Automated Verification of XACML Policies Using a SAT Solver*

Graham Hughes and Tevfik Bultan

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

Abstract. Web-based software systems are increasingly used for accessing and manipulating sensitive information. Managing access control policies in such 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 written in XACML. We introduce a formal model for XACML policies 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 a SAT solver. Our automated verification tool translates verification queries about XACML policies to a Boolean satisfiability problem. Our experimental results demonstrate that automated verification of XACML policies is feasible using our approach.

1 Introduction

Web-based applications today are used to access all types of sensitive information such as bank accounts, employee records and even health records. Given the ease of access provided by the Web, it is crucial to provide access control mechanisms for Web-based applications that deal with sensitive information. Moreover, due to increasing use of service oriented architectures, it is necessary to develop techniques for keeping the access control policies consistent across heterogeneous systems and applications spanning multiple organizations.

XACML (eXtensible Access Control Markup Language) [12] provides a common language for combining, maintaining and exchanging access control policies. XACML is an XML-based language for expressing access rights to arbitrary objects that are identified in XML. XACML provides rule and policy combining mechanisms for constructing policies from rules and metapolicies from policies, respectively.

Policies built using such mechanisms will inevitably become quite large and complex as they are used to combine access control rules and subpolicies in an organization and especially across organizations. It is possible, even likely, that the act of creating a metapolicy 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.

* This work is supported by NSF grants CCF-0341365 and CCF-0614002.

We translate XACML policies into a mathematical model, which we reduce to a normal form by 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 Boolean formulas which are satisfiable if and only if the corresponding relation is violated. We use a SAT solver to check satisfiability of these Boolean logic formulas. Using our translator and a SAT solver 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, after giving an overview of XACML, we develop a formal model for access control policies written in XACML and discuss how to transform these models into a normal form that distinguishes access permitted, access denied, and error conditions. In Section 3 we define partial ordering relations among access control policies which are used to specify their properties. We show how to check these properties automatically in Section 4. Finally, we report the results of our experiments in Section 5 and give our conclusions in Section 7.

2 Policy Specifications

An *access request* is a specially formatted XML document that defines a set of data that we call the *environment*. Given an environment, an XACML policy specification yields one of four results: Permit (*Per*), meaning that the access request is permitted; Deny (*Den*), meaning that the access request will not be permitted; Not Applicable (*NoA*), meaning that this particular policy says nothing about the request; and Indeterminate (*Ind*), which means that something unexpected came up and the policy has failed. XACML additionally defines *obligations*, which are actions that the policy must perform in some circumstances; we do not handle obligations in this work.

In XACML three classes of objects are used to specify access control policies: 1) individual rules, 2) collections of rules called policies, and 3) collections of policies called policy sets. 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 can be constructed using primitive functions such as equality, set inclusion, and ordering within numeric types, and also more complex functions such as XPath matching and X500 name matching.

Let us consider a simple example policy for an online voting system. The policy states that to be able to vote a person must be at least 18 years old and a person who has voted already cannot vote. Our environment (i.e., 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 [13] basic types,

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Policy xmlns="urn:..." xmlns:xsi="...-instance"
3    xmlns:md="http:.../record.xsd" PolicySetId="urn:example:policyid:1"
4    RuleCombiningAlgId="urn:...:deny-overrides">
5    <Target>
6      <Subjects><AnySubject/></Subjects>
7      <Resources><AnyResource/></Resources>
8      <Actions>
9        <Action>
10         <ActionMatch MatchId="urn:...:string-equal">
11           <AttributeValue DataType="...#string">vote</AttributeValue>
12           <ActionAttributeDesignator AttributeId="urn:example:action"
13                                      DataType="...#string"/>
14         </ActionMatch>
15       </Action>
16     </Actions>
17   </Target>
18   <Rule RuleId="urn:example:ruleid:1" Effect="Deny">
19     <Condition FunctionId="urn:...:integer-less-than">
20       <Apply FunctionId="urn:...:integer-one-and-only">
21         <SubjectAttributeDesignator AttributeId="urn:example:age"
22                                     DataType="...#integer"/>
23       </Apply>
24       <AttributeValue DataType="...#integer">18</AttributeValue>
25     </Condition>
26   </Rule>
27   <Rule RuleId="urn:example:ruleid:2" Effect="Deny">
28     <Condition FunctionId="urn:...:boolean-equal">
29       <Apply FunctionId="urn:...:boolean-one-and-only">
30         <SubjectAttributeDesignator AttributeId="urn:example:voted-yet"
31                                     DataType="...#boolean"/>
32       </Apply>
33       <AttributeValue DataType="...#boolean">True</AttributeValue>
34     </Condition>
35   </Rule>
36   <Rule RuleId="urn:example:ruleid:3" Effect="Permit"/>
37 </Policy>
```

Fig. 1. A simple XACML policy

as follows:

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

Here, E denotes the set of all possible environments. The first component of an environment 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 XACML policy for this example is shown in Figure 1. We will explain the semantics of this policy using a simple mathematical notation. We write all environment sets in the form $\{e \in E : C\}$ where C is a predicate whose only free variables are the components of the environment tuple e. Since we do not at this time constrain the predicate C, this does not cost us any generality.

The goal for our example policy is that if a person is 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):

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

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*).

