Verification and Change-Impact Analysis of Access-Control Policies[*]

Kathi Fisler
WPI

Shriram Krishnamurthi
Brown University

Leo A. Meyerovich
Brown University

Michael Carl Tschantz
Brown University

ABSTRACT

Sensitive data are increasingly available on-line through the Web and other distributed protocols. This heightens the need to carefully control access to data. Control means not only preventing the leakage of data but also permitting access to necessary information. Indeed, the same datum is often treated differently depending on context.

System designers create policies to express conditions on the access to data. To reduce source clutter and improve maintenance, developers increasingly use domain-specific, declarative languages to express these policies. In turn, administrators need to analyze policies relative to properties, and to understand the effect of policy changes even in the absence of properties.

This paper presents Margrave, a software suite for analyzing role-based access-control policies. Margrave includes a verifier that analyzes policies written in the XACML language, translating them into a form of decision-diagram to answer queries. It also provides semantic differencing information between versions of policies. We have implemented these techniques and applied them to policies from a working software application.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*

General Terms

Algorithms, Security, Languages, Verification.

[*]This research is partially supported by US National Science Foundation grants ITR-0218973, CCR-0305834, CCR-0305949 and CPA-0429492.

Keywords

access-control policies, change-impact analysis, verification, decision diagram, XACML

1. MOTIVATION

Important data are now on-line, and are increasingly accessible through a variety of means, ranging from thin clients (such as Web interfaces) to thick ones (distributed applications like Web services). Due to this growing variety of access methods, central databases must now provide data in a large number of different contexts, each governed by specific access-control policies.

Implementing and maintaining these policies has become increasingly difficult. In particular, it is dangerous for programmers to hard-code a policy in a program that accesses data, for several reasons. First, tracing the policy when maintaining the program becomes very difficult, since its implementation is likely to be scattered across the codebase. Second, it is onerous to share the same policy across multiple different programs and to change the policy consistently. Third, programmers must be sure to make efficient implementation decisions when hard-coding the policy. Finally, automated reasoning about the policy becomes difficult since it forces reasoning about one or more programs written in rich general-purpose languages, each of which undoubtedly contains many operations and data that are unrelated to the policy itself. For all these reasons, we notice a growing trend towards writing separate access-control policy specifications and integrating them with programs in a standardized manner.

Access control is conventionally defined using a matrix. As organizations grow, however, the explicit matrix becomes very cumbersome. Modern access-control policy languages are instead *declarative*, using rules that succinctly capture the information in the matrix. Given a request for data, an engine evaluates the request against the rules to determine whether or not to provide access.

Simply authoring a policy is, however, insufficient: an organization must also be able to analyze it. Testing, while useful, suffers from the hindrances of requiring oracles and not necessarily being exhaustive. As organizations grow their policies can become very subtle, which places particular burden on the quality of testing. The growing value of both availability and privacy of information demands a high degree of confidence in a policy. Policy deployers would thus benefit from complementing testing with more exhaustive, formal verification methods.

As we know, however, property elicitation is rarely complete, so we cannot rely solely on verification failure to identify problems, especially around sensitive information. Organizations that do not have mathematically precise statements of requirements still need to react to problems induced by changing a policy. Suppose you find your company publishing sensitive information on a public Web page. You notify your administrator, who rapidly modifies the policy. You re-load the Web page and the sensitive information is no longer on it. This is a relief; however, how do you know what *else* changed, especially given the subtle behavior of declarative rules?

Organizations in this situation would benefit from a more lightweight process that highlights changes in the effect of the policy, which administrators can explore for unintended consequences. For instance, they should welcome an analysis that shows them all the requests that used to map to deny but now map to permit, so they can examine these for information leakage or denial.

In this paper, we present a suite called Margrave[1] for analyzing access-control policies written in the XACML standard. Margrave has two components:

1. A verification system that consumes a policy and property and determines whether the policy satisfies the property. (More generally, this can be used as a query engine to investigate the behavior of a policy.)
2. A system for *change-impact analysis*. The analysis consumes two policies that span a set of changes and summarizes the differences between the two policies. Users can not only examine the summary, but also query it and verify properties of the change. This verification can happen even in the absence of formal properties about the system as a whole. (Indeed, these properties may not even hold of the entire system.)

We have implemented these ideas and successfully applied them to both organizational and software policies. The running time of Margrave on these examples suggests that it could feasibly be used in an iterative refinement scenario for policy design.

2. BACKGROUND

XACML [24] is a standardized XML language for describing access-control policies. XACML policies center around *attributes*. Attributes describe *subjects*, *actions*, and *resources*; for example, Faculty is a value for the attribute role, which describes the subject. The names of attributes (such as role) are called *attribute ids*, while the values bound to them (such as Faculty) are called *attribute values*. A (abbreviated) fragment of an XACML policy designating Faculty as a role is

```
<SubjectMatch MatchId="...:string-equal">
  <AttributeValue DataType="...#string">
    Faculty</AttributeValue>
  <SubjectAttributeDesignator
    AttributeId="role"
    DataType="...#string"/>
</SubjectMatch>
```

XACML can thus express role-based access-control [8]. Delineating access by roles (rather than by individuals) lets organizations express privileges more abstractly, and thus more easily adapt to changes in personnel.

A *rule* in XACML specifies which decision to take as a function of the attributes. The XACML engine consumes a *request*—the names and values of a set of attributes—and makes an access-control decision on it based on the specified rules. A decision can be *permit*, *deny*, or *not-applicable*; the last arises if no rule in the policy covers the request. (The full standard is more complex, but this description suffices for our purposes.)

Given that multiple rules may yield decisions for the same request (say through overlaps in rules or in roles), an XACML policy must specify the precedence between rules in the form of *rule-combining algorithms.* While the standard permits the definition of arbitrary rule-combining algorithms, Margrave supports three described in the standard: *permit-overrides* says that a permit decision takes precedence, *deny-override* is analogous, and *first-applicable* follows the decision in the first rule to report permit or deny. Sets of rules combined through these algorithms form *policies.* Policies may be combined using *policy-combining algorithms*; these have the same names, and similar semantics, as the rule-combining algorithms.

