Formal Verification of Cross-Domain Access Control Policies Using Model Checking

Mark C. Reynolds
Azer Bestavros

Boston University
Department of Computer Science
{markreyn,best}@bu.edu

Abstract. In assigning access permissions to users, formal policies fill a key role in mapping access from user attributes to resources. In a single domain a policy should be both sound (no false positives) and also complete (no false negatives). When policies from different domains are combined, several issues can arise. The most immediate issue is mapping potentially disparate attributes, roles and sources from one domain model to the other. Once this has been addressed, however, a more serious problem can arise: even if the domain policies are individually self-consistent, they may not be mutually consistent. To date, the typical approach has been a brute-force enumeration of all possible aggregate states in what might be termed "analysis of composition". This approach is manifestly not scalable. This paper describes first steps toward a new "composition of analysis" approach. This approach uses the Alloy modeling language to cast the resource accessibility problem as a constrained graph reachability problem. An Alloy model is created for each domain policy, and these models are then composed in a sound and efficient manner. This paper will describe results for the two domain and three domain cases, and sketch a general, scalable approach that can be applied to compositions of arbitrary complexity.

Key words: Alloy, policy modeling, compositional analysis

Formal Verification of Cross-Domain Access Control Policies Using Model Checking

Mark C. Reynolds   Azer Bestavros
Department of Computer Science
Boston University
{markreyn,best}@bu.edu

1. BACKGROUND AND PROBLEM STATEMENT

Access control is typically achieved through the enforcement of a set of access control *policies* that spell out for each one of a set of resources whether or not entities (individuals or programs) with a given set of attributes are entitled to access the resource. An important consideration for any access control system is the ability to verify its *soundness* – that the set of policies is consistent with one another, in the sense that if access to a resource is granted (denied) through the application of a subset of policies, then the application of any other subset of policies will not result in a contradictory decision. An access control system is said to be *safe* if it is possible to establish the soundness of the underlying access control policies.

Prior work [1] has shown that safety of an access control system is not decidable for arbitrary policy forms (*a.k.a.* access control models) such as Lampson's access matrix [2]. As a result, to render the safety verification problem decidable, most practical access control models restrict the range of policies that can be specified. Practical access control models – *e.g.*, Bell-LaPadula [3] – are even less expressive since general constraint expression languages are far too complex for typical administrators to use properly. Within a single organization, the set of policies that make up a *safe* access control system are expressed over a set of attributes that are appropriate for that organization. The safety of such an access control system implies that no contradictory grant/deny access decisions could be reached through the application of different subsets of policies within a single organization (which we call a *domain*).

In many emerging settings, it is often the case that *multiple* organizations may need to share resources in a way that adheres with the independently safe access control system of *all* organizations. This is done by defining a mapping (correspondence) between the set of attributes at the various organizations. While such a mapping enables policies from one domain to be interpreted over attributes from another domain, the resulting cross-domain access control system may be *unsafe*. In other words, the composition of individually-safe access control systems may lead to contradictory grant/deny decisions. Thus, in addition to defining a mapping between the attributes of multiple organizations, it is necessary to verify that the resulting composition of access control policies is sound.

Clearly, one approach to verifying the safety of cross-domain access control is to treat the resulting system as a *new* one, whose safety needs to be analyzed from scratch. This "analysis-of-composition" approach fails to leverage the fact that the constituent access control systems are known to be individually safe. Rather than analyzing the composition, it would be much more desirable to adopt a "composition-of-analysis" approach. Towards that end, in this paper we present a framework for the modeling and (automated) analysis of access control policies.

The framework we present in this paper uses the Alloy modeling language [4] to cast the resource accessibility problem as a constrained graph reachability problem. An Alloy model is created for each domain policy, and these models are then composed in a sound and efficient manner.

Conceptually, we view the composed system as defining a state transition graph. A node in the graph represents a state of the system, whereby a combination of attributes allow or prohibit access to a (possibly empty) set of resources. A directed edge in the graph represents an allowable transition (dictated by a specific policy rule) from one state to another. Within our framework, it is possible to prove that a composed system is *not* safe if it is possible to find a path in our conceptual graph connecting some initial system state (graph node) to another state in which some resource is both granted and denied. The existence of such a path constitutes a counter-example proof that the composed system is unsafe. The composed system is deemed *safe* if no such path exists, subject to any arbitrary upper-bound on the number of policies that can be sequentially applied. In addition to establishing the soundness of the composed system, using our framework, it is also possible to establish the completeness of the composed system, by showing that starting from any initial state, there is path (of length that does not exceed an arbitrary upper-bound) which reaches a state wherein access to a resource is either granted or denied.