A Formal Model for XACML Policies: Let $R = \{\mathit{Per}, \mathit{Den}, \mathit{NoA}, \mathit{Ind}\}$ be the set of valid results permit, deny, not applicable and indeterminate, respectively. We define the set of valid policies P as follows (semantics will be defined below):

$$\mathit{Per} \in P,\ \mathit{Den} \in P$$
$$\forall p \in P :\ \forall S \subseteq E :\ \ \mathit{Sco}(p, S) \in P \wedge \mathit{Err}(p, S) \in P$$
$$\forall p, q \in P :\ p \oplus q \in P \wedge p \ominus q \in P \wedge p \otimes q \in P \wedge p \oslash q \in P$$

Informally, we regard *Per* and *Den* as basic policies that ignore the environment and always yield *Per* or *Den*, respectively. Along these same lines, *Sco* and *Err* attach conditions to policies depending on the environment they are evaluated in: $\mathit{Sco}(p, S)$ yields p's answer if the current environment is in S, or *NoA* otherwise (i.e., *Sco* is used to define the scope of a policy); $\mathit{Err}(p, S)$ yields *Ind* if the current environment is in S or p's answer otherwise (i.e., *Err* is used to define the error conditions for a policy). The other four symbols ($\oplus, \ominus, \otimes, \oslash$) are combinators, that combine two policies as:

- **Permit-overrides:** $p \oplus q$ always yields *Per* if either p or q yield *Per*.
- **Deny-overrides:** $p \ominus q$ always yields *Den* if either p or q yield *Den*.
- **Only-one-applicable:** $p \otimes q$ requires that one of p or q yield *NoA* and then yields the other half's answer.
- **First-applicable:** $p \oslash q$ yields p's answer unless that answer is *NoA*, in which case it yields q's answer.

Our $\otimes$ and $\oslash$ operators are exactly equivalent to the only-one-applicable and first-applicable rules in XACML. However, the $\oplus$ and $\ominus$ operators we use in this paper are slightly different than the permit-overrides and deny-overrides rules in XACML. In the cases where the sub-rules do not yield *Ind*, these operators are exactly equivalent to the corresponding XACML rules. For the remaining cases, the corresponding XACML rules can be mapped to our operators with some extra work.

We formalize the semantics of these combinators in Figure 2 by defining a function $\mathrm{eff} : E \times P \to R$ that, given an environment and a policy, produces a result.

Using this notation, we can now model the XACML policy given in Figure 1 as:

$$S_0 = \{\langle a, v, o\rangle \in E : \forall x \in a\ x < 18\} \tag{1}$$
$$S_1 = \{\langle a, v, o\rangle \in E : \forall x \in v\ x\} \tag{2}$$
$$S_2 = \{\langle a, v, o\rangle \in E : \exists x \in o\ x = \texttt{vote}\} \tag{3}$$
$$S_3 = \{\langle a, v, o\rangle \in E : \neg\exists!\, a_0 \in a\} \tag{4}$$
$$S_4 = \{\langle a, v, o\rangle \in E : \neg\exists!\, v_0 \in v\} \tag{5}$$
$$r_1 = \mathit{Err}(\mathit{Sco}(\mathit{Den}, S_0), S_3) \tag{6}$$
$$r_2 = \mathit{Err}(\mathit{Sco}(\mathit{Den}, S_1), S_4) \tag{7}$$
$$p = \mathit{Sco}(r_1 \ominus r_2 \ominus \mathit{Per}, S_2) \tag{8}$$

$$\mathrm{eff}(e, \mathit{Per}) = \mathit{Per} \quad \mathrm{eff}(e, \mathit{Den}) = \mathit{Den}$$

$$\mathrm{eff}(e, \mathit{Sco}(p, S)) = \begin{cases} \mathrm{eff}(e, p) & \text{if } e \in S \\ \mathit{NoA} & \text{otherwise} \end{cases}$$

$$\mathrm{eff}(e, \mathit{Err}(p, S)) = \begin{cases} \mathit{Ind} & \text{if } e \in S \\ \mathrm{eff}(e, p) & \text{otherwise} \end{cases}$$

$$\mathrm{eff}(e, p \oplus q) = \begin{cases} \mathit{Per} & \text{if } \mathrm{eff}(e, p) = \mathit{Per} \vee \mathrm{eff}(e, q) = \mathit{Per} \\ \mathit{Ind} & \text{if } (\mathrm{eff}(e, p) = \mathit{Ind} \wedge \mathrm{eff}(e, q) \neq \mathit{Per}) \vee (\mathrm{eff}(e, q) = \mathit{Ind} \wedge \mathrm{eff}(e, p) \neq \mathit{Per}) \\ \mathit{Den} & \text{if } (\mathrm{eff}(e, p) = \mathit{Den} \wedge \mathrm{eff}(e, q) \neq \mathit{Per} \wedge \mathrm{eff}(e, q) \neq \mathit{Ind}) \\ & \quad \vee (\mathrm{eff}(e, q) = \mathit{Den} \wedge \mathrm{eff}(e, p) \neq \mathit{Per} \wedge \mathrm{eff}(e, p) \neq \mathit{Ind}) \\ \mathit{NoA} & \text{otherwise} \end{cases}$$

$$\mathrm{eff}(e, p \ominus q) = \begin{cases} \mathit{Den} & \text{if } \mathrm{eff}(e, p) = \mathit{Den} \vee (\mathrm{eff}(e, q) = \mathit{Den} \\ \mathit{Ind} & \text{if } (\mathrm{eff}(e, p) = \mathit{Ind} \wedge \mathrm{eff}(e, q) \neq \mathit{Den}) \vee (\mathrm{eff}(e, q) = \mathit{Ind} \wedge \mathrm{eff}(e, p) \neq \mathit{Den}) \\ \mathit{Per} & \text{if } (\mathrm{eff}(e, p) = \mathit{Per} \wedge \mathrm{eff}(e, q) \neq \mathit{Den} \wedge \mathrm{eff}(e, q) \neq \mathit{Ind}) \\ & \quad \vee (\mathrm{eff}(e, q) = \mathit{Per} \wedge \mathrm{eff}(e, p) \neq \mathit{Den} \wedge \mathrm{eff}(e, p) \neq \mathit{Ind}) \\ \mathit{NoA} & \text{otherwise} \end{cases}$$

$$\mathrm{eff}(e, p \otimes q) = \begin{cases} \mathrm{eff}(e, p) & \text{if } \mathrm{eff}(e, q) = \mathit{NoA} \\ \mathrm{eff}(e, q) & \text{if } \mathrm{eff}(e, p) = \mathit{NoA} \\ \mathit{Ind} & \text{otherwise} \end{cases}$$

$$\mathrm{eff}(e, p \oslash q) = \begin{cases} \mathrm{eff}(e, p) & \text{if } \mathrm{eff}(e, p) \neq \mathit{NoA} \\ \mathrm{eff}(e, q) & \text{otherwise} \end{cases}$$

Fig. 2. Semantics of policies

where 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. Note that, r_1 corresponds to the XACML rule between lines 19-27 in Figure 1, r_2 corresponds to the XACML rule between lines 28-36, and p corresponds to the XACML policy between lines 2-38.