Margrave handles a restricted subset of XACML that we intend to enlarge over time. We do not consider different attribute value datatypes, complex conditionals, multi-subject requests, and more obscure features such as hierarchical resources. Some of these, such as some classes of conditionals, fall outside the scope of static validation and will probably be better served by simulation and testing.

IBM publishes an access-control language called EPAL [25] that is very similar to XACML. It restricts rule combination to the first-applicable algorithm; besides subjects, resources and actions, EPAL also specifies purposes. There are other minor differences, but we believe Margrave can, with minor changes, handle EPAL just as well as it does XACML.

3. ILLUSTRATIVE EXAMPLE

We introduce many of the concepts in Margrave through a running example, presented from a user's perspective. The example formalizes a university's policy on assigning and accessing grades.[2] Armed with this example, we can then study the implementation of these concepts in Margrave.

All the examples discussed in this paper have been processed by Margrave. To keep the examples readable (and to stay within page limits!), the paper presents policies and properties in English. Section 4.4 provides information on obtaining source versions of these policies.

3.1 Property Verification

We initially have two roles, Faculty and Student; two kinds of resources, InternalGrades and ExternalGrades; and three actions: Assign, View and Receive. We expect the policy to initially satisfy the following (inexhaustive) properties:

Pr$_1$ There do not exist members of Student who can Assign ExternalGrades.

Pr$_2$ All members of Faculty can Assign both InternalGrades and ExternalGrades.

[1]A *margrave* (*markgraf* in German and *markgraaf* in Dutch) is a lord or keeper of borders: that is, a medieval access-control manager.

[2]This example is based loosely on Brown University's actual grading policies; the violations we discover mirror those that led over time to the current policy.

Pr$_3$ No combination of roles exists such that a user with those roles can both Receive and Assign the resource ExternalGrades.

Based on these roles, resources and actions, we can implement a policy in XACML:

Pol$_1$ Requests for students to Receive ExternalGrades, and for faculty to Assign and View both InternalGrades and ExternalGrades, will succeed.

Running Margrave immediately reports failure for properties **Pr**$_1$ and **Pr**$_3$. One of the counter-examples indicates that a student can request to receive an external grade (which **Pol**$_1$ permits) while simultaneously assigning an external grade (about which **Pol**$_1$ is silent, which implies access will not be granted). In effect, an illicit request is piggy-backing atop a legitimate one, and exploiting an underspecification in the policy. This reflects a subtlety of XACML: the language specification states that an attribute (here, the action attribute) represents an arbitrary set of values, not necessarily singletons.

The policy author must sometimes make the policy more elaborate to guard against such requests. In other cases, the author can presume that generated requests will include only a single value for certain attributes. To instruct Margrave about this presumption (discussed in section 9), the author can employ a general Margrave mechanism called an *environment constraint*. These are analogous to the environment models used in model checking, which bound the behaviors of the system by explicating details of the operating context in which the model will execute.

Returning to our example, we constrain the attributes (using *make-singleton-attribute* of section 5) governing the action and requested resource to be singletons (**Pol**$_2$). While this addresses the property violations above, this constrained policy still violates **Pr**$_1$ and **Pr**$_3$. The (sole) counter-example shows that a member of Faculty can also be a Student—naturally triggering the violations—which is really a violation of the principle of *separation-of-duty* (SoD) [6]. Fortunately, this separation is easy to encode with another environment constraint (using *constrain-policy-disjoint* of section 5). This results in policy **Pol**$_3$, which satisfies all three properties.

For the next generation of policy, we add teaching assistants (TAs). Since TAs have some of a faculty member's privileges (while still being students), a careless implementation gives them the same rights as faculty (**Pol**$_4$). Margrave catches this error by reporting a violation of **Pr**$_1$ and **Pr**$_3$. The sole counter-example shows that a student with the freedom to assign external grades is also a TA but not a faculty member. Thus:

Pol$_5$ TA can view and assign InternalGrades but *not* ExternalGrades (since faculty must take final responsibility for all external grades), combined with **Pol**$_3$.

Margrave establishes that this policy implementation successfully satisfies all the properties.

Finally, we extend the policy one more time. Now we introduce a FacultyFamily role. Faculty-family members are permitted to enroll in classes and thereby receive external grades (**Pol**$_6$), prompting one more property:

Pr$_4$ All members of role FacultyFamily can receive ExternalGrades.

Margrave analyzes **Pol**$_6$ and finds a violation of property **Pr**$_3$. This is a fundamental policy error; the corresponding counter-example lists a person who has both Faculty and FacultyFamily roles. This can indeed happen when, say, the spouse of a faculty member is also a professor! This is another violation of SoD, but it too can be successfully corrected with one more constraint (**Pol**$_7$).

3.2 Change-Impact Analysis

Suppose we had defined the same sequence of policies, but did not have formal properties to help identify problems introduced at each stage. What can change impact analysis tell us? Remarkably, simply by examining the output of change analysis closely, we can find most of the errors found by formal property analysis.

Since change-impact analysis is meant for use on extensions to stable code, we begin with policy **Pol**$_3$, the initial policy constrained with the singleton and SoD conditions. Performing change analysis on this policy against **Pol**$_4$ (the buggy addition of TAs) finds eight changes that now grant access. Four of these involve ExternalGrades. A vigilant user would immediately notice something awry, since adding TAs should not have affected external grades in any way. Thus, *even in the absence of formal properties, a policy deployer can potentially detect a dangerous leakage of access.*

Repeating this analysis on **Pol**$_3$ and **Pol**$_5$ shows changes that involve only TAs and internal grades. Since the changes all correctly grant new permissions, we can now be confident that the TA role in **Pol**$_5$ was added correctly to **Pol**$_3$.

Upon adding faculty families, we now examine changes between **Pol**$_5$ and **Pol**$_6$. The output reveals that all the changes involve receiving (as opposed to assigning, thankfully) grades, but also reveals that some changes involve the Faculty role. Output like this would, hopefully, help the policy maintainer derive situations such as two spouses who are both faculty members. Finally, analyzing **Pol**$_5$ against **Pol**$_7$ shows the expected output: faculty family members can take courses, and now requests that involve both Faculty and FacultyFamily are not permitted actions.

