Automated Certification of Authorisation Policy Resistance*

Andreas Griesmayer[1] and Charles Morisset[2]

[1] ARM, Cambridge, UK
andreas.griesmayer@arm.com
[2] Center for Cybercrime and Computer Security,
School of Computing Science, Newcastle University, UK
charles.morisset@ncl.ac.uk

Abstract. Attribute-based Access Control (ABAC) extends traditional Access Control by considering an access request as a set of pairs *attribute name-value*, making it particularly useful in the context of open and distributed systems, where security relevant information can be collected from different sources. However, ABAC enables *attribute hiding attacks*, allowing an attacker to gain some access by withholding information.

In this paper, we first introduce the notion of policy *resistance* to attribute hiding attacks. We then propose the tool ATRAP (Automatic Term Rewriting for Authorisation Policies), based on the recent formal ABAC language PTaCL, which first automatically searches for resistance counter-examples using Maude, and then automatically searches for an Isabelle proof of resistance. We illustrate our approach with two simple examples of policies and propose an evaluation of ATRAP performances.

Keywords: Attribute-based Access Control, Monotonicity, Attribute hiding, Model Checking, Proof assistant.

1 Introduction

An authorisation policy for a security mechanism is a document describing which user requests are authorised and which ones are denied. Many languages exist in the literature to define policies, most of them considering a request as a triple subject-object-mode. Recent approaches [23,22,10] consider more expressive requests, consisting of a set of pairs of attribute name and attribute values, thus defining the model known as Attribute Based Access Control (ABAC).

A major feature of ABAC is its ability to gather attribute information from different sources. This is essential in open and distributed systems, which indeed often lack a central security point providing all required information. In order to make a security decision, such systems need to combine information from different sources including the user herself (e.g., personal identification), the

* Work partially supported by the International Exchange Scheme of the Royal Society and conducted in part at Imperial College London where A. Griesmayer was supported by the Marie Curie Fellowship "DiVerMAS" (FP7-PEOPLE-252184).

J. Crampton, S. Jajodia, and K. Mayes (Eds.): ESORICS 2013, LNCS 8134, pp. 574–591, 2013.

environment (e.g., date or location), the data requested (e.g., relationship with other objects) and other users (e.g., availability of higher-ranked users).

Delegating the retrieval of security relevant information clearly brings more flexibility in open systems, but also raises the problem of information withholding, malicious or not. Tschantz and Krishnamurthi introduce in [24] the notion of safety, such that a policy is safe when "incomplete requests should only result in a grant of access if the complete one would have". More recently, Crampton and Morisset identify in [10] the notion of an *attribute hiding attack*, where a user hides some attribute values in order to get access to a resource she would be denied otherwise.

As a running example to illustrate our research problem, consider a fictional organisation, sponsored by both Austria and France, where documents can be submitted and reviewed. The organisation has a simple conflict-of-interest policy stating that a document cannot be reviewed by someone with the same nationality as the submitter. The implementation of such a policy might lead to attribute hiding attacks: Consider a policy for a document submitted by an Austrian member that states that it cannot be accessed by another Austrian member; If an Austrian attacker is able to hide her nationality, for instance by corrupting the corresponding data, then she could access the document. Such an attack is particularly relevant when a user can have multiple nationalities.

In this paper, we focus on the behaviour of authorisation policies when facing attribute hiding attacks. More precisely, we introduce the notion of *resistance*: a policy is resistant when every query obtained by adding information to an allowed query is also allowed. This definition generates the following research questions:

1. Is it possible to automatically detect whether a policy is resistant or not?
2. If a policy is not resistant, can we exhibit a counter-example?
3. If a policy is resistant, can we construct a formal proof of it?

The main contribution of this paper is to positively answer these questions, at least in a partial way. In order to do so, we present the tool ATRAP (Automated Term Rewriting for Authorisation Policies), which combines the term-rewriting tool Maude [8] and the proof assistant Isabelle/Isar [20,25] to analyse the resistance of PTaCL policies. The core of the tool is written in Java and handles the communication between Maude and Isabelle by generating the respective inputs and interpreting their results. ATRAP is capable of generating counter-examples for non resistant policies, and of building an Isabelle proof for resistant policies. Although ATRAP is sound, i.e., all counter-examples and proofs generated are indeed correct, ATRAP is not (yet) complete, as it may fail in some cases to find a counter-example in a reasonable amount of time or to build a proof.

The rest of the paper is structured as follows: in Section 2. we present the language PTaCL and introduce its evaluation in Maude. In Section 3, we define the notion of policy resistance and show how this property can be verified in practice. In Section 4, we describe our approach to find a counter-example of resistance, and in Section 5, we present how a *proof* of resistance can be automatically generated, after which we conclude and present future work in Section 6.

Related Work

The notion of resistance introduced in this paper is related to that of safety given in [24], which roughly states that the evaluation of a request should be "lower" than that of a request with strictly more information. This notion is close to that of (weak)-monotonicity [10]. As shown in Section 3, a (weakly) monotonic policy is also resistant, while the converse does not always hold.

There exist several approaches using model-checking to analyse access control policies [15,27]. For instance, SMT solving can be used to check whether a given request can eventually be granted with a particular role in Administrative-RBAC policies [1,2]. Similarly, XACML policies can be automatically compared using a SAT solver [16]. To the best of our knowledge, we are the first to automatically analyse the resistance of policy, and also to allow for the generation of a structured proof.

The automation of ATRAP relies on term-rewriting, which has already been considered for the formalisation of access control [3,11], leading to the analysis of rewrite-based policies [18]. In particular, Bertolissi and Uttha propose in [4] to relate the properties of a rewrite system, such as totality or consistency, with those of an access control policy encoded in this system. This approach allow them to use the rewrite system CiME to generate proof certificates for these properties in the proof assistant CoQ. We follow a similar objective here, in that we generate proof certificates using a rewrite system, however we focus on the notion of resistance, which is not a property of the rewrite system itself, but of the policy.

ATRAP relies on the encoding of PTaCL in Isabelle, mentioned in [10], in order to generate a proof of resistance. Brucker et al. present in [6] an encoding of an access control model into HOL, in the context of healthcare policies. Finally, it is worth mentioning that some of the techniques used in ATRAP are inspired by previous work [13], where the basic idea of using term-rewriting to generate proofs is used in the context of program refinement.