Policy Transformations: We convert the access control policies to an intermediate normal form before we verify them. This enables us to decouple the verification back-end of our tool from its front-end. This decoupling means that it is possible to support other access control languages in our verification framework as long as they can be translated to the same normal form.

We first define the equivalence between two policies:

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

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 *Per* result, the subset of E that will give a *Den* result, and the subset of E that will give an *Ind* result independently. We define the shorthand $\langle S, R, T \rangle$, where S, R and T are pairwise disjoint, as follows:

$$\langle S, R, T \rangle = \mathit{Err}(\mathit{Sco}(\mathit{Per}, S) \otimes \mathit{Sco}(\mathit{Den}, R), T)$$

Hence, $\langle S, R, T \rangle$ is simply a policy that yields *Per* for any environment in S, *Den* for any environment in R, *Ind* for any environment in T, and *NoA* for any remaining environment. We call this *triple notation* and refer to an $\langle S, R, T \rangle$ as a *triple*.

Now that we have a framework for transforming policies, we would like to transform an entire policy with *Sco*, *Err* and combinators alike into a single triple. For any policy

P a triple P_T that is equivalent to it can be written as:

$$P_T = \langle \{e \in E : \text{eff}(e, P) = \textit{Per}\}, \{e \in E : \text{eff}(e, P) = \textit{Den}\}, \{e \in E : \text{eff}(e, P) = \textit{Ind}\}\rangle.$$

However, this is not a constructive definition. In [4], we developed an eff-preserving transformation $\mathcal{T} : P \rightarrow \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E)$, such that given a policy p, $\mathcal{T}(p)$ returns a triple that is equivalent to p. Our transformation works in two stages. In the first stage, the input policy is transformed to a set of subpolicies in our triple notation combined with $\oplus, \ominus, \otimes$ and $\oslash$. In the second stage, the triples joined by combinators are transformed into a single triple. For example, applying $\mathcal{T}$ to the policy p defined in Equation (8) leads to the following:

$$\begin{aligned} p &= \textit{Sco}(\textit{Err}(\textit{Sco}(\textit{Den}, S_0), S_3) \ominus \textit{Err}(\textit{Sco}(\textit{Den}, S_1), S_4) \ominus \textit{Per}, S_2) \\ \mathcal{T}(p) &= \langle S_2 \setminus (S_0 \cup S_1 \cup S_3 \cup S_4), ((S_0 \setminus S_3) \cup (S_1 \setminus S_4)) \cap S_2, \\ &\quad ((S_3 \cup S_4) \setminus ((S_0 \setminus S_3) \cup (S_1 \setminus S_4))) \cap S_2\rangle \end{aligned}$$

3 Properties of Policies

In this section we will show that properties of policies can be expressed using 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:

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

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

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

$$S_5 = \{\langle a, v, o\rangle \in E : \exists x \in o\ x = \texttt{getresult}\}, \quad r_3 = \textit{Sco}(\textit{Err}(\textit{Per}, S_4), S_5)$$

where S_4 is defined in Equation (5). Now, we can create a composite policy $p_c = p \oplus r_3$, 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:

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

where S_0, S_1, S_2, S_3 and S_4 are from Equations (1) to (5).

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 property we wish to verify 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 *Per* 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 $\mathrm{eff}(e, p_v)$ into an *Ind* result instead of a *Per*); or we could decide that only people who have voted already can check the results.

4 Automated Verification

In this section we first formalize the syntax of formulas we use to specify sets of environments. Then we discuss how policies constructed using these formulas and policy combinators can be translated to Boolean logic formulas. After this translation we show that we can check properties of access control policies using a SAT solver.

In Section 2, we defined our formal model using subsets of the set of possible environments E. We showed that each policy can be expressed in triple form $P = \langle S, R, T \rangle$ where S, R, and T are subsets of E. We also declared that all these subsets of E are either of the form $\{e \in E : C\}$, or some combination of subsets of E using $\cup$, $\cap$ or $\setminus$. Since $\{e \in E : C_1\} \cup \{e \in E : C_2\} = \{e \in E : C_1 \vee C_2\}$ and similarly other set operations can also be expressed using logical connectives, we can regard all subsets of E as of the form $\{e \in E : C\}$.