3.3 Performance

We measured performance on an Athlon XP 1800+ at 1.5GHz with 512Mb RAM. The longest it took to parse and represent a policy was 355 milliseconds (ms), though this time was closer to 70ms once the cache was warm. No property took longer than 10ms to verify; most finished sooner than the timer could reliably measure. Margrave has a memory baseline of 4.7 Mbytes, and verification took no additional memory. The space and time consumed by change impact were similarly negligible.

4. REPRESENTING XACML POLICIES

XACML policies lend themselves naturally to various forms of boolean representations. One natural approach is to encode them using disjunctive normal form, but this has the potential for rapidly exploding in size. We prototyped this to confirm that the natural approach produced formulae that grew too large to be of any practical use. An attractive alternative is to encode the policy as a set of rules in Alloy [16]. Several others have attempted this in various guises, but these approaches also have shortcomings; we discuss this in section 8.

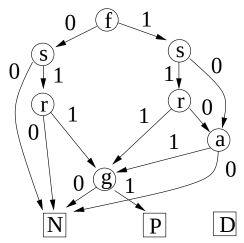

Figure 1: An MTBDD for a simple policy.

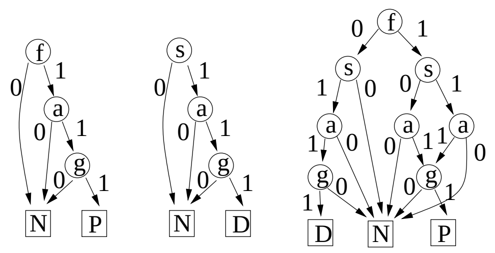

Figure 2: Sample MTBDDs for rules and rule combination.

In any chosen representation, scalability concerns are genuine. The example in section 7 is in fact illustrative of a real software system's policy: it has 50 attribute-value pairs. A member of the Corporate Information Security division of a major American financial institution has informed us that each of their typical policies has 5–20 pairs, but there are several default ones built into the policy system; in general, their only clear upper bound is about 100 pairs. The policy in Schaad, et al.'s case study [28] (not expressed in XACML) appears to need at least 432 attribute-value pairs to express.

4.1 Representing Policies Using MTBDDs

Margrave uses MTBDDs (multi-terminal binary decision diagrams) as the underlying representation of access-control policies. MTBDDs are a form of decision diagram that map bit vectors over a set of variables to a finite set of results [4, 7]. Figure 1 shows an example of an MTBDD representing a simple security policy in which faculty can assign grades and students can receive grades. The MTBDD has five variables (**f**aculty, **s**tudent, **r**eceive, **a**ssign, and **g**rades). Each combination of boolean values over these variables maps to one of three policy results (permit, deny, or not-applicable); the results are denoted by the *terminals* of the MTBDD. We refer to MTBDDs with these three terminals as "policy MTBDDs" or PMTBDDs (though as we will see in section 4.3, we will need a fourth terminal). Given an assignment of boolean values to the variables, traversing a PMTBDD from the root to a terminal according to the variable values indicates the result of the policy under that assignment.

MTBDDs have three defining characteristics, which are valuable in Margrave. First, they are constructed relative to some fixed ordering on the variables (read from root to terminal): an MTBDD would not allow two subtrees to inspect variables in different orders. This restriction makes MTBDDs a canonical representation up to the chosen variable ordering. Second, MTBDDs maximally share subtrees, in that at most one copy of a subtree appears anywhere in the decision diagram; this is evident in the three paths to the same grade node in figure 1. Third, MTBDDs collapse irrelevant variables, meaning that if both values of the same variable refer to the same subtree, the node for that variable is removed, and all references to it are redirected to the shared subtree. The combined effect of the latter two characteristics on the MTBDD in figure 1 is seen in the case where the faculty and student variables are false: rather than enumerate the remaining variables' values, the MTBDD refers directly to the not-applicable terminal.

MTBDDs are a more general form of BDDs [5], which have been credited with helping model checking scale to realistic systems in hardware verification (BDDs have only two terminals, one for each boolean value; the B stands for "binary"). While in the worst case the number of nodes in an MTBDD is exponential in the number of variables, in practice the number of nodes is often polynomial or even linear. This representation therefore gains dramatically over naïve DNF in practice. Operations that combine MTBDDs tend to be efficient in practice; most of the ones Margrave needs yield MTBDDs with at most the product of the numbers of nodes of the originals. We summarize the complexity of our policy-specific operations as we encounter them.

Margrave uses one variable for each attribute-value pair that is mentioned in the XACML policy. The variables for the policy shown in figure 1 would correspond to role=faculty, role=student, action=receive, action=assign, and finally resource=grade (the association between attributes, such as role, with values, such as faculty, is stated in the XACML policy). Margrave creates MTBDDs for the individual rules, then combines these with MTBDD-combining algorithms that implement the XACML rule- and policy-combining algorithms.

In the implementation, Margrave views the policy constants permit and deny as rules; an operation called *augment-rule* takes a boolean condition on the variables and a rule and constrains the rule to also require the given condition. Rule PMTBDDs are built using *augment-rule*. The PMTBDDs for rules saying that faculty attempting to assign grades should yield permit and students attempting to assign grades should yield deny appear as the left two PMTBDDs in figure 2. The third PMTBDD represents the result of combining these two rules using any of the three rule-combining algorithms we have discussed (these produce different results only when the conditions on the rules overlap).

