

An Authorization Framework Resilient to Policy Evaluation Failures

Jason Crampton¹ and Michael Huth²

¹ Information Security Group, Royal Holloway, University of London

² Department of Computing, Imperial College London

Abstract. In distributed computer systems, it is possible that the evaluation of an authorization policy may suffer unexpected failures, perhaps because a sub-policy cannot be evaluated or a sub-policy cannot be retrieved from some remote repository. Ideally, policy evaluation should be resilient to such failures and, at the very least, fail “gracefully” if no decision can be computed. We define syntax and semantics for an XACML-like policy language. The semantics are incremental and reflect different assumptions about the manner in which failures can occur. Unlike XACML, our language uses simple *binary* operators to combine sub-policy decisions. This enables us to characterize those few binary operators likely to be used in practice, and hence to identify a number of strategies for optimizing policy evaluation and policy representation.

1 Introduction

Many access control models and systems are *policy-based*, in the sense that a request for access to protected resources is evaluated with respect to a policy that defines which requests are authorized. Many languages have been proposed for the specification of authorization policies, perhaps the best known being XACML [4,8,12]. However, it is generally acknowledged that XACML suffers from having poorly defined and counterintuitive semantics, see e.g. [9,10]. More formal approaches have provided well-defined semantics and typically use “policy operators” to construct complex policies from simpler sub-policies [3,5,13].

Each component of an XACML policy has a so-called target, and a policy is applicable to a request only if said request “matches” that policy’s target. XACML was designed to operate in heterogeneous, distributed environments, and XACML “policies” (technically, `<PolicySet>` elements) may reference sub-policies (`<Policy>` or `<PolicySet>` elements) that may be held in remote repositories. In addition to returning the usual allow and deny decisions, the result of evaluating an XACML policy may be “not applicable” or “indeterminate”, the latter since evaluations in open, distributed systems may fail.

There are three practical drawbacks to existing, more formal algebraic approaches to policy languages: first, it becomes difficult to answer the question “Is this request authorized?”, which is central to any access-control mechanism; second, it is difficult to see how practical policies can be written in this way; and finally, no means of handling policy evaluation failures has been provided.

Our goal in this paper is to develop a practical authorization language that is resilient to authorization evaluation failures, supports different assumptions for when failures may occur, has rigorous semantics, and leads to optimization of policy evaluation and policy representation. As a side effect, our framework allows for a simple static analysis that, at times, can fully recover from evaluation failures. The contributions of our paper include

- the definition of a simple policy language, which introduces the notion of resolution function for possible decisions;
- a concise characterization of the most commonly used decision-combining algorithms as *binary* decision-combining operators;
- the identification of two important classes of decision-combining operators, and a discussion of their significance for policy specification and evaluation;
- three different and successively more robust semantics for policy evaluation and a characterization of the failures with which they can cope;
- a description of how our semantics can be implemented and a discussion of optimizations that can significantly simplify the evaluation of policies.

To reiterate, our framework presented here combines the rigor of the work on policy algebras (which tends to use binary operators to compose policies and bottom-up semantics but does not consider evaluation failures or implementation) with the practicality of the work on XACML and related languages (which tends to use decision-combining algorithms and top-down policy evaluation but lacks the rigor of the work on policy algebras). In other words, our contributions synthesize and extend existing approaches to the specification and evaluation of authorization languages whilst also dealing with evaluation failures.

In the next section, we define our policy language and discuss how binary operators can be used to implement decision-combining functions. In Sect. 3, we define our policy semantics. In Sect. 4, we explain how our formal semantics can be realized in practice, describe our techniques for optimizing policy evaluation, and discuss the implications and potential applications of these techniques. We conclude with a summary of the paper and some discussion of future work in Sect. 5.

2 A Simple Policy Language

The language we use is rather similar to (core) XACML: policies are built from other policies, and policies may or may not be applicable to requests. However, our language is much simpler syntactically, although no less expressive.

We assume that the decisions that may arise from policy evaluation are given by the set $D = \{\mathbf{a}, \mathbf{d}, \perp\}$, denoting “allow”, “deny” and “not applicable”, respectively. We assume that two decisions may be combined using any one of the possible binary operators \oplus of the form $\oplus : D \times D \rightarrow D$. We will write \oplus using infix notation: that is, we prefer $x \oplus y$ to $\oplus(x, y)$.

Policy Syntax. *Atomic policies* have the form (π, a, ϕ) or (π, d, ϕ) , where π is used to determine the applicability of the policy and ϕ is a *possible-decisions resolution-function* of type $2^D \rightarrow 2^D$. If p_1 and p_2 are policies, then $(\pi, p_1, p_2, \oplus, \phi)$ is a policy, where $\oplus : D \times D \rightarrow D$ is a *sub-decision combining-operator*. Henceforth, we will usually refer to ϕ as a resolution function and \oplus as a decision operator. We describe resolution functions and decision operators in more detail later in this section.

Policy Applicability. Informally, when evaluating a request q with respect to some policy p , we first determine whether p is applicable to q . The role of π in our policy language is similar to that of **<Target>** and **<Condition>** elements in XACML rules and policies [12], or of access predicates in [6]. We refer to π as the *applicability predicate*.¹

Hence, to build an access control mechanism, we need a language for defining the applicability predicate and a method for evaluating whether a request satisfies the predicate. In XACML, for example, the syntax for defining **<Target>** and **<Condition>** elements forms part of the core language and the evaluation method forms part of the implementation of the policy decision point (PDP). While these issues are certainly important, the concern of this paper is how to evaluate policies under the assumption that such tools are available.

Decision Operators. We assume that any policy either has no “child” policies (as in policies of the form (π, x, ϕ) , where $x \in \{a, d\}$), or exactly two child policies. Equivalently, we assume that all decision operators are binary operators. There are two main reasons for this choice. First, the three most common methods used to combine a set of decisions – allow-overrides, deny-overrides and first-applicable [12] – can all be realized using binary operators. Consider, for example, the family of allow-overrides functions $AO_n : D^n \rightarrow D$, $n \geq 2$, where

$$AO_n(x_1, \dots, x_n) = \begin{cases} a & \text{if } x_i = a \text{ for some } i, \\ \perp & \text{if } x_i = \perp \text{ for all } i, \\ d & \text{otherwise.} \end{cases}$$

Then it is easy to see that for any $n \geq 2$, $AO_n(x_1, \dots, x_n) = (\dots(x_1 \vee x_2) \vee \dots x_n)$, where \vee is defined below in Fig. 1. Similar results hold for deny-overrides and first-applicable. Second, it is very simple to characterize binary operators and this provides many opportunities for optimizing policy evaluation (as we shall see in Sect. 4). We classify operators using the following definitions.

