Automated Verification of Access Control Policies

Graham Hughes   Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{graham,bultan}@cs.ucsb.edu

ABSTRACT

Managing access control policies in modern computer systems can be challenging and error-prone, especially when multiple access policies are combined to form new policies, possibly introducing unintended consequences. In this paper we present a framework for automated verification of access control policies. We introduce a formal model for systematically specifying access to resources. We show that the access control policies in the XACML access control language can be translated to a simple form which partitions the input domain to four classes: permit, deny, error, and not-applicable. We present several ordering relations for access control policies which can be used to specify the properties of the policies and the relationships among them. We then show how to automatically check these ordering relations using an existing automated analysis tool. In particular, we translate XACML policies to the Alloy language and check their properties using the Alloy Analyzer. Our experimental results demonstrate that automated verification of XACML policies is feasible.

1. INTRODUCTION

Keeping track of permission grants, organizational policies and special cases in a modern computer system has become difficult enough on its own; keeping the policy consistent across multiple heterogeneous systems is even more difficult, as each system requires that these grants and policies be expressed in its own specific control language. Several unified access policy languages address this problem. If all the systems use the same access policy language, then access policies need only be written once and similarly only one policy needs to be kept up to date. In this paper we focus on one particular such language, the OASIS standard XACML [32].

Having a combined policy is convenient, but such a policy will inevitably become quite large and complex as all an organization's rules get placed in it. It is possible, even likely, that the act of creating a unified policy out of numerous disparate smaller policies could leave it vulnerable to unintended consequences. In this paper we investigate statically verifying properties of access control policies to prevent such errors. We translate XACML policies into a simplified mathematical model, which we reduce to a normal form separating the conditions that give rise to *access permitted*, *access denied*, and *internal error* results. We define partial orderings between access control policies, with the intention of checking whether a policy is over- or under constrained with respect to another one. We show that these ordering relations can be translated to logical expressions which evaluate to true if and only if the corresponding relation holds. We use Alloy analyzer to check the truth value of these logical expressions automatically. Using our translator and Alloy analyzer, we can check if a combination of XACML policies does or does not faithfully reproduce the properties of its subpolicies, and thus discover unintended consequences before they appear in practice.

In Section 2 we develop a formal model for access policies; in Section 2.2 we discuss how to transform these models into a normal form that distinguishes access permitted, access denied, and error conditions. In Section 4 we define partial ordering relations among access policies which are used to specify their properties. We show how to check these properties automatically in Section 4. Specifically, we discuss a hierarchy of access policies in Section 4.1 which we use in Section 4.3 to show how to translate a policy to the Alloy modeling language. Finally, we report the results of our experiments in Section 5.

Related Work: Access control itself has been extensively researched: [25, 29, 26] introduce the process, [3, 4, 27, 28] describe various models for access control, [6, 9, 8, 7] describe a particular fine grained access control for XML documents, [5] defines an algebra for composing different parts of a model into a unified whole and [2, 12, 10] speak of distributing the control so that it is consistent across a distributed system.

Access policy languages, too, are not new: [1] describes a general purpose policy language for authorization systems, [19] defines a model and language for access control and [18] and [20] present a framework for enforcing multiple access policies by expressing how to combine them in a new language. We chose XACML because it is a standardized language with tool support, and so our results are more likely to be immediately useful.

The problem with access policies becoming large and difficult to reason about has also been studied, but not in the general case: [12] speaks of verifying a hierarchy of security servers to ensure that they are implementing the whole access policy, and [23] presents an algorithm for computing the flow of permissions through the Java security model, to aid static analysis. Neither of these are exactly what we want: [12] can prove that the programs you have collectively implement the policy you specified, but their technique cannot tell you whether you have made a subtle error in creating your policy in the first place; [23] is more comprehensive but is specific to Java's security model.

XACML: XACML is an OASIS standard for specifying access policies; it is written in XML [33]. The language comprises three classes of objects—individual rules, collections of rules called policies, and collections of policies called policy sets. An XACML Policy Enforcement Point, the gateway that determines whether an action is permitted or not, takes *access requests*, which are specially formatted XML documents that define a set of data that we call the *environment*. Policy Enforcement Points yield one of four results: Permit, meaning that the access request is permitted; Deny, meaning that the access request will not be permitted; Not Applicable, meaning that this particular policy says nothing about the request; and Indeterminate, which means that something unexpected came up and the policy has failed. Which result is yielded depends on what result the policy dictates, given the environment defined in the access request.

XACML rules are the most basic object, and have a goal effect—either Permit or Deny—a domain of applicability, and conditions under which they can yield Indeterminate and fail. The domain of applicability is realized in a series of predicates about the environmental data that must all be satisfied for the rule to yield its goal effect; the error conditions are embedded in the domain predicates, but can be separated out into a set of predicates all their own. Policies combine individual rules and also have a domain of applicability; policy sets combine individual policies with a domain of applicability.

XACML predicates comprise one of a number of primitive functions, with mechanisms for extension; we consider only the core functionality. These functions include simple equality, set inclusion, ordering within numeric types, and also more complex functions such as XPath matching and X500 name matching.

2. POLICY SPECIFICATIONS

Let us consider a simple example; a person can vote if they are 18 or over, and have not voted already. Our environment, the set of information we are interested in, consists of the age of the person in question and whether they have voted already. We can represent this as a Cartesian product of XML Schema [34] basic types, as follows:

$$E = \mathcal{P}(\texttt{xsd:int}) \times \mathcal{P}(\texttt{xsd:boolean}) \times \mathcal{P}(\texttt{xsd:string})$$

The first component of the environment E is the age of the person, the second component is whether or not they have voted already, and the third component is the action they are attempting (perhaps voting, but perhaps something else). We use power sets here because in XACML all attributes describe sets of values, never singletons.

The goal for our policy is that if they are doing something other than voting, we do not really care what happens, and we require that there be only one age and one voting record presented. To do this we can divide E into four sets, E_a, E_v, E_p and E_d as follows (note that the notation $\exists!\, x\, P$ asserts that there is a unique x that satisfies a condition P):

$$\begin{aligned}
E_a &= \{\langle a, v, o\rangle \in E : \exists!\, a_0 \in a \wedge \exists!\, v_0 \in v\} \\
E_v &= \{\langle a, v, o\rangle \in E_a : \exists x \in o\; x = \texttt{vote}\} \\
E_p &= \{\langle \{a_0\}, \{v_0\}, o\rangle \in E_v : a_0 \geq 18 \wedge \neg v_0\} \\
E_d &= E_v - E_p = \{\langle \{a_0\}, \{v_0\}, o\rangle \in E_v : a_0 < 18 \vee v_0\}
\end{aligned}$$

Here, E_a is the set of all environments whose inputs are not erroneous, E_v is the set of all environments where voting is attempted, E_p is the set of all environments where the person can vote (their attempt to vote is *permitted*), and E_d is the set of all environments where the person cannot vote (their attempt to vote is *denied*). The XACML specification for this example is given in Appendix A. In the following section we will define a concise formal model for XACML policies.

