard coded version (oph). In other words, our solution does not add hidden costs.
5. Experimental Validation

![Graph showing response time vs number of concurrent users for different policy versions.]

Figure 5.4: Test Results for the pure expensive usage type in Cambridge using the optimized hard coded (oph), the optimized generated (opg) and the unoptimized (uop) version of the evaluation procedure.

5.3.3 The Moodle Policy

To ensure that the previous results do not only depend on the Cambridge policy structure, we designed a more complicated policy example. We do not explore all the mixture tests done for Cambridge, as the test results for the pure inexpensive, pure expensive and mixed 50% show similar behavior as for the Cambridge policy.

The moodle policy is a policy inspired by the Virtual Learning Environment Moodle\(^7\), which is a popular tool for professors to create web sites for their courses.

A user requesting the protected web page for a course has to satisfy the following policy:

1. He wants to read the page and is a student of the course, in that case access is granted immediately.
2. If he wants to read but he is not a student of the course, he needs to be an authenticated professor, i.e. he is in group professor and he is able to log into the authentication service. In this case access is granted.
3. If he wants to write the page, he needs to be connecting from within the University, i.e., has a whitelisted host, and he needs to be an authenticated professor. In this case access is granted if the page is not locked by another user.

This policy can be specified in PBel as follows:

1. Blacklist ($\Pi_1$):

$$\texttt{(f if from(Req,'blacklist'))}$$

\(^7\text{https://moodle.org/about/}\)
2. Read or Whitelist \((\Pi_1)\):

\[(t \text{ if } \text{is_read}\((\text{Req})\)) \lor (t \text{ if } \text{from}\((\text{Req}, 'whitelist'))\]

3. Group \((\Pi_3)\):

\[(t \text{ if } \text{inGroup}\((\text{Req}, 'course')) \lor ((t \text{ if } \text{inGroup}\((\text{Req}, 'prof')) \land (t \text{ if } \text{validUser}\((\text{Req}))))\]

4. Target \((\Pi_4)\):

\[(t \text{ if } \text{is_write}\((\text{Req})))[\bot \mapsto \top] \implies (t \text{ if } \neg \text{is_locked}\((\text{Req}))\]

5. Action in Range \((\Pi_5)\):

\[((t \text{ if } \text{is_read}\((\text{Req})) \lor (t \text{ if } \text{is_write}\((\text{Req})))[\bot \mapsto \top]\]

6. Composition:

\[(\Pi_1 \oplus \Pi_2)[t \mapsto (\Pi_3 \land \Pi_4 \land \Pi_5)][\top \mapsto \top][\bot \mapsto \bot]\]

The Moodle policy uses eight different atoms:

- \text{from}(\text{Req}, 'blacklist')
- \text{is_read}(\text{Req})
- \text{from}(\text{Req}, 'whitelist')
- \text{inGroup}(\text{Req}, 'course')
- \text{inGroup}(\text{Req}, 'prof')
- \text{validUser}(\text{Req})
- \text{is_write}(\text{Req})
- \text{is_locked}(\text{Req})

These atoms can be categorized into the four atom types, again divided into the two groups \textbf{inexpensive} and \textbf{expensive}:

\textbf{Inexpensive}:

- \text{from} is a simple string comparison just like in the Cambridge policy. We therefore assign it cost level 1 in the policy meta data.
- \text{is_read}/\text{is_write} check whether the action requested for the web page is read or write, respectively. As the action is given as a string, this atom type, like \text{from}, is also a simple string comparison. We therefore assigned it cost level 1 in the policy meta data.
5. Experimental Validation

- **validUser** scans the password file of the server for the username and the password and hash, just like in the Cambridge policy. We therefore assigned it cost level 5 in the policy meta data.

- **inGroup** connects to a local database just like the Cambridge policy. We therefore assigned it cost 10 in the policy meta data.

