PolicyMorph: Interactive Policy Transformations for a Logical Attribute-Based Access Control Framework

Michael LeMay, Omid Fatemieh, and Carl A. Gunter
University of Illinois at Urbana-Champaign

ABSTRACT

Constraint systems provide techniques for automatically analyzing the conformance of low-level access control policies to high-level business rules formalized as logical constraints. However, there are likely to be priorities for solutions that are not easy to encode formally, so administrator input is often important. This paper introduces *PolicyMorph*, a constraint system that supports interactive development and maintenance of access control policies that respect both formalized and un-formalized business rules and priorities. We provide a mathematical description of the system and an architecture for implementing it. We constructed a prototype that is validated using a case study in which constraints are imposed on a building automation system that controls door locks. PolicyMorph advances the state-of-the-art in constraint systems by suggesting predictable policy model modifications that will resolve specific constraint violations and then allowing policy administrators to select the appropriate modifications using knowledge that is not formally encoded in the constraint system.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Keywords: attribute based access control, policy administration, separation of duty, constraints

1. INTRODUCTION

Many of the challenges that arise during the development and maintenance of an access control policy are caused by the inability of the policy administrator to correctly translate high-level business requirements into low-level access control policies that can be implemented in an Access Decision Function (ADF). Several approaches to this problem have been explored, such as improving the policy languages themselves to provide more direct expressions of business requirements. Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) languages are representative outcomes of this line of investigation [18, 15]. Another common approach uses constraint languages to test properties of an ADF and point out violations to the policy administrator [11, 7]. Because constraint languages can be very expressive, they are able to encode many business rules directly, but such high-level constraints cannot be used to directly implement an ADF because they specify what access control policies satisfy the business requirements, without actually selecting any particular policy from the (usually infinite) space of acceptable ones. Thus, most constraint checkers simply report constraint violations for the formalized business rules. This generates a substantial burden on an administrator because he must not only resolve the violations manually but must also deal with all of the solution priorities that, for one reason or another, were not formalized within the constraint system. Another, more subtle, point is that business rules are often flexible: exceptions are sometimes made, and a burdensome rule may be ignored or changed. Thus the process of selecting an ADF in light of business rules benefits from formalization and automated support, but also requires significant human input.

In this paper we introduce a system called *PolicyMorph* that helps administrators *interactively* assess ABAC access control policies with respect to logical constraints. This is, PolicyMorph not only reports constraint violations, but also formulates suggestions on how to address common types of violations. It then prioritizes those suggestions, presents them, and allows the administrator to evaluate the effect of each suggestion and implement the suggestion that produces the most desirable outcome. In particular, PolicyMorph allows the administrator to evaluate the desirability of each option, without forcing him to encode all relevant constraints in a formal language. This provides a middle ground between a fully automatic system that places on the administrator a high burden of formalization and a largely manual system that provides little help in discovering and resolving specific violations.

To make these concepts more concrete, consider an access control policy for Personally Identifiable Information (PII) contained in an online retailer's database and regulated by that organization's privacy policy, which sets forth the business requirements that regulate the processing and storage of the PII collected from customers. Other works have explored the formal semantics of privacy policies and explain how to decompose policies into individual goals that can be analyzed further [1, 6, 14]. These goals can be easily converted into logical constraints over an ABAC policy. For example, consider the privacy goal "maintain confidentiality of customer information from third party partners and marketing." Let us assume that some employees hold responsibilities in multiple areas, such as both marketing and information systems support (IS). As a part of their IS duties, such an employee could be responsible for the maintenance of a customer email list. Unfortunately, her membership in the marketing department would disqualify her

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

from this role according to the privacy rule. The constraint checker can easily detect this violation, but it is unlikely to know how to optimally transfer the responsibility for managing the customer email list, since workload information, employee preferences, and other external considerations are rarely encoded into the systems hosting an access control policy. However, a human policy administrator is likely to have access to such information and can easily select between the employees who could be assigned to that task. Thus, the administrator would be aided by an analysis system that presents a list of other employees to whom the responsibility could be transferred, allowing him to make the final selection. On the other hand, if this re-assignment is viewed as impractical or excessively expensive, then the administrator may instead choose to adjust the business rules, perhaps by accepting a weaker level of protection in which the employee is asked to personally enforce the rule.

Two of the fundamental components in PolicyMorph are its logical ABAC policy language and its logical constraint language. Both of these are based on order-sorted first-order logic [16], which is capable of supporting very expressive policies [10]. We then describe our interactive environment for resolving constraint violations and provide examples of policies where an administrator's human knowledge and preferences can be used to resolve constraint violations with the help of PolicyMorph's suggestions and analysis. This analysis is complicated by the fact that PolicyMorph policies can use dynamic contextual information from external sources to make access decisions. We show how this functionality can be supported without significantly complicating policy definitions, and while still preserving safety with respect to constraints. In particular, in one of our examples the location of a subject represented within the access control system is inferred using presence information from an instant messaging protocol and has actually been implemented in our prototype. We demonstrate how our prototype operates as an access decision function using a representative policy for a building automation system in a sophisticated, object-oriented application, and also demonstrate our interactive policy administration tool using that policy.

The rest of this paper is organized as follows. In Section 2 we define our access control policy and constraint languages. In a similar fashion, Section 3 presents our transformation framework. Section 4 describes an architecture and prototype implementation of these systems. Section 5 describes an evaluation of the approach using the prototype to carry out a case study. Section 6 discusses related work. Section 7 concludes the paper and summarizes our future directions.

2. POLICIES AND CONSTRAINTS

In this section we present the major components of our system, namely the access control policies, the models needed to interpret them, and the the constraints to be imposed on them. We assume that the reader is familiar with first-order logic.

Access Control Policies.

A low-level access control policy comprises a set of predicates with a predefined signature that corresponds to the elements of an access decision request. To accommodate this signature, policies use a variety of sorts, including S (agents or principals σ, commonly known as *subjects*, that perform actions on objects), O (*objects* δ upon which subjects perform actions), *Entities* (a super-sort of both S and O), *Actions* (η), *Contexts* (runtime information γ that can be incorporated into access decisions), and *Justifications* (compound terms κ that specify every reason that a positive access decision was provided). More formally, a policy is a first-order formula which can be represented as follows:

$$f \Rightarrow \mathit{Permitted}(\sigma, \delta, \eta, \gamma, \kappa)$$

Whenever this formula is satisfied, it indicates that the corresponding access request should be granted. To understand the role of κ, one should consider it to be an output of the predicate, rather than an input parameter to be tested. It is not used in the decision making process, but is simply unified with the reasons that a positive access decision were made, as discussed in more detail later.