4.2 Combining Policies

Algorithms for combining two MTBDDs use a general algorithm called *Apply* that is parameterized over a function specifying what to do when one or both of the input MTBDDs reaches a terminal node. *Apply* traverses both MTBDDs simultaneously starting from the root node (for the least variable in the order). Each node is labeled with a variable, and has two children nodes, one for each value (true or false) of that variable; we represent this below as Node(v, B_0, B_1) where v is the variable and each B_i is the child node corresponding to truth value i. The following two equations summarize how *Apply* works on node combinations not covered by the terminal's function (assuming that $a < b$ in the variable order):

$$\text{Apply}\ (\text{Node}(a, F_0, F_1), \text{Node}(a, G_0, G_1)) = \\ \text{Node}(a, \text{Apply}(F_0, G_0), \text{Apply}(F_1, G_1))$$

```
Combine_permit−override(A1, A2) =
    if A1 = permit or A2 = permit then permit
    else if A1 and A2 are both terminals
        if A1 = deny or A2 = deny then deny
        else na
    else continue apply

Combine_first−applicable(A1, A2) =
    if A1 = permit then permit
    else if A1 = deny then deny
    else if A1 and A2 are both terminals then A2
    else continue apply
```

Figure 3: The base cases for rule-combining algorithms.

$$\text{Apply}\ (\text{Node}(a, F_0, F_1), \text{Node}(b, G_0, G_1)) = \\ \quad \text{Node}\ (a, \text{Apply}(F_0, \text{Node}(b, G_0, G_1)), \\ \qquad \text{Apply}(F_1, \text{Node}(b, G_0, G_1))$$

The functions for handling terminals in two of the rule-combining algorithms appear in figure 3. The subtree sharing and variable collapsing that characterize MTBDDs are built into the *Apply* algorithm. The equations for *Apply* illustrate that the result of any combination algorithm that uses it has at most the product of number of nodes in the input MTBDDs.

4.3 Applying Environment Constraints

Environment constraints, such as "no faculty is also a student", are easily represented as boolean conditions, and thus as BDDs. Combining a PMTBDD with a constraint BDD also builds on *Apply* (recall that a BDD is a special case of an MTBDD). When the BDD constraint indicates true, the terminal function follows the PMTBDD. When it is false, however, we must flag the path as having been eliminated by the constraint to avoid erroneously considering these variable assignments in subsequent computations. We therefore add a fourth terminal, EC, to represent that a path has been **e**xcluded by a **c**onstraint.

Adding this fourth terminal impacts rule-combination. In a first-applicable policy, for instance, if a path (i.e., a variable assignment) maps to permit in the first policy and EC in the second, it should map to permit in the resulting combined policy. In principle, this particular case will very rarely occur: presumably, all policies will be governed by the same set of environmental constraints, which means a path excluded by constraint in one policy will be excluded in all of them. Margrave does not, however, preclude the use of different constraints on different policy fragments.

Margrave updates the base cases of the rule-combinations shown in figure 3 to account for EC. Since the details are straightforward, we do not present the updated combinations here.

4.4 Implementation

Margrave is implemented atop the CUDD package [30], which provides an efficient implementation of MTBDDs (which are called ADDs in CUDD). Margrave lets users manipulate MTBDDs as data structures in Scheme [19]. Specifically, Margrave imports the CUDD representation and operations through the native function interface of PLT Scheme [9], a full-featured programming language with objects, iterators, and other modern language features. PLT Scheme's programming environment, DrScheme, provides an interactive interface to better enable incremental exploration of policies and changes.

When a policy fails to meet a property, Margrave presents the requests that lead to the violation. For instance, consider a student-TA assigning ExternalGrades due to the bug in the policy **Pol**$_4$ from section 3. Margrave's error report has this form:

```
1   1:/Resource, resource-class, ExternalGrades/
2   2:/Resource, resource-class, InternalGrades/
3   3:/Action, command, Assign/
4   4:/Action, command, View/
5   5:/Subject, role, Faculty/
6   6:/Action, command, Receive/
7   7:/Subject, role, Student/
8   8:/Subject, role, TA/
9   12345678
10  {
11  10100011
12  }
```

Line 11 represents the set of requests that comprise the counter-example. To explain this output, lines 1–8 list the subjects, resources and actions mentioned in the policy, while line 9 indexes the counter-example information against this header. The 1s in line 11 indicate which subjects, resources and actions are present in the counter-example, while the 0s indicate absence. (A third symbol, -, indicates "don't care".)

As an instance of change impact output, consider the difference between **Pol**$_3$ and **Pol**$_4$. The astute reader can find the error presented above in the output below, this time by seeing that ExternalGrades (column 3) are involved in (four) changes from not-applicable (N) to permit (P).

```
1   1:/Subject, role, Faculty/
2   2:/Subject, role, Student/
3   3:/Resource, resource-class, ExternalGrades/
4   4:/Resource, resource-class, InternalGrades/
5   5:/Action, command, Assign/
6   6:/Action, command, View/
7   7:/Action, command, Receive/
8   8:/Subject, role, TA/
9   12345678
10  {
11  00010101  N->P
12  00011001  N->P
13  00100101  N->P
14  00101001  N->P
15  01010101  N->P
16  01011001  N->P
17  01100101  N->P
18  01101001  N->P
19  }
```

The raw output from Margrave, while meaningful, is not yet refined enough for end-users. It is, however, easy to see that this information can become the input to a graphical interface that permits better exploration.

The Margrave implementation, as well as the sources of policies and properties discussed in this paper, are all available on the Web:

`http://www.cs.brown.edu/research/plt/`
`software/margrave/`

We welcome the use of these examples by developers of other tools for reasoning about access-control policies.

5. POLICY QUERYING AND VERIFICATION

Section 3.1 presents examples of properties that we check against a policy. Margrave provides operations that perform the steps needed to verify these properties.

Consider property $\mathbf{Pr}_1$. To verify this property, the user would restrict the policy to cases that permit assigning external grades, then check to see whether students are enabled in the restricted policy. This highlights three fundamental operations needed to verify policies: restricting policies to particular decisions (e.g., permit), inspecting which attribute values appear in a restricted policy, and focusing the search on specific attribute values (e.g., Assign and ExternalGrades). In general, a user might want to incrementally explore a policy through such queries. Margrave therefore provides a set of primitive combinators (composable functions) that users can employ to query policies. (We have not yet built a special-purpose query language atop these combinators; we discuss this issue further in section 10.)