**Expensive:**

- **isLocked** connects to a local database and calls a stored procedures to figure out if the requested target is being written by some other user and if this is not the case, locks the target for the requesting user. As this stored procedure needs to do different calculations on the database, this is the most expensive atom type. We therefore assigned it cost level 20 in the policy meta data.

**Test Scenarios for Moodle Policy**

We define the different request types as follows:

fail1: The request is not a read request and the requesting host is not whitelisted. Requests of this type are denied and have an average cost level of 2.

fail2: The request is a write request, the requesting host is whitelisted and the user is a valid professor, but the web page he wants to write is locked by some other user. Requests of this type are denied and have an average cost level of 37.

pass1: The request is a read request and the user is in the course. Requests of this type are allowed and have an average cost level of 11.

pass2: The request is a write request, the requesting host is whitelisted, a valid professor and the target he wants to write is not locked. Requests of this type are allowed and have an average cost level of 37.

These request types can be divided into two groups:

- The **inexpensive** request types are fail1, and pass1, which only require string comparisons and the inexpensive type of database query to decide with the optimized evaluation procedures.

  We say request type pass1 is an inexpensive request type, because the database query performed is still much less expensive than the query that checks if a page is locked. We will see in the test results that the few normal database calls do not have a big impact to the overall performance compared to the really expensive ones.
5.3. Experiments

- The **expensive** request types are the fail3 and pass2, which both always require multiple database queries and at least one expensive type of database query to decide, independent of the evaluation procedure.

We only examine one scenario with mixed request types, because the results of all the moodle tests show a very clear behavior.

The distributions used are shown in Table 5.2.

- **pure inexpensive**: 0% of the requests are of the expensive request types **fail3** and **pass2** (i.e., 100% **pass1**).

- **mixed**: 50% of the requests are of the expensive request types.

- **pure expensive**: 100% of the requests are of the expensive request types (i.e., 100% **fail3**).

<table>
<thead>
<tr>
<th></th>
<th>fail1</th>
<th>fail3</th>
<th>pass1</th>
<th>pass2</th>
<th>inexpensive</th>
<th>expensive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pure</strong></td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>inexpensive</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>mixed</strong></td>
<td>18.125%</td>
<td>12.5%</td>
<td>31.875%</td>
<td>37.5%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td><strong>pure</strong></td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td><strong>expensive</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Distributions of request types per usage type

Here, like for the Cambridge policy, the optimized versions scale much better to large number of requests and show a considerable performance improvement compared to the unoptimized version of the **inexpensive usage type** (0% expensive requests). The results are shown in Figure 5.5

![Graph showing test results](image)

Figure 5.5: Test Results for the pure inexpensive usage type in Moodle using the optimized hard coded (oph), the optimized generated (opg) and the unoptimized (uop) version of the evaluation procedure.
Also for the **mixture usage type** (50% expensive requests), the test results in Figure 5.6 clearly show a performance improvement from the unoptimized versions to the optimized versions of the evaluation procedures, in spite of a majority of the requests being of the expensive request types (fail2 and pass2).

Figure 5.6: Test Results for the mixed type for 50% expensive requests in Moodle using the optimized hard coded (oph), the optimized generated (opg) and the unoptimized (uop) version of the evaluation procedure.

Further, for the **expensive usage type** (100% expensive requests), the test results in Figure 5.7 only support the conclusions drawn in subsection 5.3.2. They show that the evaluation procedure does not influence the average response time for a request.

Figure 5.7: Test Results for the Pure Expensive usage type in Moodle using the optimized hard coded (oph), the optimized generated (opg) and the unoptimized (uop) version of the evaluation procedure.
5.3.4 Test Conclusions

The tests described for the Cambridge policy and the Moodle policy show that given a set of requests where a policy can be decided without evaluating costly atoms, the performance gains are significant. The results shown, however, only demonstrate the performance improvement for policies, which define many atoms of different costs. We have not investigated the performance behavior when the policy specifies many atoms which have the same cost, because these policies are rare in practice.

