A Framework for the Modular Specification and Orchestration of Authorization Policies

Jason Crampton[1] and Michael Huth[2]

[1] Information Security Group, Royal Holloway, University of London
jason.crampton@rhul.ac.uk
[2] Department of Computing, Imperial College London, United Kingdom
M.Huth@imperial.ac.uk

Abstract. Many frameworks for defining authorization policies fail to make a clear distinction between policy and state. We believe this distinction to be a fundamental requirement for the construction of scalable, distributed authorization services. In this paper, we introduce a formal framework for the definition of authorization policies, which we use to construct the policy authoring language APOL. This framework makes the required distinction between policy and state, and APOL permits the specification of complex policy orchestration patterns even in the presence of policy gaps and conflicts. A novel aspect of the language is the use of a switch operator for policy orchestration, which can encode the commonly used rule- and policy-combining algorithms of existing authorization languages. We define denotational and operational semantics for APOL and then extend our framework with statically typed methods for policy orchestration, develop tools for policy analysis, and show how that analysis can improve the precision of static typing rules.

1 Introduction

One of the fundamental security services in computer systems is *access control*, a mechanism for constraining the interaction between (authenticated) users and protected resources. Generally, access control is implemented by an authorization service, which includes an *authorization decision function* (ADF) for deciding whether a user request to access a resource (an *access request*, which we abbreviate by *request* henceforth) should be permitted or not. The output of an authorization decision function is usually determined by evaluating the request with respect to *authorization state*.

The protection matrix [14] is one of the earliest techniques for encoding authorization state. It assumes the existence of a set of subjects S (those entities that generate requests), a set of objects O (those entities to which access is requested), and a set of actions A (the types of interactions with objects that subjects may request). Mathematically, the protection matrix M is a total function $S \times O \to \mathbb{P}(A)$ where $M[s, o]$ is the set of interactions that subject s is authorized to engage in with object o. A request is modeled as a triple (s, o, a) and is authorized if and only if $a \in M[s, o]$. This explicit enumeration of all authorized requests in the authorization state is appealing in its simplicity. The authorization *policy*, which is implemented by the ADF, is to authorize a request if it is listed in the authorization state.

In recent years, this enumeration of authorized requests in authorization state has been refined, with authorized requests being grouped together into "targets". Authorization state can then be seen as a set of targets $\{T_1, \ldots, T_n\}$ where $T_i \subseteq S \times O \times A$. Typically, a request (s, o, a) is authorized if and only if $(s, o, a) \in T_i$ for some i. Access control lists (ACL) are one obvious example of this approach, where each target T_i is associated with a particular object o_i.

We may also extend what we call "*target-based*" authorization state by associating explicit deny and allow responses with targets, so that exceptions to requests authorized elsewhere in the authorization state can be articulated. Given an extended set of targets $\{(T_1, \mathsf{allow}), (T_2, \mathsf{deny})\}$, e.g., a request (s, o, a) is authorized if and only if $(s, o, a) \in T_1$ and $(s, o, a) \notin T_2$. Here we see that the authorization state may not be consistent: T_1 may allow a request, while T_2 may deny it. Most authorization frameworks provide a number of different ways of resolving such *conflicts* (such as "allow-overrides" or "deny-overrides"). Conversely, the authorization state could contain *gaps* and neither allows nor denies certain requests.

The literature includes many target-based specification languages for defining authorization state (e.g. [5, 10, 15]) – notably XACML [18] – and target-based policy algebras (e.g. [2, 6, 21]). In the case of XACML, we would define the authorization policy to be the specification of the policy decision point (PDP) – the algorithm that processes what the XACML standard refers to as "policies". Different implementations of the PDP may yield different authorized requests for the same "policies".

Motivation. Much recent work on access control has blurred the distinction between what we call *authorization state* and *authorization policy*. Consider the simple security property p_{SS}, defined in the Bell-LaPadula access control model [4], which says that subject s is authorized to read object o only if $\lambda(s) \geq \lambda(o)$, where $\lambda : S \cup O \rightarrow L$ is a labeling function and L is a lattice of security labels. The *policy* is that a subject is authorized to read an object only if its security classification is at least as high as that of the requested object. The *state* is defined by L and λ. To reinforce this distinction, suppose that $\lambda(o) = l_1$ at time t_1, and subsequently the contents of o are de-classified so that $\lambda(o) = l_2 < l_1$ at time $t_2 > t_1$. Now, for a subject s with $\lambda(s) = l_2$ at times t_1 and t_2, a request to read o is denied at t_1 and allowed at t_2. Thus, the decision depends on the request, the authorization state (λ is mutable), and the immutable policy ($\lambda(s) \geq \lambda(o)$).

Target-based "policies", however, do not make this distinction. The confusion arises because the protection matrix policy is to test for membership of a request in a set encoded by the protection matrix, and so the policy itself has become implicit. Although it is clearly possible to express most authorization policies using a protection matrix – by simply encoding all authorized triples in the matrix – such representations are very inefficient and "brittle": since state and policy are encoded in the matrix it will be necessary to change the matrix to re-encode the policy every time there is a state change. To encode the simple security property above, e.g., any change to $\lambda(o)$ requires adding action $read$ into entry $M[s', o]$ for all subject s' with $l_2 \leq \lambda(s')$ and $l_1 \not\leq \lambda(s')$.

The evaluation of authorization policies may also be strongly dependent on system state. The Chinese wall policy [7], e.g., is a separation of duty policy designed to prevent conflict of interest. The evaluation of this policy requires historical information about which requests have previously been made and authorized. It is not clear how to

represent or evaluate such policies using target-based policies. Similarly, stack-walking algorithms for evaluating requests in a virtual machine environment require information about the run-time state in order to determine whether a request is authorized [13].

Target-based "policies" encode authorization state and policy, so every instance of a target-based "policy" has to re-encode the semantics of the policy it seeks to enforce. In this sense, target-based policies are analogous to monolithic programs that neither benefit from the reuse of already existing authorization decision functions, nor cleanly separate authorization state from those policies. We thus believe that there is great value in a framework that supports the *modular* specification and realization of authorization policies, and that also provides for separation of state and policy.