2 PTaCL

XACML 3.0 [22] is an OASIS standard for representing authorisation policies, and given an access request, its complete request evaluation cycle can be summarised as follows: (i) the request is submitted to the Policy Enforcement Point (PEP); (ii) the PEP forwards the request to the Context Handler (CH); (iii) the CH collects all attributes necessary to the evaluation of the request; (iv) the CH forwards the complete request to the Policy Decision Point (PDP); (v) the PDP evaluates the complete request and returns the corresponding decision to the CH, which returns it to the PEP.

PTaCL [10] formalises the evaluation of the request by the PDP, which considers each request as complete, or more precisely, cannot make a distinction between incomplete and complete requests. In other words, if the CH is unable to collect some attributes, and forwards an incomplete request to the PDP, this request is evaluated in the same way than a complete one.

$\sqcap$	1	0	$\bot$
1	1	0	$\bot$
0	0	0	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

(a) Weak operators

$\sqcup$	1	0	$\bot$
1	1	1	$\bot$
0	1	0	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\tilde{\sqcap}$	1	0	$\bot$
1	1	0	$\bot$
0	0	0	0
$\bot$	$\bot$	0	$\bot$

$\tilde{\sqcup}$	1	0	$\bot$
1	1	1	1
0	1	0	$\bot$
$\bot$	1	$\bot$	$\bot$

(b) Strong operators

X	$\neg X$	$\sim X$
1	0	1
0	1	0
$\bot$	$\bot$	0

(c) Unary

Fig. 1. Binary and unary operators on the target decision set $\{1, 0, \bot\}$

We present in this section the language PTaCL through the description of an illustrative example. We introduce the different definitions required for the understanding of this example, and refer to [10] for further details about the language. We take as example the one given in the introduction, where the access to a document is based on the nationality of the requester.

2.1 3-valued Logic

The 3-valued logic extends the traditional Boolean logic $\{1, 0\}$, where 1 represents *true* and 0 represents *false*, by considering an additional value $\bot$ [19]. The usual Boolean operators, such as the conjunction, disjunction, negation, etc, can be extended to the set $\{1, 0, \bot\}$, as shown in Fig. 1. The weak operators consider the value $\bot$ as absorbing, while the strong ones "resolve" $\bot$ as much as possible.

The logic $\{1, 0, \bot, \tilde{\sqcup}, \sim, \neg\}$ is proven in [10] to be functionally complete, using a result from Jobe [17], which means that any logical operator can be built from these operators and constants. In the following, we sometimes use the set $\{1, 0, \bot, \tilde{\sqcap}, \sim, \neg\}$, since $x \,\tilde{\sqcap}\, y = \neg(\neg x \,\tilde{\sqcup}\, \neg y)$. In addition, we use the three valued logic both to represent the result of target evaluation and the result of policy composition. In order to avoid any confusion, we use $\{1_\mathrm{T}, 0_\mathrm{T}, \bot_\mathrm{T}\}$ for the former, and $\{1_\mathrm{P}, 0_\mathrm{P}, \bot_\mathrm{P}\}$ for the latter, whose meaning will be given in due course.

2.2 Target and Policy

Following recent work [23,22], PTaCL is attribute-based, meaning that a request is modeled as a set of attribute name-value pairs. Our running example uses an attribute **nat**, whose value can be either FR or AT. For instance, the request $\{(\mathbf{nat}, \mathsf{FR})\}$ represents a request made by a French national.

In addition, PTaCL is target-based [5,7,9,21,22,26], meaning that an access control policy contains a target that specifies the requests to which the policy is applicable, and a body (either a single decision or another policy) describing how applicable requests should be evaluated.

In its simplest form, an *atomic target* is a pair (n, v), where n is an attribute name and v is an attribute value. For instance, the target $(\mathbf{nat}, \mathsf{FR})$ evaluates to 1_T (match) if the request contains $(\mathbf{nat}, \mathsf{FR})$, to 0_T (no-match) if it contains $(\mathbf{nat}, \mathsf{AT})$, but not $(\mathbf{nat}, \mathsf{FR})$, and to $\bot_\mathrm{T}$ (indeterminate) if it does not contain any value for the attribute **nat**. In other words, PTaCL can distinguish between a non-matching value for an attribute and a missing attribute. More formally,

Table 1. PTaCL evaluation

	t_1	p_1	t_2	p_2
$\emptyset$	$\bot_T$	$\{1_P, 0_P\}$	$\bot_T$	$\{1_P, 0_P\}$
$\{(\mathbf{nat}, \mathsf{FR})\}$	0_T	$\{1_P\}$	1_T	$\{1_P\}$
$\{(\mathbf{nat}, \mathsf{AT})\}$	1_T	$\{0_P\}$	0_T	$\{0_P\}$
$\{(\mathbf{nat}, \mathsf{FR}), (\mathbf{nat}, \mathsf{AT})\}$	1_T	$\{0_P\}$	1_T	$\{1_P\}$

the semantics of an atomic target (n, v) for a request $q = \{(n_1, v_1), \cdots, (n_k, v_k)\}$ is given as:

$$[\![(n,v)]\!](q) = \begin{cases} 1_T & \text{if } (n, v') \in q \text{ and } v = v', \\ \bot_T & \text{if } (n, v') \notin q, \\ 0_T & \text{otherwise.} \end{cases}$$

More complex targets can be built using the logical operators not_T, for the negation of a target, opt_T, for the optional target (i.e., transform $\bot_T$ into 0_T) and and_P, for the strong conjunction of targets, interpreted by $\neg$, $\sim$ and $\sqcap$, respectively. Since this set of operators is functionally complete ($\sqcap$ can be built from $\tilde{\sqcup}$ and $\neg$ [10]), any other logical combination can be achieved with them.