Definition 1. Let $\oplus : D \times D \rightarrow D$ be a decision operator.

- If $x \oplus x = x$ for all $x \in D$, then we say \oplus is idempotent.
- If $x \oplus \perp = x = \perp \oplus x$ for all $x \in D$, then we say \oplus is a \cup -operator.
- If $x \oplus \perp = \perp = \perp \oplus x$ for all $x \in D$, then we say \oplus is an \cap -operator.
- We say \oplus is well-behaved if it is either a \cup - or an \cap -operator.

¹ If one were to use XACML syntax in order to define our applicability predicate, then our atomic policies would be analogous to XACML rules, and policies of the form $(\pi, p_1, p_2, \oplus, \phi)$ would be analogous to XACML policies and policy sets.

Idempotent operators are a natural choice, as idempotency is expected when composing access-control decisions. In total, there are $3^6 = 729$ possible idempotent, binary operators. However, far fewer operators are of practical interest. In Sect. 4, we consider how restricting attention to idempotent, well-behaved operators can considerably simplify policy evaluation.

An idempotent, well-behaved decision operator is uniquely defined by the choices of $x \oplus \perp$, $\mathbf{a} \oplus \mathbf{d}$ and $\mathbf{d} \oplus \mathbf{a}$. If we assume that \oplus is commutative, then there are only six possible choices for \oplus (and only four if we assume that $\mathbf{a} \oplus \mathbf{d} \in \{\mathbf{a}, \mathbf{d}\}$). The decision tables for two of the four commutative, idempotent, well-behaved binary operators such that $\mathbf{a} \oplus \mathbf{d} \in \{\mathbf{a}, \mathbf{d}\}$ are shown Fig. 1 as \vee and \wedge . As we noted above, the operator \vee has the same effect as the allow-overrides policy-combining algorithm in XACML, while \wedge has the same effect as the deny-overrides algorithm.² If \oplus is not commutative, then there are 18 possible choices for \oplus (and eight choices if $\mathbf{a} \oplus \mathbf{d} \in \{\mathbf{a}, \mathbf{d}\}$ and $\mathbf{d} \oplus \mathbf{a} \in \{\mathbf{a}, \mathbf{d}\}$).

The other binary operators shown in Fig. 1 are: \vee' and \wedge' , the \cap -operator analogues of \vee and \wedge ; and the non-commutative, “first-applicable”, \triangleright -operator \triangleright , which returns the first conclusive decision (\mathbf{a} or \mathbf{d}).

\wedge	\mathbf{a}	\mathbf{d}	\perp	\vee	\mathbf{a}	\mathbf{d}	\perp	\wedge'	\mathbf{a}	\mathbf{d}	\perp	\vee'	\mathbf{a}	\mathbf{d}	\perp	\triangleright	\mathbf{a}	\mathbf{d}	\perp
\mathbf{a}	\mathbf{a}	\mathbf{d}	\mathbf{a}	\mathbf{a}	\mathbf{a}	\mathbf{a}	\mathbf{a}	\mathbf{a}	\mathbf{a}	\mathbf{d}	\perp	\mathbf{a}	\mathbf{a}	\mathbf{a}	\perp	\mathbf{a}	\mathbf{a}	\mathbf{a}	\mathbf{a}
\mathbf{d}	\mathbf{d}	\mathbf{d}	\mathbf{d}	\mathbf{d}	\mathbf{a}	\mathbf{d}	\mathbf{d}	\mathbf{d}	\mathbf{d}	\mathbf{d}	\perp	\mathbf{d}	\mathbf{a}	\mathbf{d}	\perp	\mathbf{d}	\mathbf{d}	\mathbf{d}	\mathbf{d}
\perp	\mathbf{a}	\mathbf{d}	\perp	\perp	\mathbf{a}	\mathbf{d}	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\mathbf{a}	\mathbf{d}	\perp

Fig. 1. Decision tables for some binary operators

Resolution Functions. In Sect. 3 we define three different semantics for policies, which specify how a policy should be evaluated. Two of these evaluation methods handle exceptional events by considering different possible outcomes, which leads to the possibility of policy evaluation returning a set of possible decisions, rather than a single decision, as is more usual in access-control mechanisms. The express purpose of the resolution function ϕ is to modify the set of possible outcomes.

In many cases, ϕ will be the identity function ι , where $\iota(X) = X$ for all $X \subseteq D$. We will simply omit ϕ if $\phi = \iota$ (as will be the case in most subsequent examples). However, we would expect that the top-level policy would define ϕ so that for all $X \subseteq D$, $\phi(X) = \{x\}$ for some $x \in \{\mathbf{a}, \mathbf{d}, \perp\}$. In other words, evaluation of the top-level policy always results in a single response.

As for decision operators, very few resolution functions will be of practical relevance. For a policy $(\pi, p_1, p_2, \oplus, \phi)$, we might expect ϕ to be “semantically related” to \oplus : if \oplus is \vee (allow-overrides), for example, we might define $\phi(X) = \{\mathbf{a}\}$ if $\mathbf{a} \in X$ and $\phi(X) = X$ otherwise. However, it must be stressed that X represents a set of possible outcomes and (even when \oplus equals \vee) it is probably prudent to be conservative and define

² Interpreting \mathbf{a} as 1 and \mathbf{d} as 0, $x \wedge y$ is analogous to logical AND (when $x, y \in \{\mathbf{a}, \mathbf{d}\}$) and \vee is analogous to logical OR.

$$\phi(X) = \begin{cases} \{\mathbf{d}\} & \text{if } \mathbf{d} \in X, \\ \{\perp\} & \text{if } \perp \in X, \\ \{\mathbf{a}\} & \text{otherwise.} \end{cases}$$

Policy Trees. A policy tree is a convenient way of visualizing a policy and can be constructed recursively from a policy. A policy of the form $(\pi, p_1, p_2, \oplus, \phi)$ has a tree with root node (π, \oplus, ϕ) and two child sub-trees p_1 and p_2 . A policy of the form (π, x, ϕ) , where $x \in \{\mathbf{a}, \mathbf{d}\}$, is a leaf node (π, x, ϕ) . Consider, for example, the policy

$$(\pi_5, (\pi_3, (\pi_1, \mathbf{a}), (\pi_2, \mathbf{d}), \wedge), (\pi_4, \mathbf{a}), \vee),$$

whose policy tree is shown in Fig. 5(a). Henceforth, we will tend to use this tree representation of policies.