Given a set S in the form $S = \{e \in E : C\}$, our goal is to 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 not subsumed (i.e., $\not\sqsubseteq$) 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. We will discuss the details of such a translation below.

1	$SC \rightarrow A$	$SC.f := SC.v[A] \wedge \bigwedge_{i=1,i\neq A}^{k}(\neg SC.v[i])$
2	$SC \rightarrow a$	$SC.f := \bigwedge_{i=1}^{k}(SC.v[i] \leftrightarrow a[i]) \wedge (\bigvee_{i=1}^{k} SC.v[i]) \wedge$ $\bigwedge_{i=1}^{k}(SC.v[i] \rightarrow \bigwedge_{j=1,j\neq i}^{k} \neg SC.v[j])$
3	$BS \rightarrow s$	$BS.f := \bigwedge_{i=1}^{k}(BS.v[i] \leftrightarrow s[i])$
4	$BS \rightarrow e[i]$	$BS.f := \bigwedge_{j=1}^{k}(BS.v[j] \leftrightarrow e[i][j])$
5	$SE \rightarrow \{SC\}$	$SE.f := SC.f \wedge \bigwedge_{i=1}^{k}(SE.v[i] \leftrightarrow SC.v[i])$
6	$SE \rightarrow BS$	$SE.f := BS.f \wedge \bigwedge_{i=1}^{k}(SE.v[i] \leftrightarrow BS.v[i])$
7	$SE \rightarrow SE_1 \cup SE_2$	$SE.f := SE_1.f \wedge SE_2.f \wedge \bigwedge_{i=1}^{k}(SE.v[i] \leftrightarrow (SE_1.v[i] \vee SE_2.v[i]))$
8	$SE \rightarrow SE_1 \cap SE_2$	$SE.f := SE_1.f \wedge SE_2.f \wedge \bigwedge_{i=1}^{k}(SE.v[i] \leftrightarrow (SE_1.v[i] \wedge SE_2.v[i]))$
9	$SE \rightarrow SE_1 \setminus SE_2$	$SE.f := SE_1.f \wedge SE_2.f \wedge \bigwedge_{i=1}^{k}(SE.v[i] \leftrightarrow (SE_1.v[i] \wedge \neg SE_2.v[i]))$
10	$BP \rightarrow \texttt{true}$	$BP.f := BP.b \leftrightarrow \texttt{true}$
11	$BP \rightarrow \texttt{false}$	$BP.f := BP.b \leftrightarrow \texttt{false}$
12	$BP \rightarrow SC_1 = SC_2$	$BP.f := SC_1.f \wedge SC_2.f \wedge (BP.b \leftrightarrow \bigwedge_{i=1}^{k}(SC_1.v[i] \leftrightarrow SC_2.v[i]))$
13	$BP \rightarrow SC \in SE$	$BP.f := SC.f \wedge SE.f \wedge (BP.b \leftrightarrow \bigwedge_{i=1}^{k}(SC.v[i] \rightarrow SE.v[i]))$
14	$BP \rightarrow SE_1 \subseteq SE_2$	$BP.f := SE_1.f \wedge SE_2.f \wedge (BP.b \leftrightarrow \bigwedge_{i=1}^{k}(SE_1.v[i] \rightarrow SE_2.v[i]))$
15	$C \rightarrow BP$	$C.f := BP.f \wedge (C.b \leftrightarrow BP.b)$
16	$C \rightarrow \neg C_1$	$C.f := C_1.f \wedge (C.b \leftrightarrow \neg C_1.b)$
17	$C \rightarrow C_1 \vee C_2$	$C.f := C_1.f \wedge C_2.f \wedge (C.b \leftrightarrow (C_1.b \vee C_2.b))$
18	$C \rightarrow C_1 \wedge C_2$	$C.f := C_1.f \wedge C_2.f \wedge (C.b \leftrightarrow (C_1.b \wedge C_2.b))$
19	$C \rightarrow \forall a \in BS\ C_1$	$C.f := BS.f \wedge C_1.f \wedge (\bigwedge_{i=1}^{k}(BS.v[i] \rightarrow (a[i] \wedge \bigwedge_{j=1,j\neq i}^{k} \neg a[j] \wedge C_1.b)))$
20	$C \rightarrow \exists a \in BS\ C_1$	$C.f := BS.f \wedge C_1.f \wedge (\bigvee_{i=1}^{k}(BS.v[i] \rightarrow (a[i] \wedge \bigwedge_{j=1,j\neq i}^{k} \neg a[j] \wedge C_1.b)))$
21	$C \rightarrow \exists!\, a \in BS\ C_1$	$C.f := BS.f \wedge C_1.f \wedge (\bigvee_{i=1}^{k}(BS.v[i] \rightarrow (a[i] \wedge \bigwedge_{j=1,j\neq i}^{k} \neg a[j] \wedge C_1.b)))$ $\wedge(\bigwedge_{i=1}^{k}((BS.v[i] \wedge a[i] \wedge \bigwedge_{j=1,j\neq i}^{k} \neg a[j] \wedge C_1.b)$ $\rightarrow \neg \bigvee_{l=1,l\neq i}^{k}(BS.v[l] \wedge a[l] \wedge \bigwedge_{j=1,j\neq l}^{k} \neg a[j] \wedge C_1.b)))$

Fig. 3. Translation of the basic predicates and the constraints to Boolean logic formulas.