The framework we propose has two types of policies: *decision policies* and *orchestration policies*. Decision policies are similar to Boolean functions, whereas orchestration policies are similar to policy combining algorithms in XACML [18] and operators in policy algebras (e.g. [6, 19]).

Decision policies take parts of the request or authorization state (or both) as input and make either a Boolean decision or return a third value $\perp$, indicating that the policy is unable to provide a *conclusive* decision. A policy may return $\perp$ because

- the request either does not have the expected form for successful processing (e.g. the action is $delete$ but needs to be $read$ for p_{SS}), or
- the request cannot be evaluated in the authorization state (e.g. there may not be an ACL for the requested object).

Orchestration policies take other policies as input. We show that all possible orchestration requirements, even in the presence of inconsistency or lack of information, can be programmed with a single 4-case *switch operator*. Use of this operator should appeal to people familiar with such statements in mainstream programming languages. Indeed, we develop a simple typed, modular programming language in which decision and orchestration policies are distinguished by types and are declared and enforceable as parameterized methods.

Contributions. We develop a formal framework for authorization policies in which base policies encapsulate domain-specific aspects and offer an abstract interface for orchestration; all possible orchestration patterns for base policies are supported in the presence of conflict or lack of information; and authorization state and policy specifications are cleanly separated, facilitating maintenance and reuse. Policy orchestration is achieved with a switch operator that is formally analyzable and functionally complete for policy coordination (including conflict resolution). We add typed, parameterized methods to that core policy language. This not only facilitates reuse and modular analysis of policies, but these types and their analysis can also certify important run-time behavior of policy evaluation.

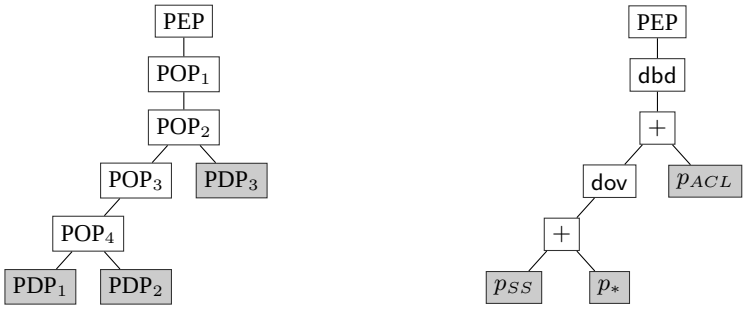
2 Authorization using Trees

We first fix terminology and provide an overview of our approach. We then describe policy orchestration before introducing *policy trees* as formal foundations for APOL.

Overview. We assume the existence of three types of entities: policy enforcement points, policy orchestration points and policy decision points.[3] As in the XACML architecture, a policy enforcement point (PEP) is responsible for ensuring that (i) every request is evaluated to determine whether it is authorized and (ii) that the request is only allowed to proceed (i.e. granted) if it is authorized.

Unlike in XACML, a policy decision point (PDP) in our architecture exists to determine whether a request is authorized by a base policy (defined below). We introduce policy orchestration points (POP) to forward requests to PDPs or other POPs for evaluation. The POP combines the decisions returned in response to those requests, according to the orchestration pattern defined for that POP. The POP then returns a decision to the PEP (or a higher level POP).

Each base policy has its own PDP. Complex authorization policies are constructed by orchestrating base policies. Hence, the authorization architecture required to evaluate an orchestrated policy will be dependent on the policy. In this respect, our architecture is quite different from existing approaches, such as XACML, which assume a *single* PEP and a *single* PDP – reflecting that the policy in target-based approaches is implicit (and is based on membership of the request in one or more targets). Figure 1(a) illustrates schematically an example of this policy evaluation architecture. Henceforth, we will blur the distinction between a PDP and the base policy it enforces and use the two terms interchangeably.



(a) Generic tree (b) BLP policy

Fig. 1. Examples of policy evaluation trees

We assume that base policies are invoked by a policy orchestration point. A base policy returns an authorization decision based on the request *and* the current authorization state of the system. Returning to the example of the simple security property p_{SS} introduced in Section 1, informally speaking and writing σ_i to denote the state at time t_i, we have $p_{\mathsf{SS}}((s, o, read), \sigma_1) = \mathsf{deny}$ and $p_{\mathsf{SS}}((s, o, read), \sigma_2) = \mathsf{allow}$. We model base policies as (partial) Boolean functions and authorization state is an input to

[3] We prefer this terminology to authorization enforcement function etc., as it is widely used and reflects the fact that access control in our setting is policy-based.

a policy. This separation of concern allows us to decouple policy semantics from the specification of authorization state, in contrast to existing approaches such as XACML.

A POP or a PDP does not necessarily take a request of the form (s, o, a), or similar, as input. Consider, for example, an authorization service that implements a policy p_{ACL} that decides requests on the basis of membership in an ACL. Then the PEP may well receive a request of the form (s, o, a), but it actually passes s, a and the ACL for o to the PDP.[4] Hence, all requests of form (s, o, a) can be processed without error (assuming that o is a valid object identifier) and are processed in the same way. In contrast, p_{SS} does not process requests in this uniform manner: it is "silent" on the evaluation of $write$ requests.

The ACL, however, is part of the authorization state, so policy p_{ACL} cannot be used to evaluate request (s, o, a) if it is not possible to locate and retrieve the ACL for object o. Indeed, the evaluation of all but the simplest policies (such as those that authorize all requests) will require authorization state as input, and is therefore acutely sensitive to the availability and consistency of such state. One cannot evaluate the simple security property p_{SS} if, e.g., $\lambda(s)$ is not available or does not belong to the security lattice L.

In summary: there will be requests for which a policy does not return a conclusive decision, simply because the policy is not designed to decide certain requests for particular authorization states; in addition, many policies cannot return a conclusive decision if there is incomplete knowledge of authorization state.

Base policies. Base policies have total functions of type $\mathsf{Req} \times \Sigma \to \{0, 1, \bot\}$ as semantics, where Req is the set of requests and Σ the set of authorization states. Mathematically, a base policy is (semantically) equivalent to a partial function $b : \mathsf{Req} \times \Sigma \to \{0, 1\}$ that has been extended to a total function $\widehat{b} : \mathsf{Req} \times \Sigma \to \{\bot, 0, 1\}$ in the obvious way.