Members of *Contexts* represent a specific set of conditions that can be defined or sensed by the overall system into which the access control system is integrated. Each context is a relation that maps arbitrary, application-defined keys to arbitrary values:

$$\forall \gamma \in \mathit{Contexts}.\ \gamma \subseteq \mathit{CtxKeys} \times \mathit{CtxValues},$$

where *CtxKeys* and *CtxValues* are application-defined sorts. Each context relation is customarily a partial function. However, the policy for each application has the freedom to define its own mechanisms for representing and processing contextual information.

Elements of *Justifications* are used to convey the exact reasons that a particular access decision is granted. Using a backtracking engine like the one built into Prolog interpreters [17], it is possible to determine all possible justifications for a particular access decision.

Each element of *Justifications* can be formally expressed as a set of individual reasons for why a positive decision was made, although the same formulation could also be used to justify negative decisions if our system supported such decisions. Justifications are simply sets of reasons and sets of labels. We now present each reason currently recognized by our framework:

$$\forall \varepsilon \in \mathit{Entities}, \forall \alpha \in A. \qquad \mathit{HasAttr}(\varepsilon, \alpha) \in \mathit{Reasons} \wedge \mathit{NotHasAttr}(\varepsilon, \alpha) \in \mathit{Reasons}$$

The *HasAttr* reason specifies that the entity ε possesses a specific attribute α, whereas *NotHasAttr* signifies that ε lacks α. This convention is also used for the following reasons. Any reason with a name prefixed by "Not" carries the opposite meaning of the positive reason with the same parameters.

$$\forall \varepsilon \in \mathit{Entities}, \forall \alpha \in A. \qquad \mathit{HasSubAttr}(\varepsilon, \alpha) \in \mathit{Reasons} \wedge \mathit{NotHasSubAttr}(\varepsilon, \alpha) \in \mathit{Reasons}$$

The *HasSubAttr* reason specifies that the entity ε possesses an attribute that has the specified attribute α as a direct or indirect parent in the attribute hierarchy (the hierarchy will be described later and is reflexive, so that the specified attribute itself is included in the set of acceptable attributes).

$$\forall \varepsilon \in \mathit{Entities}. \qquad \mathit{IsNamed}(\varepsilon) \in \mathit{Reasons} \wedge \mathit{NotIsNamed}(\varepsilon) \in \mathit{Reasons}$$

The *IsNamed* reason is required because policies can take the exact identity of the subject or object specified in an access decision request into consideration when making the decision.

These reasons can be used to describe the operation of typical policies supported by our system. Specifically, the framework supports permission terms that consider the association or disassociation of an attribute to an entity, or the identity or non-identity of an entity when making an access decision. However, administrators are not restricted to those policies that can be characterized by this justification framework. Any term that is not specially supported by the framework will be wrapped in a generic reason that simply conveys the term verbatim.

Model.

An access control model defines parameters used by an access control policy to produce access decisions. It comprises a set of entity declarations, a set of attributes, an attribute hierarchy definition, and an assignment of attributes to entities. Formally, we can represent this as a 5-tuple:

$$\Psi = \langle S, O, A, AH, AA \rangle,$$

where S and O were explained previously, and:

- A is a sort containing attributes that can be assigned to entities.
- $AH \subseteq A \times A$ is a reflexive, transitive, and antisymmetric relation defining a hierarchy over those attributes, where $(\alpha_1, \alpha_2) \in AH \implies \alpha_1$ is a sub-attribute of α_2. This hierarchy can be used when making access decisions.
- $AA \subseteq Entities \times A$ is a relation which represents the assignment of a set of attributes to each entity in the model.

Constraints.

Since access control policies in our system are expressed in first-order logic, it is easy to express and test queries over those policies using a backtracker that explores the state space of possible query instantiations. We use a backtracker facility parameterized over all entities to express high-level business requirements and other constraints.

Each constraint must conform to the following template:

$$f \Rightarrow Undesirable(\kappa),$$

where f is any first-order formula, $\kappa \in Justifications$ and the formula is only satisfied if an undesirable access is allowed in the *policy instantiation* (combination of low-level policy and access control model) being tested and κ specifies the exact reasons that the undesirable access was allowed. Using a backtracking engine, it is possible to discover all such undesirable accesses. Again, κ is actually an output from the constraint that encapsulates all of the reasons responsible for the violation.

In brief, this allows us to impose high-level constraints on low-level logical ABAC policies. Constraints are used to describe *safety* properties of an access control policy in an inverted manner (i.e. conditions or authorizations that should *never* be permitted in the policy). Such constraints should be written in a predicate calculus, in general because they must preclude assignments that are not known *a priori*. Typically, such constraints are checked whenever the policy or model is modified and before the system is deployed, to prevent the usage of any access control system that may violate the constraints.

3. MODEL TRANSFORMATIONS

When constraints are violated, it is often possible to resolve the violations by performing simple, predictable modifications to the access control model in question. It is typical that some attributes cannot be changed (like the age of the subject) whereas others can be changed under suitable authority (like the task assignment or department of the subject). The latter are the focus of our transformations related to access control and constraint satisfaction. Our system provides four predefined rules for modifying the model based on constraint violations, but additional rules can be easily added to the system. Regardless of the transformation rule in use, the overall transformation process is unmodified. Each set of transformations may resolve one or more violations, but it may also produce new violations, so it is important to re-validate the constraints after implementing transformations. The transformation process itself is represented by a function that accepts the original model and a transformation rule as arguments, and produces a modified model. PolicyMorph provides an interactive environment in which: an access control policy is analyzed with respect to a set of constraints, violations are discovered, suggestions to resolve them are provided, and transformations are applied to eliminate the violations. We now outline the basic transformations and the way in which suggestions are prioritized.

Basic Transformations.

The *elimination transformation* disassociates an attribute from an entity:

$$Transform(\langle S, O, A, AH, AA \rangle, Eliminate(\varepsilon, \alpha)) = \langle S, O, A, AH, AA - \{(\varepsilon, \alpha)\} \rangle$$

The *introduction transformation* associates an attribute with an entity:

$$Transform(\langle S, O, A, AH, AA \rangle, Introduce(\varepsilon, \alpha)) = \langle S, O, A, AH, AA \cup \{(\varepsilon, \alpha)\} \rangle$$

The *egress transfer* transformation disassociates an attribute from the entity specified in the reason and associates it with another entity that does not already possess the attribute.

$$Transform(\langle S, O, A, AH, AA \rangle, EgressTransfer(\varepsilon_1, \varepsilon_2, \alpha)) = \langle S, O, A, AH, (AA - \{(\varepsilon_1, \alpha)\})) \cup \{(\varepsilon_2, \alpha)\} \rangle$$

The *ingress transfer* transformation is the semantic complement of the egress transfer, and is included for notational convenience. It associates an attribute with the entity specified in the reason and disassociates it from another entity that already possesses the attribute. We will explain the purpose of each transformation below.