The remainder of this paper is organized as follows. In Section 2, we introduce the policy analysis problem, and describe the conceptual state transition model used for formal verification of soundness and completeness properties. In Section 3, we provide a brief summary of the salient features of model checking using Alloy. In Section 4, we propose a specific template for expressing the verification of soundness and completeness properties as Alloy model checking problems. Specifically, we show how to derive the Alloy declarations, initializers, assertions, and invariants. In Section 5, we describe results for a case study involving privilege escalation, and we sketch a general, scalable approach that can be applied to compositions of arbitrary complexity. We conclude in Sections 7 and 8 with a summary of findings and on-going work.

2. USE OF MODELING FOR POLICY ANALYSIS

There are many possible approaches to performing policy analysis in a single domain, and across multiple domains. One goal of this paper is to demonstrate that model checking is an approach that provides unique advantages in terms of the ability to reason about the resulting model. This section will provide some notional arguments as to why this approach has been selected, while the remainder of the paper will present a concrete description of the way in which these notions have been realized, and the advantages that this realization provides.

At the highest level of abstraction, a policy system can be described as a set of rules that determine who has access to what resources under what conditions. The "who" in this statement could be a human user, or it could be a software agent acting on behalf of a user. More generally, we conceive of this actor as an abstract but definite entity that possesses zero or more attributes. The set of attributes may change over time. For example, a human user who is logged out of a particular computer system can be construed as possessing only credentials (a username coupled with some authentication information, such as a password or passphrase). These credentials possess only the potential for access to further resources; it is only when they have been validated (by the login process, for example) that these further resources (access to a home directory, binding to an email account identity, etc.) are actually granted. The act of successfully logging into a system represents a state transition from the state of "possessing allegedly valid credentials" to the state of "possessing (currently) valid credentials", with all the extra resources that the latter state entails. As a first order approximation, the act of failing to successfully login leaves the user in the same state (although a more fine grained representation would actually cause a transition to a different state that took into account the failed login).

A policy system can thus be viewed as a graph of states. Each node on the graph is labeled with the **resources** associated with that particular state. The actor may be viewed as being associated with a distinguished node with the additional label **Current**. The actor itself is bound to a set of **attributes** A. These attributes may change depending on the association between the actor and the current state (node), as well as the actor's traversal history along the state graph. Finally, each edge of the graph is associated with a **policy**. This policy P is a boolean–valued function $P(Current, Next, A, H)$. If the value of this predicate is $True$ then the transition to the $Next$ state is permitted, and the resources associated with that state may be acquired. If the value of this predicate is $False$ then the transition to the $Next$ state is not permitted, and that states resource may not be acquired (at least, not by means of this particular path). The variable H in the expression above is used to denote the fact that in the most general formulation there may be *hidden* variables that contribute to the evaluation of the predicate. For example, if a user's login credentials are derived from a hardware security token, then the current time would be a hidden variable that is part of the evaluation of a $CanLogIn()$ predicate. While the formalism developed in this paper does allow for the presence of hidden variables in policy predicates, the models that are developed in this work do not use them.

A valid policy system should be both <u>sound</u> and <u>complete</u>. The soundness property expresses the fact that an actor is only granted access to resources whose accessibility is implied by the attributes that the actor possesses. The completeness property expresses the fact that if a resource R is accessible by virtue of the attributes possessed by an actor, then there is at least one sequence of state transitions (from some distinguished initial state) to a state that is bound to R. A violation of soundness would imply that a policy system permitted an actor access to a resource R via some set of state transitions that should have been denied. A standard example of violation of soundness is privilege escalation, in which an actor acquires security credentials of broader scope than those implied by the login context of that actor (for example, an unprivileged user obtaining root or SYSTEM access). A violation of completeness would imply that a policy system denied an actor access to a resource R that should have been allowed. A standard example of violation of completeness is denial of service (DOS). A DOS typically acts by consuming all instances of an ostensibly available resource R such that when an actor attempts to acquire an instance of R it is denied by virtue of no remaining instances being available (an inability to connect to a remote service because all available network sockets have been allocated, for example).