2.1 Formal Model

Let $R = \{\mathit{Permit}, \mathit{Deny}, \mathit{NotApp}, \mathit{Indet}\}$ be the set of valid results. Now, we can define the set of valid policies P as follows (semantics will be defined later):

$$\begin{aligned}
\mathit{Permit} &\in P \\
\mathit{Deny} &\in P \\
\forall p \in P\ \forall S \subseteq E\ \mathit{Scope}(p, S) &\in P \\
\forall p \in P\ \forall S \subseteq E\ \mathit{Err}(p, S) &\in P \\
\forall p, q \in P\ p \oplus q &\in P \\
\forall p, q \in P\ p \ominus q &\in P \\
\forall p, q \in P\ p \otimes q &\in P \\
\forall p, q \in P\ p \oslash q &\in P
\end{aligned}$$

Here, *Permit* and *Deny* are *primitive policies* that always return permit or deny. We can also enclose policies in *scopes* using the *Scope* function or attach *error conditions* to them the *Error* function. We can further combine policies using *policy combinators* such as $\oplus$, $\ominus$, $\otimes$ and $\oslash$; these policy combinators correspond to the permit overrides ($\oplus$), deny overrides ($\ominus$), only one applicable ($\otimes$), and first applicable ($\oslash$) operations in the XACML specification. To define the semantics of these policies, we define a function eff that takes an environment and policy as inputs and returns a result as an output: $\mathrm{eff} : E \times P \to R$; which is defined in Figure 1.

Our semantics are slightly different from that defined in the XACML standard; specifically, *Permit* does not override *Indet* in $\oplus$ and similarly with $\ominus$. We decided that since *Indet* is an error and not a normal condition, for the purposes of analysis error conditions should not be covered up; using the original semantics does not change the analysis technique, merely some of the normal form transformations.

Using this notation, we can now model our example as fol-

$$
\begin{array}{rcl}
\mathrm{eff}(e, \mathit{Permit}) & = & \mathit{Permit} \\
\mathrm{eff}(e, \mathit{Deny}) & = & \mathit{Deny} \\
\mathrm{eff}(e, \mathit{Scope}(p, S)) & = & \begin{cases} \mathrm{eff}(e,p) & \text{if } e \in S \\ \mathit{NotApp} & \text{otherwise} \end{cases} \\
\mathrm{eff}(e, \mathit{Err}(p, S)) & = & \begin{cases} \mathit{Indet} & \text{if } e \in S \\ \mathrm{eff}(e,p) & \text{otherwise} \end{cases} \\
\mathrm{eff}(e, p \oplus q) & = & \begin{cases} \mathit{Indet} & \text{if } \mathrm{eff}(e,p) = \mathit{Indet} \\ & \quad \vee \mathrm{eff}(e,q) = \mathit{Indet} \\ \mathit{Permit} & \text{if } (\mathrm{eff}(e,p) = \mathit{Permit} \\ & \quad \wedge \mathrm{eff}(e,q) \neq \mathit{Indet}) \\ & \vee(\mathrm{eff}(e,q) = \mathit{Permit} \\ & \quad \wedge \mathrm{eff}(e,p) \neq \mathit{Indet}) \\ \mathit{Deny} & \text{if } (\mathrm{eff}(e,p) = \mathit{Deny} \\ & \quad \wedge \mathrm{eff}(e,q) \neq \mathit{Permit} \\ & \quad \wedge \mathrm{eff}(e,q) \neq \mathit{Indet}) \\ & \vee(\mathrm{eff}(e,q) = \mathit{Deny} \\ & \quad \wedge \mathrm{eff}(e,p) \neq \mathit{Permit} \\ & \quad \wedge \mathrm{eff}(e,p) \neq \mathit{Indet}) \\ \mathit{NotApp} & \text{otherwise} \end{cases} \\
\mathrm{eff}(e, p \ominus q) & = & \begin{cases} \mathit{Indet} & \text{if } \mathrm{eff}(e,p) = \mathit{Indet} \\ & \quad \vee \mathrm{eff}(e,q) = \mathit{Indet} \\ \mathit{Deny} & \text{if } (\mathrm{eff}(e,p) = \mathit{Deny} \\ & \quad \wedge \mathrm{eff}(e,q) \neq \mathit{Indet}) \\ & \vee(\mathrm{eff}(e,q) = \mathit{Deny} \\ & \quad \wedge \mathrm{eff}(e,p) \neq \mathit{Indet}) \\ \mathit{Permit} & \text{if } (\mathrm{eff}(e,p) = \mathit{Permit} \\ & \quad \wedge \mathrm{eff}(e,q) \neq \mathit{Deny} \\ & \quad \wedge \mathrm{eff}(e,q) \neq \mathit{Indet}) \\ & \vee(\mathrm{eff}(e,q) = \mathit{Permit} \\ & \quad \wedge \mathrm{eff}(e,p) \neq \mathit{Deny} \\ & \quad \wedge \mathrm{eff}(e,p) \neq \mathit{Indet}) \\ \mathit{NotApp} & \text{otherwise} \end{cases} \\
\mathrm{eff}(e, p \otimes q) & = & \begin{cases} \mathrm{eff}(e,p) & \text{if } \mathrm{eff}(e,q) = \mathit{NotApp} \\ \mathrm{eff}(e,q) & \text{if } \mathrm{eff}(e,p) = \mathit{NotApp} \\ \mathit{Indet} & \text{otherwise} \end{cases} \\
\mathrm{eff}(e, p \oslash q) & = & \begin{cases} \mathrm{eff}(e,p) & \text{if } \mathrm{eff}(e,p) \neq \mathit{NotApp} \\ \mathrm{eff}(e,q) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 1: Semantics of policies

lows:

$$
\begin{array}{rcll}
S_0 & = & \{\langle a,v,o\rangle \in E : \forall x \in a\ x < 18\} & (1) \\
S_1 & = & \{\langle a,v,o\rangle \in E : \forall x \in v\ x\} & (2) \\
S_2 & = & \{\langle a,v,o\rangle \in E : \exists x \in o\ x = \texttt{vote}\} & (3) \\
S_3 & = & \{\langle a,v,o\rangle \in E : \neg\exists!\, a_0 \in a\} & (4) \\
S_4 & = & \{\langle a,v,o\rangle \in E : \neg\exists!\, v_0 \in v\} & (5) \\
r_1 & = & \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_0), S_3) & (6) \\
r_2 & = & \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_1), S_4) & (7) \\
p & = & \mathit{Scope}(r_1 \ominus r_2 \ominus \mathit{Permit}, S_2) & (8)
\end{array}
$$

Here, S_0 is the set of environments that fail the age requirement, S_1 is the set of environments that fail the voting requirement, S_2 is the set of environments where someone's trying to vote, etc.

2.2 Policy Transformations

We would like to perform analysis on this model, and it would be easier to do this analysis if we could bring the model into a normal form. To do this, first we define equivalence:

$$P_1 \equiv P_2 \text{ iff } \forall e \in E\ \ \mathrm{eff}(e, P_1) = \mathrm{eff}(e, P_2)$$

We call a function f that takes a policy and returns another policy an eff-*preserving transformation* if $\forall p \in P\ f(p) \equiv p$.