Since a justification is invalidated if any one of its reasons is invalidated, the administrator is usually not required to implement a transformation for every reason before the policy instantiation becomes conformant. To minimize the number of transformations, we combine the reasons from all violations and sort them in decreasing order according to the frequency with which they each occur. Then, we iterate through the list and generate all possible transformations that will eliminate the reason in question. The administrator is allowed to evaluate the results of any of the suggested transformations and then implement the most desirable transformation. This process continues until the administrator decides to re-evaluate the policy's conformance.

We define a function that suggests transformations to eliminate a specific reason as follows:

$$SR : Reason \rightarrow 2^{Transformers},$$

where *Transformers* is a sort containing all possible transformation rules, containing at a minimum those just discussed. This function can be extended when additional transformation rules are introduced into the system. We show the basic definition of the function here:

$$\begin{aligned} SR(HasAttr(\varepsilon, \alpha)) &= \{Eliminate(\varepsilon, \alpha)\} \cup \\ &\quad \{EgressTransfer(\varepsilon, \varepsilon_d, \alpha) | \\ &\quad (\varepsilon \in S \implies \varepsilon_d \in S) \wedge \\ &\quad (\varepsilon \in O \implies \varepsilon_d \in O) \wedge \\ &\quad \alpha \notin AA(\varepsilon_d)\} \\ SR(HasSubAttr(\varepsilon, \alpha)) &= \bigcup_{(\beta,\alpha) \in AH \wedge \beta \in AA(\varepsilon)} SR(HasAttr(\varepsilon, \beta)), \end{aligned}$$

where $AA(\varepsilon)$ is the set of attributes assigned to ε by the relation AA. You may have noticed that *Eliminate* is the functional complement of *Introduce*, and *EgressTransfer* is the complement of *IngressTransfer*. Thus, it should not be surprising that we use *Introduce* and *IngressTransfer* as the suggested transformations for negated reasons:

$$\begin{aligned} SR(NotHasAttr(\varepsilon,\alpha)) &= \{Introduce(\varepsilon,\alpha)\} \cup \\ &\quad \{IngressTransfer(\varepsilon,\varepsilon_s,\alpha) | \\ &\quad (\varepsilon \in S \implies \varepsilon_s \in S) \wedge \\ &\quad (\varepsilon \in O \implies \varepsilon_s \in O) \wedge \\ &\quad \alpha \in AA(\varepsilon_s)\} \\ SR(NotHasSubAttr(\varepsilon,\alpha)) &= \bigcup_{(\beta,\alpha)\in AH} SR(NotHasAttr(\varepsilon,\beta)) \end{aligned}$$

Finally, *SR* does not produce any suggestions for *IsNamed* reasons, since those reasons typically point to underlying problems in the model (presence of a banned entity) or low-level policy. Neither problem lends itself to an automated solution in our system.

Suggestion Prioritization.

As we will explain shortly, our tool presents potential transformations that will resolve constraint violations to policy administrators. Unfortunately, an overwhelming number of possible transformations may be generated in large access control models, causing policy administrators to abandon the tool or make non-optimal decisions. Thus, we first prioritize reasons in justifications for violations in descending order according to the frequency with which they appear in all the justifications. Then, for each of these reasons, we prioritize the possible transformations and present the most likely transformations to administrators first.

Unfortunately, it is not possible to formulate a general prioritization scheme; a comparator must be defined that explicitly handles each type of transformation in the system:

$$CompareSuggestion : Transformers \times Transformers \rightarrow \{first, second\},$$

where a result of *first* indicates that the first transformation rule should be given precedence, and *second* indicates that the second transformation should be given precedence.

Again, this comparator should be extensible and accommodate the addition of new transformations, but we explain how it will operate on the standard transformation types here.

Introductions and eliminations will always be given precedence over transfers, since they do not interact with any entity in the system except the one identified in the reason. Thus, they are less likely to introduce new violations.

Ingress and egress transfers need only be prioritized among themselves, since both types of transfers will never be suggested for a single reason (besides, they are semantically equivalent).

In a well-defined access control model, attributes will typically be transferred between entities with similar sets of attributes, since these attributes should represent the similarity of the entities. In our privacy policy example, the "customer email list administrator" attribute should most likely be transferred to another entity that already possesses attributes similar to those possessed by the source entity. In this example, someone who already possesses the IS departmental attribute is a likely candidate. Thus, we quantify the similarity between entities specified in a transfer transformation and prefer transfers between similar entities. More formally, we evaluate the following function on the two entities in the transfer and prefer transformations that result in higher values for the function:

$$EntitySimilarity : Entities \times Entities \rightarrow \mathbb{R},$$

where

$$\begin{aligned} &\forall \varepsilon_1, \varepsilon_2.\ EntitySimilarity(\varepsilon_1,\varepsilon_2) = \\ &|\{\alpha | \alpha_1 \in AA(\varepsilon_1) \wedge \alpha_2 \in AA(\varepsilon_2) \wedge \\ &\quad (\alpha_1,\alpha) \in AH \wedge (\alpha_2,\alpha) \in AH\}|, \end{aligned}$$

and AA and AH are drawn from the access control model.

Of course, this scheme could be greatly improved by considering how relevant attributes are to each other, so that in our example the IS attribute is given more consideration than other attributes when transferring the email list administrator attribute. Unfortunately, this may require the policy administrator to specify these relationships, although they could possibly be constructed by analyzing the correlation between attributes assigned to each object in a representative model.

4. IMPLEMENTATION

To evaluate the constructs discussed in this paper, we implemented a prototype access control engine. Several requirements helped direct the system's design. First, it must be straightforward to design and encode policies, models, and constraints. Second, it must be easy to evaluate the access control model using those constraints, and then to implement transformations to bring the access control model into conformance. Finally, we needed a full-featured foreign language API to demonstrate the usefulness of our access control engine in a sophisticated, object-oriented security application.

There are now several logical programming languages available that could have potentially satisfied many of our requirements, but we settled upon Prolog due to its maturity and popularity [17]. We use the SWI-Prolog interpreter (`swi-prolog.org`) because it provides a sophisticated foreign language API, is freely available, and has respectable performance [8].

Our system implements the major logical constructs discussed previously, including support for context-aware policies. The foreign language API supports bi-directional data transfer, to allow external programs to query the ADF, and to allow permission predicates to utilize contextual information from external sources.