In both cases, these violations can be expressed in terms of reachability in the state transition graph. If a forbidden state (a state holding a forbidden resource) can be reached, that is a violation of soundness. If a permitted state cannot be reached, that is a violation of completeness. What determines reachability? What determines what is permitted and what is forbidden? It would be a tautology to say the set of policy predicates P themselves determine reachability, because then everything that was reachable would be permitted, and nothing that was unreachable would be forbidden, and thus no violations could ever occur. In fact, there are a set of rules distinct from the policy system as embodied by the set of predicates P. These rules can be expressed as constraints on possible attribute–resource combinations. For example, in the case of login credentials one can posit two resource containers *unpriv* and *priv* and require that $\{\forall R \mid R \in priv \oplus R \in unpriv\}$ where R is a resource. It can be further required that if the login context for an actor contains no privileged resources, then all successor states to that login context must also not contain any privileged resources. If R_A is used to denote the resources held in the login context of an actor A, then this can be expressed as

$$\{\forall r \in R_A | r \in unpriv \Rightarrow \forall s \in SUCC^{+}(R_A) | s \in unpriv\}$$

where $SUCC$ denotes the successor function (which is multi–valued in this case, since a given state may have multiple possible successors), and $^{+}$ denotes reflexive transitive closure. A policy system is thus an expression of these rules, which may also be thought of as constraints or invariants. The predicates P should be constructed such that they express these rules. In standard terminology, the rules are the "policy" of the system, while the policy predicates P are the "mechanism" of the system. This distinction between the two is critical. It is certainly possible for the rules themselves to be unsound or incomplete. It is also possible that the set or rules is sound, or complete, or both, but the implementation fails to properly reflect the properties of the rules.

It would be highly desirable to develop a systematic methodology for checking the soundness and completeness of a policy system, both at the abstract level represented by the rules, and also at the concrete level represented by the implementation. Modeling provides such a methodology. Within a model framework, the top level rules can be expressed as invariants of the model. The implementation is then expressed as a particular instance of the model. The instantiation is formed by combining the rules with the particular instances of model variables and relations implied by the particular implementation being modeled. If the modeling system itself is sound, it can then be used to check the soundness of the policy model. Any counterexample represents a violation of one of the rules, either because the implementation is inconsistent with the invariants, or because the invariants themselves are inconsistent. If the modeling system itself is complete, then the absence of counterexamples can be taken as a proof that the policy model itself is complete. Unfortunately, completeness is a difficult property to demonstrate; the modeling system that will be used in this paper (like many similar systems) is, in fact, only complete within scope. This means that it can only assert completeness within a given search depth in the state space. If the model shows no counterexamples in a complete–within–scope modeling system, two possibilities arise. Either one must demonstrate that the search scope is sufficiently large that it must encompass all reachable states, or one must accept that fact that the absence of counterexamples is only a lower bound on their reachability.

The next section will describe Alloy, a simple yet powerful modeling system based on first order logic. Subsequent sections will describe how Alloy was used to build a model for a generic policy system. A counterexample driven approach will be used to illustrate the power of the Alloy approach, with a special focus on the privilege escalation case.

3. ALLOY MODELING LANGUAGE

Alloy [5] is a modeling and specification language based on first order logic. The goal of using Alloy is to create a model for a specification or implementation, constraint it with the rules of the system being modeled, and then check properties about the model. Properties can be checked using positive logic (searching for an instance) of negative logic (searching for a counterexample). Alloy always checks within a finite scope that is specified explicitly in the model, so it can only assert the lack of a counterexample within that scope. If it can be shown that the scope is sufficient to encompass all states, then Alloy can produce a proof of satisfiability. Note that Alloy never produces false positives: it is always the case that a counterexample violates one or more model constraints, and the Alloy analyzer GUI can be used to graphically show a complete state backtrace from the counterexample.

Alloy enables the model author to recognize if a particular predicate has been satisfied or refuted. This approach is particularly useful for analysis of specifications for consistency, since one is typically interested in demonstrating the absence of specification violations rather than developing a full fledged reachability proof for each valid state.