For elements $e \in E$, we name the components of e $e[0], \ldots, e[n]$. We use s, $s_0, \ldots,$ s_n to denote set variables, a, $a_0, \ldots, a_n$ to denote scalar variables, and A, $A_0, \ldots, A_n$ to denote constants. BP is a set of basic predicates which we define as:

$$SC \rightarrow A \mid a \qquad BS \rightarrow s \mid e[i]$$
$$SE \rightarrow BS \mid \{SC\} \mid SE \cup SE \mid SE \cap SE \mid SE \setminus SE$$
$$BP \rightarrow \texttt{true} \mid \texttt{false} \mid SC = SC \mid SC \in SE \mid SE \subseteq SE$$

The above grammar is sufficient for specifying policies with finite domain types and the operations $\neg$, $=$, $\in$, $\subseteq$. We will discuss extension to other domains later in this section.

Assuming that all subsets of E are specified in the form $\{e \in E : C\}$, where there are no free variables save e in C, C is defined as follows:

$$C \rightarrow BP \mid C \wedge C \mid C \vee C \mid \neg C \mid \forall a \in BS\ C \mid \exists a \in BS\ C \mid \exists!\, a \in BS\ C$$

Recall that we use $\exists!$ to mean there exists exactly one instance that holds. We can express all set definitions on unordered and enumerated types that are permitted in XACML using the expressions above.

We will explain our translation of a constraint C defined by the above grammar to a Boolean logic formula using attribute grammars. We will first discuss the translation of the basic predicates BP. In order to simplify our presentation we will assume that domains of all scalar variables have the same size k. We will encode a set of values from any domain using a Boolean vector of size k. Given a Boolean vector v, we will denote its components as $v[1]$, $v[2]$, $\ldots$, $v[k]$ where $v[i] \leftrightarrow \texttt{true}$ means that element i is a member of the set represented by v whereas $v[i] \leftrightarrow \texttt{false}$ means that it is not. We encode a set variable s and each component of the environment tuple e using the

same encoding, i.e., as a vector of Boolean values. To simplify our presentation we also encode a scalar variable a as a set using a vector of Boolean values but restrict it to be a singleton set by making sure that at any time only one of the Boolean values in the vector can be true. In our actual implementation scalar variables are represented using $\log_2 k$ Boolean variables where k is the size of the domain.

The production rules 1 to 14 in Figure 3 show the attribute grammar for basic predicates. Each production rule has a corresponding semantic rule next to it. The semantic rules describe how to compute the attributes of the nonterminal on the left hand side of the production rule using the attributes of the terminals and nonterminals on the right hand side of the production rule. In the attribute grammar shown in Figure 3, the nonterminals SC, BS and SE have two attributes. One of them is a Boolean vector v denoting a set of values, and the other one is a Boolean logic formula f which accumulates the frame constraints.

Rule 1 in Figure 3 states that a scalar constant A is encoded as a singleton set that contains only A. This singleton set is represented as a Boolean vector v, such that $v[A]$ is set to true and all the rest of the elements of the vector are set to false. This condition is stored in the frame constraint f. Rule 2 states that a scalar variable is also encoded as a Boolean vector v. The frame constraint f makes sure that the elements of the Boolean vector v are same as the elements of the Boolean vector representing the scalar variable a and exactly one of the elements in a or v is set to true in any given time. Rules 3 and 4 show that the set variables (s) and components of the environment tuple ($e[i]$) are also encoded as Boolean vectors.

Rule 5 creates a singleton set from a scalar constant SC. However, since we encode scalar constants as singleton sets, this simply means that the Boolean vectors encoding the scalar constant ($SC.v$) and the set ($SE.v$) are equivalent and the frame constraint $SE.f$ expresses this constraint. Note that in the attribute grammar shown in Figure 3, the frame constraint of a nonterminal on the left hand side of a production is a conjunction of the frame constraints of the nonterminals on the right hand side of the production plus some other constraints that are added based on the production rule.

Rules 7, 8 and 9 encode the set operations: union, intersection and set difference. Each set operation on two set expressions SE_1 and SE_2 results in the creation of a new Boolean vector $SE.v$. The value of an element in $SE.v$ is defined based on the corresponding elements in $SE_1.v$ and $SE_2.v$. For example, for the union operation, $SE.v[i]$ is true if and only if $SE_1.v[i]$ is true or $SE_2.v[i]$ is true. The intersection and set difference are defined similarly.

The nonterminal BP corresponds to the basic predicates and it has two attributes. One of them is a boolean variable b representing the truth value of the predicate and the other one is a Boolean logic formula f that accumulates the frame constraints. Rules 10 and 11 create two basic predicates which have the truth value true and false, respectively. Rule 12 is a basic predicate that corresponds to an equality expression comparing two scalars. Since scalars are expressed as Boolean vectors, the Boolean variable encoding the truth value of the predicate is true if and only if all elements of the Boolean vectors encoding the two scalar values are the same. This constraint is added to the frame constraint of the basic predicate.

Rule 13 creates a basic predicate that corresponds to a membership expression testing membership of a scalar to a set expression. Rule 14 creates a basic predicate that corresponds to a subset expression testing if a set expression is subsumed by another set expression. Since we encode scalars as singleton sets, the frame constraints generated for rules 13 and 14 are very similar. They state that if a value is a member of the set on the left hand side, then it should also be a member of the set on the right hand side.