Our prototype system can serve at least two different purposes. First, it can be used to ensure that policies conform to a set of constraints representing business requirements. However, it can also be used to evaluate the effects of business requirement changes on existing policies. To perform this evaluation, the administrator should first ensure that the affected policy instantiations are compliant with the original constraints. Then, they should re-evaluate those instantiations using the new constraints. The violations discovered by the system will inform the administrator of what new constraints will necessitate changes in the policy instantiations, which can then be used to evaluate the practical effects of the constraint changes on the legacy policies.

4.1 Access Control Policies

Policies must be written in Prolog as a number of predicates. Each predicate must have a *head* (Prolog terminology referring to the signature of the predicate) like the following: `permitted(Subj, Obj, Act, Ctx, Just)`, where `Subj` is a subject, `Obj` is an object, `Act` is the action that is to be performed by `Subj` on `Obj`, `Ctx` is the current context of the system, and `Just` is an output *variable* to which a justification structure will be assigned whenever the rule is satisfied. The following rule is a fairly complex and illustrative example. It specifies that professors are allowed to access their secretaries' resources:

```
permitted(entity(subject, ProfId, _), Obj,
```

```
        Act, Ctx, Just) :-
  justification_none(prof_secretary_res, JN),
  is_subject(Sec),
  jb(subject_has_attr(secretary(ProfId),
     Sec), JN, J0),
  permitted(Sec, Obj, Act, Ctx, J1),
  jb_join(J0, J1, Just).
```

Recall that in Prolog all of the terms separated by commas after the head and reverse-implication symbol (`:-`) must be satisfied for the entire predicate to be satisfied. Multiple predicates with the same name and number of arguments are joined into a logical disjunction, so that the entire formula can be satisfied if at least one predicate in the procedure is satisfied. Capitalized terms represent variables, whereas lowercase terms are simple values. Lowercase terms written in function notation are structures, such as `secretary(ProfId)`. Notice that structures may contain variables. We now describe each line individually, since this rule uses some syntax that we have not yet introduced:

`permitted(entity(subject, ProfId, _), Obj, Act, Ctx, Just)`: The first argument uses Prolog's ability to decompose structures within predicate heads. In our system, both subjects and objects are represented using `entity` structures. The first field of the structure indicates whether the entity is a subject or an object. The second field identifies the entity, and the third field may contain the set of attributes associated with the entity. It is not necessary to populate the final field in normal usage, since helper predicates are provided to populate that field whenever it becomes necessary. In this case, we are only interested in the identity of the person who must be a professor for the predicate to be satisfied, so we extract the ID from the entity and refer to it as `ProfId`.

`justification_none(prof_secretary_res, JN)`: Create an empty justification structure assigned to `JN` that assigns the name `prof_secretary_res` to the predicate. The name is simply used to record in a human-understandable format which predicates played a role in deciding to allow this access if it is in fact permitted.

`is_subject(Sec)`: Due to the backtracker in Prolog, this single line will actually enumerate all subjects in the entity database, unifying `Sec` with them individually. This line does not play a significant role in deciding whether to permit the access, so it does not contribute to the justification structure.

`jb(subject_has_attr(secretary(ProfId), Sec), JN, J0)`: The inner part of this line, `subject_has_attr(secretary(ProfId), Sec)`, is only satisfied when `Sec` is the secretary of the professor identified by `ProfId`.

The outer part of this line, `jb(..., JN, J0)`, serves two purposes. First, it interprets the inner part of the line as just discussed. If the inner term is satisfied, the outer term will then add a reason representing that inner part to the justification, unifying `J0` with the new structure. The outer term is satisfied iff the inner term is satisfied.

Seven generic reasons of four major types are supported. They correspond to the formal reasons discussed in previous sections:

(\)has_attr(Type, Id, Attr): Specifies that the decision was based on the fact that the entity with the given `Type` (`subject` or `object`) and `Id` was associated (or not associated if the negation symbol \ is used) with the attribute `Attr`.

(\)has_subattr(Type, Id, Attr): Specifies that the decision was based on the fact that the entity with the given `Type` and `Id` was associated (or not associated) with a sub-attribute of `Attr`.

(\)is_named(Type, Id): Specifies that the decision was based on the fact that an entity with the given `Type` is or is not identified by the given `Id`. Typically, all three of the reasons just presented identify one of the entities passed into the permission predicate producing the justification, but they can also be applied to other entities in the system.

satisfied(Pred): This is a default reason that encapsulates a predicate that does not correspond to any of the other reasons just described.

In our example, the `jb` predicate adds the `has_attr(subject, Sec, secretary(ProfId))` reason to the justification, where all of the variables (capitalized terms) are replaced with the specific terms that cause the larger term to be satisfied.

The next line in our predicate, `permitted(Sec, Obj, Act, Ctx, J1)`, is a chained reference to the permitted predicate, to determine whether the secretary just selected has access to the resource in question. Notice that this invocation generates its own justification.

The final line in the predicate, `jb_join(J0, J1, Just)`, simply merges the two justifications, `J0` and `J1`, to form the single justification `Just` that is finally returned to the process invoking the predicate. The merging process constructs a justification structure containing a union of the labels and reasons from both justifications being merged.

Contextual Information.

The previous rule did not explicitly deal with any contextual information. The next rule demonstrates how contextual information can be used to make access decisions. This example is drawn from an academic environment, and centers on an admissions committee comprising both faculty and students. It specifies that students on the admissions committee are permitted to enter rooms designated for admission committee meetings only after the context indicates that the system is at least 50% confident that all professors on the committee have entered the room. The final confidence level is an unweighted average of the confidence levels that each individual professor on the committee is in the room. Thus, if the system is 100% confident that half of the committee members are present the predicate will be satisfied, and so forth.

```
permitted(Subj, Room, enter, Ctx, Just) :-
  justification_none(adm_comm, JN),
  jb(object_has_subattr(adm_comm_rm, Room),
     JN, J0),
  jb(subject_has_subattr(adm_comm_mbr, Subj),
     J0, J1),
  (
    jb(subject_has_subattr(professor, Subj),
       J1, Just)
  ;
    jb(adm_comm_meeting(Room, Ctx),
       J1, Just)
  ).
```

We explain only the significantly different aspects of this predicate here:

`object_has_subattr(adm_comm_rm, Room)`: This predicate only deals with admission committee rooms. Thus, this term filters out all irrelevant rooms by only accepting objects with an attribute specifying that the room is a designated meeting place for admission committee members.

`subject_has_subattr(adm_comm_mbr, Subj)`: In a similar spirit, this predicate filters out subjects that are not members of the admission committee.