In what follows, we employ the following type domains:

$$\begin{array}{rcll} \text{Dec} & = & \{\text{P, D, NA, EC}\} & \textit{decision} \\ \text{av} & = & \text{id} \times \text{val} & \textit{attribute-value pair} \\ \text{avc} & = & \mathcal{P}(\text{av}) \rightarrow \text{bool} & \text{(described below)} \\ \text{Pol} & = & \mathcal{P}(\text{av}) \rightarrow \text{Dec} & \textit{policy} \end{array}$$

where *id* is a set of legal attribute identifier names, *val* is a set of corresponding attribute values, and $\mathcal{P}(\cdot)$ is the powerset operator.

The key internal data structure in Margrave is called the *attribute-value cohort*, or AVC, which represents a set of requests. (As section 2 defines, a request is a set of attribute-value pairs.) A canonical use of an AVC would be to represent the restriction of a policy to a particular kind of decision:

$$\textit{restrict-to-dec} \quad : \quad \text{Pol} \times \text{Dec} \rightarrow \text{avc}$$

The result is the set of requests that yield the specified decision under the given policy.

To implement *restrict-to-dec*, Margrave replaces the terminal for the given decision in the policy MTBDD with logical true and all other terminals with logical false. This yields a decision diagram with only boolean terminals, i.e., a BDD.

Given an AVC, a set of combinators helps extract information from them. The more primitive operators implement quantification over variables associated with sets of attribute values. The more convenient operators include the following. *get-present-matches* computes the set of all *id*=*val* pairs present in at least one request in the set represented by the AVC. *get-present-attrValues* returns just those values that match a given id.

$$\begin{array}{rcl} \textit{get-present-matches} & : & \text{avc} \rightarrow \mathcal{P}(\text{av}) \\ \textit{get-present-attrValues} & : & \text{avc} \times \text{id} \rightarrow \mathcal{P}(\text{val}) \end{array}$$

get-present-matches builds on two CUDD operations (viz. `Cudd_SupportIndex` and `Cudd_FindEssential`) that determine which variables are used on paths to true in a given BDD. *get-present-attrValues* uses a Scheme-level mapping from CUDD variables to their corresponding pairs of attribute ids and values to project the result of *get-present-matches* to the values for the given id. These operations are linear in the size of the BDD.

The above two operations, in conjunction with *restrict-to-dec*, can already perform rudimentary queries of a policy. For instance, we can easily determine all the roles that have write access to a file. We cannot, however, verify properties of the form "Ensure that students cannot write grades", because we have neither predicates, nor ways of focusing our attention on the subset of requests that have common attribute values (such as those whose role attribute has the value student).

To enable verification, Margrave provides a set of AVC combinators:

$$\begin{array}{rcl} \textit{avc-empty?} & : & \text{avc} \rightarrow \text{bool} \\ \textit{avc-equal?} & : & \text{avc} \times \text{avc} \rightarrow \text{bool} \\ \textit{avc-}\sqcap & : & \text{avc} \times \text{avc} \rightarrow \text{avc} \\ \textit{avc-}\sqcup & : & \text{avc} \times \text{avc} \rightarrow \text{avc} \\ \textit{avc-not} & : & \text{avc} \rightarrow \text{avc} \end{array}$$

Other operators include additional comparisons (such as subset) and more boolean combinations. In addition,

$$\textit{make-avc} \quad : \quad \text{id} \times \text{val} \rightarrow \text{avc}$$

takes an attribute identifier (such as action) and a value for that attribute (such as assign) and creates an AVC that represents all requests that possess the given attribute-value pair (such as action=assign).

Recall that AVCs are represented as BDDs. The *make-avc* operation takes the given attribute and value and returns a BDD that maps any request containing that attribute-value pair to true, and all other requests to false. The remaining AVC operations are implemented as standard boolean operations on BDDs: *avc-*$\sqcap$ computes the intersection using BDD-and, *avc-empty?* checks whether the BDD is the constant false, and so on. All of these standard BDD operations are built into CUDD. Binary operations on BDDs take time linear in the size of each BDD and result in a BDD of size at most the product of the input BDDs. The unary operations all take constant time due to particulars of their implementation within CUDD.

At this point, we have not yet presented the representation of environment constraints. Conceptually, just as an AVC can indicate which attribute values and combinations to focus on, it can also indicate those to which the policy applies. Specifically, *constrain-policy* takes a policy and a constraint (represented as an AVC) and returns a constrained policy in which all cases failing to meet the constraint now map to EC. To simplify constraining policies, Margrave provides *constrain-policy-disjoint*, which takes a policy and a set of conditions (such as action=assign and action=receive) and constrains the policy to those cases that satisfy at most one of the given conditions. A simplified form of *constrain-policy-disjoint* is *make-singleton-attribute*, which takes an attribute identifier (such as resource) and restricts the policy to cases where exactly one value is associated with the given identifier. The implementation of *make-singleton-attribute* makes a closed-world assumption, namely that it knows the names of all the attribute values that can occur in a policy.

$$\begin{array}{rcl} \textit{constrain-policy} & : & \text{Pol} \times \text{avc} \rightarrow \text{Pol} \\ \textit{constrain-policy-disjoint} & : & \text{Pol} \times \mathcal{P}(\text{avc}) \rightarrow \text{Pol} \\ \textit{make-singleton-attribute} & : & \text{Pol} \times \text{id} \rightarrow \text{Pol} \end{array}$$

constrain-policy uses the *Apply* operation from section 4.2, replacing any path for which the AVC yields false with EC. The other two operations are implemented via *constrain-policy*: *constrain-policy-disjoint* forms a BDD representing the exclusive-or of the given AVCs, combines it using *avc*-$\sqcup$ with the negation of the *avc*-$\sqcap$ of the given AVCs, and then behaves as *constrain-policy*; *make-singleton-attribute* is similar, but the exclusive-or is over all variables corresponding to the given attribute. The complexity of these operations therefore comes from the complexity of *Apply* (polynomial in the size of its inputs).