The intuition and assumption is that a base policy b returns a *conclusive* decision (0 for prohibitions or 1 for authorizations) for all well-formed requests as input that can be properly evaluated in the current state. Base policy p_{ACL}, e.g., makes a conclusive decision for requests (s, o, a) if object o has an ACL, and returns $\bot$ if o has no ACL. We can express the simple security property and the *-property [4] as the following base policies, where σ is understood to include an encoding of the security function λ.

$$p_{\mathsf{SS}}((s,o,a),\sigma) = \begin{cases} 1 & \text{if } \lambda(s) \geq \lambda(o) \text{ and } a \text{ is read} \\ 0 & \text{if } \lambda(s) \not\geq \lambda(o) \text{ and } a \text{ is read} \\ \bot & \text{otherwise} \end{cases}$$

$$p_{*}((s,o,a),\sigma) = \begin{cases} 1 & \text{if } \lambda(s) \leq \lambda(o) \text{ and } a \text{ is write} \\ 0 & \text{if } \lambda(s) \not\leq \lambda(o) \text{ and } a \text{ is write} \\ \bot & \text{otherwise} \end{cases}$$

We work with a set of *base policies* $\mathcal{B}$ that have the above type and from which more complex policies are orchestrated. Actual members of $\mathcal{B}$ will depend on context and

[4] The request received by the PEP may also be called an *application* request or *native* request, and the one passed to the PDP by the PEP may be called a *decision* or *authorization* request.

requirements. We might have $\mathcal{B} = \{p_{\mathsf{ACL}}, p_{\mathsf{SS}}, p_*\}$, for example. Base policies also fit nicely with a view of authorization as a service, where the focus is on the orchestration of base policies informed by the known and trusted behavior of these base policies.

Joining policies. Policy orchestration may be useful where policies are developed independently and their respective results need to be combined before reaching an authorization decision. Alternatively, we may simply need to construct authorization policies out of simpler sub-policies. The BLP model, for example, "orchestrates" three policies: the simple security property, the *-property and the discretionary security property (which requires that the request be authorized by a protection matrix M) [4].

Many existing languages, therefore, include the possibility of combining the decisions returned by two policies, and our language is no exception. We write $p_1 + p_2$ to denote the *join* of p_1 and p_2, and define

$$(p_1 + p_2)(r, \sigma) = p_1(r, \sigma) \oplus p_2(r, \sigma),$$

where $\oplus$ is a binary relation on $\{\bot, 0, 1, \top\}$ defined by the following table.

$\oplus$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

A similar join operator was proposed and used in the work of [8, 9], but for policies of different types. We write $\mathcal{P}$ for the set of policies orchestrated from base policies in $\mathcal{B}$.

The orchestration of base policies means that policies in general have a richer type, as total functions $p\colon \mathsf{Req} \times \Sigma \to \{\bot, 0, 1, \top\}$. By abuse of notation, we may write 0 and 1, respectively, for the constant policies $(r, \sigma) \mapsto 0$, and $(r, \sigma) \mapsto 1$. We also write 1_T, where $T \subseteq \mathsf{Req}$, to denote the base policy that returns 1 if $r \in T$ and 0 otherwise.

The switch operator. Many policy algebras and policy languages define ways of resolving gaps ($\bot$) and conflicts ($\top$) in policies as they occur [6, 21], thereby reducing the range of all policies to some subset of $\{\bot, 0, 1, \top\}$. Reducing the range to $\{\bot, 0, 1\}$, e.g., removes conflicts, and reducing the range to $\{0, 1, \top\}$ removes gaps. XACML, e.g., uses *rule-combining* and *policy-combining algorithms* to remove conflicts [18].

We introduce the *switch* policy operator, which can be used *inter alia* to remove gaps and conflicts. Informally, this operator is a total function of type $\mathcal{P}^5 \to \mathcal{P}$, where the decision computed by evaluating the first policy determines which of the other four policies should be evaluated to obtain the overall decision. More formally, we have:

Definition 1. *Let p, $q_\bot$, q_0, q_1 and $q_\top$ be policies. Then the formal expression $(p : q_\bot, q_0, q_1, q_\top)$ is a policy with* switch policy *p such that for all $(r, \sigma) \in \mathsf{Req} \times \Sigma$,*

$$(p : q_\bot, q_0, q_1, q_\top)(r, \sigma) \stackrel{\text{def}}{=} \begin{cases} q_\bot(r, \sigma) & \text{if } p(r, \sigma) = \bot, \\ q_0(r, \sigma) & \text{if } p(r, \sigma) = 0, \\ q_1(r, \sigma) & \text{if } p(r, \sigma) = 1, \\ q_\top(r, \sigma) & \text{if } p(r, \sigma) = \top. \end{cases} \tag{1}$$

The switch operator is surprisingly versatile. We now describe some of its more immediate applications. We can define a policy $!p$ that reverses those decisions for which p is conclusive, and a policy (p/F) where F filters policy p as follows:

$$!p \stackrel{\text{def}}{=} (p : \bot, 1, 0, \top) \tag{2}$$

$$(p/F) \stackrel{\text{def}}{=} (1_F : \%, \bot, p, \%) \tag{3}$$

Since 1_F cannot output $\bot$ or $\top$, we use the reserved symbol $\%$ to denote "dead code", policies that cannot be reached by the switch operator. In particular, we write $\%$ for $q_\top$ whenever the switch policy p is a base policy (and so cannot produce $\top$ as a decision).

Policy p/F models the *if* operator of [8, 9] where base policy F models what is called an "access predicate" in those papers. We may think of F as a filter and of p as the filtered policy. Indeed, if p is the constant 0 or 1 policy, p/F models an XACML rule with target F and effect p: policy expression $(1_F : \%, \bot, 0, \%)$, for example, encodes an XACML rule with target F and effect deny.