`subject_has_subattr(professor, Subj)`: This predicate is satisfied if the subject in question is a professor. It is one of two predicates in a disjunctive compound term. One or both of this predicate and the following one must be satisfied for the rule as a whole to be satisfied. The intention of this arrangement is to allow professors on the admission committee to access the room unconditionally, but perform further checks otherwise.

`adm_comm_meeting(Room, Ctx)`: This is a custom predicate that processes the current context to determine if `Room` is currently occupied by a number of admission committee members. For full details, examine our sample policy included in the software distribution at seclab.uiuc.edu/policymorph. One important provision included in the predicate is special handling for the context value used during constraint checks. Obviously, constraint checks occur in an isolated environment, so it is not usually possible to predict what context values will be used when the access control system is deployed. Thus, we use a special context structure value, `context(some)`, when performing constraint checks. All predicates that perform context processing must be satisfied if some context value could satisfy the predicate. By convention, all permission predicates that incorporate context must use satisfied context checks to provide additional permissions beyond those normally provided with unsatisfied context checks. This permits us to perform constraint checking on the most permissive access control system supported by the policy instantiation in question.

To illustrate this, consider how our permission predicate would behave if the term `adm_comm_meeting(Room, Ctx)` were replaced with the term `\+(adm_comm_meeting(Room, Ctx))`, where the standard Prolog `\+(...)` predicate is only satisfied when the predicate provided as its sole argument is unsatisfied. For the overall permission predicate to be satisfied, the subject must be a professor, or an admissions committee meeting must *not* be in progress. In this case, the `adm_comm_meeting` predicate will be satisfied when the `context(some)` value is passed to it, as before, and this will actually result in the most restrictive permission predicate available.

Arbitrary contextual information can be considered in policies, from any conceivable source. The only fixed structure imposed on the context is that it take the form of a dictionary structure (set of key-value pairs where the keys are unique). We will discuss specific types of context below, such as instant messenger presence status generated by individuals in a building.

4.2 Attribute Hierarchies

Besides the main permission rules, policies also contain predicates specifying the attribute hierarchy. Since attributes can be parameterized, and since the declarations are true Prolog predicates, sophisticated hierarchies can be constructed with very little effort:

In each predicate shown below, the first argument specifies the type of the entities affected by the declaration, the second argument specifies the attribute that is lower in the hierarchy, and the final argument specifies the parent attribute. As with any Prolog predicate, an underbar indicates a *don't care*, meaning any term can be inserted in its place, and any capitalized term is a variable. For example, `entity_subattr(subject, professor(cs421), ta(_))` would indicate that `professor(cs421)` is a sub-attribute of any `ta` attribute, while `entity_subattr(subject, professor(X), ta(X))` could indicate that `professor(cs411)` is a sub-attribute of `ta(cs411)`. Policies are responsible for interpreting the attribute hierarchy. We have included an example of a subject and object attribute hierarchy here:

```
entity_subattr(subject, secretary, staff).
entity_subattr(subject, professor, staff).
entity_subattr(subject, secretary(_),
                        secretary).
entity_subattr(subject, ta(_), ta).
entity_subattr(subject, student(_), student).
entity_subattr(subject, professor(_),
                        professor).
entity_subattr(subject, ta(X), student(X)).
entity_subattr(subject, professor(X), ta(X)).
entity_subattr(object, class_rm(_),class_rm).
entity_subattr(object, secretary_rm(_),
                        secretary_rm).
```

4.3 Constraints

One of the most important features of our tool is its policy validation engine that checks constraints against a specific policy instantiation and uses its results as inputs for the transformation engine.

As we noted previously, it is often the case that systems have been running with legacy procedures and access control policies for years, before business rules are suddenly changed. Ideally, before such changes are ratified, the effect upon legacy access control policies and models should be evaluated. Our tool can evaluate business rule changes by discovering violations of those new rules in existing policy instantiations. If the business rule changes are still enacted after this evaluation, our tool can help administrators discover and resolve all violations that are introduced.

First, let us consider a simple constraint for enforcing Bell-LaPadula confidentiality properties on a multi-level secure system:

```
uncleared_access(Just) :-
  is_subject(Subj),
  is_object(Res),
  object_has_attr(classified(Level), Res),
  \+(subject_has_subattr(cleared(Level), Subj)),
  permitted(Subj, Res, enter, _, Just).
```

Just as we did for our logical policy predicates, let us explore this constraint one line at a time:

`uncleared_access(Just)`: Unlike policy statements, we do not use a standard naming convention for constraints. Instead, the names of each constraint predicate should be meaningful to a human and we explicitly list constraints by specifying their names in a separate `policy_constraint` predicate, so that the policy validation engine can efficiently enumerate all constraints. Again, the `Just` variable will be unified with a justification for whatever policy violation is detected by the constraint.

Just like we do in our first policy predicate, we enumerate all subjects in the policy system using the `is_subject` predicate. We also enumerate all objects in the system using the `is_object` predicate.

Next, we filter out the objects that have classification attributes with the `object_has_attr(classified(Level), Res)` predicate. `Level` is unified with the actual classification in use (e.g. `secret` or `top_secret`).

For the constraint to be violated, we must locate a subject that can perform the `enter` operation on the classified resource, but does not possess the required clearance. This bit of logic is encoded as `\+(subject_has_subattr(cleared(Level), Subj))`, assuming that the `cleared(Level)` attribute indicates that the subject has been granted a clearance at the indicated level, and that clearance levels are arranged appropriately in the attribute hierarchy, so that `top_secret` is a child of `secret`, etc.

4.4 Policy Validation and Transformation

In this section we describe how potential violations of high-level constraints by low-level policies detected as shown in the previous section are interactively resolved by our tool. The basic interactions are depicted in Figure 1.

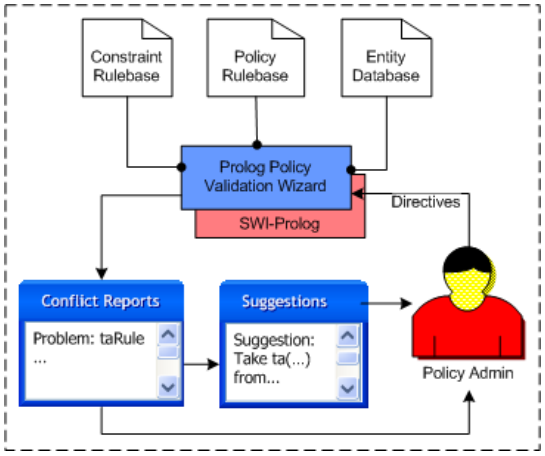

Figure 1: Interactive policy validation and transformation.