3 Policy Semantics

Policies are used to evaluate whether an access request is authorized. When a policy is evaluated, one first checks whether the policy is applicable to the request, which will be determined by the request and π . Under normal circumstances, the evaluation of the applicability of a policy returns either **true** or **false**.

However, if we wish to account for exceptional circumstances – perhaps it is not possible to retrieve certain information due to communication, software or hardware failures, or perhaps the request is malformed – then it may not be possible to evaluate some component of a policy. As we noted above, it is natural to then consider the possible outcomes that *could* have arisen from evaluating the policy. We use a resolution function ϕ to combine these possible outcomes.

The evaluation of policy $p = (\pi, p_1, p_2, \oplus, \phi)$ at request q is determined by

- the applicability of the policy to q (π);
- the evaluation of the sub-policies of p (p_1 and p_2) at q ;
- the method by which evaluation results of the sub-policies are combined (\oplus);
- the combination of different possible evaluations of p (using ϕ).

We wish to account for indeterminacy that might arise in the evaluation of policy applicability and in the retrieval of policies. To this end, we consider the following possibilities.

1. Normal evaluation, where all policy components can be retrieved and the applicability of all sub-policies can be determined.
2. Indeterminate applicability of sub-policies, where all sub-policies can be retrieved, but the applicability of a sub-policy may be impossible to determine.
3. Indeterminate applicability or non-retrievability of some sub-policies.

The second and third items account for differing types of exceptional behavior that might occur during policy evaluation. These differences are reflected in the evaluation of the parent policy.

In the presence of indeterminacy, we adopt a conservative evaluation strategy and consider all possible outcomes. If the applicability of a sub-policy cannot be determined, then we consider two possibilities when evaluating the parent policy: that the sub-policy was applicable and that the sub-policy was not applicable. If a sub-policy cannot be retrieved, then we consider three possibilities: that the sub-policy was applicable and returned \mathbf{a} , that the sub-policy was applicable and returned \mathbf{d} , and that the sub-policy was not applicable. We use ϕ to modify the set of possible outcomes: ϕ could, for example, reduce a set of two or more possible outcomes to a single outcome.

Below, we treat the three assumptions on evaluation failures separately, but we prove that in each case the semantics are extended in such a way that the semantics for simpler assumptions are preserved (see Proposition 1, for example).

Our technical development assumes the existence of an evaluation function e that determines whether policy p is applicable to request q . We define three methods of evaluation for policies, corresponding to the failure assumptions identified above. We refer to them as Type 1, 2 and 3 semantics, respectively.

For a policy p , we write $[[p]]_i(q)$ to mean the result of evaluating p at point q using Type i semantics. We write $[[p]]_i = [[p']]_i$ if and only if for all requests q , we have $e(p, q) = e(p', q)$ and $[[p]]_i(q) = [[p']]_i(q)$.

Type 1 Semantics. In this case, we assume that for all policies p and all requests q , either $e(p, q) = \mathbf{true}$ or $e(p, q) = \mathbf{false}$. Henceforth, we write \mathbf{t} and \mathbf{f} for true and false, respectively.

Our Type 1 semantics is depicted in Fig. 2. An alternative form of the same semantics, explained in the next section, is given in Fig. 3(a).

$$\begin{aligned}
 [[(\pi, p_1, p_2, \oplus), \phi]]_1(q) &= \begin{cases} [[p_1]]_1(q) \oplus [[p_2]]_1(q) & \text{if } e(p, q) = \mathbf{t}, \\ \perp & \text{otherwise;} \end{cases} \\
 [[(\pi, x, \phi)]_1(q) &= \begin{cases} x & \text{if } e(p, q) = \mathbf{t} \text{ and } x \in \{\mathbf{a}, \mathbf{d}\}, \\ \perp & \text{otherwise.} \end{cases}
 \end{aligned}$$

Fig. 2. Type 1 semantics

Note that for all policies p and all requests q , we have that $[[p]]_1(q) \in \{\mathbf{a}, \mathbf{d}, \perp\}$. This can be proved by a simple induction on the depth of the policy tree.

Type 2 Semantics. For ease of exposition (and to aid implementation), we introduce a “dummy” policy that is applicable to every request: if p is a policy, then \hat{p} is a policy that is identical to p except that π is replaced with the reserved word \mathbf{t} . Hence:

- if $p = (\pi, x, \phi)$, where $x \in \{\mathbf{a}, \mathbf{d}\}$, then $\hat{p} \stackrel{\text{def}}{=} (\mathbf{t}, x, \phi)$;
- if $p = (\pi, p_1, p_2, \oplus, \phi)$, then $\hat{p} \stackrel{\text{def}}{=} (\mathbf{t}, p_1, p_2, \oplus, \phi)$.

By definition, $e(\hat{p}, q) = \mathbf{t}$ for all policies p and all requests q .

Type 2 policy semantics are presented in Fig. 3(b). Note that we can also define Type 1 semantics for p using \hat{p} as shown in Fig. 3(a). The uniform presentation of the semantics in Fig. 3 illustrates how Type 2 semantics are related to Type 1, and how Type 3 are related to Type 2.

In defining Type 2 semantics, we do not assume that $e(p, q)$ takes a unique value in $\{t, f\}$. Hence, we must provide a method of evaluating p if $e(p, q) \neq t$ and $e(p, q) \neq f$. In this case, we consider two possible evaluations of the policy tree: one when the policy is applicable and one when the policy is not (when the response is \perp). In Fig. 3(b), we see that $[[p]]_2(q)$ introduces a third option to explicitly handle this possibility.

Now, of course, the evaluation of a policy may return a set of possible responses, rather than a single response. Hence $[[p]]_2$ returns a set of responses, to which the resolution function ϕ is applied.