We may also express the join operator $p_1 + p_2$ (introduced above) using the switch operator:

$$(p_1 + p_2) \stackrel{\text{def}}{=} (p_1 : p_2, q_0, q_1, \top) \tag{4}$$

where $q_0 = (p_2 : 0, 0, \top, \top)$ and $q_1 = (p_2 : 1, \top, 1, \top)$. Moreover, all the usual policy modifiers can be expressed using the switch operator. We have, for example,

$$\mathsf{dov}(p) = (p : \bot, 0, 1, 0) \qquad \mathsf{aov}(p) = (p : \bot, 0, 1, 1) \qquad \mathsf{agr}(p) = (p : \bot, 0, 1, \bot)$$
$$\mathsf{dbd}(p) = (p : 0, 0, 1, \top) \qquad \mathsf{abd}(p) = (p : 1, 0, 1, \top)$$

where: dov denotes "deny-overrides; dbd denotes "deny-by-default"; aov denotes "allow-overrides"; abd denotes "allow-by-default"; and policy $\mathsf{agr}(p)$ returns the same decision as p if p returns a conclusive decision and returns $\bot$ otherwise. Most of these constructions are supported in XACML; agr, however, is not.

Clearly the switch operator provides a very economical and uniform way of specifying a wide variety of orchestration patterns. For example, we can easily specify an orchestration aed ("allow-overrides-else-deny") that returns 1 whenever p does and returns 0 otherwise: namely $\mathsf{aed}(p) = (p : 0, 0, 1, 1)$.

The switch statement $(p : p', 0, 1, p')$ is the policy that returns whatever policy p returns when p returns a conclusive decision, and returns whatever policy p' returns otherwise. In other words, $(p : p', 0, 1, p')$ is a "first-applicable" binary policy operator, commonly used to evaluate firewall rule-sets. Finally, one can also program "majority-out-of-n" for any odd natural number $n > 2$ with nested switch operators.

The policy $q_\top$ in (1) can be seen as a *conflict handler* for the switch policy p. The choice of this handler depends on the context within which $q_\top$ itself is orchestrated. For example, if the switch statement for $q_\top$ is the outermost policy, a conservative approach would make it always prohibit all requests. But $q_\top$ may have negative polarity (with respect to applications of ! above) as a sub-policy within an orchestration and so prohibiting all requests may be unwise. Polarity issues aside, $q_\top$ may be non-constant since it could make its decision depend on further switch statements that try to differentiate the source of conflict – a common idiom in programming with exceptions. In this context we note that the join operator distributes through all policies q_v in (1), but not through the switch policy p.

Example 1. We can orchestrate the full policy defined in the BLP model as

$$p_{\mathsf{BLP}} = \mathsf{dbd}(\mathsf{dov}(p_{\mathsf{SS}} + p_*) + p_{\mathsf{ACL}}).^5$$

Figure 1(b) shows the policy's evaluation architecture. An alternative, and perhaps more natural, orchestration can be defined using the switch operator:

$$p_{\mathsf{BLP}} = (p_{\mathsf{SS}} + p_* : \bot, 0, p_{\mathsf{ACL}}, 0)$$

Writing $p_{\mathsf{IF}} = \mathsf{dov}(p_{\mathsf{SS}} + p_*)$ to denote the *information flow* policy of the BLP model, we have $p_{\mathsf{BLP}} = \mathsf{dbd}(p_{\mathsf{IF}} + p_{\mathsf{ACL}})$. We may then view p_{IF} as a base policy since any conflicts arising from the evaluation of p_{SS} and p_* are resolved by the dov operator.

Note that an organization may choose to construct a new base policy (e.g. p_{IF} above) from existing ones and "hide" the original base policies (e.g. p_{SS} and p_*) from policy orchestrators. A modular policy language should therefore support such encapsulation.

Note also that the type of policy composition described in Example 1 is not possible in target-based policy languages, because the "policies" written in such languages encode state as well as policy and are therefore inextricably bound to the context in which they were authored.

The following result establishes that our language is powerful enough to realize any orchestration function.

Theorem 1. *For each $n > 0$, all total functions of type $\{\bot, 0, 1, \top\}^n \to \{\bot, 0, 1, \top\}$ can be generated from the constants $\bot$, 0, 1, $\top$ and the switch operator.*

Arieli and Avron [1] consider the expressive power of the language incorporating the connectives $\{\bot, 0, 1, \top, \neg, \vee, \wedge, \oplus, \otimes, \supset, \rightarrow, \leftrightarrow\}$. They prove that the language using connectives from $\{\neg, \wedge, \supset, \bot, \top\}$ is *functionally complete* and no proper subset of these connectives has this property [1, Theorem 3.8]. We use this result as the basis for the proof of Theorem 1.

Proof (Sketch). We show that the operators $\neg$, $\supset$ and $\wedge$ can be encoded using $\bot$, $\top$, 0, 1 and the switch operator. The definitions of $\supset$ and $\neg$ given below in (5) are equivalent to those given by Arieli and Avron (using connectives from $\{\neg, \wedge, \supset, \bot, \top\}$).

$$x \supset y \stackrel{\text{def}}{=} (x : 1, 1, y, y) \quad \text{and} \quad \neg x \stackrel{\text{def}}{=} (x : \bot, 1, 0, \top) \tag{5}$$

We now consider the $\wedge$ operator (which is analogous to conjunction in classical logic), the "truth" table for which is shown below.

x	y	$x \wedge y$
$\bot$	$\bot$	$\bot$
$\bot$	0	0
$\bot$	1	$\bot$
$\bot$	$\top$	0

x	y	$x \wedge y$
0	$\bot$	0
0	0	0
0	1	0
0	$\top$	0

x	y	$x \wedge y$
1	$\bot$	$\bot$
1	0	0
1	1	1
1	$\top$	$\top$

x	y	$x \wedge y$
$\top$	$\bot$	0
$\top$	0	0
$\top$	1	$\top$
$\top$	$\top$	$\top$

[5] We have chosen to implement the discretionary security property using the standard interpretation of a protection matrix as a set of ACLs.

p, q ::=		(Policy Trees)
$\mathsf{allow} \mid \mathsf{deny} \mid \mathsf{na} \mid \mathsf{conflict}$		Constant Policy
b		Base Policy
$(p\colon q_\bot, q_0, q_1, q_\top)$		Policy Switch