Three distinct input files are used to instantiate the system. The access control model is split between the *Policy Rulebase* and the *Entity Database*. The attribute declarations and attribute hierarchy are contained within the Policy Rulebase, along with the permission rules. The subject and object declarations and attribute assignment are contained within the Entity Database. The Entity Database contains declarations that resemble Prolog sentences, but they are actually encoded in a simplified format that is programmatically expanded to form valid Prolog declarations, to enhance ease of use and encapsulation. Finally, the *Constraint Rulebase* encodes high-level constraints. At the conclusion of the policy validation and transformation process, a modified entity database may be produced and saved as a new file.

To begin the validation process, all of the violations for a particular constraint are collected. Then, the individual reasons within the justifications for those violations are extracted and sorted according to the frequency with which they occur in the justifications. Each of these reasons is handled in turn, until the administrator believes that the constraints should be re-validated to see which violations have been eliminated and which new violations have been introduced.

The processing for each reason entails generating a list of possible transformations, prioritizing those transformations as described previously, and then presenting that list to the administrator. Then, the administrator is permitted to select one of the suggested transformations and either apply it immediately or evaluate its effects on the ADF. To evaluate the effects on the ADF, the system generates every possible access request using both the old and new models and presents the differences in the decisions to the administrator. Of course, this exhaustive evaluation takes a long time in large models, but is provided as a convenience when it is practical.

We have not yet actually implemented the transformation prioritization scheme in our current tool, but we plan to do so in the future. Prioritization is most helpful in access control systems with large entity databases.

The policy validation/resolution phase is complete when the administrator has resolved all violations or chooses to halt the process prematurely. More details are provided in our case study below.

5. CASE STUDY

In this section we demonstrate the usage of our system to control access to rooms within a Building Automation System (BAS) simulator called *Janus* (`seclab.uiuc.edu/janus`). The overall architecture of the system is depicted in Figure 2. Notice that the Constraint Rulebase used during policy validation is not used by the ADF.

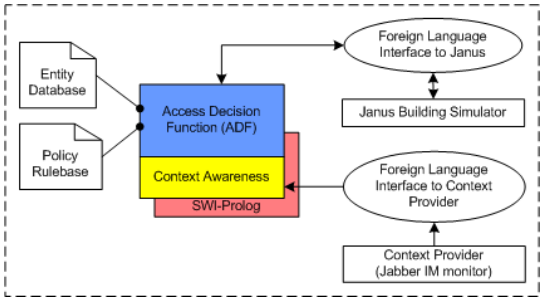

Figure 2: System architecture of PolicyMorph-regulated Janus BAS simulator.

5.1 Integration with Building Simulator

Originally, Janus was developed as a backend simulator for the Janus' Map [5] location detection system. It simulates parts of the building automation system actually installed in the Siebel Center for Computer Science at the University of Illinois at Urbana-Champaign (UIUC). Janus simulates rooms and doors installed in the building, as well as imaginary users of the building. In the original version of Janus, access to resources was based on a rudimentary discretionary access control system. It is written in Java and uses a backend database to store and access information about resources, individuals, and (static) access control policies.

As implied above, it was necessary for us to modify the Janus backend slightly to support our improved access control scheme. In order to evaluate access control decisions in real-time using our Prolog engine, we modified Janus to perform a dynamic call to our system whenever an access request is issued. As discussed previously, our Prolog kernel interfaces with Java and can provide real-time access decisions to Janus.

5.2 Integration with Context Provider

We integrated our system with the Jabber Instant Messaging (IM) system (`jabber.org`) to infer the locations of subjects in the access control system that are also Jabber users. In addition, we can infer contextual information from successful access requests to the rooms in the building. Therefore, we use context from a few sources:

Jabber presence status - Location inference: We provide support for mapping Jabber users to subjects in the access control model. Then, when a subject modifies their IM status, the system modifies its dynamic contextual information to indicate how confident it is that the subject is currently at the location occupied by the IM client. We use the *resource* variable that can be set in any Jabber client to identify a physical space represented in the access control system. Of course, this indicator can be spoofed, but it simply serves as an example. The context map contains entries formatted as `presence(SubjId, RoomId)-ConfLvl`, where `ConfLvl` is a floating-point number between 0.0 and 100.0 that indicates how confident the system is that the subject is currently in the room. We update the confidence level when the user changes their Jabber presence. For instance, if the presence changes from *Available* to *Idle*, we lose some confidence that the subject is still in the room.

Successful access requests: Whenever an entity successfully requests entrance to a room, we become more confident that they are in the room, and update the same confidence level that is updated by the Jabber location inference scheme.

Jabber presence status - Do Not Disturb: Jabber supports a special presence value: Do Not Disturb. Normally, that means that the user is present at their computer, but do not wish to chat. When this presence status is set, our system records the fact using a similar format to that shown above: `dnd(SubjId, RoomId)-X`, where `X` is set to `yes` when the subject wishes to not be disturbed, and `no` otherwise. See our *Experiment Integrity Protection* scenario for a usage example.

5.3 Example Scenarios

In this section, we identify representative security and constraint models for access control policies from the literature and demonstrate their realization in PolicyMorph. In the last part of this section, we also illustrate a dynamic, context-aware access control scenario.

Model.

The model we use in the following scenarios is an approximate representation of the Department of Computer Science in UIUC. The subjects include 60 faculty members, 28 postdoctoral researchers (referred to as postdocs), 300 grad students, and 20 secretaries. The objects include 20 class/seminar rooms, 12 public areas and print rooms, 20 research labs, and 120 offices for faculty, postdocs, secretaries and TAs. All the entities are assigned attributes as described in the previous sections. These attributes and the attribute hierarchy reflect the academic realm they model, and are incrementally introduced as necessary in the following discussion.

Separation of Duty Constraints.

Separation of duty (SOD) constraints are by far the most exhaustively explored set of constraints in the literature. Jaeger and Tidswell in [12] and Crampton in [7] provide the most comprehensive set of examples in the literature on SOD constraints. We pick a few non-trivial SOD scenarios which reflect the SOD requirements in our academic environment and show how they can be supported in our system. Other constraints for this environment could be based on Bell-LaPadula [3] and Biba [4] security models, which can be represented in our system as illustrated in 4.3.

Individuals within academic organizations occupy many different roles on a daily basis. For example, a Teaching Assistant (TA) for a particular class is usually also a student in several other classes. In the idealized world represented by typical RBAC systems, the individual is able to perform a full *context-switch* when switching between these distinct roles. In the real world, however, that simply is not possible. Even when the individual is working in their TA role, they will still be motivated to enhance their performance as a student. Consider TAs, Amber and Curtiss, who are assigned to work in the same room. If Amber is a TA for the operating systems class (CS523) in which Curtiss is a student, Curtiss may be tempted to take advantage of Amber's materials that are available in the room while she is away.