Alloy is capable of modeling both static and dynamic behavior. In particular, it can not only model properties of states, it can also model state transitions. Alloy has a simple yet rich syntax. This makes Alloy particularly suitable for metaprogramming. Prior work has demonstrated that it is straightforward to write translators that convert from actual implementations (as opposed to specification) into Alloy. The result model can then be used to investigate properties of the implementation itself, without the need to provide annotations to the code, and then invoking the modeling tool or proof assistant on the annotated code blocks.

Alloy has a visualizer GUI that can be used for forward and backward trace exploration. Alloy is also supported by an Eclipse plug–in. Alloy uses a SAT solver for constraint analysis. The solver is pluggable such that any solver that supports Alloy's simple API (including KodKod, Sat4J and minisat, among others) can be used. Alloy is open source, and is written in Java. Alloy has been in wide use for several years, and the code base for the current release (version 4) is extremely stable.

4. THE ALLOY MODEL

The Alloy model used for policy analysis consists of two components: a static component and a dynamic component. The static component is the same for all policies, and describes the generic structure of what constitute a policy system. The dynamic component is dependent on the policy being analyzed, and will vary from system to system. In particular, it is important to note that the Alloy model that we now be described can be used in analyzing policy systems associated with single domains, as well as for analyzing policy systems that are formed by aggregating two or more policy subsystems. The syntax of Alloy and the structure of the Alloy policy model allows us to deal with policy systems without having to consider how they were constructed.

4.1 Sets and Relations: Declarations

Conceptually, a policy system consists of nodes and edges. The nodes represent state, while the edges represent possible state transition. Nodes are decorated with the resources that are accessible from that node. There is a distinguished node known as "current", that represents the state of the actor (we assume only a single actor). An actor, in turn is decorated with a set of attributes. Without loss of generality we assume that the set of attributes is static. We can always convert a system with dynamic attribute derived from resources into a system with only static attributes, by requiring that the policy functions in the latter take both the attributes and resources as arguments, instead of only operating on attributes. Finally, each edge of the graph is associated with exactly one policy function. This policy function is a boolean–valued function. It operates on the current source node (state). If its value is $True$ the state transition to the destination node (state) is allowed, otherwise it is forbidden.

It would be straightforward to create an Alloy model exactly as described above. However, for the sake of efficiency, we instead create a more compact model containing only Alloy signatures for **Resources**, **Attributes**, **Edges**, and **Policies**.

(There is also a special signature known as Conflict that will be described below.) Having an Alloy signature for a node is not needed. We associate the set of resources that would have been held by a node to each edge that enters that node. Similarly, we associate the set of attributes to each edge that exits from any node that could be a current node. An edge, of course, is also always associated with a policy. We therefore have the following declarations as part of the static component of the Alloy model.

```
abstract sig Policy {}

abstract sig Resource {}

abstract sig Attribute {
    e:      Edge
}

abstract sig Edge {
    ar:     Attribute+Resource,
    p:      Policy
}
```

The Policy and Resource signatures have no relations. The Attribute signature has a single relation e that maps an Attribute to an associated Edge. This could have been declared as a set of Edges without changing the semantics of the model. The Edge signature has two relations: the *ar* relation maps the Edge to the union of the Attribute and Resource sets, while the p relation maps the Edge to a single policy.

All of these signatures are declared as abstract. This is because the static part of the Alloy model does not contain any information about a specific policy system. It is only when a specific policy system is considered that concrete signatures will be created that will extend these abstract signatures. A simple example is shown in subsection 4.2 below.

The remaining signature in the Alloy model is the Conflict signature. Even though this signature is not a part of the static component of the Alloy model, it is sufficiently important to the model semantics that it is described here. The Conflict signature is a singleton: the Alloy analyzer is told to create only one such set by using a cardinality constraint. The Conflict signature has one relation for each concrete Policy signature in the model. Essentially, the Conflict signature is a container for policy conflicts; the Policy relations allow a backward mapping to the Policy that caused the conflict to arise. The presence of a conflict is signaled by a counterexample to the *noattrconflict* assertion which is described in detail in subsection 4.3 below. For a policy system with only two policies, the declaration of the Conflict signature would be

```
sig Conflict {
    p1:   Policy,
    p2:   Policy
} {
    #Conflict = 1
}
```