The test results also demonstrate that independent of the optimization capacity of the request types, our solution does not add cost compared to a naive implementation of the policy. If the policy does not provide any optimization capability, i.e., if the environment, in which the policy is applied, only provides requests which fall into the expensive test cases, applying our method will not harm performance.
In this thesis we consider the problem of generating an efficient and correct evaluation procedure for a given PBel access control policy. The efficiency requirement for the generated evaluation procedure rests on the following assumptions:

- Every condition of the policy evaluates to true and false with 50% probability.
- The cost of evaluating a given request condition is given.

The efficiency requirement for a PBel evaluation procedure is stated as:

Given a policy $\Pi$, an evaluation procedure $\text{dec}$, and the set of uniformly distributed requests $R$ for $\Pi$, there does not exist another evaluation procedure $\text{dec'}$ such that:

$$\sum_{r} \text{time}(\text{dec'}, \Pi, r) < \sum_{r} \text{time}(\text{dec}, \Pi, r),$$

where $r \in R$ and $\text{time}(\cdot, \Pi, r)$ denotes the amount of time needed by the procedure to evaluate $r$ against $\Pi$.

The problem of optimizing evaluation in access control by automated generation of an efficient evaluation procedure for a formal policy language has never been explicitly defined. Compared to previous work, which focus their research on heuristics and only perform experimental validations of their solutions, we examine the optimization potential of the policy evaluation process for a formal access control policy language. Moreover we link the problem to the setting of Markov Decision Processes, for which standardized optimal solutions exist. The formal semantics of the language also allow us to formally prove the correctness and show optimality of our proposed solution.
6. Conclusion

Although the method presented in this thesis only focuses on PBel, the idea of mapping an access control policy evaluation process to an MDP structure and calculating an optimal strategy in this setting is not restricted to a particular access control policy language. Our approach can be generalized to any formal access control policy whose semantics implement domain-independence and provide an analysis procedure.

Furthermore the proposed solution is implemented for generating evaluation code for the Apache web server. We evaluate our solution for two different access control policies. The test results from the stress tests on the resulting evaluation procedures, show that our optimized evaluation process increases performance of the overall system compared to a naive evaluation process for formal policies.

As all methods for planning problems, our method suffers from state-space explosion. Therefore, its applicability and the optimality it guarantees are limited to access control policies which have a limited number of request conditions. However the number of rules defined in the policy is not restricted.

We hope that this work, being one of the first steps in exploring efficient access control evaluation for formal policies, can serve as the basis for further investigations of this problem. In particular, we have identified the following areas for future research:

- **Handling State-Explosion:** The limitation of our approach to policies which only specify a very small number of request conditions can be a restriction to its applicability. Especially, analyzing the optimality loss of methods developed to tackle the state-space explosion problem could provide more information about possible evaluation performance loss of the suboptimal evaluation code generated by our approach when adapted to those methods. One possible way of integrating the method of MDP factorization [9] [13] to our presented generation method could be to divide the policy into multiple sub policies, each containing only a subset of all the request conditions, and to optimize each sub policy separately. The strategy chosen for the overall policy would then be a combination of the calculated optimal strategies for the sub policies.

In addition to dealing with the complexity problem, we see the potential for extensions of our solution to more adaptive approaches of efficient access control policy evaluation.

- **Using Heuristics for Transition Probability Estimation:** Many approaches presented in Chapter 2 use heuristics for fine-tuning and adaptation of their solutions to changing environments. Similarly, introducing heuristics to estimate state transition probabilities and atom costs could
generalize our presented solution to different situations, where more information about requests is available. One possible implementation could use the no-information approach presented here as a starting point, which can be updated periodically using the data heuristics measured in production.