For any given policy, we want to regard the subset of E that will give a *Permit* result, the subset of E that will give a *Deny* result, and the subset of E that will give an *Error* result independently. We define the shorthand $\langle S, R, T\rangle$, where S, R and T are pairwise disjoint, as follows:

$$\langle S,R,T\rangle = \mathit{Err}(\mathit{Scope}(\mathit{Permit}, S) \otimes \mathit{Scope}(\mathit{Deny}, R), T)$$

We call this *triple notation* and refer to individual nodes $\langle S,R,T\rangle$ as *triples.*

Now that we have a framework for transforming policies, we would like to transform an entire policy with *Scope*, *Error* and combinators alike into a single triple. We know that for any policy P a triple P_T that is equivalent to it exists: the triple is just $P_T = \langle\{p \in P : \mathrm{eff}(p) = \mathit{Permit}\}, \{p \in P : \mathrm{eff}(p) = \mathit{Deny}\}, \{p \in P : \mathrm{eff}(p) = \mathit{Error}\}\rangle$. However, this is not a constructive definition. To transform the policies to the triple form, we define two functions f and g, both eff-preserving transformations, such that $g(f(p))$ is a triple for all p. The f function transforms the policy into an equivalent one that is composed of triples joined by combinators. The g function combines triples joined by combinators into a single triple. The two together generate the triple representation. We define f in Figure 2, and g in Figure 3.

As an example, applying f to the policy p defined in Equa-

$$
\begin{array}{l}
f : P \to P \\
f(\mathit{Scope}(\mathit{Scope}(X,S),R)) = f(\mathit{Scope}(X, R \cap S)) \\
f(\mathit{Scope}(\mathit{Err}(X,S),R)) = f(\mathit{Err}(\mathit{Scope}(X, R \setminus S), S \cap R)) \\
f(\mathit{Scope}(X \oplus Y, S)) = \mathit{Scope}(f(X),S) \oplus \mathit{Scope}(f(Y),S) \\
f(\mathit{Scope}(X \ominus Y, S)) = \mathit{Scope}(f(X),S) \ominus \mathit{Scope}(f(Y),S) \\
f(\mathit{Scope}(X \otimes Y, S)) = \mathit{Scope}(f(X),S) \otimes \mathit{Scope}(f(Y),S)) \\
f(\mathit{Scope}(X \oslash Y, S)) = \mathit{Scope}(f(X),S) \oslash \mathit{Scope}(f(Y),S) \\
f(\mathit{Scope}(P,S)) = \mathit{Scope}(f(P),S) \\
\qquad \text{if no other rules apply} \\
f(\mathit{Err}(\mathit{Err}(X,S),R)) = f(\mathit{Err}(X, R \cup S)) \\
f(\mathit{Err}(\mathit{Scope}(X,S),R)) = f(\mathit{Err}(\mathit{Scope}(X, S \setminus R), R)) \\
\qquad \text{if } S \cup R \neq \emptyset \\
f(\mathit{Err}(X \oplus Y, S)) = \mathit{Err}(f(X),S) \oplus f(Y) \\
f(\mathit{Err}(X \ominus Y, S)) = \mathit{Err}(f(X),S) \ominus f(Y) \\
f(\mathit{Err}(X \otimes Y, S)) = \mathit{Err}(f(X),S) \otimes f(Y) \\
f(\mathit{Err}(X \oslash Y, S)) = \mathit{Err}(f(X),S) \oslash f(Y) \\
f(\mathit{Err}(P,S)) = \mathit{Err}(f(P),S) \\
\qquad \text{if no other rules apply} \\
f(\mathit{Permit}) = \mathit{Permit} \\
f(\mathit{Deny}) = \mathit{Deny}
\end{array}
$$

Figure 2: eff-preserving transformations for reduction to normal form

$$
\begin{array}{ll}
g : P \rightarrow \langle S, R, T\rangle & \\
g(\langle S_1, R_1, T_1\rangle \oplus \langle S_2, R_2, T_2\rangle) = \langle (S_1 \cup S_2) \setminus (T_1 \cup T_2), & \\
\qquad (R_1 \setminus (S_2 \cup T_2)) \cup (R_2 \setminus (S_1 \cup T_1)), T_1 \cup T_2\rangle & \\
g(\langle S_1, R_1, T_1\rangle \ominus \langle S_2, R_2, T_2\rangle) = & \\
\qquad \langle (S_1 \setminus (R_2 \cup T_2)) \cup (S_2 \setminus (R_1 \cup T_1)), & \\
\qquad (R_1 \cup R_2) \setminus (T_1 \cup T_2), T_1 \cup T_2\rangle & \\
g(\langle S_1, R_1, T_1\rangle \otimes \langle S_2, R_2, T_2\rangle) = & \\
\qquad \langle (S_1 \cup S_2) \setminus ((S_1 \cap S_2) \cup T_1 \cup T_2), & \\
\qquad (R_1 \cup R_2) \setminus ((R_1 \cap R_2) \cup T_1 \cup T_2), & \\
\qquad T_1 \cup T_2 \cup (S_1 \cap S_2) \cup (R_1 \cap R_2)\rangle & \\
g(\langle S_1, R_1, T_1\rangle \oslash \langle S_2, R_2, T_2\rangle) = \langle S_1 \cup (S_2 \setminus (R_1 \cup T_1)), & \\
\qquad R_1 \cup (R_2 \setminus (S_1 \cup T_1)), T_1 \cup (T_2 \setminus (S_1 \cup R_1))\rangle & \\
g(\langle S_1, R_1, T_1\rangle) = \langle S_1, R_1, T_1\rangle & \\
g(P_1 \oplus P_2) = g(g(P_1) \oplus g(P_2)) & \text{if no other rules apply} \\
g(P_1 \ominus P_2) = g(g(P_1) \ominus g(P_2)) & \text{if no other rules apply} \\
g(P_1 \otimes P_2) = g(g(P_1) \otimes g(P_2)) & \text{if no other rules apply} \\
g(P_1 \oslash P_2) = g(g(P_1) \oslash g(P_2)) & \text{if no other rules apply}
\end{array}
$$

Figure 3: eff-preserving transformations for $\langle S, R, T\rangle$ reduction

tion (8) leads to the following:

$$
\begin{aligned}
p &= \mathit{Scope}(\mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_0), S_3) \\
&\qquad \ominus \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_1), S_4) \\
&\qquad \ominus \mathit{Permit}, S_2) \\
f(p) &= \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_2 \cap S_0 \setminus S_3), S_3 \cap S_2) \\
&\qquad \ominus \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_2 \cap S_1 \setminus S_4), S_4 \cap S_2) \\
&\qquad \ominus \mathit{Scope}(\mathit{Permit}, S_2)
\end{aligned}
$$

Note that the function f pushes all Scope forms down to the leaves of the policy tree, and all Err forms down to just above the leaves.