Finally, an *authorisation policy* can be defined as single decision, i.e., either 1_P (allow) or 0_P (deny), a targeted policy (t, p), where t is a target, or a logical composition of two policies, using the operators not_P for the negation of a policy, dbd_P for the deny-by-default of a policy, or and_P for the conjunction of two policies, interpreted by $\neg$, $\sim$ and $\tilde{\sqcap}$, respectively. Here again, these three operators suffice to build any other logical operator. Given an access request, the evaluation of a policy returns the set of all possible decisions. The logical operators are therefore extended in a point-wise way, and the evaluation of a targeted policy (t, p) for a request q is given by:

$$[\![(t,p)]\!]_P(q) = \begin{cases} [\![p]\!]_P(q) & \text{if } [\![t]\!]_T(q) = 1_T, \\ \{\bot_P\} & \text{if } [\![t]\!]_T(q) = 0_T, \\ \{\bot_P\} \cup [\![p]\!]_P(q) & \text{otherwise.} \end{cases}$$

where $\bot_P$ represents the not-applicable decision. For instance, the policy p_1 that explicitly denies any access to Austrian citizens and otherwise allows the access can be defined as:

```
t1 :: (Tatom "nat" "AT")
p1 : Pnot (Pdbd (Pnot (Ptar t1 (Patom Zero))))
```

We adopt a declarative syntax, where the double-colon is used for target definition, and a single-colon for policy definition. In the above, the target `t1` is defined as the atomic target (**nat**, AT) with the keyword `Tatom`, `Patom Zero` represents the atomic policy that always returns 0_P (whereas `One` represents the decision 1_P), `Ptar t1 (Patom Zero)` is the above policy guarded by the target `t1`, and thus evaluates to: $\{0_P\}$ if `t1` evaluates to 1_T; to $\{\bot_P\}$ if `t1` evaluates to 0_T; and to $\{0_P, \bot_P\}$

if t1 evaluates to $\perp_T$. Furthermore, Pnot and Pdpd defines the negation and deny-by-default operators, and therefore the constructor Pnot (Pdbd (Pnot x)) acts as an allow-by-default operator, i.e., transforms $\perp_P$ to 1_P. Similarly, the policy p_2 that explicitly authorises any access to French citizens and otherwise denies the access can be defined as:

```
t2 :: (Tatom "nat" "FR")
p2 : Pdbd (Ptar t2 (Patom One))
```

The evaluation of p_1 and p_2 for four different requests is given in Table 1. Note that the evaluation might return more than one decision, which can be interpreted as an inconclusive decision. In XACML, such decisions are defined by the Indeterminate decision. The way an inconclusive decision is concretely interpreted by the PEP is left to the implementer, and might vary from a risk-advert approach (e.g., any inconclusive decision is interpreted as 0_P) to a risk-prone approach (e.g., if 1_P is a possible decision, then the PEP allows the request).

2.3 Maude Evaluation

ATRAP uses the term rewriting system Maude [8] to model PTaCL and dynamically generate and evaluate requests for a given policy. The syntax for the PTaCL terms in Maude closely resembles the syntax given above. Based on this formalisation, we define *equations* and *rewrite rules* to manipulate the syntax tree based on pattern matching.

To evaluate a request, we model the operators for *targets*, *policies* and the three valued logic used in PTaCL. The definition of an operator starts with the keyword op, followed by a pattern that allows parameters at positions marked with "_" (underline), and the signature of the operator after a ":" (colon), where the list on the left hand side of -> defines the parameters, and the right hand side the result type. The definition is completed by a "." (full stop). To exemplify the notation we give the definitions for decisions and policy operations:

```
op ALLOW : -> decision .
op DENY : -> decision .
op BOT : -> decision .

op Patom _ : decision -> policy .
op Pnot _ : policy -> policy .
op Pdbd _ : policy -> policy .
op Pand _ _ : policy policy -> policy .
```

The first three lines give parameter-free operators to define the decisions for 1_P (ALLOW), 0_P (DENY) and $\perp_P$ (BOT) respectively. The decisions are prefixed with the keyword Patom to form a basic policy, and combined with Pnot, Pdbd and Pand to form more complex expressions. Similar operators exist for the targets, where the basic element Tatom holds a key-value pair for the attributes. While these operators capture the structure of the policies, operators can also be associated with equations to modify them or evaluate requests. Equations correspond to operators and are defined using the keyword eq. When one of the

patterns on the left of the = matches, it is replaced by the pattern on the right hand side.

```
op dbd _ : decision -> decision .
eq dbd DENY = DENY .
eq dbd ALLOW = ALLOW .
eq dbd BOT = DENY .

op strongand _ _ : decision decision -> decision [assoc comm] .
eq strongand ALLOW d = d .
eq strongand DENY d = DENY .
eq strongand d d = d .
```

While the *deny-by-default* operator `dbd` has only one parameter and replaces possible `BOT` values by `DENY`, the equations can also contain variables which match all possible patterns for the respective type. The keywords `assoc` and `comm` declare that the operator for strongand is associative and commutative respectively, which is considered by Maude in pattern matching (e.g., we only need `strongand ALLOW d` and can omit `strongand d ALLOW`). Note that while in general associative-commutative rewriting is NP complete, Maude supports effective algorithms for handling the equational rewriting steps for typical patterns in time proportional to the logarithm of the term size [12]. The definitions for equations are evaluated from top to bottom. That is, for the equations above, `eq strongand d d = d` is only considered if both of the parameters are `BOT`. In addition to the policies, we define *requests* as sets of `(key =? value)` pairs and an operation `peval` that evaluates a request on a given policy.

ATRAP uses this formalisation in three ways: to evaluate a request against a policy, to compute a counter-example demonstrating that a policy is not resistant, and to search for a proof tree that shows the resistance of a policy.

3 Policy Resistance

To introduce the notion of policy resistance, consider the evaluation of the request $\{(\mathbf{nat}, \mathsf{FR}), (\mathbf{nat}, \mathsf{AT})\}$ with the policy p_1, as given in Table 1: this request, corresponding to a user with both French and Austrian citizenships, is initially denied; however, if the attribute (**nat**, AT) is "removed", then the request becomes allowed.

In general, several reasons can explain the absence of an attribute in a request, such as an error during the transmission of the attributes, the expiration of the attribute certificate, the non-existence of this attribute, etc. In particular, an attribute might be withheld intentionally (e.g., a user does not want to disclose her address for privacy reasons), by mistake (e.g., a user is not aware of the fact that it should be disclosed) or maliciously (e.g., a user wants to hide some "negative" information).

In the latter case, the omission of an attribute can be seen as an *attribute hiding attack* [10] from a user, trying to gain a better answer by hiding some information. A policy is resistant when it is able to resist to such attacks:

Definition 1. *A policy p is resistant if, and only if, for any requests q and q', if $q' \subseteq q$ and if $[\![p]\!](q') = \{1_P\}$, then $[\![p]\!](q) = \{1_P\}$.*

In other words, if a request q is not allowed, then any sub-request $q' \subseteq q$ is also not allowed. For instance, we can observe that the policy p_1 is not resistant, while the policy p_2 is.