- **Online Adaptation:** Instead of only using the MDP structure and the calculated optimal strategy to generate evaluation code, these structures could be used online, where an optimal solution is calculated at request evaluation. For such an approach the optimal strategy would have to define the probabilities of choosing a particular successor action in a given MDP state, such that information about all possible state transitions could be collected and could influence future strategy calculations. Such an approach would have the major advantage of immediate reaction to environment changes, such that e.g. denial of service attacks could be handled differently from normal usage at peak request times.
Appendix A

Translation of the Optimal Strategy to C-Code

The function performing the translation from a given MDP optimal strategy into C-code is described in Algorithm A.1. It has an argument state of the optimal strategy, which is an MDPState-object as described in subsection 4.2.1 and an argument code, which is a string containing the generated C-based evaluation code.

An MDPState-object is identified by its assignment and has a decision which is one of \{t, f, ⊤, ⊥, Pending\} and a best action which is an MDPAction-object also described in subsection 4.2.1.

Every time generate_evaluation_code is called, the code in code is copied to the String variable new_code, to which further branches of the generation are added.

Finally, new_code is returned to the caller, which writes the generated String into a file and compiles it as library.
Algorithm A.1 \textit{generate\_evaluation\_code}(state, code)

\begin{verbatim}
new\_code = code #string

if (state.decision != Pending):
    #every decision state is translated into a return statement in the code
    new\_code += to\_string(state.decision) + "\n"

else:
    #every undecided state is translated into the evaluation of the best next atom
    action = state.best\_action
    atom = action.atom
    variable = "var" + to\_string(atom.id)
    new\_code += variable + " = evaluate(" + to\_string(atom) + ") \n"

    #and a branching depending on the value of the atom
    new\_code += "if (var" + to\_string(atom.id) + "){ \n"
    new\_code += translate\_to\_evaluation\_code(action.true\_state, new\_code)
    new\_code += "} else{ \n"
    new\_code += translate\_to\_evaluation\_code(action.false\_state, new\_code)
    new\_code += "} \n"

return new\_code
\end{verbatim}
In this Chapter we describe the user motivations for the different requests made in the mixture usage types in the Cambridge Policy (subsection 5.3.2). The distributions under

As every protected web page has honest users and users trying to circumvent the security system, we divide the users requesting the protected cambridge web page into these two groups. The honest users are moreover split according to their functionality in the University.

- An honest student requesting the web page is enrolled in course1.

  There are two reasons why a honest student could be denied access and we assume them to be equally likely:

  1. The student misspelled either his username or password.

  2. The students host was set on the blacklist because of a mistake.

  Both request types fall into the case fail1.

- An honest researcher requesting the web page belongs to the Computer Laboratory group (lab).

  The two reasons why a honest researcher could be denied access are the same as for honest students and again we assume them to be equally likely:

  1. Either the username or password was misspelled, or

  2. The host was set on the blacklist because of a mistake.

  Both request types fall into the case fail1.
B. Experiment Mixture Description

- An honest course1-helper requesting the web page is an assistant of course1.

The two reasons why a honest course assistant could be denied access are the same as for honest students and researchers and again we assume them to be equally likely:

1. Either the username or password was misspelled, or

2. The host was set on the blacklist because of a mistake.

Both request types fall into the case fail1.

- All other requests of honest students, honest researchers and honest course assistants will be allowed access. The reasons for this to happen depends on other activities of the users but independent of these activities this request falls into the case pass2.

- An honest administrator requesting the web page only has to connect to the web page if some sort of error occurs.

If an administrator misspells either his username or password the request falls into the case fail1.

In any other case, the request is allowed access, which falls into the case pass1.

- As curious users we denote users which are not associated to course1 in any way, but still try to get the web pages content. These users could be prospective students trying to prepare for the course, students by mistake clicking on a link to the course web page or web crawlers. Depending on their functionality within the University, their requests get denied for different reasons:

  - The request of a user which has no connection to the University falls into the case fail1.

  - The request of a user which is a member of the University but uses a blacklisted host also falls into the case fail1.