In this conflict of interest example, the following negative logical constraint will ensure that the desired properties for TA room assignments hold:

$$\forall s \in S, \neg\exists(c_0, c_1, r) \in (C \times C \times R).$$
$$(ta(s, c_0) \wedge taroom(c_0, r)) \wedge (enrolled(s, c_1) \wedge taroom(c_1, r)),$$

where C is the set of all courses, $R \subseteq O$ is the set of all rooms, $ta(s, c)$ holds when subject s is the TA for class c, $taroom(c, r)$ holds when r is the assigned TA room for course c, and $enrolled(s, c)$ holds when subject s is enrolled in course c.

If this constraint holds for the ABAC policy defined in the system, then it will never be true that any TA shares a TA room with another TA from one of the courses in which the first TA is enrolled. We encode this negative constraint as a rule in Prolog that will never be satisfied by a valid policy:

```
coi_ta_student(JF) :-
  justification_none(coi_ta_student, JN),
  is_subject(SubjA),
  is_subject(SubjB),
  jb(subject_has_attr(ta(ACrs), SubjA),
     JN, J0),
  jb(subject_has_attr(ta(BCrs), SubjB),
     J0, J1),
  ta_room(ACrs, Room, J2),
  ta_room(BCrs, Room, J3),
  enrolled(SubjB, ACrs, J4),
  jb_join(J2, J3, J23),
  jb_join(J23, J4, J234),
  jb_join(J234, J1, JF).
```

This overall rule makes use of some other "convenience" rules, whose definitions are omitted for clarity: *1)* `ta_room(Course, Room, JF)`: Satisfied when object `Room` is a designated TA room for the course identified by `Course`. *2)* `course_ta(Course, Ta, JF)`: Satisfied when the subject `Ta` is a designated TA for `Course`. *3)* `enrolled(Subj, Course, JF)`: Satisfied when subject `Subj` is a student in `Course`.

With our current policy and entity database, we encounter violations. Here is an excerpt from the output of our policy validator:

```
*** coi_ta_student found some violations:
justified by[coi_ta_student,enrolled,ta_room]:
  has_attr(object,room(rm4023),ta_room(cs461))
  has_attr(object,room(rm4023),ta_room(cs523))
  has_attr(subject,curtiss,student(cs523))
  has_attr(subject,curtiss,ta(cs461))
  has_attr(subject,amber,ta(cs523))
```

This output informs the administrator that the validator detected a violation of the `coi_ta_student` constraint. It also explains why the constraint was violated, and thus how the violation can be resolved. The labels in square brackets are the labels of the permission predicates that were used to grant the violating access. The reasons following those labels show that the violation occurred because room 4023 is the TA room for both the CS461 and CS523 courses, and because Curtiss is both a TA using that room and the student of another TA using the same room, Amber.

What follows is a subset of the transformations that the system suggests to resolve this violation. Note that any or all of the reasons used to generate these suggestions may appear in violations generated by other constraints or permission rules. Thus, the number of transformations ultimately suggested to the administrator is not directly related to the size or complexity of the policy or its constraints, although it is directly related to the number of entities in the system. Some obvious user interface improvements could be applied to manage the complexity associated with large entity sets.

```
remove ta(cs461) from the subject curtiss
transfer ta(cs461) to amber
transfer ta(cs461) to corwin
transfer ta(cs461) to alice
...
remove student(cs523) from the subject curtiss
```

```
transfer student(cs523) to alice
...
remove ta(cs523) from the subject amber
transfer ta(cs523) to curtiss
transfer ta(cs523) to corwin
transfer ta(cs523) to alice
...
remove ta_room(cs523) from the object room(rm4023)
transfer ta_room(cs523) to room(rm4001)
transfer ta_room(cs523) to room(rm4002)
...
remove ta_room(cs461) from the object room(rm4023)
transfer ta_room(cs461) to room(rm4001)
transfer ta_room(cs461) to room(rm4002)
...
```

Implementing any one of these transformations would invalidate the justification for this violation.

As another example, consider a graduate student Bob that graduates with a Ph.D. degree under Dr. Carlson. However, after graduation, he accepts a position as a postdoctoral researcher under Dr. Smith. There exists an obvious Static Separation Of Duty (SSOD) rule [12] in the department that prohibits users from being assigned both the student and postdoc attributes, and there should also exist only one advisor for each student or postdoc in the system. The administration could inadvertently violate these rules by adding the new `postdoc` and `advised_by` attributes (and hence respective privileges) to Bob's account without removing his old status and adviser information. We show how to encode the constraints in the system and how the policy validator discovers and reports these violations below:

```
sod_grad_postdoc(Just) :-
  justification_none(sod_gradpostdoc, JN),
  is_subject(Subj),
  jb(subject_has_attr(gradstudent,Subj),
     JN, J0),
  jb(subject_has_attr(postdoc,Subj),
     J0, Just).

sod_one_advisor(Just) :-
  justification_none(sod_one_advisor, JN),
  is_subject(Subj),
  jb(subject_has_attr(advised_by(Prof1),
                      Subj), JN, J0),
  jb(subject_has_attr(advised_by(Prof2),
                      Subj), J0 ,J1),
  jb(Prof1 \= Prof2, J1, Just).
```

Just as it did in the TA room assignment scenario, PolicyMorph's policy validator successfully identifies the constraint violations and suggests appropriate attribute removals or transfers. We omit the output for the sake of brevity.

Context Sensitive Access Restriction.

Experiment Integrity Protection. In a research-oriented academic environment, a professor may decide to give his students and secretary access to his office. Under normal circumstances, this access would be justified by the need for students to access the books, journals, or presentation laptops often stored in the professor's office. However, temporary circumstances may arise during which such access should be denied. We explain one such scenario here, and use it to demonstrate the ability of PolicyMorph to consider external contextual information when making access decisions.

Suppose the professor must run a light-sensitive experiment in his office over the weekend. He would not want any of his students to enter the room while this experiment is being performed, since the light flooding through an open door would invalidate the results. Such access restrictions can be easily enforced using our context-aware environment. By adding the required rules to our policy, we can dynamically enforce the policy in the following manner:

Access to the professor's office will be restricted so that his students are unable to enter whenever he sets the standard "Do Not Disturb" flag in his Jabber instant messenger. Of course, he will set this flag when he initiates the sensitive experiment. Once the experiment is completed, he will remove the flag and his students will immediately be able to access his office once again. To encode this in Prolog, the following rule is used:

```
permitted(Stu, Office, enter, Ctx, Just) :-
  justification_none(dnd_stu_access, JN),
  Office=entity(_, OffId, _),
  jb(object_has_subattr(office(Prof), Office),
     JN, J0),
  jb(subject_has_subattr(advised(Prof), Stu),
     J0, J1),
  jb(dnd_flag_cleared(Prof, OffId, Ctx),
     J1, Just).

dnd_flag_cleared(_, _, context(some)) :- !. *

dnd_flag_cleared(Prof, Office, Ctx) :-
  context_lookup(dnd(Prof, Office)-X, Ctx),
  !,
  X = no.

dnd_flag_cleared(_, _, _).
```

The rule above basically says that students of the professor are allowed to access his office only if the "Do Not Disturb" flag is cleared. The starred line is necessary to accommodate the policy validator. It ensures that the policy validator operates on a maximally permissive policy.