Note that for Type 2 and Type 3 semantics, we have assumed that for all ϕ and all $x \in D$, $\phi(\{x\}) = \{x\}$, the intuition being that if the evaluation of a

$$[[p]]_1(q) = \begin{cases} [[\hat{p}]]_1(q) & \text{if } e(p, q) = t, \\ \perp & \text{otherwise;} \end{cases}$$

$$[[\hat{p}]]_1(q) = \begin{cases} [[p_1]]_1(q) \oplus [[p_2]]_1(q) & \text{if } p = (\pi, p_1, p_2, \oplus, \phi), \\ x & \text{if } p = (\pi, x, \phi), x \in \{a, d\}. \end{cases}$$

(a) Type 1

$$[[p]]_2(q) = \begin{cases} \{[[\hat{p}]]_2(q)\} & \text{if } e(p, q) = t, \\ \{\perp\} & \text{if } e(p, q) = f, \\ \phi(\{\perp\} \cup \{[[\hat{p}]]_2(q)\}) & \text{otherwise;} \end{cases}$$

$$[[\hat{p}]]_2(q) = \begin{cases} \phi(\{x \oplus y : x \in [[p_1]]_2(q), y \in [[p_2]]_2(q)\}) & \text{if } p = (\pi, p_1, p_2, \oplus, \phi), \\ \{x\} & \text{if } p = (\pi, x, \phi), x \in \{a, d\}. \end{cases}$$

(b) Type 2

$$[[p]]_3(q) = \begin{cases} \{a, d, \perp\} & \text{if } p \text{ cannot be retrieved,} \\ \{[[\hat{p}]]_3(q)\} & \text{if } e(p, q) = t, \\ \{\perp\} & \text{if } e(p, q) = f, \\ \phi(\{\perp\} \cup \{[[\hat{p}]]_3(q)\}) & \text{otherwise;} \end{cases}$$

$$[[\hat{p}]]_3(q) = \begin{cases} \phi(\{x \oplus y : x \in [[p_1]]_3(q), y \in [[p_2]]_3(q)\}) & \text{if } p = (\pi, p_1, p_2, \oplus, \phi), \\ \{x\} & \text{if } p = (\pi, x, \phi), x \in \{a, d\}. \end{cases}$$

(c) Type 3

Fig. 3. Three types of policy semantics corresponding to our three failure assumptions

policy returns a single outcome, then ϕ should return that outcome unmodified. Then Type 2 semantics are an extension of Type 1 semantics, and preserve Type 1 semantics in the following sense.

Proposition 1. *Let p be any policy comprising sub-policies p_1, \dots, p_k and let $e(p, q) \in \{\mathbf{t}, \mathbf{f}\}$ and $e(p_i, q) \in \{\mathbf{t}, \mathbf{f}\}$ for all i . Then $[[p]]_2(q) = \{[[p]]_1(q)\}$.*

Proof. By induction on the depth of the policy tree. Consider the base case when the tree has depth 1, whence $p = (\pi, x, \phi)$ for some $x \in \{\mathbf{a}, \mathbf{d}\}$. Then $[[p]]_1(q) = x$ if $e(p, q) = \mathbf{t}$ and $[[p]]_1(q) = \perp$ otherwise. Now, by assumption, $e(p, q)$ is known. Therefore, $[[p]]_2(q) = [[\hat{p}]]_2(q) = \{x\}$ if $e(p, q) = \mathbf{t}$, and $[[p]]_2(q) = \{\perp\}$ if $e(p, q) = \mathbf{f}$. Hence the result holds if the tree has depth 1.

Now suppose that the result holds for all trees of depth less than or equal to n and suppose the tree for $p = (\pi, p_1, p_2, \oplus, \phi)$ has depth $n + 1$. Then, by assumption, $e(p, q), e(p_1, q), e(p_2, q) \in \{\mathbf{t}, \mathbf{f}\}$. Moreover,

$$[[p]]_1(q) = \begin{cases} [[p_1]]_1(q) \oplus [[p_2]]_1(q) & \text{if } e(p, q) = \mathbf{t}, \\ \perp & \text{otherwise.} \end{cases}$$

Now let us consider $[[p]]_2(q)$. If $e(p, q) = \mathbf{t}$, then $[[p]]_2(q)$ equals $[[\hat{p}]]_2(q) = \phi(\{x \oplus y : x \in [[p_1]]_2(q), y \in [[p_2]]_2(q)\})$. By the inductive hypothesis $[[p_1]]_2(q) = \{[[p_1]]_1(q)\}$ and $[[p_2]]_2(q) = \{[[p_2]]_1(q)\}$. Hence,

$$\begin{aligned} [[p]]_2(q) &= \phi(\{x \oplus y : x \in [[p_1]]_2(q), y \in [[p_2]]_2(q)\}) \\ &= \phi(\{[[p_1]]_2(q) \oplus [[p_2]]_2(q)\}) \\ &= \{[[p_1]]_2(q) \oplus [[p_2]]_2(q)\} \\ &= \{[[p]]_1(q)\} \end{aligned}$$

Alternatively, if $e(p, q) = \mathbf{f}$ then $[[p]]_2(q) = \{\perp\} = \{[[p]]_1(q)\}$. Hence, the result follows by induction. \square

In other words, if the applicability of all component policies can be determined, then the evaluation of p with respect to Type 2 semantics returns a unique response which is that obtained by using Type 1 semantics.

Type 3 Semantics. In this case, we do not assume that we can always retrieve a sub-policy, so it may be the case that we have no sub-policy to evaluate. We can still attempt to evaluate the root policy by considering all possible responses that could be returned by that sub-policy. This is reflected in the Type 3 semantics illustrated in Fig. 3(c), where the evaluation of a policy p simply returns the set $\{\mathbf{a}, \mathbf{d}, \perp\}$ if p cannot be retrieved.

It is very easy to see that if all policies can be retrieved, the evaluation of a policy will be the same whether Type 2 or Type 3 semantics are used. More formally, we have the following result.