There are many ways to prove that a policy p is resistant, the most straightforward one being to exhaustively check any pair of requests q, q' such that $q' \subseteq q$. We describe an implementation of this approach in Section 4 using Maude, together with the notion of a normal form for requests, allowing us to reduce the set of requests to check.

In some cases, we can also use the *structure* of the policy to prove its resistance. For instance, the policy `Ptar t (Patom Zero)` clearly evaluates, for any request, either to $\{0_P\}$ or to $\{0_P, \bot\}$, regardless of the definition of `t`, and is therefore trivially resistant. Generalising this example, we can observe (and formally prove) that if a policy cannot return 1_P, then it is resistant. Furthermore:

- if `p` cannot return 1_P, then `Ptar t p` cannot return 1_P, for any `t`;
- if `p` cannot return 1_P, then `Pdbd p` cannot return 1_P;
- if `p` cannot return 1_P, then `Pand p p1` and `Pand p1 p` cannot return 1_P;
- if `p` cannot return 1_P, then `Pnot p` cannot return 0_P;
- if `p` cannot return 0_P, then `Pnot p` cannot return 1_P;
- if `p` cannot return 0_P, then `Ptar t p` cannot return 0_P.

In other words, it might be possible to prove that a policy is resistant simply by inspecting its structure, without checking each possible pair of requests. As another example, a policy without any target clearly evaluates identically for any request (since all requests are equally applicable), and therefore is resistant.

In addition, the notion of weak-monotonicity is introduced in [10], such that, intuitively speaking, a target is weakly monotonic if removing information from a request lowers the evaluation of the target.

Definition 2. *A target t is* weakly monotonic *if for all requests q and for every $q' \subseteq q$, $[\![t]\!](q') \preccurlyeq [\![t]\!](q)$, where $\preccurlyeq$ is the reflexive closure of $\bot_T \prec 0_T \prec 1_T$. A policy p is weakly monotonic if and only if every target in p is weakly monotonic.*

Any atomic target (n, v) is weakly-monotonic, and the operators opt_T and and_T preserve the weak-monotonicity [10]. In other words, any policy whose targets are built without the operator not_T is weakly monotonic. Moreover, as a direct result from Theorem 6 of [10] (which is recalled in [14]), we have:

- If `p` is weakly monotonic and built without dbd_P, then it is resistant;
- If `p` is weakly monotonic and built without not_P, then it is resistant;

Finally, the notion of resistance can be proved in a compositional way:

- If `p` is resistant, then `Pdbd p` is resistant;
- If `p1` and `p2` are resistant, then `Pand p1 p2` is resistant.

These rules make it possible to prove the resistance of a conjunctive policy in a different way for each sub-policy. Clearly, all the rules presented in this section are only implications, and a policy might be resistant even though it does not satisfy any of them. We show in Section 5 how ATRAP can use these rules, together with their encoding in Isabelle and Maude, to automatically build a proof of resistance.

Remark 1. A (strongly) monotonic policy, as defined in [10] is also trivially resistant. However, in order to prove the (strong) monotonicity of an atomic target, attributes must be assumed to be *compact*, i.e., either all the values of an attribute are given, or none are. For instance, the compactness of the attribute **nat** would mean that a user can either hide all of her nationalities, or none of them, but cannot only hide one. Hence, proving resistance using (strong) monotonicity requires the assumption of compactness from the environment, whereas we aim here at generating *complete* proofs, i.e., without assumption. The integration of such assumptions when all other strategies have failed is left for future work.

4 Search for Non-resistance

We use the PTaCL encoding in Maude to automatically search for possible counter-examples to resistance. A naive approach for a policy p would simply consist in checking any two requests q and q', such that $q' \subseteq q$ and $[\![p]\!](q') = \{1_P\}$, in order to ensure that $[\![p]\!](q) = \{1_P\}$. However, the set of all possible requests can potentially be very large. For instance, in our previous example, focusing only on the **nat** attribute, the United Nations Organisation currently counts 193 members[1], meaning there are 193 pairs $(\mathbf{nat}, v)$ possible, and thus $\mathbf{card}(\mathcal{Q}) = 2^{193}$.

In order to simplify the search, we first introduce the *normal form* of a request for a given policy. We then describe our correct and complete search for counter-examples in Maude, and finally we present some experimental results.

4.1 Normal Form of Requests

Intuitively speaking, the evaluation of a request against a policy mostly depends on whether the request contains the atomic targets present in the policy: given an attribute n, all pairs (n, v) that do not explicitly appear in the policy are evaluated in the same way. For instance, consider the policy p_1: it is clear that the pairs $(\mathbf{nat}, \mathsf{FR})$, $(\mathbf{nat}, \mathsf{DE})$ or $(\mathbf{nat}, \mathsf{UK})$ are evaluated similarly.

Hence, given a policy p, we write $\mathcal{A}(p)$ for the set of atomic targets appearing in p. For instance, $\mathcal{A}(p_1) = \{(\mathbf{nat}, \mathsf{AT})\}$ and $\mathcal{A}(p_2) = \{(\mathbf{nat}, \mathsf{FR})\}$. Given an attribute n and a policy, we write $fv(p, n)$ for a fresh value of n with respect to p, i.e., a value such that $(n, fv(p, n)) \notin \mathcal{A}(p)$. If p explicitly mentions all possible values for n, we define $fv(p, n)$ to return a random value for n. Finally, the normal form of a request q for a policy p is given by keeping all pairs (n, v) that appear both in q and $\mathcal{A}(p)$, and replacing any (n, v) in q that does not appear in $\mathcal{A}(p)$ by $(n, fv(p, n))$. More formally:

[1] `http://www.un.org/depts/dhl/unms/whatisms.shtml#states`

Definition 3. *The normal form of a request q is given by:*

$$\mathsf{nf}_p(q) = (q \cap \mathcal{A}(p)) \cup \{(n, fv(p, n)) \mid \exists v\ (n, v) \notin \mathcal{A}(p) \wedge (n, v) \in q\}$$

Given a set of requests $\mathcal{Q}$ and a policy p, we write $\mathsf{NF}_p(\mathcal{Q}) = \{\mathsf{nf}_p(q) \mid q \in \mathcal{Q}\}$. In the following, we omit the subscript when p is clear from context. For instance, the set of requests in normal form for the policy p_1 is given by:

$$\mathsf{NF}_{p_1}(\mathcal{Q}) = \{\emptyset, \{(\mathbf{nat}, \mathsf{AT})\}, \{(\mathbf{nat}, \mathsf{NV})\}, \{(\mathbf{nat}, \mathsf{NV}), (\mathbf{nat}, \mathsf{AT})\}\}$$

where NV represents the fresh value for the attribute **nat**, i.e., $\mathsf{NV} = fv(p_1, \mathbf{nat})$.