(a) Grammar

$$[\![\mathsf{allow}]\!](r,\sigma) \stackrel{\text{def}}{=} 1 \qquad [\![\mathsf{deny}]\!](r,\sigma) \stackrel{\text{def}}{=} 0$$

$$[\![\mathsf{na}]\!](r,\sigma) \stackrel{\text{def}}{=} \bot \qquad [\![\mathsf{conflict}]\!](\mathsf{r},\sigma) \stackrel{\text{def}}{=} \top$$

$$[\![b]\!](r,\sigma) = \rho(b)(r,\sigma)$$

$$[\![(p\colon q_\bot, q_0, q_1, q_\top)]\!](r,\sigma) \stackrel{\text{def}}{=} [\![q_v]\!](r,\sigma) \quad \text{where } [\![p]\!](r,\sigma) = v$$

(b) Denotational semantics

$$\frac{}{(\mathsf{allow}, r, \sigma) \rightsquigarrow 1}\ C1 \qquad \frac{}{(\mathsf{deny}, r, \sigma) \rightsquigarrow 0}\ C0 \qquad \frac{}{(\mathsf{conflict}, r, \sigma) \rightsquigarrow \top}\ C\top$$

$$\frac{}{(\mathsf{na}, r, \sigma) \rightsquigarrow \bot}\ C\bot \qquad \frac{\rho(b)(r,\sigma) = v}{(b, r, \sigma) \rightsquigarrow v}\ Base \qquad \frac{(p, r, \sigma) \rightsquigarrow v \ \ (q_v, r, \sigma) \rightsquigarrow v'}{((p\colon q_\bot, q_0, q_1, q_\top), r, \sigma) \rightsquigarrow v'}\ Switch$$

(c) Inference rules for operational semantics

Fig. 2. Grammar and semantics for policy trees

Noting that $0 \wedge y = 0$ and $1 \wedge y = y$ for all $y \in \{\bot, 0, 1, \top\}$, it is easily seen that we can encode $x \wedge y$ as $(x : z, 0, y, z')$, where

$$z = (y : \bot, 0, \bot, 0) \quad \text{and} \quad z' = (y : 0, 0, \top, \top).$$

Note that: (i) we cannot encode $\supset$ without 1; (ii) we cannot encode $\wedge$ without 0; and (iii) we cannot encode $\neg$ without 0 and 1.

Grammar and semantics for policy orchestration. We now summarize and formalize the preceding discussion by giving a grammar and formal semantics for policy trees. Figure 2(a) depicts the grammar for policies, which are built out of some set of base policies, constant policies (of which all except conflict are base policies), and the switch operator.

The denotational semantics for policy trees are shown in Figure 2(b), relative to an environment ρ that gives semantics $\rho(b) : \mathsf{Req} \times \Sigma \to \{\bot, 0, 1\}$ to all base policies b. Figure 2(c) shows a "big-step" structural operational semantics of policy trees, again relative to an environment ρ. An induction on the height of derivation trees for judgments $(p, r, \sigma) \rightsquigarrow v$ can be used to establish that the operational and denotational semantics compute the same meaning:

`pol` ::=		(Policy)
	`allow` | `deny` | `na` | `conflict`	Constant Policy
	$\%$	Dead Code
	`base`	Base Policy
	`switch` {`pol`: `pol`; `pol`; `pol`; `pol`}	Policy Switch

Fig. 3. Core policy-orchestration language APOL for some set of base policies

Theorem 2. *For all policy trees $p \in \mathcal{P}$ and environments ρ, equation $[\![p]\!](r, \sigma) = v$ holds if and only if $(p, r, \sigma) \rightsquigarrow v$ can be derived using the inference rules in Figure 2(c).*

3 APOL: a typed, modular policy language

We now develop a treatment of policies as *typed, modular programs*. Modularity facilitates compositionality, maintainability, and reuse of policies. Types are conservative mechanisms for preventing certain kinds of "run-time" errors during request evaluation. For example, types can certify that the use of $\%$ to represent dead code is safe for the evaluation of a policy for any request and authorization state.

The grammar of a core programming language APOL ("authorization policy orchestration language") is shown in Figure 3: it is essentially the grammar for policy trees shown in Figure 2(a), extended with a symbol $\%$ for dead code, and presented in a form more amenable to programming. Below, we present a static type system where types are inferred from the syntax of policies without evaluating them. We then also demonstrate that deeper semantic analysis is useful for such static type inference.

Let `p` be an expression of APOL and let $\rho(\mathtt{b})$ be defined for all base policies `b` occurring in `p`. Then we point out that the operational semantics of `p` in Figure 2(c) is well defined by matching clauses of the grammars in Figures 2(a) and 3. However, as there is no rule for $(\%, r, \sigma) \rightsquigarrow \ldots$, dead code does not evaluate to any value, meaning that its evaluation is stuck and constitutes an error.

Typed methods for APOL. We extend our core policy language with typed, parameterized methods. The types τ in the extended language are base (for base policies, that cannot output $\top$), pol (for orchestrated policies), and prd (for base policies that cannot output $\bot$ and so represent Boolean predicates). Each method has $\tau \in \{\mathsf{base}, \mathsf{pol}, \mathsf{prd}\}$ as return type, specifying that method invocations in-lined with correctly typed input policies render a policy of type τ. The following method, for example, explicitly implements the deny-overrides orchestration pattern. Note the return type `base` and the parameter type `pol` for its argument policy:

```
base deny-overrides(P:pol) { switch{P : na; deny; allow; deny} }
```

The type system for such methods is presented in Figure 4: judgments "expression `e` has type τ in context Γ" have the form $\Gamma \vdash \mathtt{e} : \tau$, where $\tau \in \{\mathsf{pol}, \mathsf{base}, \mathsf{prd}\}$ and Γ is a context binding variables x_i to types τ_i. Each rule specifies a possible inference: if all