The f function transforms a policy to a collection of expressions of the form $\mathit{Err}(\mathit{Scope}(A, B), T)$ (where $A \in \{\mathit{Permit}, \mathit{Deny}\}$, $B, T \subseteq E$, and $B \cap T = \emptyset$) combined using $\oplus, \ominus, \otimes$ and $\oslash$. Since $\text{eff}(X \otimes \mathit{Scope}(Y, \emptyset)) = \text{eff}(X)$, we can further rewrite these expressions in the form $\mathit{Err}(\mathit{Scope}(\mathit{Permit}, S)$ $\otimes$ $\mathit{Scope}(\mathit{Deny}, R), T)$ combined with $\oplus, \ominus, \otimes$ and $\oslash$ where $S = B$ and $R = \emptyset$ if $A = \mathit{Permit}$ and $S = \emptyset$ and $R = B$ if $A = \mathit{Deny}$. Since S, R and T are all pairwise disjoint this is exactly the required form for our triple notation. Hence, after applying the function f we have a set of subpolicies in our triple notation combined with $\oplus, \ominus, \otimes$ and $\oslash$. We define the function g in Figure 3. The transformations for function g all preserve the disjointness property, and using the function g we can transform the policy generated by function f to a single triple $\langle S, R, T\rangle$ for some $S, R, T \subseteq E$.

When we apply the function g to our example we get the following:

$$
\begin{aligned}
f(p) &= \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_2 \cap S_0 \setminus S_3), S_3 \cap S_2) \\
&\quad \ominus \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_2 \cap S_1 \setminus S_4), S_4 \cap S_2) \\
&\quad \ominus \mathit{Scope}(\mathit{Permit}, S_2) \\
&= \langle \emptyset, S_2 \cap S_0 \setminus S_3, S_3 \cap S_2\rangle \\
&\quad \ominus \langle \emptyset, S_2 \cap S_1 \setminus S_4, S_4 \cap S_2\rangle \\
&\quad \ominus \langle S_2, \emptyset, \emptyset\rangle \\
g(f(p)) &= \langle S_2 \setminus (S_0 \cup S_1 \cup S_3 \cup S_4), \\
&\quad ((S_0 \cup S_1) \setminus (S_3 \cup S_4)) \cap S_2, \\
&\quad (S_3 \cup S_4) \cap S_2\rangle
\end{aligned}
$$

Now that we have our policy into a form that is convenient for analysis, we can begin to prove things about it.

3. PROPERTIES OF POLICIES

In this section we will show that properties of policies can be expressed based on several partial ordering relations. For example, we might want to prove that a (possibly very complex) policy at least protects as much as some simpler policy, and similarly we might want to guarantee that a (possibly very complex) policy does not say anything outside of its scope. Such properties can be expressed using the ordering relations defined below.

Let $P_1 = \langle S_1, R_1, T_1\rangle$ and let $P_2 = \langle S_2, R_2, T_2\rangle$ be two policies. We define the following partial orders:

$$
\begin{array}{rcl}
P_1 \sqsubseteq_P P_2 & \text{iff} & S_1 \subseteq S_2 \\
P_1 \sqsubseteq_D P_2 & \text{iff} & R_1 \subseteq R_2 \\
P_1 \sqsubseteq_E P_2 & \text{iff} & T_1 \subseteq T_2 \\
P_1 \sqsubseteq_{P,D,E} P_2 & \text{iff} & P_1 \sqsubseteq_P P_2 \wedge P_1 \sqsubseteq_D P_2 \wedge P_1 \sqsubseteq_E P_2
\end{array}
$$

Note that, we can define a partial order for for any combination of of P, D and E. We define $P_1 \sqsubseteq P_2 \equiv P_1 \sqsubseteq_{P,D,E} P_2$. We can regard $P_1 \sqsubseteq P_2$ as stating that for any $e \in E$ where $\text{eff}(P_1, e) \neq \mathit{NotApp}$, $\text{eff}(P_2, e) = \text{eff}(P_1, e)$.

To demonstrate the use of these ordering relations, let us create a new policy; people are permitted to check the current results of the election, for exit polls. We encode this with the following policy

$$
\begin{aligned}
S_5 &= \{\langle a, v, o\rangle \in E : \exists x \in o\ x = \texttt{getresult}\} \\
r_3 &= \mathit{Scope}(\mathit{Err}(\mathit{Permit}, S_4), S_5)
\end{aligned}
$$

where S_4 is defined in Equation (5). Now, we can create a composite policy as follows $p_c = p \oplus r_3$, where p is defined in Equation (8). This policy has a bug—specifically, it permits people under 18 to vote in certain circumstances—and we will demonstrate the usefulness of our technique by showing this. First, we perform our translations on this new policy as above, getting:

$$
\begin{aligned}
g(f(r_3)) &= \langle S_5 \setminus S_4, \emptyset, S_4 \cap S_5\rangle \\
g(f(p_c)) &= \langle (((S_2 \setminus (S_0 \cup S_1 \cup S_3 \cup S_4)) \\
&\qquad \cup (S_5 \setminus S_4)) \setminus (((S_3 \cup S_4) \cap S_2) \cup (S_4 \cap S_5)), \\
&\quad (((S_0 \cup S_1) \setminus (S_3 \cup S_4)) \cap S_2) \\
&\qquad \setminus ((S_5 \setminus S_4) \cup (S_4 \cap S_5)), \\
&\quad ((S_3 \cup S_4) \cap S_2) \cup (S_4 \cap S_5)\rangle
\end{aligned}
$$

where S_0, S_1, S_3 and S_4 are from Equations (2) to (5). Using set algebra we can simplify the expression for policy p_c to

$$\begin{aligned} g(f(p_c)) = \langle\, & (S_2 \setminus (S_0 \cup S_1 \cup S_3 \cup S_4)) \\ & \cup (S_5 \setminus ((S_3 \cap S_2) \cup S_4))\,, \\ & ((S_0 \cup S_1) \cap S_2) \setminus (S_3 \cup S_4 \cup S_5), \\ & ((S_3 \cup S_4) \cap S_2) \cup (S_4 \cap S_1 \cap S_5)\rangle \end{aligned}$$

Now, we insist that this combined policy deny anyone trying to vote who is under 18. This is itself a policy, which we call p_v:

$$p_v = \langle \emptyset, (S_0 \cap S_2) \setminus (S_3 \cup S_4), (S_3 \cup S_4) \cap S_2\rangle$$

The interesting thing here is whether or not $p_v \sqsubseteq_D p_c$, i.e., does the policy p_c deny every input that is denied by p_v. That would mean that everyone trying to vote who is under 18 is denied, and that our policy combination has not done any harm. However, the environmental tuple

$$e = \langle \{17\}, \{\texttt{true}\}, \{\texttt{vote}, \texttt{getresult}\}\rangle$$

demonstrates that that is not the case. Input e passes the second part of the $Permit$ requirement and so is permitted by p_c (which means that it is *not* denied by p_c) but denied by p_v, i.e., e demonstrates that $p_v \not\sqsubseteq_D p_c$. The error is that, we do not enforce that only one action be given in the third component of the input, and because of this we have the surprising result that someone who is under eighteen and has already voted, but asks for the voting results at the same time as trying to vote will be permitted, and so can cast any number of ballots. To fix this, we could insist upon a new condition, that $\exists!\, x \in o$; or we could use $\otimes$ instead of $\oplus$, which would ensure that only one of the sub-policies could be definitive on any given point (and so turn $\mathsf{eff}(e, p_v)$ into an $Indet$ result instead of a $Permit$); or we could decide that only people who have voted already can check the results.