Proposition 2. *Let p be any policy comprising sub-policies p_1, \dots, p_k and suppose that it has been possible to retrieve all p, p_1, \dots, p_k . Then $[[p]]_3(q) = [[p]]_2(q)$.*

Proof. Since we assume that all policies are retrievable, the result follows directly from the definitions of Type 2 and Type 3 semantics. \square

4 Policy Evaluation

In this section, we consider several evaluation strategies that realize the semantics defined in the previous section. We first present a simple algorithm that can be used to implement Type 1 and Type 2 semantics.

Naïve Algorithm. We can implement Type 2 semantics directly using an algorithm of the form shown in Fig. 4. We assume that a tree representation of the entire policy can always be constructed. (In other words, Type 2 semantics are sufficient to derive a decision.) The function *evaluateTree*(\cdot) takes a pointer to the root of the policy tree and a request and returns the set of possible authorization decisions for that request with respect to the policy tree. The function *evaluateApplicability*(\cdot) determines whether a policy is applicable to a request (in other words, it is a realization of the function e used in the previous section), taking a policy and a request as input and returning **t**, **f** or neither.

We assume each node in a policy tree has form $(\pi, lptr, rptr, effect, \phi)$, where *effect* may be a decision **a** or **d** or it may be a decision operator \oplus . Hence, we model a policy of the form (π, a, ϕ) , for example, as $(\pi, \text{null}, \text{null}, a, \phi)$, and $(\pi, p_1, p_2, \oplus, \phi)$ as $(\pi, lptr, rptr, \oplus, \phi)$. In an attempt to keep the pseudo-code easy to read, we refer directly to the components of a node, so we write π in preference to $p.\pi$ or $p \rightarrow \pi$, for example.

```
[Inputs: pointer to policy tree  $p$ ; request  $q$ ]
[Outputs: set of decisions]
evaluateTree( $p, q$ )
  if ( $\pi == t$ ) then
    if ( $lptr == \text{null}$ ) and ( $rptr == \text{null}$ ) then
      return {effect}
    else
       $X = \text{evaluateTree}(lptr, q)$ 
       $Y = \text{evaluateTree}(rptr, q)$ 
       $result = \emptyset$ 
      for all  $x \in X$ 
        for all  $y \in Y$ 
           $result = result \cup \{x \oplus y\}$ 
      return  $\phi(result)$ 
  else
    if (evaluateApplicability( $\pi, q$ ) == t) then
       $\pi = t$ 
      evaluateTree( $p, q$ )
    else-if (evaluateApplicability( $\pi, q$ ) == f) then
      return { $\perp$ }
    else
       $\pi = t$ 
      return  $\phi(\{\perp\} \cup \text{evaluateTree}(p, q))$ 
```

Fig. 4. A possible implementation of Type 2 semantics

Let us now consider the evaluation of the policy illustrated in Fig. 5(a), where the operators \wedge and \vee are as defined in Fig. 1. We will write p_i to refer to the

policy (sub-)tree with root (π_i, \oplus_i) . (The indices assigned to the node identifiers correspond to a post-order traversal [1] of the tree.)

An *evaluation tree* (for request q) is obtained by labeling each node of the policy tree with its applicability (with respect to q) and its response.

Let us now consider the effect of evaluating a request for which we cannot decide whether certain policies are applicable or not. First, if $e(p_2, q) \neq t$ and $e(p_2, q) \neq f$, with all other policies being applicable, then p_2 returns the set of (possible) responses $\{d, \perp\}$. Then $a \wedge \perp = a$ and $a \wedge d = d$, which means that p_3 returns $\{d, \perp\}$. Hence, p_5 returns $\{a\}$. This example is illustrated in Fig. 5(b).

Finally, suppose that $e(p_3, q) \neq t$ and $e(p_3, q) \neq f$, the applicability of other policies being shown in Fig. 5(c). Then we evaluate p_1 and p_2 and combine the results using \wedge to obtain d . To this we add the response \perp to account for the possibility that p_3 may not have been applicable. Hence, the set of possible responses for p_3 is $\{d, \perp\}$. If $e(p_4, q) = f$ (as shown in Fig. 5(c)), then p_5 returns $\{d, \perp\}$. If $e(p_4, q) = t$ (not illustrated), then p_5 returns $\{a\}$.

Well-Behaved Operators. A policy $p = (\pi, p_1, p_2, \oplus, \phi)$ may not return a conclusive decision (a or d) even if p is applicable, because neither p_1 nor p_2 may be applicable. There are two standard interpretations of what might be termed the “effective applicability” of a policy $p = (\pi, p_1, p_2, \oplus, \phi)$.

1. One is to regard p as being effectively applicable to every request for which p is applicable and at least one of p_1 or p_2 is applicable, as in XACML.
2. The other is to regard p as being applicable to a request only if p, p_1 and p_2 are all applicable.

In the first case, $[[p]](q)$ is defined if $e(p, q) = t$ and either $e(p_1, q) = t$ or $e(p_2, q) = t$. In the second case, $[[p]](q)$ is defined if $e(p, q) = e(p_1, q) = e(p_2, q) = t$. This interpretation appears in several papers on “policy algebras” (see [5], for example). In both cases, if the policy p is applicable to request q , then the decision returned by the policy is $[[p_1]] \oplus [[p_2]]$.

In fact, each of these interpretations can be realized provided \oplus is chosen appropriately. If we want the first interpretation, then we ensure that for all $x \in \{\perp, a, d\}$, $x \oplus \perp = \perp \oplus x = x$. That is, \oplus is a \cup -operator. A \cup -operator effectively ignores all \perp values, ensuring that p will return a value whenever at least one of p_1 or p_2 is applicable. All the standard policy-combining algorithms

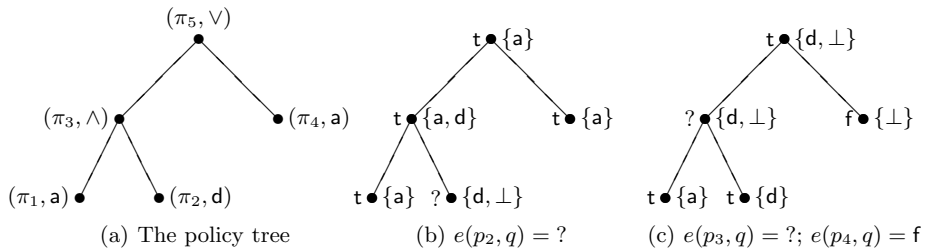


Fig. 5. Policy and evaluation trees for policy $(\pi_5, (\pi_3, (\pi_1, a), (\pi_2, d), \wedge), (\pi_4, a), \vee)$

in XACML [12] have this behavior. If \oplus is a \cup -operator, then we say any p of the form $(\cdot, \cdot, \cdot, \oplus, \cdot)$ is a \cup -policy.