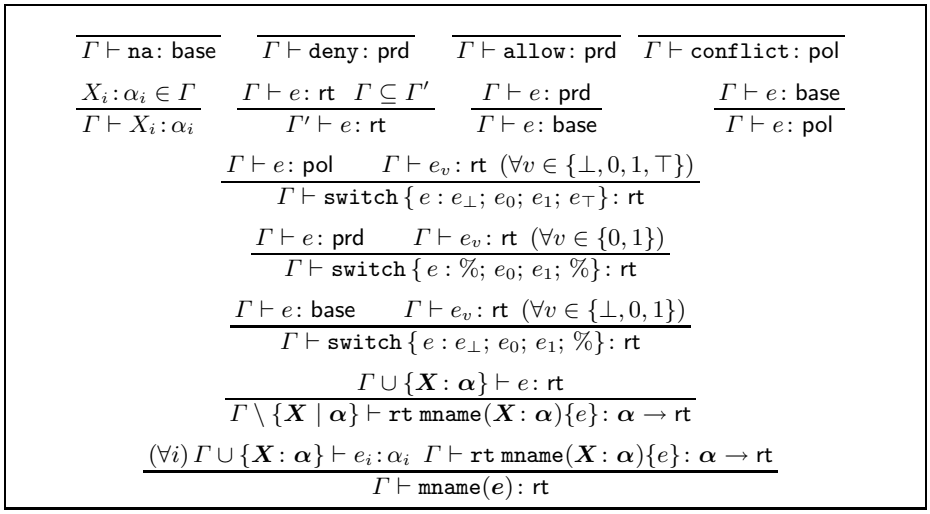


Fig. 4. Rules for type checking judgments $\Gamma \vdash e\colon \mathtt{t}$ for APOL expressions, where Γ is a type context binding variables to types, $\mathtt{rt} \in \{\mathsf{base}, \mathsf{pol}, \mathsf{prd}\}$, and $\boldsymbol{X} : \boldsymbol{\alpha}$ is a list of typed parameters

judgments on top of the line of a rule have been inferred or are given, then the judgment below the line of that rule can be inferred.

The first row in Figure 4 states the types of constant policies, reflecting that `na` is a base policy, `conflict` a non-base policy, and that conclusive decisions are predicates. These are axioms as their inferences do not depend on any judgments. The second row uses standard structural and type casting rules for type inference in the presence of subtyping and states that prd is a subtype of base and that the latter is a subtype of pol. The third row handles type inference for the switch operator: the switch policy must have type pol (through casting of subtypes, if needed), and all argument policies must agree on the output type (again, through casting, if needed).

The next two rows depict two important patterns for reasoning about the safe use of the dead code symbol $\%$. The first one shows that $r = \mathtt{switch}\{\, e : \%;\, e_0;\, e_1;\, \%\}$ has return type `rt` by first showing that the switch policy e has type `prd`, and then showing that both e_0 and e_1 have type `rt`. These type inference rules are safe: for no request and for no authorization state does the operational semantics of r ever evaluate one of the two occurrences of $\%$. This rule can be used to certify that our definition of (p/F), when written in APOL, is type safe. The second rule is very similar but – since it only has a dead code symbol for the case when the switch policy outputs a conflict – it only has to show that the switch policy has type `base`. Again, such type inference guarantees that the symbol $\%$ will never be encountered in the operational semantics.

The final two rows show standard type inference rules for parameterized methods. The first rule assumes the type bindings of the method head, uses those bindings to infer a type for the method body, and that type is then the method return type (but in a type context that no longer relies on these assumptions). The second rule captures that well typed method invocations return the specified type.

Example 2. We illustrate the type system on method `deny-overrides` specified above. To show that this method has output type base, we assume that `P` has type pol as declared in the method header (and so Γ records that binding) and show that the switch statement has output type base under that assumption. But this follows easily from the type inference rule for the switch statement, since all argument policies have subtypes of base or have that type, and so all argument types can be cast into type base, if needed.

Note that variables occurring in a context Γ cannot be constants of APOL. In particular, it is not possible to assign a type to $\%$, and so one can also not give a type to a policy switch with switch policy $\%$.

Our type system can be fine-tuned to enable a richer semantic typing. For example, consider the typed APOL method `foo`, declared by

```
prd foo(P:pol) { switch{P : deny; P; P; deny} }
```

Its argument has type pol and so we cannot assume that this policy is a base policy or free of conflict. Our type system therefore forces that all four argument types of the switch statement be cast into their least common supertype, which is pol. Therefore, the type system can only infer pol as output type. But when `P` is executed as q_0 we know that its output is a `deny`. Similarly, when `P` is executed as q_1 its output is an `allow`. Therefore, it is intuitively safe to assume that all four arguments have type prd and so prd is a safe output type for method `foo`. We now sketch a semantic analysis that formalizes such intuitions. This analysis can then be used to extend our static type inference.

Analysis of APOL policies. For any policy r, let $r \Uparrow 1$ be a propositional formula whose atoms are expressions of the form $b \Uparrow 1$ and $b \Uparrow 0$ for base or constant policies b occurring in policy r. Intuitively, $r \Uparrow 1$ is true if policy r authorizes the (implicit) request, and $r \Uparrow 0$ is true if policy r denies it. The constraint $r \Uparrow 1$ (respectively, $r \Uparrow 0$) therefore expresses the conditions on the base policy decisions for the orchestrated policy r to either authorize (respectively, deny) the request or to report a conflict.

The definition of $r \Uparrow 1$ and $r \Uparrow 0$ is by induction over terms of APOL. For constant policies, these conditions merely express the obvious meaning of these constants. For example, `conflict` $\Uparrow 1$ and `conflict` $\Uparrow 0$ both are the truth constant $true$, whereas `na` $\Uparrow 1$ and `na` $\Uparrow 0$ both are the truth constant $false$. For the other two logical constants, we set `deny` $\Uparrow 1 = false$, `deny` $\Uparrow 0 = true$ and `allow` $\Uparrow 1 = true$, `allow` $\Uparrow 0 = false$. Assuming that dead code is type safe, we set $\% \Uparrow 1$ and $\% \Uparrow 0$ both to be $false$.

