SPAN: A Unified Framework and Toolkit for Querying Heterogeneous Access Policies

Swati Gupta[1]*, Kristen LeFevre[2], and Atul Prakash[2]†
[1] Indian Institute of Technology, Delhi [2] University of Michigan, Ann Arbor, MI

Abstract

Incorrect policy configurations are a major cause of security failures in large-scale systems. Policy analyzers and testing tools can help with this, but often the tools are specific to one type of policy (e.g., firewalls). In contrast, the most insidious security problems often require understanding the interactions of policies across systems (e.g., firewalls, SSH, file systems, etc.). Currently, much of this analysis must be done manually. In this paper, we propose a common framework called SPAN (Security Policy Analyzer) to help analyze policies from heterogeneous systems. On the front-end, SPAN presents administrators with a simple, unified, abstraction and flexible query language. Internally, policies and queries are implemented compactly and efficiently using decision diagrams.

1 Introduction

Security experts and system administrators agree that good security policies are essential to protecting their systems. However, it can be difficult to set security policies correctly, without making mistakes. Many system administrators recount experiences in which a system resource (e.g., a computer or file) was accessed inappropriately, not by hacking, but due to misconfiguration. Similarly, users may be improperly denied access to resources.

Unfortunately, it can be difficult to troubleshoot and fix security policies. In today's complex systems, requests for system resources must often pass through multiple services, each with its own access policy. For example, consider a web application and a user who requests access to a particular entry in a back-end database (DBMS). This request will likely need to pass through a firewall policy, policies used by the web application server, the DBMS policy, and the access control policies in place in the underlying file system that stores DBMS tables.

Often, gaps in security arise at the boundaries between system policies, which are often written by different people, making implicit assumptions about the guarantees provided by other policies. Detecting and fixing errors requires system administrators to examine multiple policies, analyzing the flow of requests between them in order to determine the cause of a problem. In heterogeneous environments, this task can be difficult and error-prone.

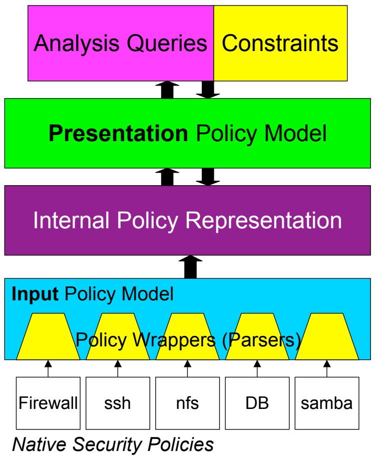


Figure 1: SPAN System Components

Clearly, there is a need for better solutions. In this paper, we propose a novel framework and system called SPAN (Security Policy Analyzer). The main distinguishing design insight behind SPAN is a three-tiered approach to policy abstraction and analysis (shown in Figure 1):

- At the bottom level, policies are consumed in their various native formats (i.e., configuration files).
- At the top level, policies are presented to system administrators in terms of a unified *presentation policy model*, which abstracts away the numerous semantic and syntactic differences between input policies. To understand a set of heterogeneous policies, a system administrator can now write simple queries, expressed in terms of the presentation model, rather than attempting to reason about the input policies.
- Finally, in the middle, policies are represented using an internal policy representation selected for efficiency. (In the current implementation, this representation is a form of decision diagram.) Queries, which are specified in terms of the presentation policy model, are converted automatically to operations on the internal policies. Thus, the internal policy model is completely hidden from view.

*Work done while author was visiting the University of Michigan.
†Supported in part by NSF Grant 0705672.

Proto	Source	SPort	Dest.	DPort	Action
*	192.168.*.*	*	*.*.*.*	*	ACCEPT
*	$dom(Source) - \{$192.168.*.*$\}$	*	*.*.*.*	*	DROP

(a) Firewall presentation policy (fw)

Username	Action
root	ACCEPT
alice	ACCEPT
bob	ACCEPT
$dom(Username) - \{root, alice, bob\}$	DROP

(b) SSH presentation policy (ssh)

Proto	Source	fw.Action	Username	ssh.Action
*	192.168.*.*	ACCEPT	alice	ACCEPT
*	192.168.*.*	ACCEPT	bob	ACCEPT
*	192.168.2.4	ACCEPT	root	ACCEPT

(c) Policy join result under the constraint that $Username \in \{root\} \rightarrow Source \in \{$192.168.2.4$\}$

Figure 2: Example integrating heterogeneous policies

2 SPAN - Users' View

SPAN provides a surprisingly simple and powerful model for analyzing security policies; users are presented a relational view of policies, even though the underlying implementation is very different from a relational database. The "rows" in the relational view describe (exhaustively) the relationships between inputs and access control decisions. For example, Figure 2 shows a relational-style view of sample firewall and SSH policies[1]. Users can query these policies as if they were relational tables.