4. AUTOMATICALLY PROVING PROPERTIES OF POLICIES

Given the formal model defined in Section 2.1 and properties defined in Section we would like to check properties of access policies automatically. To do this we first formalize the syntax of formulas we use to specify subsets of E. Then we discuss how policies constructed using these formulas and policy combinators can be translated to the Alloy language. One can check properties of access policies by translating them to SAT problems. Instead of targeting a SAT problem directly, our translator targets the Alloy language, which is in turn translated into a SAT problem by the Alloy Analyzer. This approach simplifies our translator and permits us to remain closer to the syntax of the original problem statement.

4.1 Policy Classes

In Section 2.1, we defined our formal model using subsets of E for the definition of $Scope$ and Err. Since E is the Cartesian product of the power sets of what are in most cases infinite sets, analyzing these properties would seem to require enumerating these subsets, an impossible job. Fortunately, we are translating from XACML to our foreign model, and totally arbitrary subsets are not possible in XACML. We can structure the subsets we will get from XACML policies in a more useful fashion: they end up looking like the sets $S_0, S_1, \ldots, S_4$ in Equation (1).

To formalize this, we fix the following notational conveniences: for elements $e \in E$, we name the components of e to be $e[0], e[1], \ldots, e[n]$. We use $s, s_0, s_1, \ldots, s_n$ to denote set variables, $c, c_0, c_1, \ldots, c_n$ to denote scalar variables, and $C, C_0, C_1, \ldots, C_n$ to denote constants. Finally, BP is a set of basic predicates which we will define below. We fix our structure in the following fashion: all subsets of E are specified in the form $\{e \in E : P\}$, where there are no free variables save e in P and P is defined as follows:

$$\begin{array}{rcl} P & \rightarrow & BP \ \mid\ \forall c \in s\ P \ \mid\ \exists c \in s\ P \\ & \mid & \exists!\, c \in s\ P \ \mid\ \exists!\, c \in e[i]\ P \\ & \mid & P \wedge P \ \mid\ P \vee P \ \mid\ \neg P \end{array}$$

Below, we will define four different basic predicate sets with increasing expressive power, such that $BP_1 \subseteq BP_2 \subseteq BP_3 \subseteq BP_4$. Which version of BP is used is important for translation. For example, if a policy can be written using only predicates from BP_1 then we can make certain guarantees about its translation that may not hold if we must simulate predicates from a larger BP.

Our first class is that of policies using only enumerated types (which obviously have finite domains) and the simple operations $\neg, =, \in, \subseteq$. We define the first class of basic predicates, BP_1, as follows:

$$\begin{array}{rcl} SCAL & \rightarrow & c \ \mid\ C \\ BSET & \rightarrow & s \ \mid\ e[i] \\ SET & \rightarrow & SET \cup SET \ \mid\ SET \cap SET \\ & \mid & SET \setminus SET \ \mid\ \{SCAL\} \ \mid\ BSET \\ BP_1 & \rightarrow & SET \subseteq SET \ \mid\ SCAL \in SET \\ & \mid & SCAL = SCAL \ \mid\ \texttt{true} \ \mid\ \texttt{false} \end{array}$$

We can express all set definitions on unordered and enumerated types that are permitted in XACML using the expressions above.

Given a set in the form $S = \{e \in E : P\}$ where P is defined based on the above syntax, one can generate a boolean logic formula B which encodes the set S. The encoding will map each $e \in E$ to a valuation of the boolean variables in B and B will evaluate to true if and only if $e \in S$. Based on such an encoding we can convert questions about different policies (such as if one subsumes the other one) to SAT problems and then use a SAT solver to check them. For example, we can generate a boolean formula which is satisfiable if and only if an access policy is subsumed (i.e., $\sqsubseteq$) by another one. If the SAT solver returns a satisfying assignment to the formula, then we can conclude that the property is false, and generate a counterexample based on the satisfying assignment. If the SAT solver declares that the formula is not satisfiable then we can conclude that the property holds. Details of such a translation for the Alloy language is given in [14], and as we will show below the policies specified with the syntax described above can be translated to Alloy language.

The second class of basic predicates extends the first one to handle types which have a total order relation $<$, as well.

We define BP_2 as follows:

$$BP_2 \;\rightarrow\; BP_1 \;\;|\;\; SCAL < SCAL$$

Sets described using this class of predicates can also be translated to a boolean logic formula. We can encode a type with a domain of n ordered elements using n^2 boolean variables, one for each pair of values in the domain.

The third class of policies extends the second class to include infinite domains. Although the syntax for BP_2 and BP_3 are the same for BP_3 we permit E to be composed of power sets of infinite domains. Note that we cannot translate sets described using such predicates directly to boolean logic formulas. So instead we limit the scope of our investigations: we artificially limit the size of the set to a given fixed size and then perform analysis upon it as though it were a finite enumerated set of that size. The problem is that if no counterexample is found, then that does not necessarily mean that no counterexample exists—perhaps if we had increased the scope just a little more we would have found one. The small scope hypothesis (discussed in [15], and tested and confirmed for some data structure algorithms in [21]) suggests that small scopes could be sufficient in practice. Note that if a counterexample is found, that counterexample is definite and can be translated into an error in the original policy.

The fourth and final class of policies extends the third class to handle arbitrary Boolean functions, with any scalars or sets as arguments. We also handle one special function on sets, that being the magnitude operation.

$$BP_4 \;\rightarrow\; BP_3 \;\;|\;\; f(SET, \ldots, SCAL, \ldots) \;\;|\;\; |SET|$$

To translate sets described using such predicates to boolean logic formulas we use uninterpreted functions, i.e., we create a Boolean variable for encoding the value of a boolean function and we create a (bounded) integer variable for encoding the size of set. We generate constraints which guarantee that the value of the function is the same if its arguments are the same. Other than this restriction the variables encoding the functions can get arbitrary values. Note that this brings an extra level of imprecision to our analysis. We were not able to trust the positive results because of the scope restriction, but now it is also possible that counterexamples may be spurious, and will need to be validated against the original policy. However, we think that such automated analysis can still be useful in uncovering errors in access policies.

The above suffices for modeling every function in core XACML, with the exception of the higher order functions. Those functions invariably operate on sets, complicate the analysis, and—if these functions are restricted to using the predicates that we can model directly—can ultimately be expressed using the predicates we can already handle.

In the above we defined the syntax for four policy classes and argued that properties of policies described with the above syntax can be translated to SAT problems. Our analysis tool instead targets the modeling language Alloy [16, 13, 14]. Alloy permits a more literal translation of our model, simplifying the translation tremendously. After introducing Alloy briefly, we will show how we translate our four classes of policies to Alloy.

4.2 Alloy

Alloy is a declarative modeling language out of MIT equipped with an analyzer that can verify assertions about models written in the language. Alloy analyzer achieves this by converting assertions to Boolean logic formulas which are fed to a SAT solver. Alloy is based in first order relational logic, and is intended to model complex structures. It does so through extensive set manipulation, and this manipulation permits an easy translation from our mathematical model. Alloy has been used to automatically extract object models [17, 31], to analyze the behavior of filesystem synchronization utilities [24], to model virtual functions [22] and to automatically check structural properties of data on the heap [30].