The meaning of $b \Uparrow 1$ and $b \Uparrow 0$ for base policies b will depend on the application domain and concrete nature of those base policies. For example, base policies may be written over equations of attributes and so these constraints may be propositional formulas over such attribute conditions.

It remains to specify the conditions $r \Uparrow 1$ and $r \Uparrow 0$ when r is the switch statement $(p : q_\bot, q_0, q_1, q_\top)$. Then we can define $r \Uparrow 1$ in the following way:

$$\begin{aligned} r \Uparrow 1 = {} & (\neg(p \Uparrow 0) \wedge \neg(p \Uparrow 1) \wedge (q_\bot \Uparrow 1)) \vee \\ & ((p \Uparrow 0) \wedge \neg(p \Uparrow 1) \wedge (q_0 \Uparrow 1)) \vee \\ & (\neg(p \Uparrow 0) \wedge (p \Uparrow 1) \wedge (q_1 \Uparrow 1)) \vee \\ & ((p \Uparrow 0) \wedge (p \Uparrow 1) \wedge (q_\top \Uparrow 1)) \end{aligned} \tag{6}$$

The definition of $r \Uparrow 0$ merely changes all $q_v \Uparrow 1$ in (6) to $q_v \Uparrow 0$. The intuition of these constraints should be clear. Each disjunct specifies, in its first two conjuncts, the intended continuation location of the switch statement, and captures the intended condition for that continuation in its third conjunct. These constraints are therefore disjunctions of conjunctions of similar constraints for subpolicies.

Given such formulas, one can then build other constraints, e.g., $\neg(r \Uparrow 0) \wedge (r \Uparrow 1)$, which states the conditions for policy r to authorize a request (and so, in particular, not to report a conflict). In order to determine whether a policy r has a conflict, for example, is equivalent to determining whether $(r \Uparrow 1) \wedge (r \Uparrow 0) \wedge \bigwedge_b \neg((b \Uparrow 1) \wedge (b \Uparrow 0))$ is satisfiable, where b ranges over all base policies occuring in r. Note that the third conjunct rules out spurious witnesses since base policies cannot return $\top$.

We can apply this analysis to infer the more informative type `prd` of method `foo`, which our static type system could not do. For that, it suffices to show that the body r of the method always returns 0 or 1, for any switch policy p of type `pol`; that is to say, that $r \Uparrow 1$ is equivalent to $\neg(r \Uparrow 0)$ if we interpret $p \Uparrow 0$ and $p \Uparrow 1$ as atomic propositions.

Applying (6) to that r, and noting that `Deny` $\Uparrow 1 = false$, we compute

$$\begin{aligned} r \Uparrow 1 = {} & (\neg(p \Uparrow 0) \wedge \neg(p \Uparrow 1) \wedge false) \vee \\ & ((p \Uparrow 0) \wedge \neg(p \Uparrow 1) \wedge (p \Uparrow 1)) \vee \\ & (\neg(p \Uparrow 0) \wedge (p \Uparrow 1) \wedge (q_1 \Uparrow 1)) \vee \\ & ((p \Uparrow 0) \wedge (p \Uparrow 1) \wedge false)) \\ = {} & \neg(p \Uparrow 0) \wedge (p \Uparrow 1) \end{aligned} \tag{7}$$

Noting that `Deny` $\Uparrow 0 = true$, we similarly compute $r \Uparrow 0 = (p \Uparrow 0) \vee (\neg(p \Uparrow 0) \wedge \neg(p \Uparrow 1))$. But it is easily seen that $\neg(r \Uparrow 0)$ and $r \Uparrow 1$ are equivalent formulas.

4 Discussion

We now discuss our contributions and put them into perspective.

Separating state and policies. We have argued that policy and state should be separated, and have developed a framework that meets this criterion. In a practical setting, enterprise security requirements must be encoded in authorization policies, which may require complex orchestration patterns. However, the authorization state is independent of these orchestrations, and different parts of the state would typically be maintained by different local administrators who are responsible for correctly associating users and resources with security-related attributes.

Informing future authorization languages. We now consider how our work is related to target-based approaches to authorization and suggest how these connections might usefully inform the development of authorization languages.

Recall that the protection matrix authorizes a request if that request is encoded in the matrix. The policy is to allow if that request is encoded in the matrix and deny otherwise – the simplest possible way of deciding whether a request is authorized. In our view, the authorization state is the matrix M (or some other suitable data structure, such as a collection of ACLs). Accordingly, we define the base policy p_{allow}, where $p_{\mathsf{allow}}(r, M) = 1$ if $r \in M$, and $\bot$ otherwise.

We can, of course, prohibit certain requests using the orchestration $!p_{\mathsf{allow}}$ for an appropriate choice of M. Complex policies can be built from authorization state components $M_1, \ldots, M_k$ and appropriate orchestrations using $+$ and ! (or the switch operator). Hence, policy authors can define orchestrations, while local administrators can update M_i to reflect changes to personnel and resources.

Informing XACML. We now reflect on how the features of our framework might usefully be applied in XACML. We focus our attention on XACML, because it is a well known standard that provides a framework for the specification and evaluation of target-based authorization policies. In conflating state and policy, target-based approaches such as XACML mean either that policy authors must be aware of authorization state or that local administrators must be able to author XACML policies. This makes it more difficult to author and maintain policies. In our framework, a component of authorization state can be regarded as (just) another resource and can be protected by an authorization policy like any other resource. In XACML, there is no structured support for policy updates, which therefore continue to be a problematic issue.

We believe the switch operator could usefully be deployed in XACML (and other authorization languages). Currently, the XACML standard requires that a number of rule- and policy-combining algorithms be supported by the PDP. Moreover, XACML rules and policies are indistinguishable in terms of semantics, so it is unnecessary to have both rule- and policy-combining algorithms. It has also been observed that there are certain pathological cases in which the combination of rule- and policy-combination algorithms (compliant with the XACML standard) leads to unexpected results [17]. Hence, we make three suggestions: (i) XACML should remove the (artificial) distinction between rules and policies; (ii) a single algorithm based on the switch operator should be supported (since we showed that it can encode the standard policy-combination algorithms); and (iii) types and modularity (as sketched in this paper) should be supported to provide safer policy orchestration for policy authors.