This notion of normal is consistent both with policy evaluation and request inclusion (the proofs can be found in [14]).

Proposition 1. *For any policy p and any request q, $[\![p]\!](q) = [\![p]\!](\mathsf{nf}(q))$.*

Proposition 2. *Given any requests q and q', if $q' \subseteq q$, then $\mathsf{nf}(q') \subseteq \mathsf{nf}(q)$.*

Remark 2. The simplicity of the normal form comes from the fact that PTaCL does not allow for complex atomic targets, such as the comparison of attribute values. For instance, one cannot directly write the atomic target stating that **age** must be lower than 18, and must instead generate the disjunction of all atomic target for **age** between 0 and 18. Note however that, as long as attributes have a finite domain, all possible targets can be defined, hence PTaCL can be seen here as a low-level language, designed for policy analysis. The design of richer atomic targets is planned for future work.

4.2 Search for Counter-Examples

As a consequence of Proposition 1 and Proposition 2, the resistance of a policy can be decided only by looking at the set of requests in normal form.

Proposition 3. *A policy p is resistant if, and only if, given q and q' in $\mathsf{NF}_p(\mathcal{Q})$, if $q' \subseteq q$ and if $[\![p]\!](q') = \{1_{\mathrm{P}}\}$, then $[\![p]\!](q) = \{1_{\mathrm{P}}\}$.*

In other words, we can restrict our attention to $\mathsf{NF}_p(\mathcal{Q})$, whose size is bounded by $2^{|\mathcal{A}(p)|+n}$, where n stands for the number of attributes, instead of $\mathcal{Q}$, whose size is 2^N, where N is the number of all possible attribute name-values pairs. Finally, it is worth observing that, in order for a policy p to be resistant, it is enough to check, for any allowed request q, whether removing any pair attribute value changes the decision. More formally:

Proposition 4. *A policy p is resistant if, and only if, for any request q such that $[\![p]\!](q) \neq \{1_{\mathrm{P}}\}$, if $[\![p]\!](q \setminus \{(n, v)\}) \neq \{1_{\mathrm{P}}\}$, for any attribute n and any value v.*

Combining Propositions 4 and 3, we conclude that to find a counter-example to resistance, we only need to check all pairs (q, q'), where $q, q' \in \mathsf{NF}_p(\mathcal{Q})$ and $q = q' \cup \{(n, v)\}$. Proposition 4 allows us to reduce the number of comparisons needed

for the search for counter-examples from an upper bound of $2^{|\mathcal{A}(p)|+n} \times 2^{|\mathcal{A}(p)|+n}$ (comparison of all subsets) to $2^{|\mathcal{A}(p)|+n} \times (|\mathcal{A}(p)| + n)$, i.e., where each subset needs to be checked against at most $|\mathcal{A}(p)| + n$ direct subsets.

The seach for counter-examples is performed by generating and evaluating the largest possible request $q_m = \mathcal{A}(p) \cup \{(n, fv(n)) \mid (n, v) \in \mathcal{A}(p)\}$, and systematically removing attributes to see if a reduction of a request (hiding of an attribute) can lead to an increase of access in the policy. This manipulation of requests is performed using *rewrite rules*, which are defined similarly to equations, but in contrast to them are not evaluated deterministically, i.e., may be executed whenever the left hand side pattern matches.

The Maude command for the search has the following form:

search sres(bldevallist (policy Requests Defs DecsList)) =>* error (x:DecsList).

where policy is the ID of a top level policy to check, the set Requests is the maximal request in normal form, and Defs are the policy definitions. The operator bldevallist repeatedly removes attribues from the request, evaluates it with respect to the policy, and stores the result in DecsList. The operator sres traverses the list and searches for pairs that violate resistance, in which case the violating request pair is wrapped into an error operator. Removing an element from the set Request is nondeterministic, and thus may generate different lists of decisions. The maude command search systematically explores all the possible outcomes and returns those that match the search command, resp. can be rewritten to an error label. For instance, when analysing the policy p_1, ATRAP outputs:

```
Counter-example #1
["nat" =? "new_value"]:                [ALLOW]
["nat" =? "AT", "nat" =? "new_value"]: [DENY]
```

4.3 Experimental Results

We experiment the search for counter-examples by randomly generating some policies, and executing the search for each policy. We write $P\langle m, n, k, l, r\rangle$ for a set of r random policies, such that m stands for the maximal height of each policy, n for the maximal width of each target, k for the number of attributes and l for the number of values for each attribute.

Figure 2 represents the execution time (on 2 GHz Intel core i7 with 8GB of RAM) for the search of counter-examples for each $p \in P\langle 4, 4, 4, 4, 300\rangle$, indexed by $\mathcal{A}(p)$ and with a logarithmic scale for the execution time. Over the 300 policies, 252 are resistant, and this ratio of 0.85 is consistent with other experiments, and seems to be independent of the policy dimensions. Note that the times shown in this Figure are for complete searches, i.e., searches finding *all* possible counter-examples. Hence, the time required to analyse a policy is the same whether the policy is resistant or not. As expected from the theoretical analysis of the previous section, the search for counter is exponential in the size of $\mathcal{A}(p)$.