If we want the second interpretation, then we ensure that for all x , $x \oplus \perp = \perp \oplus x = \perp$. That is, \oplus is a \cap -operator, which has the effect of returning \perp whenever at least one of p_1 or p_2 is not applicable. If \oplus has this property, then we say any p of the form $(\cdot, \cdot, \cdot, \oplus, \cdot)$ is an \cap -policy.

Optimizing Policy Evaluation. We now show how a policy evaluation tree can be pruned, *without* changing its meaning, when the decision operators are known to be well-behaved (and all sub-policies can be retrieved). Suppose that p uses the identity resolution function, and so $p = (\pi, p_1, p_2, \oplus)$. Table 1 illustrates the evaluation of p given the applicability of p , p_1 and p_2 , and the nature of \oplus . Abusing notation slightly, we write $[[p_1]] \oplus [[p_2]]$ for $\{x_1 \oplus x_2 : x_1 \in [[p_1]], x_2 \in [[p_2]]\}$. We write “ \perp ” to denote that the applicability of a sub-policy is irrelevant to the evaluation of p .

Table 1. Optimized evaluation of $p = (\pi, p_1, p_2, \oplus)$ when \oplus is well-behaved

Applicability			[[p]]	
p	p ₁	p ₂	\cup -operator	\cap -operator
t	t	t	$[[p_1]] \oplus [[p_2]]$	$[[p_1]] \oplus [[p_2]]$
t	t	f	$[[p_1]]$	$\{\perp\}$
t	f	t	$[[p_2]]$	$\{\perp\}$
t	f	f	$\{\perp\}$	$\{\perp\}$
t	t	?	$([[p_1]] \oplus [[p_2]]) \cup [[p_1]]$	$\{\perp\} \cup ([[p_1]] \oplus [[p_2]])$
t	?	t	$([[p_1]] \oplus [[p_1]]) \cup [[p_2]]$	$\{\perp\} \cup ([[p_1]] \oplus [[p_2]])$
t	?	?	$([[p_1]] \oplus [[p_2]]) \cup [[p_1]] \cup [[p_2]] \cup \{\perp\}$	$\{\perp\} \cup ([[p_1]] \oplus [[p_2]])$
t	f	?	$\{\perp\} \cup [[p_2]]$	$\{\perp\}$
t	?	f	$\{\perp\} \cup [[p_1]]$	$\{\perp\}$
?	–	–	$\{\perp\} \cup ([[p_1]] \oplus [[p_2]])$	$\{\perp\} \cup ([[p_1]] \oplus [[p_2]])$
f	–	–	$\{\perp\}$	$\{\perp\}$

Given a request, we now assume the applicability of every sub-policy is first evaluated. We can then apply re-writing rules to the policy-evaluation tree on the basis of the applicability of each sub-policy and the semantics shown in Table 1. To illustrate this point, we define and prove the correctness of one such re-write rule in Proposition 3. Similar results exist for the other re-writing rules, but are omitted due to space constraints.

Proposition 3. *Let $p = (\pi, p_1, p_2, \oplus)$, where \oplus is well-behaved, and let q be a request such that $e(p, q) = e(p_1, q) = t$ and $e(p_2, q) = f$. Then*

$$[[p]]_2(q) = \begin{cases} \{\perp\} & \text{if } \oplus \text{ is an } \cap\text{-operator,} \\ \{[[p_1]]_2(q)\} & \text{if } \oplus \text{ is a } \cup\text{-operator.} \end{cases}$$

Proof. By definition, $[[p]]_2(q) = \{x \oplus \perp : x \in [[p_1]]_2(q)\}$. If \oplus is an \cap -operator, then $x \oplus \perp = \perp$ for all $x \in \{a, d, \perp\}$; hence $[[p]]_2(q) = \{\perp\}$. If \oplus is a \cup -operator, then $x \oplus \perp = x$ for all $x \in \{a, d, \perp\}$; hence $[[p]]_2(q) = [[p_1]]_2(q)$. \square

Using these re-writing rules we can simplify the evaluation of a policy considerably. Also, these re-writing rules can be implemented easily using a recursive post-order tree traversal algorithm. To illustrate, consider the policy represented by the tree in Fig. 6(a), where each node has been assigned an identifier of the form p_i to facilitate explanation. (The indices assigned to the node identifiers correspond to a post-order traversal of the tree.) The applicability of each sub-policy for some request q is indicated to the left of each node.

If all decision operators are \cap -operators, then we can simplify this evaluation tree for q to a single node comprising a non-applicable policy. This is because p_2 is not applicable (since one of its children is not applicable), which in turn means that p_4 and p_5 are not applicable.

The simplified policy-evaluation tree, when all operators are \cup -operators, is shown in Fig. 6(b). The sub-tree rooted at policy p_9 reduces to an evaluation of p_7 and the evaluation of p_7 reduces to an evaluation of p_5 . Suppose that the (relevant) leaf policies are $p_0 = (\pi_0, a)$, $p_5 = (\pi_5, a)$ and $p_9 = (\pi_9, d)$. Let \oplus_i denote the decision operator for the policy at node p_i . Now $x \oplus_i \perp = \perp \oplus x_i = x$ for all operators \oplus_i in the policy tree (since, by assumption, \oplus_i is a \cup -operator) and assuming that \oplus_i is idempotent, we have $[[p_0]]_2(q) = \{a\}$, $[[p_5]]_2(q) = \{\perp, a\}$, $[[p_9]]_2(q) = \{\perp, d\}$ and $[[p_{11}]]_2(q) = \{\perp, a, d, a \oplus_{11} d\}$ from which we obtain

$$\begin{aligned} [[p_{12}]]_2(q) &= \{a, a \oplus_{12} a, a \oplus_{12} d, a \oplus_{12} (a \oplus_{11} d)\} \\ &= \{a, a \oplus_{12} d, a \oplus_{12} (a \oplus_{11} d)\}. \end{aligned}$$

By specifying \oplus_{11} and \oplus_{12} we can compute $[[p_{12}]]_2(q)$. For example:

$$[[p_{12}]]_2(q) = \begin{cases} \{a\} & \text{if } \oplus_{12} = \vee, \\ \{a, d\} & \text{if } \oplus_{12} = \wedge. \end{cases}$$

Thus one may derive a conclusive decision for p (namely $\{a\}$ in the case that $\oplus_{12} = \wedge$) even if the applicability of some sub-policies cannot be determined.