4.2 Initializers

In order to instantiate the model, initializers must be provided for all relations that are non–empty. These initializers obviously will depend on the specifics of the policy system being analyzed, and thus are part of the dynamic component of the Alloy model. We consider an extremely simple system in which there are two policies, two edges, two resources and two attributes. Even with this extremely simple system, we will see that it is possible to create a meaningful example. In anticipation of that example, which is described in section 5 below, we will add the following initializers to the Alloy model:

```
one sig Policy1, Policy2 extends Policy {}
one sig RPriv, RUnpriv extends Resource {}
one sig EPriv, EUnpriv extends Edge {}
one sig Root, User extends Attribute {}
```

Because of the way these signatures are declared, each is guaranteed to be a singleton. Also creating concrete signatures from abstract signatures in this manner insures that they are disjoint, so that, for example $RPriv \cap RUnpriv = \varnothing$. These declarations only give the signature initializers, they do not give the relation initializers. The relation initializers must be declared differently, using Alloy facts to establish them as model invariants. Section 5 shows the relation initializers for the privilege escalation example.

4.3 Invariants and Assertions

As stated above, the Conflict signature fills a very specific purpose in this model as a container that holds any conflicts that are discovered. Its relations must be initialized dynamically, but unlike the other relations, it can be done in a uniform manner. It is still part of the dynamic component of the Alloy model, but the initialization of the Conflict relation only depends on the number of policies that are present in the policy system, and their signature names, not on the structure of the state graph, or the assignment of attributes and resources to edges. For the simple case of a system with two policies, the following initializer suffices:

```
fact pfact {
    Policy1 in Conflict.p1 and Policy2 in Conflict.p2
}
```

Finally, there must be something to check. The Alloy analyzer requires at least one assertion that expresses the top level rule to be satisfied or refuted. In our case, we wish to capture a single top level rule, namely that it is not possible to find to find two disjoint paths such that two different agents may come to hold the same set of resources. How can this statement be captured in Alloy? If the Conflict set is the container that holds conflicting results, then how is such a conflict ultimately derived from Attributes. Observe that an Attribute is in the range of the *ar* mapping on an Edge, and that a Policy is in the range of the *p* mapping on an Edge. Further, a Policy is also in the range of one of the (disjoint) relations on Conflict. Thus, starting from an Attribute if we invert the ar relation to arrive at an Edge, and then apply the p relation to arrive at a Policy, we can express the stated exclusivity rule as the statement that there is no pair of Attributes that are in conflict. In Alloy this becomes

```
assert noattrconflict {
    no a1, a2: Attribute | a1.~ar.p in Conflict.p1 and
                           a2.~ar.p in Conflict.p2
}
```

Recall from section 3 that the dot operator (.) denotes forward evaluation of a relation, while the dot–tilde operator (.) denotes inverse evaluation of a relation. In addition to the state initializers, which will be given in the next section, there is still one thing missing in order for this to be a complete Alloy model. We must specify a scope for checking the assertion, namely we must supply the value of N in an Alloy statement of the form:

```
check noattrconflict for N
```

If we wish to prove that there are no attribute conflicts, we must specify a value of N large enough to encompass all possible realizations of the Alloy sets that might be created that have distinct semantics. If we merely wish to verify that no conflicts can be reached in a maximum of N transitions, choose N accordingly. In the latter case it need not encompass all distinct signature realizations. It is clear that the value of N need to prove the absence of conflicts is strongly dependent on the structure of the state transition graph. For the privilege escalation case, given in the next section, the value of N to use will be immediately derivable by simply drawing the transition graph, counting the number of distinct states, and adding one.

5. PRIVILEGE ESCALATION EXAMPLE

In order to complete the model, all relations other than the *px* relations in the Conflict signature, must be initialized. In particular, this means that the *ar* and *p* relations for Edges must be initialized, and the *e* relation for Attributes must be initialized. For demonstration purposes, it suffices to create the simplest non–trivial example that shows privilege escalation, which we construed to mean an unprivileged agent (one with the **User** Attribute) gaining access to an **RPriv** resource. Because of the extreme simplicity of this case, it is possible to write down all the initializers manually. In the case of an actual policy system, this manual approach would be highly undesirable, as it could itself lead to transcriptions errors. For a real system, one would like to have a policy configuration file that would describe all the attributes, resources, edges and policies. An automatic translation program would then generate the corresponding Alloy initializers from this configuration file.