Given such a model, the following are examples of the types of analysis that SPAN can do on the policies.

What requests are accepted? The first example is the simplest; it requests the set of all source IP addresses such that policy fw accepts some requests from these addresses via the TCP protocol.

SELECT $Source, Action$ FROM fw
WHERE $Action = accept$ AND $Proto = TCP$

Change analysis: Using the query language, it is also easy to express change analysis. Suppose that we have two versions of the same policy, P_1 and P_2, and we want to know which requests are accepted by P_2, but not by P_1. This query is easily expressed as follows:

SELECT $F_1, ..., F_n, Action$ FROM P_2
WHERE $Action = accept$
EXCEPT
SELECT $F_1, ..., F_n, Action$ FROM P_1
WHERE $Action = accept$

Comparing with a reference policy: Often, an organization will develop a reference policy, expressing best security practices. Using a similar query as for change analysis, it is easy to check whether a policy P accepts any requests that are not accepted by the reference policy R.

Querying Heterogeneous Policies While our query language can express common analysis and verification tasks for policies of a single type (e.g., firewall policies), much of the language's power actually lies in its ability to combine policies expressed in terms of different input types (*schemas*). For example, we can write a query to find inputs that will be accepted by both a firewall and an SSH policy:

SELECT $fw.Proto$, $fw.Source$, $fw.Action$, $ssh.Username$, $ssh.Action$ FROM fw, ssh
WHERE $fw.Action = accept$
AND $ssh.Action = accept$

The variables in different schemas can be associated through a set of (user-provided) *instance-level constraints*. An example set of constraints (which contains only one constraint) is $\{Username \in \{root\} \rightarrow Source \in \{$192.168.2.4$\}\}$. Let's say the user gives this a name "SSHConstraints". This defines a relationship between fw and ssh, indicating that authentication credentials for username root (e.g., private key) are only available at IP address 192.168.2.4. The modified query under the instance-level constraint is:

SELECT $fw.Proto$, $fw.Source$, $fw.Action$, $ssh.Username$, $ssh.Action$
FROM fw, ssh WHERE $fw.Action = accept$
AND $ssh.Action = accept$ AND $SSHConstraints$

The result of such a query contains the set of all requests that, in light of the constraints, could *possibly* be accepted by both policies. Figure 2(c) shows the query results. Notice that requests with $Username = root$ from arbitrary addresses are ruled out by the constraint. SSH requests from IP addresses other than 192.168.*.* are ruled out by the firewall.

3 Related Work

Security policy analysis and verification is a growing area of research. Most related to our work is *Margrave* [2], developed as a verification toolkit for XACML policies. XACML is a single policy language, but it is capable of expressing various policy semantics. (That is, policies can be viewed as a sequence of rules, and XACML supports various rule combining algorithms such as first-applicable, deny-override, etc.) Margrave is built around

[1]For ease of presentation, we have not enumerated all of the requests in the example. For example, in Figure 2(b), the SSH policy accepts requests from users $root$, $alice$, and bob, but drops all other requests.

two policy representations: multi-terminal binary decision diagrams (MTBDDs) and binary decision diagrams (BDDs). Initially, each policy is represented as a reduced MTBDD. Margrave provides several operations that can be used to combine and query policies, and the results of these operations are expressed as BDDs.

While the internal policy representation used in SPAN is similar to that of Margrave, and the two systems support a similar set of query operators, SPAN provides a simple and unified presentation policy abstraction, which allows the user to think of policies in terms of simple tables and make queries in a familiar SQL-like syntax. In contrast, Margrave requires users to understand and manipulate MTDDs and BDDs. Furthermore, Margrave does not provide a natural operator for combining policies expressed in terms of different vocabularies (what we will call *schemas*). For example, firewall policies are expressed in terms of IP addresses, and file system policies are expressed in terms of users. SPAN permits a user to combine such heterogeneous policies in a natural way using a join operation.

Other than Margrave, the vast majority of work is security policy verification and testing has focused on a single type of policy, and is thus unable to reason about misconfigurations spanning heterogeneous systems.