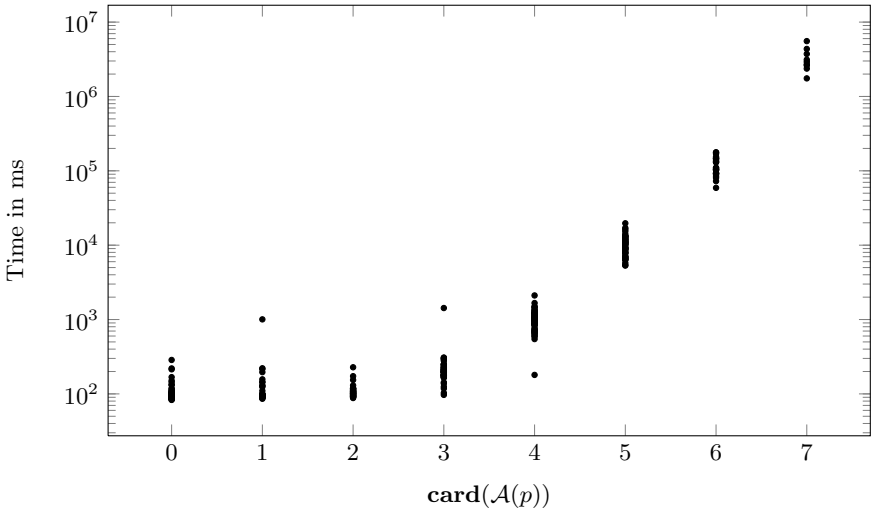

Fig. 2. Automatic search for counter-examples for $P\langle 4, 4, 4, 4, 300\rangle$

5 Certification of Resistance

If no counter-example is found, ATRAP tries to build a proof of resistance of the policy, following a proof-obligation discharge approach, where Maude is used to automatically generate a proof that can be checked in Isabelle [20,25]. To reach this goal, ATRAP encodes the policy and the deduction mechanism in rewrite logic and calls Maude to perform a search for the proof. The result is parsed into Java classes and mapped into a corresponding proof in Isabelle, which is then called to validate the proof. It is worth mentioning that the encoding in Maude of the deduction mechanism is independent to the counter-examples search described in the previous section, although both techniques share the encoding of PTaCL.

Our approach revolves around two main entities: proof obligation and proof technique. A *proof obligation* corresponds to a goal, i.e., to a property that we want to prove. The top-level goal is a proof of resistance, which in turn may require further sub-goals like weak-monotonicity to form a complete proof tree. A *proof technique* describes a method to discharge a proof obligation, using some rules described in Section 3.

Rather than encoding the proofs manually in Isabelle, we use Maude to search for viable proof trees that are then encoded and checked in Isabelle. To facilitate this goal, each proof obligation and technique have corresponding facets in Maude, Isabelle and Java. In Maude, the facet of a proof obligation is represented as an operator (e.g., `isResistant`) that takes a policy or target ID and a proof technique. In Java this is implemented in form of public classes, which have a field `statement` corresponding to the Isabelle definition of the proof obligation. For instance, the field `statement` of the class `ResistanceProof` is equal to `resistant p q1 q`, where `p` is the name of the policy over which this class is

defined and where q and q1 are the universally quantified variables representing the variables q and q' of Definition 1. The Isabelle facet of a proof obligation is then a definition over either a target or a policy. For instance, the definition of resistance in Isabelle is given as:

```
definition resistant :: " policy ⇒ request ⇒ request ⇒ bool" where
" resistant p q1 q ≡ (set q1)⊆(set q) → peval p q1 = {One} → peval p q = {One}"
```

The Maude facet of a proof technique is an operator defined over policies and/or targets. For instance, let r_1 be the rule stating that a policy is resistant if it is weakly-monotonic and without the not_P operator. Note that for technical reasons, we also need to impose that a policy is well-formed, i.e., each atomic policy is either 1_P or 0_P, and not $\bot_\mathrm{P}$. This rule is represented in Maude as the operator ResWFWMWNProof. A proof technique is implemented in Java as a class local to that corresponding to the proof obligation. For instance, in order to define the rule r_1, the class ResistanceProof comes with a local class WeakMonotonicityWithoutNotWF. This class is defined with three fields: WellFormedProof, WithoutNotProof, WeakMonotonicPolicyProof, each being a public class. In other words, in order to use the rule r_1, one must first exhibit a proof that the policy is well-formed, a proof that it is built without the not_P constructor, and a proof that it is weakly-monotonic. Finally, this structure is mapped to Isabelle, where the proof technique for a proof obligation is a proven lemma whose goal is that proof obligation. For instance, the lemma corresponding to rule r_1 is defined in Isabelle as[2]:

```
lemma weak_monotonic_without_not_resistant :
" well_formed_policy p ⇒ weak_pmonotonic p q1 q ⇒ policy_without_not p
  ⇒ resistant p q1 q"
```

To generate the proof tree, each rule of Section 3 is modelled in Maude in a basic, compositional form. Starting from the initial goal of proving resistance of a policy p, the proof generation then follows the structure of the policy to generate new proof obligations for sub-proofs according to the components and properties of p.

While binary operators like Pand trigger sub-proofs for both operands, the proof generation is also guided by properties and preconditions of the lemma to apply. To show, e.g., resistance using the rule r_1, we need to establish well-formedness, weak-monotonicity, and check that the policy does not use not_P. Well-formedness and use of not_P can easily be checked by Maude doing a syntactic check. Only if both conditions are fulfilled, a proof for weak monotonicity is instantiated. When all conditions for a proof are fulfilled, the *proof obligation* is replaced by a description of the actual proof. For instance, the rewrite rule corresponding to the rule r_1 is given as:

```
   genproof( isResistant (p, noproof),
             isWF(p, pr1), isWN(p, pr2), isWM(p, pr3),pis | defs )
=> genproof(isResistant(p, ResWFWMWNProof(p)),
             isWF(p, pr1), isWN(p, pr2), isWM(p, pr3), pis | defs)
```

[2] All ATRAP lemmas are available at `http://www.morisset.eu/atrap/`.

```
theory p2 imports atrap begin
definition t2 :: target where "t2 = (Tatom ''nat'' ''FR'')"
definition pone :: policy where "pone = Patom One"
definition pt :: policy where "pt = Ptar t2 pone"
definition p2 :: policy where "p2 = Pdbd pt"

lemma "resistant p2 q1 q" proof −
  have wf: " well_formed_policy p2"
    by (simp add: p2_def pt_def pone_def t2_def)
  have without_not: " policy_without_not p2"
    by (simp add: p2_def pt_def pone_def t2_def)
  have weak_monotonic: "weak_pmonotonic p2 q1 q" proof −
    have wm_p: "weak_pmonotonic pt q1 q" proof −
        have wm_p: "weak_pmonotonic pone q1 q" by (simp add: pone_def)
        have wm_t: "weak_tmonotonic t2 q1 q"
          by (simp add: tatom_weak_monotonic t2_def)
        from wm_p wm_t show ?thesis by (simp add:pt_def) qed
    from wm_p show ?thesis by (simp add:p2_def) qed
  from wf without_not weak_monotonic show ?thesis
    by ( insert weak_monotonic_without_not_resistant [of p2 q1 q], simp)
qed
```

Fig. 3. Generated Isabelle Proof

where noproof indicates that the proof obligation was not fulfilled yet, and pr1, pr2, and pr3 are previously generated subproofs. The variables pis and defs hold the available sub-proofs and policy definitions respectively. This approach also allows for manual intervention by the user by supplying the generation mechanism by external information about the system or predefining proof obligations with their respective techniques.