Use Case

As a sample use of the combinators for verification, the following Margrave query checks that students cannot assign final grades. (We use traditional rather than Scheme notation for the benefit of the parenthetically-challenged.)

stu-asn-ext = *avc*-$\sqcap$ (*make-avc* (role, Student),
make-avc (resource, ExternalGrades),
make-avc (action, Assign))
perm-requests = *restrict-to-dec* (*policy*, permit)
perm-stu-asn-ext = *avc*-$\sqcap$ (*perm-requests*, *stu-asn-ext*)
answer = *avc-empty?* (*perm-stu-asn-ext*)

When *policy* is bound to $\mathbf{Pol}_4$ (the buggy policy involving TAs), *answer* is bound to false. We investigate the problem by running *get-present-matches* (*perm-stu-asn-ext*), which indicates that errors occur when the student is a TA.

6. CHANGE-IMPACT ANALYSIS

Change-impact analysis must function even in the absence of an externally-defined property. All we can assume to be given is two (hopefully similar) versions of a policy. How do we extract differences between the two? In particular, can we reuse our PMTBDD representation of a policy for performing this analysis?

The natural approach is to attempt to compute the difference between two PMTBDDs. The difference is well-defined for binary decision diagrams, but is not as clearly defined for multi-terminal decision diagrams such as a PMTBDD. To define this operation in our context, it is worth examining what form of answer would be most useful.

When a user analyzes a change, they are naturally interested in which requests will produce different answers in the two policies. In many cases, a deny changing to a permit will be regarded as dangerous; sometimes, however, the purpose of a change is to publish information, so the inverse change should equally evoke caution. Because of these different use cases, Margrave does not rank either change higher than the other. Rather, the important lesson is that users will care not only about what changed but also how.

To implement change analysis, Margrave introduces a decision diagram called a *change-analysis decision diagram* or CMTBDD. A CMTBDD has *sixteen* terminals, one for each ordered pair of results from the policies being compared (such as permit-to-permit, deny-to-ec, permit-to-not-applicable, and so on). Margrave then provides a suite of combinators for processing a CMTBDD.

Change impact analysis extends the Margrave library with the following new types:

Chg	=	{P→P, P→D, P→NA, P→EC, D→P, ... }	*change kinds*
Cmp	=	$\mathcal{P}$(av) → Chg	*policy difference*

At an intuitive level, the operations we want to perform on CMTBDDs are similar to those on PMTBDDs, such as restricting a CMTBDD to a particular kind of change and determining which variable values can lead to particular kinds of changes. The operators for creating and manipulating CMTBDDs have the following types:

compare-policies	:	Pol × Pol → Cmp
restrict-cmp-to	:	Cmp × Chg → avc
get-attrVal-in	:	Cmp × id × Chg → $\mathcal{P}$(val)

Given two polices (as PMTBDDs), *compare-policies* creates the CMTBDD showing the changes between the policies. This operation is implemented using the *Apply* algorithm. The terminals function applies only when both input CMTBDDs are at terminal nodes, and the returned value is the appropriate terminal for the pair of values at the respective nodes. Having sixteen terminals instead of four may reduce subtree sharing, but the number of nodes remains at most the product of node counts in the original policies.

The *restrict-cmp-to* operator on CMTBDDs uses the same implementation as on PMTBDDs. Margrave supports various *get-attrVal* functions, which use the same core implementation techniques as *get-present-attrValues* but on CMTBDDs. Operation *get-attrVal-in* returns the set of values for the given attribute that can contribute positively to the given change, while others like *get-attrVal-in-change* returns the set of values for attributes that contribute positively to *any* change (rather than a particular change).

Quering and Verifying Changes

Invoking the *compare-policies* procedure on two PMTBDDs constructs a CMTBDD, whose printed representation shows which combinations of changes occur between the policies. More interestingly, the user can then query the result of change analysis. The user can, for instance, apply *get-attrVal-in-change* to get the roles that trigger D→P changes. To further explore changes from deny to permit, the user can employ *restrict-cmp-to* to obtain an AVC.

Having generated an AVC, the user can now employ the rich set of AVC primitives from section 5. *get-present-matches* can extract the contributing pairs of attribute ids and values; the AVC boolean connectives can further refine the set for the AVC predicates to verify properties over changes, akin to verifying a single policy. For example, the user can verify that a change did not affect access to ExternalGrades. This enables a form of *lightweight policy exploration.*

7. ANALYZING A CONFERENCE MANAGER

We have applied Margrave to an access-control policy modeling CONTINUE [13, 22], a working software system initially written by the second author. CONTINUE is a Web-based application that supports the submission, review, discussion, and notification phases of conferences. CONTINUE has been used by several conferences to date, including the International Symposium on Software Testing and Analysis (ISSTA), 2004.

We were forced to adopt a role-based information presentation policy for CONTINUE after discovering role-conflict errors during testing. This policy was implemented in a special-purpose access-control language we had developed (since we were not aware of XACML at that time). For this paper, we have translated this policy into XACML, generated properties from a requirements exercise, and gathered information on Margrave's performance.

Translating and verifying the CONTINUE policy is valuable for assessing Margrave. First, it represents the policy of a real software system, in particular one whose verification has tangible benefits for the user community of researchers submitting papers to conferences. Second, the policy describes a software system found "in a state of nature", so it is not contrived to suit Margrave. In particular, it is a valuable evaluation of the utility of the XACML fragment that Margrave handles. Finally, the bugs we have found in CONTINUE are in fact based on role-conflicts, making this an *interesting* example for encoding and verification.

The CONTINUE policy has 50 PMTBDD variables. The MTBDD has 1268 nodes; upon applying environment constraints, it shrinks to 817 nodes. Parsing and converting it to this representation took 2050ms and constraining it another 20ms.

We verified twelve properties against this policy. Each property verification took less time than could reliably be measured by the millisecond-resolution clock. The entire process increased memory consumption by 316,288 bytes above the baseline (due to CUDD) after all the verification runs. The verification process did report errors; some are an artifact of our modeling, while others appear to be legitimate software bugs that we are investigating further.

We also conducted change analysis on different versions of the CONTINUE policy. Given the two policies converted into PMTBDDs, the change itself computed in 2 milliseconds. The resulting CMTBDD had 1133 nodes and consumed 16.3Kb of additional memory.