The Alloy relation initializers for the privilege escalation example are:

```
fact fact1 { EPriv.p = Policy1 }
fact fact2 { EUnpriv.p = Policy2 }
fact fact3 { Root.e = EPriv }
fact fact4 { User.e = EUnpriv }
fact fact5 { EPriv.ar = RPriv }
fact fact6 { EUnpriv.ar = RPriv }
```

Drawing the state transition graph for the above system shows immediately that a search depth of 4 is sufficient, so we add the following check statement in order to complete the model:

```
check noattrconflict for 4
```

When the Alloy analyzer is run on the resulting complete model, a counterexample is quickly discovered. The output of the analyzer is shown in Figure 1.

Note that by starting from the box (set) labeled Conflict we can trace the two Policy relations, *p1* and *p2* to the Policy1 and Policy2 sets. Inverting the *p* relations from these sets leads us back to the EUnpriv and EPriv sets. Following the *ar* relations forward immediately leads us to the RPriv set. This is the source of the conflict. From the Alloy code we can immediately see the error. The **fact6** fact was given as

```
fact fact6 { EUnpriv.ar = RPriv }
```

when it should have been

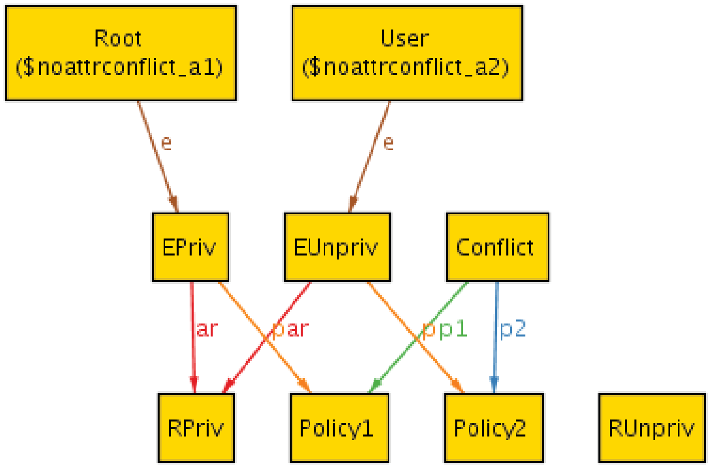

Figure 1: Privilege Escalation Example

```
fact fact6 { EUnpriv.ar = RUnpriv }
```

In this simple case it would have been easy to spot this error without recourse to viewing the output of the Alloy analyzer. In a more complicated system mere visual examination of the policy specification would, of course, not be sufficient. The automated approach provided by converting the policy specification into Alloy provides the opportunity for uncovering inconsistencies in the specification that would be very unlikely to be found by inspection.

6. RELATED WORK

There is a large amount of literature addressing aspects of access control and identity management, both within and across domains. XACML [6] and SELinux [7], [8] are two widely used access control enforcement technologies. XACML, which is used in Service-Oriented Architecture (SOA) systems to restrict access to critical services, defines architectural components as well as an XML-based policy language for expressing access control rules. XACML allows the expression of policy decision points (PDPs) and policy enforcement points (PEPs). SELinux, on the other hand, allows fine-grained restriction of the functions, services, and resources that processes are allowed to access (as it relates to file system and networking stack operations). The work presented in this paper could be seen as necessary to establish the safety of access control policies expressed in XACML and SELinux.

With respect to multi-domain access control, a number of identity management frameworks exist that provide this federated capability. OpenID [9], Shibboleth [10] and OpenPERMIS [11] are examples of such systems. They enable single "sign on" capabilities across a federation of domains – e.g., in grid environments – and provide support for unification across different policy representations. While many of these systems target some of the same issues described in this paper, they do not address the issues of safety (lack of conflicts between multi-domain policies), and do not have support for automated checking that scales with the number of users, attributes, and resources in the system.