Alloy models consist of sets of concrete objects, called *signatures*, facts about these sets, and relations on these sets. Distinguished subsets of signatures are possible; these new signatures are said to *extend* the superset. Unlike some other modeling languages, Alloy does not require that these relations be completely specified. After defining signatures and facts about them, one can ask Alloy to verify that certain properties hold in all possible models that conform to the facts given, or that there exists a model capable of satisfying all the facts given. Alloy cannot, in general, prove assertions about all possible models; it can, however, prove assertions for all models within a fixed scope, which is what we have to settle for analyzing access policies in general as well.

One oddity about Alloy is that it unifies singleton sets and scalars; this is done for technical reasons, but it has some implications for our translation that will be discussed in the next section as they arise.

4.3 Translation to Alloy

The general structure we will be using here is as follows: to prove that $P_1 \sqsubseteq P_2$, we need to prove that each individual component of P_1 is a subset of each individual component of P_2. This part of the generated Alloy code is as follows:

```
static sig P1 extends Triple {} {
    ...
}
static sig P2 extends Triple {} {
    ...
}
assert Subset {
    P1.permit in P2.permit
    P1.deny in P2.deny
    P1.error in P2.error
}
```

That is, we define two models `P1` and `P2`, and then check that the components of the one are contained in the other. Since Alloy unifies sets and singletons, `in` can do double duty as set membership and subset testing. Similarly we do not need to specifically handle the conversion of scalar variables to singleton sets.

We can translate P_1 and P_2 in our mathematical model in the following manner. First, we distinguish predicates outside of existential and universal quantifiers from predicates

$$\text{translate}(\langle r,s,t\rangle) \Rightarrow \begin{array}{r}\texttt{permit} = \text{translate}_{P'}(r)\\ \texttt{deny} = \text{translate}_{P'}(s)\\ \texttt{error} = \text{translate}_{P'}(t)\end{array}$$

For predicates outside quantifier formulas
$\text{translate}_{P'}(P'_1 \cap P'_2) \Rightarrow \text{translate}_{P'}(P'_1)$ `&` $\text{translate}_{P'}(P'_2)$
$\text{translate}_{P'}(P'_1 \cup P'_2) \Rightarrow \text{translate}_{P'}(P'_1)$ `+` $\text{translate}_{P'}(P'_2)$
$\text{translate}_{P'}(\neg P'_1) \Rightarrow$ `E -` $\text{translate}_{P'}(P'_1)$
$\text{translate}_{P'}(\forall c \in s\ P'') \Rightarrow \text{extract}($`all` $c\colon \text{translate}_{P''}(s)$
`|` $\text{translate}_{P''}(P''))$
$\text{translate}_{P'}(\exists c \in s\ P'') \Rightarrow \text{extract}($`some` $c\colon \text{translate}_{P''}(s)$
`|` $\text{translate}_{P''}(P''))$
$\text{translate}_{P'}(\exists!\, c \in s\ P'') \Rightarrow \text{extract}($`one` $c\colon \text{translate}_{P''}(s)$
`|` $\text{translate}_{P''}(P''))$
$\text{translate}_{P'}(BP) \Rightarrow \text{translate}_{BP}(BP)$

For predicates inside quantifier formulas
$\text{translate}_{P''}(P''_1 \cap P''_2) \Rightarrow \text{translate}_{P''}(P''_1)$ `&&` $\text{translate}_{P''}(P''_2)$
$\text{translate}_{P''}(P''_1 \cup P''_2) \Rightarrow \text{translate}_{P''}(P''_1)$ `||` $\text{translate}_{P''}(P''_2)$
$\text{translate}_{P''}(\neg P''_1) \Rightarrow$ `!` $\text{translate}_{P''}(P''_1)$
$\text{translate}_{P''}(\forall c \in s\ P'') \Rightarrow \text{extract}($`all` $c\colon \text{translate}_{P''}(s)$
`|` $\text{translate}_{P''}(P''))$
$\text{translate}_{P''}(\exists c \in s\ P'') \Rightarrow \text{extract}($`some` $c\colon \text{translate}_{P''}(s)$
`|` $\text{translate}_{P''}(P''))$
$\text{translate}_{P''}(\exists!\, c \in s\ P'') \Rightarrow \text{extract}($`one` $c\colon \text{translate}_{P''}(s)$
`|` $\text{translate}_{P''}(P''))$
$\text{translate}_{P''}(BP) \Rightarrow \text{translate}_{BP}(BP)$

Figure 4: Basic translation rules

inside: that is, we split P into P' and P'' as follows:

$$\begin{array}{rcl} P' & \rightarrow & BP \ \mid\ \forall c \in s\ P'' \ \mid\ \exists c \in s\ P''\\ & \mid & \exists!\, c \in s\ P'' \ \mid\ \exists!\, c \in e[i]\ P''\\ & \mid & P' \wedge P' \ \mid\ P' \vee P' \ \mid\ \neg P''\\ P'' & \rightarrow & BP \ \mid\ \forall c \in s\ P'' \ \mid\ \exists c \in s\ P''\\ & \mid & \exists!\, c \in s\ P'' \ \mid\ \exists!\, c \in e[i]\ P''\\ & \mid & P'' \wedge P'' \ \mid\ P'' \vee P'' \ \mid\ \neg P''\end{array}$$

We translate tuples according to the rules in Figure 4, and some of these tuples will create auxiliary sets much like the sets S_0, S_1, S_2, S_3 (Equations (2) to (5)) we used in Section 2.1. The function extract defines a new subset of the environment `E` based on its argument (which is a formula) and then returns the name of this subset. So extract(e) would return $\texttt{S}_i$ and generate the following definition: `sig` $\texttt{S}_i$ `extends E {} {` e `}`.

Now, we just need to know how to translate the BP hierarchy to Alloy. Translating the various sorts of basic predicates in BP_1 is mostly straightforward but with minor complications: Alloy equates scalar quantities and sets with only one element. So, $\subseteq$ is the same operation as $\in$. To create constant elements, we create a field in a structure composed entirely of static constants which looks like:

```
static sig CONSTANTS {
    x1 : scalar Integer,
    x2 : scalar Integer,
    ...
}
```

We use constant(C) to describe the operation that inserts the constant C into this table if it is not already present, and returns a name to refer to it (for example, `CONSTANTS.x1` for the first field). Since the Boolean constants `True` and `False` are already defined in Alloy we do not need to go through this operation for Boolean constants, and can translate those directly.

The $e[i]$'s are represented as fields of the `E` structure which is declared as:

```
sig E {
    age : set Integer,
    voted : set Bool,
    actions : set String
}
```

for the environment E defined in Equation (2) for our running example. We use env($e[i]$) to give the translation for the environmental set $e[i]$. For the running example, env($e[0]$) = `age` and env($e[2]$) = `actions`.