A feature of XACML and target-based policy languages and algebras is the immediate resolution of conflicts. As can be seen from our framework, there is no theoretical necessity for doing this and there may be good practical reasons for postponing the computation of a conclusive decision. A convincing case for decoupling policy composition from conflict resolution has been made in [8] already. In XACML terminology, the rule- and policy-combining algorithms should not be required attributes of policy and policy set elements, respectively, thereby enabling XACML policies and policy sets to return conflicts.

Related work. The work of Bruns *et al.* [8, 9] is closest to that reported in this paper. This work suggested the Belnap space $\{\bot, 0, 1, \top\}$ as carrier of meaning for authorization policies, built atomic policies of the form e/T (in our notation) where T is a subset of Req, employed policy combinators that act on policies in a pointwise manner, and showed that the resulting policy language is as expressive as it can be (thinking of targets T as predicates) [8]. Subsequently, various policy analyses are reduced to checking the satisfiability of formulae in an NP fragment of first-order logic, and such satisfiability checks are extended to a limited form of assume-guarantee reasoning [9]. Our work creates policy languages that have equal expressiveness of orchestration. But our framework separates requests from authorization state, supports types and modularity

for policy orchestration and enforcement, and has a very simple policy analysis in the form of Boolean satisfiability checks (based on a sole policy operator `switch`).

There are a number of authorization frameworks in which policies are based on logic-programming languages such as Datalog (see [3, 11, 12], for example). Although these frameworks are not target-based and can easily express policies such as $p_{\mathtt{SS}}$, the orchestration patterns that are available are limited by the semantics of rule evaluation in the underlying programming language.

5 Conclusions

Summary. The motivation of this paper was to overcome most of the shortcomings of existing target-based authorization-policy languages such as XACML. We began with a semantic view of a policy as a 4-valued function whose input comprises a request and relevant authorization state. This resulted in a language of policy trees with $\{0, 1, \bot\}$-valued base policies where $\bot$ models that a request is not applicable for the policy interface, or that it cannot be evaluated in the authorization state. We then transferred these ideas into the programming language APOL, first within a core language of policy trees and then for an extension of that core language to parameterized, typed methods. We also gave equivalent denotational and operational semantics to the core language. Finally, we discussed how our ideas and results could leverage policy analysis – for example to reason about the type safe identification of dead code – and inform new versions of the XACML standard and future authorization languages.

Future work. Our most recent work shows that programs of type `pol` are representable, and hence implementable, as ternary decision diagrams [20]; that the remote evaluation of sub-programs can be accommodated by the use of decision diagrams that have four kinds of edges and terminals; and that our policy analysis methods can be used to enrich our type system with a notion of semantic (i.e. behavioral) types. We intend to continue our preliminary work in this area.

In our current approach, the output value $\bot$ is overloaded, as we use it to denote both the inapplicability of a base policy to a request and the inability to evaluate a request given current knowledge of the state. If we want to make a distinction between these cases, we would require a 5-valued meaning space $\{0, 1, \sharp, \flat, \top\}$, where 0, 1, and $\top$ retain their meaning and $\sharp$ and $\flat$ denote problems with policy evaluation and policy applicability (respectively). We also hope to develop an implementation of our framework, perhaps making use of XACML-like syntax to define platform-independent authorization state. Thirdly, we mean to devise a modular policy analysis for APOL based on the familiar idea of programming by contracts [16].

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

References

1. O. Arieli and A. Avron. The value of the four values. *Artificial Intelligence*, 102(1):97–141, 1998.

2. M. Backes, M. Dürmuth, and R. Steinwandt. An algebra for composing enterprise privacy policies. In *Proc. of the 9th European Symp. on Research in Computer Security*, pages 33–52, 2004.
3. M.Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *Proc. of 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, 2004.
4. D.E. Bell and L. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, Mitre Corporation, Bedford, Massachusetts, 1976.
5. E. Bertino, S. Castano, and E. Ferrari. Author-$\mathcal{X}$: A comprehensive system for securing XML documents. *IEEE Internet Computing*, 5(3):21–31, 2001.
6. P. Bonatti, S. de Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.
7. D. Brewer and M. Nash. The Chinese Wall security policy. In *Proc. of the 1989 IEEE Symp. on Security and Privacy*, pages 206–214, 1989.
8. G. Bruns, D.S. Dantas, and M. Huth. A simple and expressive semantic framework for policy composition in access control. In V. D. Gligor and H. Mantel, editors, *Proc. of the Fifth Workshop on Formal Methods in Security Engineering: From Specifications to Code*, pages 12–21, 2007.
9. G. Bruns and M. Huth. Access control via Belnap logic: Effective and efficient composition and analysis. In A. Sabelfeld, editor, *Proc. of the 21st IEEE Computer Security Foundations Symp.*, pages 163–176, 2008.
10. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security*, 5(2):169–202, 2002.
11. J. DeTreville. Binder, a logic-based security language. In *Proc. of the 2002 IEEE Symp. on Security and Privacy*, pages 105–113, 2002.
12. D.J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Proc. of Automated Reasoning, Third International Joint Conference, IJCAR 2006*, pages 632–646, 2006.
13. L. Gong. *Inside Java 2 Platform Security*. Addison Wesley, 1999.
14. M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.
15. R. Jagadeesan, W. Marrero, C. Pitcher, and V. Saraswat. Timed constraint programming: A declarative approach to usage control. In *Proc. of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 164–175, 2005.
16. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.
17. Q. Ni, E. Bertino, and J. Lobo. D-Algebra for composing access control policy decisions. In *Proc. of 4th ACM Symp. on Information, Computer and Communications Security*, pages 298–309, 2009.
18. OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*, 2005. OASIS Committee Specification (T. Moses, editor).
19. C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies and complex constraints. In *Proc. of the Network and Distributed System Security Symp. (NDSS)*, pages 89–107, February 2001.
20. T. Sasao. Ternary decision diagrams: Survey. In *Proc. of the 27th International Symp. on Multiple-Valued Logic (ISMVL '97)*, pages 241–250, 1997.
21. D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Transactions on Information and System Security*, 6(2):286–235, 2003.