In practice, all decision operators are likely to be well-behaved. Indeed, all the standard policy-combining algorithms in XACML (the equivalent of our decision operators) are \cup -operators. Note that we neither require that all operators in the policy tree are \cup -operators nor that they are all \cap -operators in order to use our re-write rules, simply that they are all well-behaved.

Under the assumption that all decision operators are well-behaved, it is always possible to perform tree re-writing, thereby simplifying policy evaluation. A flag in each policy could indicate if it is a \cup - or an \cap -policy, thereby indicating how \oplus should treat the \perp value. This provides sufficient information to re-write the evaluation tree and means that we only need to define $x \oplus y$ for $x, y \in \{a, d\}$.

Indeed, recalling the discussion in Sect. 2, we can completely specify any idempotent, well-behaved operator with three pieces of information: a flag indicating

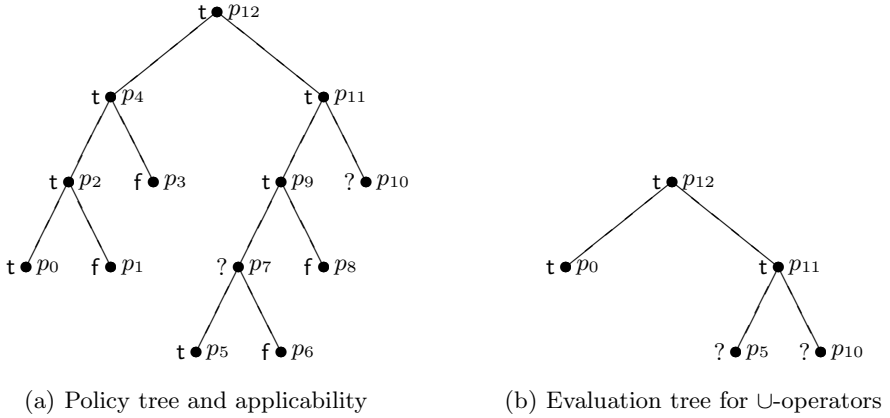


Fig. 6. Policy evaluation by tree re-writing

whether it is a \cup - or \cap -operator, the value of $\mathbf{a} \oplus \mathbf{d}$ and the value of $\mathbf{d} \oplus \mathbf{a}$. This information can be included in the policy definition (rather than providing a pointer to a decision table) and used directly by the *evaluateTree*(,) function.

Partial Evaluation Trees. Type 3 semantics are only relevant when we are unable to build a complete policy tree. Such a situation could arise when policies are not self-contained, in the sense that they may reference sub-policies stored in remote repositories.

Under these operating assumptions, we build an evaluation tree at request evaluation time. This evaluation tree may not be isomorphic to the policy tree, since some policy (that would give rise to sub-trees) may not be retrievable at evaluation time. In constructing the evaluation tree, we label the nodes with an applicability value (if possible) or with the decision set $\{\mathbf{a}, \mathbf{d}, \perp\}$ otherwise.

To illustrate, let us evaluate the policy depicted in Fig. 6 for request q under assumption that we cannot retrieve policy p_9 . Then we construct the (partial) evaluation tree shown in Fig. 7(a). If all operators are \cup -operators, we can re-write this evaluation tree to obtain the tree shown in Fig. 7(b).

Then we have

$$\begin{aligned}
 [[p_{11}]]_3(q) &= \{\mathbf{a} \oplus_{11} \mathbf{d}, \mathbf{d} \oplus_{11} \mathbf{a}, \perp \oplus_{11} \mathbf{d}\} = \{\mathbf{a} \oplus_{11} \mathbf{d}, \mathbf{d}\}; \\
 [[p_{12}]]_3(q) &= \{\mathbf{a} \oplus_{12} (\mathbf{a} \oplus_{11} \mathbf{d}), \mathbf{a} \oplus_{12} \mathbf{d}\}.
 \end{aligned}$$

Assuming that \oplus_{11} and \oplus_{12} belong to $\{\vee, \wedge\}$, it can be shown that $[[p_{12}]]_3(q)$ is a conclusive decision, except when $\oplus_{12} = \wedge$ and $\oplus_{11} = \vee$.

Applications. The fact that we can obtain conclusive results from a partial evaluation of a policy opens up interesting possibilities. We sketch two of them here briefly.

A first application is an access-control architecture in which there are two PDPs: one is co-located with the policy-evaluation point (PEP) and is used to

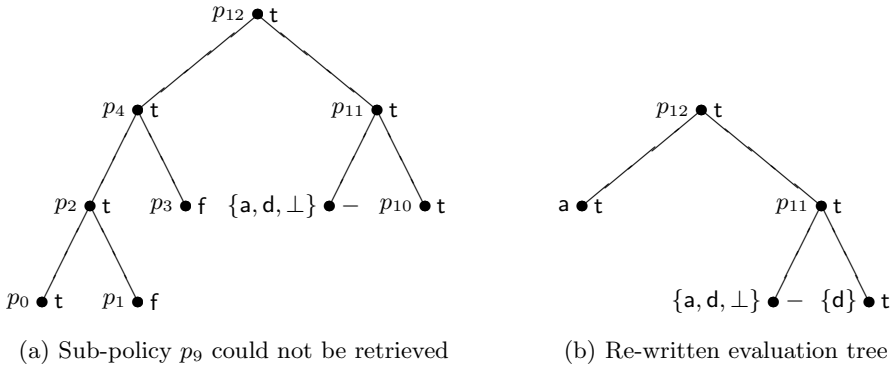


Fig. 7. Partial request-time evaluation tree and its rewrite

make rapid decisions where possible, while the other may be remote.³ The “local” PDP we envisage is provided with a partial representation of an authorization policy that returns a (comparatively) quick response to the PEP. If the response is not conclusive, then the PEP forwards the request to the other PDP which evaluates the full policy tree and returns a decision.