One particular area of focus has been in analyzing and testing firewall policies. Two of the first systems were *Fang* [7] and *Lumeta* [9], which support a specific class of queries over multiple firewalls in a network. Queries in Fang are described by triples of the form (set of source IP addresses, set of destination IP addresses, set of services), asking "Which Source IP addresses can send which services to which destination IP addresses?". Lumeta extends Fang by automatically selecting queries. In both cases, however, query evaluation is implemented by simulating the behavior of all packets described by the query, rather than through static analysis.

In contrast to the simulation approach, a variety of tools have recently been proposed that use static verification (specifically, tools built on decision diagrams) to verify or query one or more firewall policies:

- *ITVal* [6, 5] Marmorstein and Kearns provide an analysis tool for one or more firewalls, with an implementation built on multi-way decision diagrams (MDDs). The system provides a simple firewall-specific query language for asking questions about which packets are accepted.
- *Structured Firewall Query Language (SFQL)* [4] Liu et al. proposed a simple SQL-like language for specifying queries on a single firewall policy. These queries are implemented using a structure the authors refer to as a firewall decision tree. However, the system does not support queries on multiple firewalls or queries on other types of policies.
- *Fireman* [10] Yuan et al. propose a tool based on binary decision diagrams (BDDs) to check for misconfigurations and inconsistencies in one or more firewalls. While the system can process some of the same analyses as SPAN, it is designed specifically for firewalls, and it is not clear whether it would generalize to other classes of policies.

Finally, Gouda et al. proposed using (reduced) interval-based decision diagrams to produce consistent, compact, and complete firewall policies [3]. However, the tool provides no support for user-specified querying.

4 Preliminaries

4.1 Input Policy Model

We begin by describing (in abstract terms) the kinds of input policies that we will support. It is common in many domains to express access policies in terms of a set of rules, each of which is expressed in terms of input and decision variables.[2] The set of variables can vary by policy type. For example, a firewall policy is specified in terms of source and destination IP addresses, among other things, while SSH policies are specified in terms of usernames. We capture this idea through the idea of a *policy schema*, borrowing the terminology from relational databases.

Definition 1 (Policy Schema) *A policy schema* $\mathfrak{S}$ *consists of a set of input variables* $F_1, ..., F_n$ *and a decision variable D, each with a finite domain denoted* $dom()$.

Consider firewall policies, for example. In this case, the input variables describe packets; we might have $F_1 = Source\ IP$, $F_2 = Source\ Port$, etc. The decision variable describes the decisions that can be made for a particular packet (i.e., $dom(D) = \{accept, drop\}$).[3]

Definition 2 (Input Policy Instance) *An* input policy instance I *is defined by an ordered sequence of policy rules, each expressed in terms of policy schema* $\mathfrak{S}$*, and a* rule combining algorithm. *Each* rule *is a statement of the form* $predicate \rightarrow decision$*, where* $predicate$ *is a boolean expression of the form* $F_1 \in S_1 \wedge ... \wedge F_n \in S_n$*, such that each* $S_i \subseteq dom(F_i)$*, and* $decision \in dom(D)$.

Policies are applied to *requests*, which are tuples of the form $(f_1, ..., f_n) \in dom(F_1) \times ... \times dom(F_n)$. For example, firewall requests are individual packets. A request is said to *match* a particular policy rule R if $f_1 \in S_1 \wedge ... \wedge f_n \in S_n$.

In this work, we consider two specific rule combining algorithms. In the *first applicable* algorithm (similar to

[2]Our input policy abstraction does not inherently distinguish between properties of requests (e.g., usernames) and properties of the resources being requested (e.g., computers or files). Both concepts are simply incorporated as input variables.

[3]Technically, it is not required that the policy produce a decision for every input (i.e.., $dom(D)$ could include a value *no decision*).

typical firewalls), the decision rendered for a particular request is determined by the first policy rule to match the request. The *decision precedence* algorithm assigns a total order to all values in $dom(D)$, and if multiple rules match a particular request, it takes the value of D with the highest precedence. Examples of the latter algorithm include *accept override* and *deny override*.

Of course, notice that two input policies that are syntactically different may in fact yield the same decisions for all requests, in which case we will say that the two policies are *semantically equivalent*.

4.2 Presentation Policy Model

In even this abstract view of input policies, there can be considerable semantic heterogeneity. For example, firewall policies typically use a first-applicable rule combining algorithm, but SSH typically uses deny-override, which makes it difficult to compare different policies, or reason about them in combination. For these reasons, we developed a unified *presentation policy model*, which masks the heterogeneity in input policies without losing information. The system administrator / analyst interacts with a set of *presentation policies*, each of which has a unique *policy name*.