8. RELATED WORK

The similarity between policies and functions from boolean variables to booleans suggests encoding them using DNF [2]. This can, however, require exponentially many clauses relative to the number of variables. This especially arises from rule-combining algorithms such as first-applicable, since the expression must encode the order of evaluation. These problems are compounded by deny rules, policy sets, etc. A similar explosion occurs when performing change analysis. Indeed, our experiments suggest that DNF does not scale beyond all but the simplest policies.

Alloy [16] seems an obvious choice for modelling access-control policies, but suffers from several potential pitfalls. First, it forces users to bound the sizes of each set, so its analysis is valid only for problem instances up to the stated sizes. Second, Alloy is designed to support first-order reasoning: a natural approach to modelling XACML policies in Alloy exploits the first-order features, which results in a substantial increase in the number of variables created as compared to Margrave. Both Margrave and Alloy require a variable per attribute value in the policy. Any other entity that is modelled explicitly (such as roles, people, or individual rules) introduces additional variables that can make the analysis intractable in practice. In our own experience embedding XACML in Alloy using first-order constructs, we found the tool unable to handle even a third of the policy we describe in section 7 without exhausting memory (after a few minutes).

An alternate Alloy approach is to employ the language's propositional constructs (as do Bultan and Hughes [15], and as suggested to us by Daniel Jackson [personal communication]). This approach is, however, somewhat unsatsifying. Implementing the rule- and policy-combining algorithms requires explicitly modeling sub-policies, each of which introduces an additional variable into the analysis as compared to Margrave. Furthermore, an efficient Alloy model of a policy effectively mimics the MTBDD operations within Alloy. It is unclear what this gains relative to using MTBDDs directly, since satisfiability solvers (which Alloy employs to find solutions) and BDD packages perform different optimizations. Finally, the results of an analysis are an internal Alloy representation that can only be explored with Alloy's visualization tools (which, designed for first-order variables, are a bit clumsy for propositional expressions). In contrast, the results of an analysis in Margrave are given as MTBDDs, which enables using the policy query language to explore and refine counter-examples.

A tempting alternative to Alloy is the use of logic programming systems. While such a solution is effective at querying policies, it does not provide support for change analysis. In particular, encoding policies as full-fledged logic programs can potentially make the change analysis problem as hard as trying to difference general source programs written in richly expressive languages, a difficult problem and one whose solutions would provide little insight to the policy developer.

Schaad and Moffett [29] examine the problem of verifying a policy that is subject to change proscribed by another policy. They describe how an access-control policy under the RBAC96 model, a policy governing access to the access-control policy under the ARBAC97 model, and a set of separation-of-duty constraints can be translated into Alloy, which can then be used to check that the ARBAC97 policy does not allow for roles to be assigned to users in ways that will violate the separation-of-duty constraints. Ahmed and Tripathi [1] propose translating access-control policies into PROMELA. From this model, a sub-model is produced for each of four types of query. They employ SPIN to analyze the appropriate model given a query. It is difficult to offer a direct comparison since neither work discusses algorithms, tool support, or experimental results. Our experience, however, is that state-exploration in the style of Alloy may not scale well to large policies.

Kolaczek [21] proposes to translate access-control policies into a role-based access control modal model (RMM), then into Horn clauses for conversion into Prolog. Although he provides a proof that RMM may be represented in Horn clauses, he does not discuss tool support, experimental evaluation or change analysis.

Guelev, Ryan, and Schobbens present a formal language for expressing access-control policies and queries [10]. Their work presents algorithms for deciding queries. They have implemented one of these algorithms in Prolog but do not report on the scalability of this work. They state, moreover, that their policy and property languages are "not meant to be [...] for users" [10]. Their follow-up work [31] provides a translator from their language to XACML that preserves the verified properties.

Guttman and Herzog use BDDs to implement verification tools for network security goals [11]. Their models capture connections between regions or devices within networks and operations (such as packet filtering) within regions. Their verification algorithms search for paths through the network that either allow classes of packets to reach undesirable regions, or violate authentication or confidentiality properties. BDDs scaled very well for their applications (including realistic Cisco routing policies and models of IPSec). Their use of a network model is both a strength and a weakness: the analysis is not effective without such a model but, given one, can provide interesting information about data flow. Their work does not consider change impact analysis.

Several systems have analyzed the policies of the Security-Enhanced Linux system (SELinux). Most of these efforts [12, 27, 23] build models of a SELinux policy and translate it into various notations such as tabled Prolog, BDDs and a model checker's input. These tools are, however, optimized for determining information flow. Since analyzing an XACML policy is largely concerned with resolving rule-conflicts, the efforts appear orthogonal and complementary. Gokyo [18] provides the ability to reason about a policy's structure in terms of underspecification, overlaps, and conflicts, and provides means to resolve these. It does not address verification relative to external properties.

Koch, Mancini, and Parisi-Persicce study the change impact problem [20]. They propose using graph transformations to represent policy change and integration. Although they present examples of changes and of the result as graphs, they present no algorithms or tools, nor suggest methods for eliciting policy change from graph differences.

Backes, Karjoth, Bagga, and Schunter [3] propose refinement relations as a technique for comparing policies. Policy A is said to refine policy B if for all requests, A produces the same decision as B and a set of obligations that include all the obligations B would issue for that request. Their paper presents an algorithm to determine whether one EPAL policy refines another. Since Margrave does not track obligations, it is incapable of validating refinement. However, while this relation is useful in some circumstances, at other times administrators purposely break refinement, so knowing they have done so is of little use.

The change-impact analysis computes, in effect, the difference between two programs. The general problem of program differencing has a venerable research lineage [14]. By working with programs in a restricted domain, however, we can compute semantically richer differences than differencing tools that cater to general-purpose languages.

9. PERSPECTIVE ON PROGRAM ANALYSIS

A comprehensive software-security analysis would employ Margrave as a policy-analysis component. A program that consults a policy engine for sensitive data must ultimately act on its results. The result of the access-control check must thus be combined with, say, secure information flow analysis [26]: if the access-control engine responds with deny, then no information about the secure datum should leak to the client. Margrave isolates the policy verification from the analysis of the enclosing program. This may simplify and enhance some program analyses because critical decisions that were earlier embedded in the (harder-to-analyze) general-purpose language are now expressed in a domain-specific access-control policy language.