The total translation for BP_1 can be done as follows:

$$\begin{array}{rl} \text{translate}_{BP}(s) & \Rightarrow s\\ \text{translate}_{BP}(c) & \Rightarrow c\\ \text{translate}_{BP}(e[i]) & \Rightarrow \text{env}(e[i])\\ \text{translate}_{BP}(C) & \Rightarrow \text{constant}(C)\\ \text{translate}_{BP}(s_i \cup s_j) & \Rightarrow \text{translate}(s_i)\ \texttt{+}\ \text{translate}_{BP}(s_j)\\ \text{translate}_{BP}(s_i \cap s_j) & \Rightarrow \text{translate}(s_i)\ \texttt{\&}\ \text{translate}_{BP}(s_j)\\ \text{translate}_{BP}(s_i \setminus s_j) & \Rightarrow \text{translate}(s_i)\ \texttt{-}\ \text{translate}_{BP}(s_j)\\ \text{translate}_{BP}(\{c\}) & \Rightarrow \text{translate}(c)\\ \text{translate}_{BP}(c_i = c_j) & \Rightarrow \text{translate}(c_i)\ \texttt{=}\ \text{translate}_{BP}(c_j)\\ \text{translate}_{BP}(c \in s) & \Rightarrow \text{translate}(c)\ \texttt{in}\ \text{translate}_{BP}(s)\\ \text{translate}_{BP}(s_i \subseteq s_j) & \Rightarrow \text{translate}(s_i)\ \texttt{in}\ \text{translate}_{BP}(s_j)\end{array}$$

Using all of this, we can show as an example, a translation of S_1 in Equation (2) into Alloy which results in:

```
sig S1 extends E {} { all x : voted | x = False }
```

For policies of the second class, we add the predicate <. To accommodate this, we define an Alloy function `LessThan` and enforce its transitivity as follows:

```
fact {
    all a,b,c:Type {
        LessThan (a, b) = True &&
        LessThan (b, c) = True =>
            LessThan (a, c) = True
    }
}
```

Now we can simply translate

$$\text{translate}(a < b) \Rightarrow \\ \texttt{LessThan}(\text{translate}(a), \text{translate}(b))\ \texttt{= True}$$

As an example, we translate S_0 in Equation (1) which looks like:

```
static sig CONSTANTS {
    x1 : scalar Integer
}
sig S1 extends E {} { all x : age |
    LessThan (a, CONSTANTS.x1) = True }
```

Policies of the third class require no special translation; they merely require that Alloy be informed of the scope requirements when it attempts to analyze the policy.

Policies of the fourth class are accommodated by defining a new Alloy relation about which we specify nothing. For example, suppose we wanted to analyze a policy involving XACML's embedded XPath matching. Since we will encode this as an uninterpreted function all we need to know about XPath matching is that it returns a Boolean value. We define a set S_6 as follows:

$$S_6 = \{\langle a, v, o\rangle \in E : \texttt{xpathnodematch}(\texttt{/actions}, o)\}$$

where `/actions` is an XPath expression. We translate this by first introducing a new function as follows:

```
static sig Functions {
    expr1 : E -> Bool // xpathnodematch(/actions, o)
}
```

and we use it in a new subset of `E` as follows:

```
sig S6 extends E {} { this.(Functions.expr1) = True }
```

We make no other claims about `Functions.expr1`, and as a result `S6` represents an arbitrary subset of `E`.

5. EXPERIMENTS

Our tool generates Alloy code which is then run through the Alloy Analyzer to do the analysis. It is targeted for proving things about $\sqsubseteq$ relations, but we can use it for simpler questions as well.

One such question might be 'give a tuple e such that $\text{eff}(e, p) = \textit{Permit}$' (where p is defined as in our running example). We generate the Alloy code for the XACML policy, as normal, and then append the following:

```
fun CheckTuple {
    some T0.permit
}
run CheckTuple for 2 but 2 Bool, 1 Triple, 8 Type
```

The numbers after `CheckTuple` establish how deeply we will look; we're looking for such a tuple in a universe where there are two Boolean values, one Triple (`T0`, the one generated through translate(p), 8 domain types (strings, integers, and the like)), and two elements of everything else. If we run this, the analyzer tells us that the tuple $\langle\{\texttt{Type\_6}\}, \{\texttt{Bool\_1}\}, \{\texttt{Type\_7}\}\rangle$ generates a *Permit*, and further examination of the output shows that `Bool_1` is `True`, `Type_6` is 18, and `Type_7` is the string `vote`. Through similar means we can discover that the tuple $\langle\{18\}, \{\}, \{\texttt{vote}\}\rangle$ will generate an error.

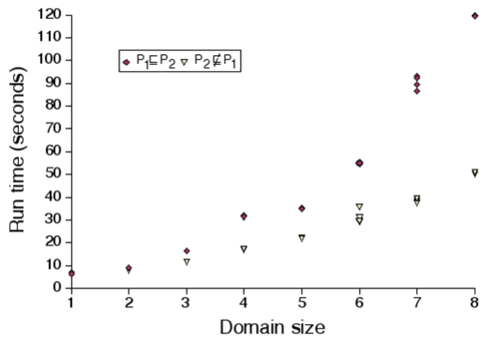

Figure 5: Run time vs. domain size plotted for P_1 and P_2

Domain size	Comparison	
	$P_3 \sqsubseteq P_4$	$P_4 \not\sqsubseteq P_3$
1	10.0 s	12.2 s
2	21.7 s	42.0 s
3	35.5 s	
4	63.6 s	

Table 1: Median run time of Medico example (P_4) and subset (P_3) in seconds

For something more interesting, we try to show that $p_v \sqsubseteq_D p_c$, as we proved manually. Our coda now becomes

```
assert Subset {
    T0.deny in T1.deny
}
check Subset for 2 but 2 Bool, 2 Triple, 10 Type
```

We get a counterexample almost immediately, giving us the tuple we constructed in Section 4. If we modify the policy so as to restrict result checking to only those who voted successfully, then the subset relation holds, and no counterexample is produced.

5.1 Timing Data

These are but small examples, so they do not reflect the time required to solve large problems that one might reasonably ask. Since the underlying problem (SAT) is NP complete, it is reasonable to ask whether these techniques are useful at all as the problem size increases.

There are two sides to this. One side is that larger and more complex policies will inescapably take longer than small policies. The other is that—in the context of problems in BP_3 and BP_4, where we must restrain the domains to a certain finite size—the amount of computation involved as the size of the domain increases may trigger the exponential worst case behavior.

To demonstrate that the analysis is still feasible, we have collected two sets of data. The first, which we have charted in Figure 5, shows the time required to verify or refute a relationship between two example policies. It shows how the technique scales as the size of the domain sets increases. The second set of data proves and refutes a relationship between the Medico policy from Section 4.2 of the XACML specification [32] and a subset of itself. This shows that the technique is feasible for larger policies. All these benchmarks were performed on a 1 GHz PowerPC, and all times are the median of five runs, to smooth out irregularities. In each case, trials were run until the formula proved too large for Alloy Analyzer to handle; past the given sizes the analyzer would fail cryptically. "Domain size" means the number of elements we are using for our analysis, in each domain; so a domain size of 8 for our example environment E means we simulate every $e \in E$ where $|e[i]| \leq 8$ for each component in E.

The data indicates that the time required for analysis is exponential in the size of the scope, which is to be expected for a SAT based algorithm. However, all times are under two minutes, and our technique can clearly prove important properties of these problems in a useful amount of time.

6. CONCLUSION

We have presented a formal model for access policies, and shown how to analyze interesting properties about such models in an automated way. We have implemented a tool to translate XACML policies into this model and to yield code suitable for analysis. The experimental results indicate that automated analysis of nontrivial access policies is feasible.