Running Example. The policy p_2 is automatically proven to be resistant by ATRAP, which executes the following Maude command (where the policy pt is introduced as an intermediary step):

```
rew genproof( isResistant (P "p2", noproof) | (T "t2"::( Tatom "nat" "FR"),
  P "pone"=Patom ALLOW, P "pt"=Ptar T "t2" P "pone", P "p2"=Pdbd P "pt")).
```

This proof-obligation is automatically discharged using the rules described above, and the following proof-obligation is returned:

```
isResistant (P "p2", ResWFWMWNProof(P "p2")), isWF(P "p2",WFBFProof(P "p2")),
isWN(P "p2", WNBFProof(P "p2")), isWM(P "p2", WMPdbd(P "p2",P "pt")),
isWM(P "pt", WMwithPtar(P "pt",T "t2",P "pone")),
isWM(T "t2" ,WMwithTAtom(T "t2")), isWM(P "pone", WMwithPAtom(P "pone")))
```

Informally, this proof can be read as follows: p2 is resistant, since it is well-formed, weakly-monotonic and without-not; p2 is well-formed which can checked by "brute-force", i.e., by checking the definition of p2; p2 is without the not_P operator, which can also be checked by "brute-force"; p2 is weakly-monotonic,

since it is the deny-by-default of the weakly-monotonic policy pt; pt is weakly-monotonic, since it is the composition of the weakly-monotonic target t2 and of the weakly-monotonic policy pone; t2 is weakly-monotonic, since it is atomic; and pone is weakly-monotonic since it is atomic.

ATRAP parses this Maude proof-obligation, and using the Java mechanism described above, the following Isabelle theory is automatically generated. This theory is built on the logic atrap.thy, which includes the definition of the three-valued logic, the definition of PTaCL, and the lemmas corresponding to the different rules described in Section 3.

Fig. 3 presents the generated proof for the running example. For the sake of compactness, we do not go through Isabelle/Isar's syntax, but intuitively, the structure of the proof follows the informal description given above, and the tactics used are limited to the simplification tactic (which unfolds the definition of the entities involved in the proof), and the insertion of existing lemmas. It is worth observing that the generated proof is human readable, and is structurally very close to the corresponding mathematical proof. We believe this aspect to be particularly important as a security designer is not necessarily an expert in proof techniques, and ATRAP provides a high-level proof, without having to rely blindly on a verification tool.

5.1 Experimental Results

We now evaluate the performance of ATRAP for the generation of resistance proof. The complexity of the proof generation mostly depends on the number of constructors of a policy p, which we refer to by $size(p)$, since the number of applicable rules directly depends on that number. Figure 4 shows some evaluation times for $P\langle 8, 3, 3, 3, 500\rangle$, where the y axis is logarithmic. As expected, the complexity of the proof search is exponential in $size(p)$, making the proof search most of the time faster than the search for counter-examples (the few exceptions to that rule come from policies p with large sizes, but small $\mathcal{A}(p)$, i.e., policies where a same target is repeated a large number of times).

However, although usually faster than the counter-examples search, the accuracy of the proof generation decreases with $size(p)$. For instance, writing T_n for $P\langle n, n, 2, 2, 1000\rangle$, for T_1, we generated the proof for all of the 957 resistant policies, this ratio falls to $905/920 \approx 0.98$ for T_2, to $762/883 \approx 0.86$ for T_3, to $659/864 \approx 0.76$ for T_4, to $558/852 \approx 0.66$ for T_5, to $503/861 \approx 0.58$ for T_6, etc.

Theses results, together with those of Section 4.3, should be seen as a validation of the ATRAP approach, rather than a "real-world" characterisation of policy resistance. Indeed, the policies analysed are randomly generated, and therefore the samples do not necessarily represent policies defined in a concrete context. In particular, it is not necessarily the case that about 85% of the policies enforced in existing information systems are resistant. Similarly, even though the accuracy of the proof search decreases with the number of constructors used, a very large policy whose targets consist only of conjunctions of atomic targets, and using only the operator $\mathsf{and_P}$, could be easily proved resistant.

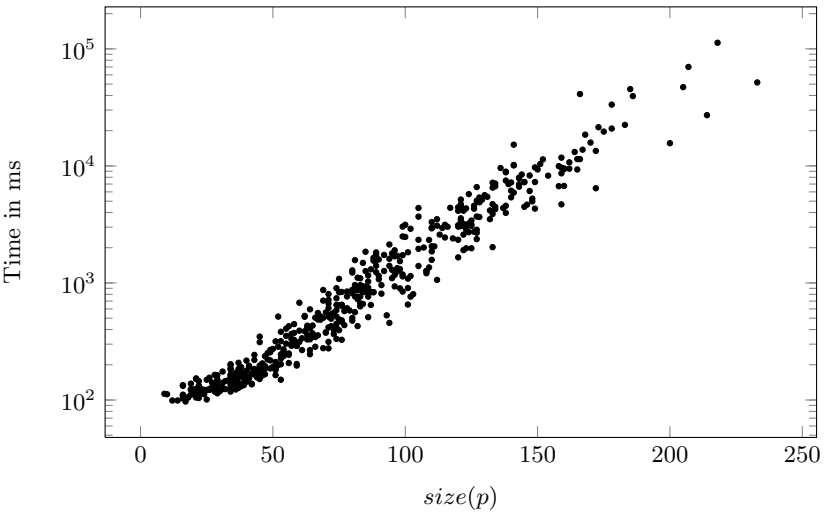

Fig. 4. Automatic search for proof for $P\langle 8, 3, 3, 3, 500\rangle$

6 Conclusion - Future Work

This paper presents the tool ATRAP, which, given a PTaCL policy, is capable of positively answer the three questions stated in the Introduction, i.e., can automatically detect the resistance of a policy by exhibiting a counter-example when it is not resistant, and, in some cases, by generating an Isabelle proof when it is. The different mechanisms are illustrated with two simple policies.

Using Proposition 3, we are able to limit the number of requests to evaluate when searching a counter-example, while maintaining the correctness and completeness of the approach. We have also implemented a collection of proof techniques allowing to prove efficiently the resistance of a policy, although large policies might fail to be proven automatically. However, the rules presented in Section 3 can also be seen as a policy construction guide, since a policy built using only those rules is resistant, by construction.