The production rules 15 to 21 in Figure 3 show the attribute grammar for the constraints. The nonterminal C has two attributes. One of them is a boolean variable b representing the truth value of the constraint, and the other one is a Boolean logic formula f that accumulates the frame constraints. Again, the frame constraint of a nonterminal on the left hand side of a production is a conjunction of the frame constraints of the nonterminals on the right hand side of the production plus some other constraints that are added based on the production rule.

Rule 15 is just a syntactic rule expressing that a constraint can be a basic predicate. Rule 16 defines the negation operation. As expected the frame constraint states that the value of the constraint on the left hand side of the production rule is the negation of the value of the constraint on the right hand side of the production rule. Rules 17 and 18 define the disjunction and conjunction operations. The frame constraints generated in Rules 17 and 18 state that the value of the constraint on the left hand side of the production rule is the disjunction or the conjunction of the values of the constraints on the right hand side of the production rule, respectively.

Rules 19, 20 and 21 deal with quantified constraints. In the rules 19, 20 and 21, a denotes a scalar variable which is quantified over a basic set expression BS which is either a set variable s or a component of the environment tuple $e[i]$. The quantified variable a can appear as a free variable in the constraint expression on the right hand side (C_1). Universal quantification is expressed as a conjunction which states that for all the members of the set s or $e[i]$, the constraint C_1 should evaluate to true. This is achieved by restricting the value of the scalar variable a to the value of a different member of the set for each conjunct. Existential quantification is expressed similarly as a disjunction by restricting the value of the scalar variable a to the value of a different member of the set for each disjunct.

Rule 21 is an existentially quantified constraint which evaluates to true if and only if the constraint C_1 evaluates to true for exactly one member of the set s or $e[i]$. This is expressed by first stating that there is at least one member of the set s or $e[i]$ for which the constraint C_1 evaluates to true (which is equivalent to existential quantification) and then adding an extra conjunction which states that the constraint C_1 does not evaluate to true for two different members of the set s or $e[i]$.

The translation we described above does not handle domain specific predicates, e.g., ordering relations on types such as integers. When we translate sets described using such predicates to boolean logic formulas, we represent them as uninterpreted Boolean functions. We create a Boolean variable for encoding the value of the uninterpreted boolean function and 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 introduces some

imprecision to our analysis. It is possible that counterexamples may be spurious, and will need to be validated against the original policy.

Note that, it is possible to fully interpret ordering relations in order to reduce the imprecision in the analysis. 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, representing the ordering relations. However, XACML uses many complex functions such as XPath matching and X500 name matching which can be lead to very complex formulas if one tries to fully interpret them in the Boolean logic translation. Hence, we believe that using uninterpreted functions for abstracting such complex functionality is a justified approach which enables us to handle a significant portion of the XACML language. Also, we would like to note that the imprecision caused by abstraction of such complex functions has not led to any spurious results in the experiments we performed so far.

Property Verification: As discussed in Section 3, we specify properties of policies using a set of partial ordering relations. These partial ordering relations can be used to state that a certain type of outcome for one policy subsumes the same type of outcome for another policy. In this section we will only focus on the $\sqsubseteq$ relation. Translation of properties specified using other relations are handled similarly.

Given a query like $P_1 \sqsubseteq P_2$, our goal is to generate a Boolean logic formula which is satisfiable if and only if $P_1 \not\sqsubseteq P_2$. As we discussed earlier our tool first translates policies P_1 and P_2 to triple form, such that $P_1 = \langle S_1, R_1, T_1 \rangle$ and $P_2 = \langle S_2, R_2, T_2 \rangle$ where each element of each triple is specified with a constraint expression as follows:

$$S_1 = \{e \in E : C_{S_1}\}, R_1 = \{e \in E : C_{R_1}\}, T_1 = \{e \in E : C_{T_1}\}$$
$$S_2 = \{e \in E : C_{S_2}\}, R_2 = \{e \in E : C_{R_2}\}, T_2 = \{e \in E : C_{T_2}\}$$

After translating policies P_1 and P_2 in to the triple form our translator generates boolean logic formulas for the constraints C_{S_1}, C_{R_1}, C_{T_1}, C_{S_2}, C_{R_2} and C_{T_2} based on the attribute grammar rules described in Figure 3. For example, after this translation the truth value of the constraint C_{S_1} is represented with the Boolean variable $C_{S_1}.b$ and the frame constraint $C_{S_1}.f$ states all the constraints on the Boolean variable $C_{S_1}.b$.

Recall that, given $P_1 = \langle S_1, R_1, T_1 \rangle$ and $P_2 = \langle S_2, R_2, T_2 \rangle$, $P_1 \sqsubseteq P_2$ holds if and only if $S_1 \subseteq S_2$ and $R_1 \subseteq R_2$ and $T_1 \subseteq T_2$. Based on this, we generate a formula F such that $F = \texttt{true}$ if and only if $P_1 \sqsubseteq P_2$ as follows:

$$F = (C_{S_1}.f \wedge C_{R_1}.f \wedge C_{T_1}.f \wedge C_{S_2}.f \wedge C_{R_2}.f \wedge C_{T_2}.f) \rightarrow$$
$$((C_{S_1}.b \rightarrow C_{S_2}.b) \wedge (C_{R_1}.b \rightarrow C_{R_2}.b) \wedge (C_{T_1}.b \rightarrow C_{T_2}.b))$$

Finally, we send the property $\neg F$ to the SAT solver. If the SAT solver returns a satisfying assignment for the Boolean variables encoding the environment tuple e (which are the only free variables in the formula $\neg F$), the satisfying assignment corresponds to a counter-example environment demonstrating how the property is violated. If the SAT solver states that $\neg F$ is not satisfiable, then we conclude that the property holds, i.e., $P_1 \sqsubseteq P_2$.