It would be interesting to investigate using predicate abstraction to generate more precise models for the functions that we cannot directly simulate. Translating directly to a SAT solver may be more efficient than going through Alloy, but we would have to try it to be sure. We would also like to experiment on more and larger policies.

7. REFERENCES

[1] J. L. Abad-Peiro, H. Debar, T. Schweinberger, and P. Trommler. PLAS — Policy language for authorizations. Technical Report RZ 3126, IBM Research Division, 1999.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

[4] P. Bonatti, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. An access control model for data archives. In *Proceedings of the 16th international conference on information security: trusted information*, pages 261–276, Paris, France, 2001. Kluwer International Federation For Information Processing Series.

[5] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information Systems Security*, 5(1):1–35, 2002.

[6] E. Damiani, S. De Capitani di Vimercati, E. Fernández-Medina, and P. Samarati. Access control of SVG documents. In *Proceedings of DBSec 2002*, pages 219–230, 2002.

[7] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. *Computer Networks*, 33(1–6):59–75, 2000.

[8] 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 Systems Security*, 5(2):169–202, 2002.

[9] E. Damiani, P. Samarati, S. De Capitani di Vimercati, and S. Paraboschi. Controlling access to XML documents. *IEEE Internet Computing*, 5(6):18–28, 2001.

[10] S. De Capitani di Vimercati and P. Samarati. Access control in federated systems. In *Proceedings of the 1996 workshop on new security paradigms*, pages 87–99, Lake Arrowhead, California, United States, 1996. ACM Press.

[11] M. Heckman and K. N. Levitt. Applying the composition principle to verify a hierarchy of security servers. In *HICSS (3)*, pages 338–347, 1998.

[12] D. Jackson. Micromodels of software: Modeling and analysis with Alloy. `http://sdg.lcs.mit.edu/alloy/reference-manual.pdf`.

[13] D. Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT COnf. Foundations of Software Engineering*, November 2000.

[14] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7), July 1996.

[15] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of International Conference on Software Engineering*, Limerick, Ireland, June 2000. IEEE.

[16] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *IEEE Transactions on Software Engineering*, February 2001.

[17] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[18] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, USA, 1997. IEEE Press.

[19] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD'97*, pages 474–485, Tucson, AZ, May 1997.

[20] D. Marinov, A. Andoni, D. Danilinc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.

[21] D. Marinov and S. Khurshid. VAlloy: Virtual functions meet a relational language. In *11th International Symposium of Formal Methods Europe (FME 2002)*, Copenhagen, Denmark, July 2002.

[22] G. Naumovich. A conservative algorithm for computing the flow of permissions in Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 33–43, July 2002.

[23] T. Nolte. Exploring filesystem synchronization with lightweight modeling and analysis. Master's thesis, MIT, August 2002.

[24] P. Samarati and S. De Capitani di Vimercati. *Foundations of Security Analysis and Design*, chapter 3, pages 137–196. Springer Verlag, 2001.

[25] R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys*, 28(1):241–243, 1996.

[26] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[27] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[28] R. S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994 1994.

[29] M. Vaziri and D. Jackson. Checking heap-manipulating procedures with a constraint solver. In *TACAS'03*, Warsaw, Poland, 2003.

[30] A. Waingold. Automated extraction of abstract object models. Master's thesis, MIT, May 2001.

[31] eXtensible Access Control Markup Language (XACML) version 1.0. OASIS Standard, February 2003.

[32] Extensible markup language (XML) 1.0 (second edition), 2000.

[33] XML Schema part 2: Datatypes. W3C Recommendation, May 2001.

APPENDIX

A. XACML REPRESENTATION OF VOTING EXAMPLE AND ALLOY TRANSLATION

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy
  xmlns="urn:..."
  xmlns:xsi="...-instance"
  xmlns:md="http://www.medico.com/schemas/record.xsd"
  PolicySetId="urn:example:policyid:1"
  RuleCombiningAlgId="urn:...:deny-overrides">
  <Target>
    <Subjects><AnySubject/></Subjects>
    <Resources><AnyResource/></Resources>
    <Actions>
      <Action>
        <ActionMatch MatchId="urn:...:string-equal">
          <AttributeValue DataType="...#string">
            vote
          </AttributeValue>
          <ActionAttributeDesignator
            AttributeId="urn:example:action"
            DataType="...#string"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
  <Rule RuleId="urn:example:ruleid:1" Effect="Deny">
    <Condition FunctionId="urn:...:integer-less-than">
      <Apply FunctionId="urn:...:integer-one-and-only">
        <SubjectAttributeDesignator
          AttributeId="urn:example:age"
          DataType="...#integer"/>
      </Apply>
      <AttributeValue DataType="...#integer">
        18
      </AttributeValue>
    </Condition>
  </Rule>
  <Rule RuleId="urn:example:ruleid:2" Effect="Deny">
    <Condition FunctionId="urn:...:boolean-equal">
      <Apply FunctionId="urn:...:boolean-one-and-only">
        <SubjectAttributeDesignator
          AttributeId="urn:example:voted-yet"
          DataType="...#boolean"/>
      </Apply>
      <AttributeValue DataType="...#boolean">
        True
      </AttributeValue>
    </Condition>
  </Rule>
  <Rule RuleId="urn:example:ruleid:3" Effect="Permit"/>
</Policy>
```

B. ALLOY TRANSLATION

```
module foo
open std/bool
sig Triple {
  permit : set E,
  deny : set E,
  error : set E
}
sig Type {}
disj sig Integer extends Type {}
disj sig String extends Type {}
static sig E {
  env2 : set String, // urn:example:action
  env0 : set Bool, // ...::urn:example:voted-yet
  env1 : set Integer // ...::urn:example:age
}
static sig S {
  static2 : scalar String, // vote
  static0 : scalar Bool, // True
  static1 : scalar Integer // 18
}
static sig Functions {
  lessthan: Type -> Type -> Bool
}
det fun LessThan (a, b: Type) : scalar Bool {
  result = if a = b then False else b.(a.(Functions.lessthan))
}
det fun LessEqual (a, b: Type) : scalar Bool {
  result = if LessThan (a, b) = True || a = b then True else False
}
det fun GreaterThan (a, b: Type) : scalar Bool {
  LessThan (b, a) = result
}
det fun GreaterEqual (a, b: Type) : scalar Bool {
  LessEqual (b, a) = result
}
fact {
  all a,b,c:Type {
        LessThan (a, b) = True && LessThan (b, c) = True => LessThan
        (a, c) = True
  }
}
sig S0 extends E {} { S.static2 in env2 }
sig S1 extends E {} { env0 = S.static0 }
sig S2 extends E {} { one env0 }
sig S3 extends E {} { LessThan (env1, S.static1) = True }
sig S4 extends E {} { one env1 }
static sig T0 extends Triple {} {
  permit = S0 & (E - (S1 & S0 & (E - (E - S2) & S0) + (E - S2) &
  S0)) & (E - (S3 & S0 & (E - (E - S4) & S0) + (E - S4) & S0))
  deny = (S3 & S0 & (E - (E - S4) & S0) + S1 &
  S0 & (E - (E - S2) & S0) & (E - (E - S2) & S0)) &
  (E - ((E - S4) & S0 + (E - S2) & S0))
  error = (E - S4) & S0 + (E - S2) & S0
}
```