Programs also need to be verified with respect to the environment constraints applied during Margrave's analysis. At a technical level, the XACML components known as the PEP, PIP, and context-handler should only produce combinations of attribute values that are consistent with the applied constraints. Ultimately, however, this analysis must occur at a managerial level, as decisions about which attribute values may overlap are organizational in nature.

10. CONCLUSION AND FUTURE WORK

This paper argues that multi-terminal decision diagrams are a scalable and flexible representation for formal analysis of access-control policies. We have discussed two forms of analysis for security policies: querying and verification relative to known properties, and change analysis between two policies. We have presented techniques for performing these operations, and discussed their implementation in the Margrave tool suite. Margrave has been run on policies extracted from real software, and its performance has been extremely attractive. In particular, its running time on these examples is sufficiently low (often well under a second) that it can conceivably be used to iteratively design policies.

Ultimately, Margrave's scalability will depend on the number of variables used in large-scale policies and whether those variables can be ordered to produce compact MTBDDs. BDDs are regularly used for verifying systems with hundreds of variables, which is within the scale discussed in sections 4 and 7. Furthermore, Margrave uses the BDD operations that commonly cause blow-up only when projecting variables to explore counter-examples. We are therefore optimistic about the scalability of this approach.

In conjunction with reasoning about programs, we will also need richer reasoning about data in policies. We currently employ the standard technique of using uninterpreted symbols, constrained by the environment, to simulate reasoning about data. To encode this information directly in a policy would require explicit representation of the data, which would explode the size of the MTBDD. Other reasoning tools may be more effective at this: for instance, theorem provers can typically perform inductive reasoning over structured data, while a tool such as Alloy can perform optimizations such as eliminating isomorphs [17]. These tools therefore perform operations that complement those in Margave, so they may be usable in conjunction (for instance, by generating instances of problems for Margrave).

Section 5 describes several procedures for querying and analyzing a policy. Ideally, the user should be able to express queries in a more concise language. We have intentionally not tried to synthesize such a language, preferring to wait until we have enough experience to generate a catalog of common query paradigms.

Margrave presents output using a simple front-end to display CUDD's sum-of-product representations. Algorithmic and heuristic techniques could be used to eliminate redundant output, partition results around attributes, and provide traceability from counter-examples and changes to policy source files. This is a rich area for future investigation.

Acknowledgments

We thank Seth Proctor, Anne Anderson, and their fellow researchers at Sun Labs East. Seth introduced us to XACML and has since answered numerous questions. We had interesting discussions on this topic with Tevfik Bultan, Dimitar Guelev, and Mark Ryan. Pete Hopkins, co-author of CONTINUE, was helpful (and patient!) during the property elicitation phases for the software. Daniel Jackson helped us create better Alloy representations for policies and discussed representation tradeoffs. Michelle Engel initated the work on our Alloy translator. We benefited from discussions with William Cook, Michael Greenberg and Manos Renieris. The anonymous referees made helpful suggestions for improvement. We thank Robin Fairbairns for the moreverb package for LaTeX.

11. REFERENCES

[1] T. Ahmed and A. R. Tripathi. Static verification of security requirements in role based CSCW systems. In *Symposium on Access Control Models and Technologies*, pages 196–203, 2003.

[2] A. Anderson. Evaluating XACML as a policy language. Technical report, OASIS, Mar. 2003. Document identifier: wd-xacml-wspleval-03.

[3] M. Backes, G. Karjoth, W. Bagga, and M. Schunter. Efficient comparison of enterprise privacy policies. In *Symposium on Applied Computing*, pages 375–382, 2004.

[4] R. Bahar, E. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer-Aided Design*, pages 188–191, 1993.

[5] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8):677–691, 1986.

[6] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium of Security and Privacy*, pages 184–194, 1987.

[7] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *International Workshop on Logic Synthesis*, 1993.

[8] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, 2003.

[9] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[10] D. P. Guelev, M. D. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *Information Security Conference*, Lecture Notes in Computer Science. Springer-Verlag, Sept. 2004.

[11] J. D. Guttman and A. L. Herzog. Rigorous automated network security management. *International Journal of Information Security*, Dec. 2004.

[12] J. D. Guttman, A. L. Herzog, and J. D. Ramsdell. Verifying information flow goals in security-enhanced Linux. In *Workshop on Issues in the Theory of Security*, January 2004.

[13] P. W. Hopkins. Enabling complex UI in Web applications with send/suspend/dispatch. In *Scheme Workshop*, 2003.

[14] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[15] G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, University of California, Santa Barbara, 2004.

[16] D. Jackson. Automating first-order relational logic. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Nov. 2000.

[17] D. Jackson, S. Jha, and C. A. Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems*, 20(2):302–343, Mar. 1998.

[18] T. Jaeger, X. Zhang, and F. Cacheda. Policy management using access control spaces. *ACM Transactions on Information and System Security*, 6(3):327–364, 2003.

[19] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), Oct. 1998.

[20] M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the specification and evolution of access control policies. In *Symposium on Access Control Models and Technologies*, pages 121–130, 2001.

[21] G. Kolaczek. Specification and verification of constraints in role based access control for enterprise security system. In *International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 190–195, 2003.

[22] S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, January 2003.

[23] F. Mayer. Tools and techniques for analyzing type enforcement policies in security enhanced Linux. In *Annual Computer Security Applications Conference*, Dec. 2003.

[24] T. Moses. eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS, Feb. 2003.

[25] C. Powers and M. Schunter. Enterprise privacy authorization language (EPAL 1.2). W3C Member Submission, November 2003.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[27] B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced Linux. In *Workshop on Issues in the Theory of Security*, pages 1–12, April 2004.

[28] A. Schaad, J. Moffett, and J. Jacob. The role-based access control system of a European bank: a case study and discussion. In *Symposium on Access Control Models and Technologies*, pages 3–9, 2001.

[29] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Symposium on Access Control Models and Technologies*, pages 13–22, 2002.

[30] F. Somenzi. CUDD: The CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[31] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *ACM Workshop on Formal Methods in Security Engineering*, 2004.