We could use this same translation to verify logical properties of a policy directly at the cost of introducing a new language that our users would be forced to learn. We feel that subsumption is sufficiently powerful and the advantages of using only one language are sufficiently compelling that we do not support this at this time.

Since the majority of the SAT solvers expect their input to be expressed in Conjunctive Normal Form (CNF), the last step in our translation (before we send the formula

Property	IO	Transform	Boolean	SAT	Lines of XACML	Variables	Clauses	Result
C1	1.85s	0.17s	1.35s	0.11s	13157	56	114	Property holds
C2	2.07s	0.19s	1.41s	0.39s	13175	42	83	Property holds
C3	1.88s	0.16s	1.36s	0.12s	13108	51	108	Property holds
C4	1.94s	0.17s	1.33s	0.11s	13103	52	106	Property holds
C5	1.82s	0.16s	2.00s	0.16s	13108	79	166	Property holds
C6	2.12s	0.15s	2.53s	0.15s	13150	89	190	Property fails
C7	2.56s	0.34s	3.70s	0.10s	13203	95	218	Property fails
C8	1.99s	0.18s	1.21s	0.11s	13101	42	83	Property fails
C9	1.92s	0.16s	1.49s	0.11s	13107	51	106	Property fails
C10	1.88s	0.19s	3.47s	0.11s	13107	108	250	Property fails
C11	1.89s	0.16s	5.18s	0.15s	13151	129	297	Property fails
M1	0.75s	0.02s	15.10s	0.22s	457	109	280	Property holds
M2	1.00s	0.03s	14.78s	0.13s	405	108	279	Property holds
V1	0.73s	0.14s	5.86s	0.12s	102	52	123	Property fails

Table 1. Results for the CONTINUE (C1-11), Medico (M1-2) and voting (V1) examples.

$\neg F$ to the SAT solver) is to convert $\neg F$ to CNF. For conversion to CNF we have implemented the structure preserving technique from [10].

5 Experiments

Our tool generates a Boolean formula in Conjunctive Normal Form (CNF), which we then give to a SAT solver; in particular, we use the `zchaff` [9] tool. To demonstrate the value of our tool we conducted some experiments. One of the policies we used for our experiments is the CONTINUE example [7], encoded into XACML by Fisler et al. [3]. CONTINUE is a Web-based conference management tool, aiding paper submission, review, discussion and notification. In addition, we used the Medico example from the XACML [12] specification, which models a simple medical database meant to be accessed by physicians. Finally, we have encoded our online voting example from Section 3 into XACML and applied our tool to the discovery of the error which we know to exist. We tested 11 properties:

- C1 tests whether the conference manager denies program committee chairs the ability to review papers he/she has a conflict with.
- C2 and C7 test properties concerning reviews for papers co-authored by program committee members.
- C3 and C8 test properties concerning access to the conference manager if the user has no defined role.
- C4 and C5 test properties regarding read access to information about meetings.
- C6 tests whether program committee members can read all parts of a review.
- C9 tests which roles can set meetings.
- C10 and C11 test under what conditions program committee members can see reviews.
- M1 and M2 test whether the unified Medico policy upholds the required access properties about the medical records.
- V1 is the voting property we discussed in Section 3.

The performance results shown in Table 1 indicate that analysis time is dominated by the initial parsing of the policies and by the conversion from triple form to a Boolean formula; sometimes the Boolean conversion is strongly dominant, as in the Medico examples. The resulting formulas are unexpectedly small and analysis time is so small the

startup and I/O overhead of the `zchaff` tool is probably dominating. This was unexpected; our tool goes to some length to simplify the Boolean formula on the assumption that run times would be dominated by the SAT solver. The results show that our assumption was wrong. These results are very encouraging in terms of the scalability of the proposed approach. Among the different components of our analysis, SAT solving is the one with worst case complexity. Since the examples we tested so far were easily handled by the SAT solver we believe that our approach will be feasible for analysis of very large XACML policies.

There appears to be no relationship between lines of XACML and the number of Boolean formulas required to represent them, which is counterintuitive. This reflects a difference in structure between the Medico and voting example and the CONTINUE conference manager. The CONTINUE conference manager was written by Fisler et al. [3] for their Margrave tool, which supports only simple conditionals in the `<Target>` block of an XACML specification. Accordingly, the policy files require far more text to describe simple Boolean combinations than would be the case if `<Condition>` elements were used. We use their example because it is the largest XACML example that we could find, but it is instructive that the Medico example from the XACML specification is as or more complex despite using an order of magnitude less lines of XACML.

The number of variables in our Boolean formulas is quite large, approximately half the number of clauses. We have made a deliberate tradeoff to get this; our translation machinery from Section 4 introduces large numbers of tightly constrained variables, and our CNF conversion uses the structure preserving technique [10] which generates even more variables. In exchange we get a relatively small formula, and the search space is not so large as might be presumed because of the constraints. A different CNF conversion that would embody a different tradeoff between the CNF conversion and SAT solving might be worth exploring.

Our experimental results clearly demonstrate that the subsumption property is practical to analyze, and we believe total runtime could be lowered by optimizing the Boolean formula generation and CNF transformation steps.