Prior work that is concerned with the analysis of access control policy conflicts include the Lobster domain specific language (DSL) [12] which enables expression of high-level security policies through graphs representing information flow between system entities. Lobster also provides tools for automated translation of these policies into SELinux policies, and has limited capabilities to check policies for consistency through definition and evaluation of policy assertions (which themselves are not verified). Perhaps the closest work to ours is [13] – a semantic web policy management framework. KAoS provides limited conflict analysis through the use of the Stanford Java Theorem Prover [14].

As we mentioned earlier in the paper, to be practical, access control models in use must restrict the range of policies that can be specified. Examples of these models include the Bell-LaPadula model [3] and the BIBA model [15]. Both of these models allow the expression of constraints – of fairly limited expressiveness to ensure decidability/tractability – on access control rules that prevent violation of data integrity. The work we present in this paper is similar in its reliance on a rigorous formalism that enables the specification of valid and invalid interaction patterns. However, these models are becoming increasingly untenable due to the need to seamlessly handle dynamically unfolding scenarios. Our approach aims at establishing safety guarantees before deployment (*i.e.*, independent of how many users are in the domains, or the types of resources, *etc.*) and is additionally compositional.

7. FUTURE DIRECTIONS

In developing the Alloy model shown above we explicitly eliminated the notion of having a Node signature, since it was shown that all the properties of such a signature could be applied to other signatures without any loss in expressive semantics.

In order to extend the applicability of this type of Alloy modeling to large policy systems, in particular to systems that are actually compositions of policy systems, more work needs to be done. In particular, it would be highly desirable if individual policy systems could be passed through the Alloy analyzer, not only to demonstrate the lack of counterexamples, but also to produce the highly reduced form of the model. In effect, we would like the analysis process to strip out any internal details of the model that would not be visible when the policy system was composed with other systems. Having the ability to produce such model "skeletons" would be of great benefit in performing a composition of analyses, rather than the more brute–force analysis of compositions. Work in this area is ongoing.

The ability to automatically produce all the dynamic parts of the Alloy model from a policy configuration file is also an active area of research.

8. CONCLUSION

This paper has demonstrated that Alloy is an extremely powerful tool for performing constraint analysis on policies. Even at this stage of development, meaningful results have been obtained. Extensions to this work are ongoing, with the goal of increasing the scope of multi–domain policy checking and further refining and improving the analysis process.

Acknowledgments

The authors would like to acknowledge the support and collaboration of XXX.

9. REFERENCES

[1] Harrison, M. A., Ruzzo, W. L., Ullman, J. D. 1976: Protection in operating systems. Communications of the ACM 19, 8 (Aug).

[2] Lampson, Butler W.: Protection. Proc. 5th Princeton Conf. on Information Sciences and Systems, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[3] Bell, D., La Padula, L.: Secure computer systems: Mathematical foundations (Volume 1). Tech. Rep. ESD-TR-73-278. Mitre Corporation. 1973.

[4] Alloy website, `http://alloy.mit.edu`

[5] Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)

[6] Moses, T.: eXtensible Access Control Markup Language (XACML) Versions 2.0, 2005;

[7] McCarty, W.: SELinux: NSAâĂŹs Open Source Security Enhanced Linux.: OâĂŹReilly, 2005.

[8] MacMillan, K., Caplan, D., Mayer, F.: SELinux by Example, Open Source Software Development Series, Prentice Hall, 2007.

[9] OpenID web site `http://openid.net`

[10] Shibboleth web site `http://shibboleth.internet2.edu`

[11] Chadwick, D. and Zhao, G.: PERMIS: a modular authorization infrastructure, Concurrency and Computation: Practice and Experience, vol. 20, no. 11, pp. 1341-1357, 2008.

[12] White, P.: Security Configuration Domain Specific Language (DSL), 2008 SELinux developer summit `http://selinuxproject.org/files/2008_selinux_developer_summit/2008_summit_white.pdf`

[13] Tonti, G., Bradshaw, J. M., Jeffers, R., Montanari, R., Suri, N., and Uszok, A.: Semantic Web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. International Semantic Web Conference (ISWC 2003). Sanibel Island, Florida.

[14] Java Theorem Prover web site `http://www-ksl.stanford.edu/software/jtp`

[15] Biba, K. J. Integrity Considerations for Secure Computer Systems, MTR-3153, The Mitre Corporation, April 1977.