Consider, for example, the policy tree in Fig. 6(a) and suppose that we expect that p_{10} will be applicable to a large percentage of requests. Then we might choose to provide the local PDP with the tree depicted in Fig. 8 (deliberately preventing the local PEP from evaluating the whole policy tree by omitting the relatively complex policies p_4 and p_9).

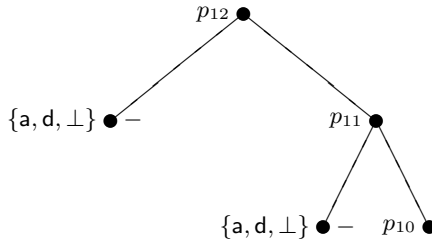


Fig. 8. The policy evaluated by the local PDP

Now suppose that p_{10} , p_{11} and p_{12} are applicable to q . Then

$$[[p_{12}]]_3(q) = \begin{cases} \{a\} & \text{if } [[p_{10}]]_3(q) = \{a\} \text{ and } \oplus_{11} = \oplus_{12} = \vee, \\ \{d\} & \text{if } [[p_{10}]]_3(q) = \{d\} \text{ and } \oplus_{11} = \oplus_{12} = \wedge, \\ \{a, d\} & \text{otherwise.} \end{cases}$$

³ This architecture is structurally similar to those used for authorization recycling that cache previous authorization decisions at the PEP to improve response times [7].

Clearly, it would be worth providing the local PDP with the reduced policy in Fig. 8 if $\oplus_{11} = \oplus_{12}$ (and it is known that p_{10} is applicable to many requests).

A second application of our authorization framework is to define a PDP that can process multiple requests in a single pass through the evaluation tree. There are many practical instances where it is necessary to decide several different access requests in order to determine whether an attempted subject-object interaction is authorized. Two obvious examples are:

- In Unix, a subject is authorized to access an object only if it is authorized to access every directory (multiple objects) in said object’s path name.
- In the stack-walk algorithm used in Java, where it is necessary to check that every subject on the call stack (multiple subjects) is authorized.

In this case, the PDP processes all requests at the same time, treating each of these as possible evaluations of the tree. We introduce the top-level resolution function ϕ_{\forall} , where $\phi_{\forall}(X) = \{\mathbf{a}\}$ if $X = \{\mathbf{a}\}$ and $\phi_{\forall}(X) = \{\mathbf{d}\}$ otherwise. In contrast, a role-based PDP can evaluate multiple requests, one for each role for which the requester is authorized, and use the ϕ_{\exists} resolution function to compute a final decision, where $\phi_{\exists}(X) = \{\mathbf{a}\}$ if $\mathbf{a} \in X$ and $\phi_{\exists}(X) = \{\mathbf{d}\}$ otherwise.

5 Concluding Remarks

We have presented a framework for tree-like authorization policies that are resilient to evaluation failures. This resiliency is achieved by defining three different semantics for those policies, representing three different sets of assumptions about the operational environment in which these failures may occur.

We have provided a succinct characterization of decision operators, which yields numerous opportunities for optimizing policy evaluation and policy representation. Our semantics improve on existing work in enabling policy evaluation to be completed even if it is not possible to recover one or more sub-policies. Our approach is conceptually similar to static analysis [11]. In particular, if our semantics return a conclusive decision, then our over-approximation of decisions is precise (Propositions 1 and 2). Our work also enables the design of efficient PDPs and novel access-control architectures, to be explored in future work.

There are many other ways in which our work could and should be extended. From a technical perspective, it is important to establish whether our language can accommodate a fourth decision value to represent conflicting decisions from sub-policies [6]. Equally important is to establish what set of binary operators would be sufficient to articulate any desired policy. There are clear parallels here with establishing a minimal set of logical connectives that is functionally complete [2]. From a practical perspective, it would be interesting to develop an XML schema for our policy language, perhaps re-using those parts of XACML that are used to specify `<Target>` and `<Condition>` elements, and to develop a PDP that implements our policy semantics.

Acknowledgements. The authors would like to thank the anonymous referees for their comments.

References

1. Aho, A., Hopcroft, J., Ullman, J.: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading (1975)
2. Aireli, O., Avron, A.: The value of the four values. *Artificial Intelligence* 102, 97–141 (1998)
3. Backes, M., Dürmuth, M., Steinwandt, R.: An algebra for composing enterprise privacy policies. In: *Proceedings of the 9th European Symposium on Research in Computer Security*, pp. 33–52 (2004)
4. Bertino, E., Castano, S., Ferrari, E.: Author- \mathcal{X} : A comprehensive system for securing XML documents. *IEEE Internet Computing* 5(3), 21–31 (2001)
5. Bonatti, P., Vimercati, S.D.C.D., Samarati, P.: An algebra for composing access control policies. *ACM Transactions on Information and System Security* 5(1), 1–35 (2002)
6. Bruns, G., Huth, M.: Access-control policies via Belnap logic: Effective and efficient composition and analysis. In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pp. 163–176 (2008)
7. Crampton, J., Leung, W., Beznosov, K.: The secondary and approximate authorization model and its application to Bell-LaPadula policies. In: *Proceedings of 11th ACM Symposium on Access Control Models and Technologies* (2006)
8. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security* 5(2), 169–202 (2002)
9. Li, N., Wang, Q., Qardaji, W., Bertino, E., Rao, P., Lobo, J., Lin, D.: Access control policy combining: Theory meets practice. In: *Proceedings of 14th ACM Symposium on Access Control Models and Technologies*, pp. 135–144 (2009)
10. Ni, Q., Bertino, E., Lobo, J.: D-algebra for composing access control policy decisions. In: *Proceedings of 2009 ACM Symposium on Information, Computer and Communications Security*, pp. 298–309 (2009)
11. Nielson, F., Nielson, H., Hankin, C.: *Principles of Program Analysis*. Springer, Heidelberg (1999)
12. OASIS: eXtensible Access Control Markup Language (XACML) Version 2.0. In: Moses, T. (ed.) *OASIS Committee Specification* (2005)
13. Wijesekera, D., Jajodia, S.: A propositional policy algebra for access control. *ACM Transactions on Information and System Security* 6(2), 286–235 (2003)