6. RELATED WORK

Constraint conflicts were clearly distinguished from policy conflicts in [11]. That work also showed how constraints could be expressed in propositional logic and used to evaluate SELinux Type Enforcement policies. Furthermore, it showed several ways in which the policy could be modified to resolve constraint conflicts. Our system is applicable to a completely different policy language, and modifies the access control model rather than the policy itself. Additionally, we have created a powerful tool to automatically implement our conflict resolutions.

An alternative constraint expression language was put forth in [12]. It expresses constraints graphically, as binary relationships between sets of subjects and objects, etc. Our system has similar capabilities to this alternative, however, it uses first-order logic for this purpose, and additionally provides interactive suggestions to remedy the constraint violations.

Other work explored the various types of SOD constraints that are useful in the context of RBAC systems, as well as more general constraints, which it collectively (with SOD constraints) refers to as *authorization constraints* [7]. It represents these constraints using set-based notation. Our system also supports generalized authorization constraints, but expresses them differently and provides automated support for constraint violation resolution.

Another access control system based on first-order logic was presented in [2]. The system supports an extended RBAC model with support for positive and negative policy statements, temporal constraints and authorization constraints. It does not provide suggestions on how to resolve the violations. However, the system uses Constraint Logic Programming (CLP) to express policies and constraints, so it is possible to efficiently express numeric ranges for

arithmetic variables. This is a feature that may prove beneficial to PolicyMorph if implemented in the future.

Our system uses justifications to encode the reasons that a particular positive access control decision was made. These justifications are used to audit the normal operation of the system and also to resolve constraint violations. Other systems have been developed that generate reasons for negative access control decisions, such as *Know* [13]. However, these reasons are presented to the subjects themselves, rather than being reserved for those with access to the policy administrative system and audit log. Thus, Know must take special precautions to ensure that subjects are unable to manipulate the system and determine significant portions of the access control policy from the denial justifications they receive. PolicyMorph and Know provide two different examples of how access decision justifications can be useful to both users and administrators.

Margrave is a system for analyzing access control policies written in a subset of XACML [9]. It allows authorization constraints to be encoded and validated, and also provides change analysis functionality, so that all the permission changes introduced by policy changes can be enumerated. Our system provides similar functionality for a logical policy language, and also provides an interactive environment that suggests resolutions to constraint violations.

An Attribute-Based Access Control framework constructed in Constraint-Logic Programming was motivated and presented in [18]. It has a good discussion of why ABAC is often preferable to RBAC and other models. The logical model used in [18] is focused on sets of attributes and services, rather than subjects and objects, although it provides similar capabilities to our underlying policy system. The paper also has a discussion of policy optimization techniques that could be applied to optimize our policies. Again, it lacks resolution strategies.

7. CONCLUSIONS AND FUTURE WORK

In conclusion, PolicyMorph advances the state-of-the-art in logical ABAC access control policy and constraint models by introducing an interactive policy validation and transformation methodology that leverages the knowledge and preferences of a human administrator while still assisting the administrator in the decision process and providing comprehensive analysis of transformation effects. We have demonstrated the utility of our system using realistic building automation scenarios drawn from an academic setting and integrating dynamic contextual information.

Since our policy administration tool is fully functional, in the future we would like to develop a graphical interface to assist with iterative access control policy and model design and maintenance.

We would like to thank Sundeep Reddy for participating with the authors on this project. Nikita Borisov, Michael Reiter, Xinming Ou, Marrianne Winslett, Roy Maxion, Anupam Datta, and Matt Bishop all provided helpful comments on our project that influenced this paper. This work was partially supported by: NSF CNS05-5170 CNS05-09268 CNS05-24695, ONR N00014-04-1-0562 N00014-02-1-0715, and a grant from MacArthur Foundation. Michael LeMay was supported on an NDSEG fellowship from the AFOSR.

8. REFERENCES

[1] A. Antón, J. Earp, D. Bolchini, Q. He, C. Jensen, and W. Stufflebeam. The Lack of Clarity in Financial Privacy Policies and the Need for Standardization. *IEEE Security & Privacy*, 2(2):36–45, 2004.

[2] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546, 2003.

[3] D. Bell and L. Lapadula. Secure computer systems: Mathematical foundations (volume 1). Technical report, 1973.

[4] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, Apr. 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[5] J. P. Boyer, K. Tan, and C. A. Gunter. Privacy sensitive location information systems in smart buildings. In *SPC '05: Proceedings of the 3rd International Conference on Security in Pervasive Computing*, 2005.

[6] T. Breaux and A. Antón. Deriving Semantic Models from Privacy Policies. *Proc. IEEE 6 thWorkshop on Policies for Distributed Systems and Networks, Stockholm, Sweden*, pages 67–76, 2005.

[7] J. Crampton. Specifying and enforcing constraints in role-based access control. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 43–50, New York, NY, USA, 2003. ACM Press.

[8] B. Demoen and P. Nguyen. Odd Prolog benchmarking. *KU Leuven, CW report*, 312, 2001.

[9] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.

[10] J. Halpern and V. Weissman. Using first-order logic to reason about policies. *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 187–201.

[11] T. Jaeger, R. Sailer, and X. Zhang. Resolving constraint conflicts. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 105–114, New York, NY, USA, 2004. ACM Press.

[12] T. Jaeger and J. E. Tidswell. Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Secur.*, 4(2):158–190, 2001.

[13] A. Kapadia, G. Sampemane, and R. H. Campbell. Know why your access was denied: regulating feedback for usable security. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 52–61, New York, NY, USA, 2004. ACM Press.

[14] M. J. May, C. A. Gunter, and I. Lee. Privacy APIs: Access control techniques to analyze and verify legal privacy rules. In *Computer Security Foundations Workshop (CSFW '06)*, Venice, Italy, July 2006. IEEE.

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

[16] J. G. Stell. A framework for order-sorted algebra. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 396–411, Reunion Island, France, September 2002.

[17] L. Sterling and E. Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994.

[18] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 45–55, New York, NY, USA, 2004. ACM Press.