6 Related Work

There has been earlier work on automated analysis of access control policies. [11] and [14] analyze role based access control schemas using the Alloy analyzer. However [11] uses Alloy to verify that the composition of specifications is well formed and is silent about their content, whereas we introduce a formal model of and a partial ordering on XACML specifications specifically designed for analyzing the semantics. Zao, Wee et al. [14] model RBAC schema in Alloy and then check these models against predicates, also written in Alloy. We introduce a formal model for XACML with a partial ordering on policies that we then automatically check using a SAT solver as a back end; we do not insist that the user write predicates in another language and operate solely on XACML.

The Alloy Analyzer also uses a SAT solver as a back-end to solve verification queries [5, 6]. Hence, translating XACML policies to Alloy in order to verify them is in effect an indirect way of using a SAT solver for verification. In fact, we also used Alloy analyzer for verification of XACML policies in our earlier work [4]. However our experience has shown that a direct translation to SAT is much more effective then translating

the verification queries to Alloy. In our experience the Alloy Analyzer is not always capable of dealing with the sizes of problems we are dealing with. It is certainly the case that our direct translation generates a customized encoding of the problem, whereas the translation from the Alloy Analyzer is optimized for a more general class of models; hence it may not necessarily be efficient for types of verification queries we are interested in. Mankai and Logrippo [8] also use Alloy Analyzer to analyze interactions and conflicts among access control policies expressed in XACML. The translation to Alloy appears to have been done by hand, in contrast to our automated translator; as well, cited runtimes are around a minute for simple policies whereas our current approach takes seconds to analyze the most complex policies we could find.

Zhang, Ryan and Guelev [15] have developed a new language named *RW*, on which they can perform verification through an external program, and which can be compiled to XACML. It is not obviously possible to translate arbitrary XACML policies to *RW*, and so no analysis on arbitrary XACML policies can be done within their framework, unlike ours.

Bryans [2] modeled XACML using the Communicating Sequential Processes (CSP) process algebra, and then used the FDR model checker to provide some automatic verification, including comparing policies. Bryans uses process interleavings to model rule and policy combination operations which is likely to add unnecessary nondeterminism and increase the state space. In fact, Bryans does not handle all policy combination operations in XACML due to efficiency concerns.

Recently, Fisler et al. [3] used multi-terminal decision diagrams to verify properties of XACML policies with the Margrave tool. Verification queries in [3] are expressed in the Scheme language. We use relationships between policies instead, and since this does not require learning a separate query language, we believe this makes our tool easier to use. Margrave does not handle as much of XACML as we do, and so our tools are not directly comparable; we handle more datatypes, and also complex conditionals as in `<Condition>` elements, whereas Margrave can only handle simple conditionals in the `<Target>` block. For example, the predicate $x < 18$ as in our subpolicy S_0 cannot be expressed in Margrave, not even as an uninterpreted Boolean variable, because it can only be written in a `<Condition>` element. Of our examples, none of M1, M2 or V1 can be expressed using Margrave. The verification underpinnings of our tools are also different; a verification approach that uses decision diagrams is more likely to be successful for incremental analysis techniques, and so are probably the appropriate representation to use for the change-impact analysis presented in [3]. However, for the type of verification queries we discuss in this paper we expect a verification approach based on SAT solvers to perform better than a verification approach based on decision diagrams.

Agrawal et al. [1] discuss techniques for analyzing interactions among policies and propose algorithms for policy ratification. They use techniques from constraint, linear and logic programming domains for policy analysis. Compared to the approach presented in [1] we focus on the XACML language and use a Boolean SAT solver for analysis. Unlike the approach discussed in [1], we are not proposing new mechanisms for combining different policies. Rather, the approach we present in this paper is useful for automated analysis of existing policies and finding possible errors in them.

7 Conclusions

We have presented a formal model for access control policies, and shown how to verify interesting properties about such models in an automated way. In particular we translate queries about access control policies to Boolean satisfiability problems and use a SAT solver to obtain an answer. We express properties about access control policies as subsumption queries between two policies. We implemented a tool that implements the proposed approach and our experimental results indicate that automated verification of nontrivial access control policies is feasible using our approach.

References

1. D. Agrawal, J. Giles, K. W. Lee, and J. Lobo. Policy ratification. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 223–232, 2005.
2. Jeremy Bryans. Reasoning about xacml policies using csp. Technical Report 924, Newcastle University, School of Computing Science, July 2005.
3. K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering*, pages 196–205, St. Louis, Missouri, May 2005.
4. Graham Hughes and Tevfik Bultan. Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, September 2004.
5. Daniel Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, November 2000.
6. Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of International Conference on Software Engineering*, Limerick, Ireland, June 2000. IEEE.
7. S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, January 2003.
8. Mahdi Mankai and Luigi Logrippo. Access control policies: Modeling and validation. In *Proceedings of the 5th NOTERE Conference*, pages 85–91, Gatineau, Canada, August 2005.
9. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC 2001)*, Las Vegas, June 2001.
10. David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
11. Andreas Schaad and Jonathan Moffet. A lightweight approach to specification and analysis of role-based access control extensions. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, June 2002.
12. eXtensible Access Control Markup Language (XACML) version 1.0. OASIS Standard, February 2003.
13. XML Schema part 2: Datatypes. W3C Recommendation, May 2001.
14. John Zao, Hoetech Wee, Jonathan Chu, and Daniel Jackson. RBAC schema verification using lightweight formal model and constraint analysis. In *Proceedings of the eighth ACM symposium on Access Control Models and Technologies*, 2003.
15. Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems in xacml. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 56–65, New York, NY, USA, 2004. ACM Press.