  - The request of a user which requests the web page from a University host or which is a member of the University falls into the case fail2.

In the three mixture usage types of Cambridge policy the user requests are distributed according to Table B.1, Table B.2 and Table B.3 to the four request types.
<table>
<thead>
<tr>
<th>Role</th>
<th>fail1</th>
<th>fail2</th>
<th>pass1</th>
<th>pass2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>honest student in course1</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>50%</td>
<td>22.5%</td>
<td></td>
<td>27.5%</td>
<td></td>
</tr>
<tr>
<td>(of which 45% fail, 55% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>honest researchers in lab</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td>11.25%</td>
<td></td>
<td>13.75%</td>
<td></td>
</tr>
<tr>
<td>(of which 45% fail, 55% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>honest helper of course1</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td>4.5%</td>
<td></td>
<td>5.5%</td>
<td></td>
</tr>
<tr>
<td>(of which 45% fail, 55% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>honest administrators</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5%</td>
<td>2.25%</td>
<td></td>
<td>2.75%</td>
<td></td>
</tr>
<tr>
<td>(of which 45% fail, 55% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>curious users</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>10%</td>
<td>5%</td>
<td></td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>45.5%</td>
<td>5%</td>
<td>2.75%</td>
<td>46.75%</td>
</tr>
</tbody>
</table>

Table B.1: Mixture 1 for 51.75% expensive requests

<table>
<thead>
<tr>
<th>Role</th>
<th>fail1</th>
<th>fail2</th>
<th>pass1</th>
<th>pass2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>honest student in course1</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>50%</td>
<td>12.5%</td>
<td></td>
<td>37.5%</td>
<td></td>
</tr>
<tr>
<td>(of which 25% fail, 75% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>honest researchers in lab</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td>6.25%</td>
<td></td>
<td>18.75%</td>
<td></td>
</tr>
<tr>
<td>(of which 25% fail, 75% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>honest helper of course1</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td>2.5%</td>
<td></td>
<td>7.5%</td>
<td></td>
</tr>
<tr>
<td>(of which 25% fail, 75% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>honest administrators</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5%</td>
<td>1.25%</td>
<td></td>
<td>3.75%</td>
<td></td>
</tr>
<tr>
<td>(of which 25% fail, 75% pass)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>curious users</strong></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>10%</td>
<td>5%</td>
<td></td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>27.5%</td>
<td>5%</td>
<td>3.75%</td>
<td>63.75%</td>
</tr>
</tbody>
</table>

Table B.2: Mixture 2 for 68.75% expensive requests
### Table B.3: Mixture 3 for 81.5% expensive requests

<table>
<thead>
<tr>
<th>Role</th>
<th>fail1</th>
<th>fail2</th>
<th>pass1</th>
<th>pass2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>honest student in course1</strong></td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>50% (of which 10% fail, 90% pass)</td>
<td>5%</td>
<td></td>
<td></td>
<td>45%</td>
</tr>
<tr>
<td><strong>honest researchers in lab</strong></td>
<td>✔️</td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>25% (of which 10% fail, 90% pass)</td>
<td>2.5%</td>
<td></td>
<td></td>
<td>22.5%</td>
</tr>
<tr>
<td><strong>honest helper of course1</strong></td>
<td>✔️</td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>10% (of which 10% fail, 90% pass)</td>
<td>1%</td>
<td></td>
<td></td>
<td>9%</td>
</tr>
<tr>
<td><strong>honest administrators</strong></td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>5% (of which 10% fail, 90% pass)</td>
<td>0.5%</td>
<td></td>
<td></td>
<td>4.5%</td>
</tr>
<tr>
<td><strong>curious users</strong></td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10% (of which 10% fail, 90% pass)</td>
<td>5%</td>
<td>5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>14%</td>
<td>5%</td>
<td>4.5%</td>
<td>76.5%</td>
</tr>
</tbody>
</table>
Bibliography