An interesting lead to explore for future work is to increase the interactivity between the search for counter-examples and the proof generation, by leveraging the different complexity of each approach. More precisely, we should take advantage of Maude to rewrite a policy to an equivalent one, such that the latter policy could be proven to be resistant more easily. This leads to the question of the existence of a normal form for policies, such that one could build a complete collection of resistance rules, i.e., if a policy in normal form is resistant, then there exists a set of structural proofs to prove it. At this stage, this remains as an open question.

We however believe that one of the strengths of our approach is its flexibility, and rules can be incrementally added. In order to so, one needs to provide the corresponding rule in Maude, together with Isabelle lemma and the Java class

linking the two. Clearly, an interesting future work would consist in formalising this extendability, by having an explicit, abstract notion of rule, with multiple facets, i.e., one facet for Maude, one for Isabelle, one for Java. We could also extend our approach to other properties of access control policies, for instance by relying on the underlying properties of the rewrite system [4].

Another relevant problem is the one of fixing a policy, in order to transform a non-resistant policy into a resistant one. This problem raises the question of policy "closeness", i.e., given a non-resistant policy, it is not enough to create a resistant one, we also need to ensure that the new policy is close enough to the original one. It is worth noting that ATRAP can generate partial proof of resistance, i.e., even if the whole policy is not resistant, it might identify some sub-policies that are, which might be helpful to fix a given policy.

Finally, as we mentioned, the current version of PTaCL can be seen as a low level language, and it would therefore be worth interfacing a "higher" level language with PTaCL, in order to analyse more complex policies, for instance include relational targets. XACML 3.0 would be a good candidate for such an extension, since both languages are attribute and target-based. In general, we plan to release ATRAP as an open-source software, and to generally optimise the searches for counter-examples and proofs. In this regard, it would be worth looking at parallel/distributed computation, since the evaluation of each pair of requests $(q, q \setminus \{(n, v)\}$ can be performed independently.

Acknowledgements. The authors would like to thank Jason Crampton for valuable discussions about PTaCL and policy resistance.

References

1. Alberti, F., Armando, A., Ranise, S.: Efficient symbolic automated analysis of administrative attribute-based rbac-policies. In: Proceedings of the 6th ACM ASIACCS 2011, pp. 165–175. ACM, New York (2011)
2. Armando, A., Ranise, S.: Scalable automated symbolic analysis of administrative role-based access control policies by smt solving. Journal of Computer Security 20(4), 309–352 (2012)
3. Barker, S., Fernández, M.: Term rewriting for access control. In: Damiani, E., Liu, P. (eds.) Data and Applications Security 2006. LNCS, vol. 4127, pp. 179–193. Springer, Heidelberg (2006)
4. Bertolissi, C., Uttha, W.: Automated analysis of rule-based access control policies. In: Proceedings of the 7th Workshop on Programming Languages Meets Program Verification, PLPV 2013, pp. 47–56. ACM, New York (2013)
5. Bonatti, P., De Capitani Di Vimercati, S., Samarati, P.: An algebra for composing access control policies 5(1), 1–35 (2002)
6. Brucker, A.D., Brügger, L., Kearney, P., Wolff, B.: An approach to modular and testable security models of real-world health-care applications. In: Proceedings of ACM SACMAT 2011, pp. 133–142. ACM, New York (2011)
7. Bruns, G., Huth, M.: Access control via Belnap logic: Intuitive, expressive, and analyzable policy composition. ACM Transactions on Information and System Security 14(1), 9 (2011)

8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
9. Crampton, J., Huth, M.: An authorization framework resilient to policy evaluation failures. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 472–487. Springer, Heidelberg (2010)
10. Crampton, J., Morisset, C.: PTaCL: A language for attribute-based access control in open systems. In: Degano, P., Guttman, J.D. (eds.) POST 2012. LNCS, vol. 7215, pp. 390–409. Springer, Heidelberg (2012)
11. Dougherty, D.J., Kirchner, C., Kirchner, H., Santana de Oliveira, A.: Modular access control via strategic rewriting. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 578–593. Springer, Heidelberg (2007)
12. Eker, S.: Associative-commutative rewriting on large terms. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 14–29. Springer, Heidelberg (2003)
13. Griesmayer, A., Liu, Z., Morisset, C., Wang, S.: A framework for automated and certified refinement steps. Innov. Syst. Softw. Eng. 9(1), 3–16 (2013)
14. Griesmayer, A., Morisset, C.: Automated certification of authorisation policy resistance. CoRR, abs/1306.4624 (2013), `http://arxiv.org/abs/1306.4624`
15. Guelev, D.P., Ryan, M., Schobbens, P.-Y.: Model-checking access control policies. In: Zhang, K., Zheng, Y. (eds.) ISC 2004. LNCS, vol. 3225, pp. 219–230. Springer, Heidelberg (2004)
16. Hughes, G., Bultan, T.: Automated verification of access control policies using a sat solver. Int. J. Softw. Tools Technol. Transf. 10(6), 503–520 (2008)
17. Jobe, W.: Functional completeness and canonical forms in many-valued logics. Journal of Symbolic Logic 27(4), 409–422 (1962)
18. Kirchner, C., Kirchner, H., Santana de Oliveira, A.: Analysis of rewrite-based access control policies. Electron. Notes Theor. Comput. Sci. 234, 55–75 (2009)
19. Kleene, S.: Introduction to Metamathematics. D. Van Nostrand, Princeton (1950)
20. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
21. OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0. Committee Specification (2005)
22. OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0. Committee Specification 01 (2010)
23. Rao, P., Lin, D., Bertino, E., Li, N., Lobo, J.: An algebra for fine-grained integration of XACML policies. In: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, pp. 63–72. ACM, New York (2009)
24. Tschantz, M., Krishnamurthi, S.: Towards reasonability properties for access-control policy languages. In: Ferraiolo, D., Ray, I. (eds.) 11th ACM Symposium on Access Control Models and Technologies, Proceedings, SACMAT 2006, pp. 160–169. ACM (2006)
25. Wenzel, M.: The isabelle/isar reference manual (2007)
26. Wijesekera, D., Jajodia, S.: A propositional policy algebra for access control 6(2), 286–235 (2003)
27. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems through model checking. J. Comput. Secur. 16(1), 1–61 (2008)