Definition 3 (Presentation Policy) *A presentation policy P consists of two parts: a policy schema $\mathfrak{S} = \{F_1, ..., F_n, D\}$, and a policy instance, which is a set of unique tuples in $dom(F_1) \times ... \times dom(F_n) \times dom(D)$.*

Observe that every input policy instance can be expressed as a unique *canonical* presentation policy (that is, a presentation policy in which each unique value of $F_1, ..., F_n$ is associated with a single unique value of D). Conceptually, this can be done by enumerating all requests in $dom(F_1) \times ... \times dom(F_n)$ and applying the policy to each. Of course, we do not advocate actually enumerating such requests, or materializing the full presentation policy. We will return to the internal policy representation in Section 6.

5 SPAN-QL Query Language

To query policies, we propose a simple query language called SPAN-QL, the syntax and semantics of which are inspired by (a subset of) the relational database query language SQL, which is familiar to most system administrators. In order to provide a uniform interface, all queries are expressed in terms of presentation policies.

5.1 Algebra Operators

It is straightforward to define the relational algebra operators atop the presentation policy model: Selection ($\sigma_C(P)$); Projection ($\pi_{projection\ list}(P)$); Set Difference ($P1 - P2$, where P_1 and P_2 have the same schema); Set Intersection ($P1 \cap P2$, where P_1 and P_2 have the same schema); Union ($P1 \cup P2$, where P_1 and P_2 have the same schema). We also define the policy Cross Product ($P_1 \times P_2$) and Conditional Join ($P_1 \bowtie_C P_2$), which allow us to combine two or more policies with different schemas. Like relational algebra, the algebra of presentation policies is *closed*. That is, each operator takes one or more presentation policies as input, and produces a presentation policy as output. Thus, algebra operators can be composed. Further, the operators have the same algebraic properties as relational algebra and can be reordered following the same rules.

5.2 SPAN-QL and Constraints

System administrators interact with SPAN by issuing simple queries, which are specified in terms of the presentation policies using a subset of the SQL syntax. Like SQL, queries are expressed declaratively in SPAN-QL, but then translated into algebra expressions. The basic form of a query is as follows, which is equivalent to the algebra expression $\pi_{A_1,...,A_m}(\sigma_H(P_1 \bowtie_C P_2))$.

SELECT $A_1, ..., A_m$ FROM P_1, P_2
WHERE C AND H

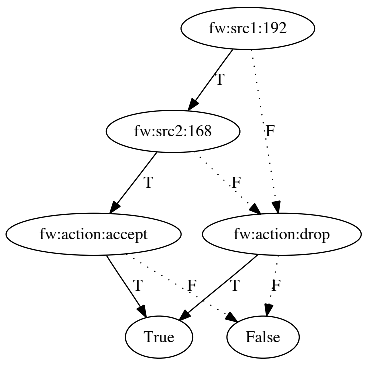
In this example, C is a standard boolean condition. In addition, in order to easily relate schemas, we incorporate *named constraint sets*, denoted H, where constraints are instance-level rules of the form $F_1 \in \{f_1^1, ..., f_m^1\} \rightarrow F_2 \in \{f_1^2, ..., f_n^2\}$, where $F_1 \subseteq schema(P_1)$ and $F_2 \subseteq schema(P_2)$ and each $f_i^1 \in dom(F_1)$ and each $f_i^2 \in dom(F_2)$. This is just syntactic sugar, but the idea is that a set of constraints can be loaded from file and used repeatedly. Another form of constraints that SPAN supports are *equality constraints*, where a variable in one policy schema is always equal to a variable in another policy schema. Such constraints are useful for handling network connectivity.

In addition, we use SQL syntax to express set operations. In each of the following, P_1 and P_2 are two presentation policies expressed in terms of the same schema: Set Difference (P_1 EXCEPT P_2); Set Intersection (P_1 INTERSECT P_2); Union (P_1 UNION P_2).

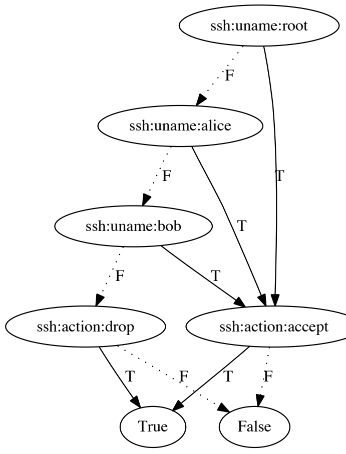
6 Internal Policy Representation

Returning to the system architecture (Figure 1), the remaining challenge is bridging the gap between input policies and presentation policies. A naive approach would take each input policy and actually enumerate all possible requests. However, this is clearly not satisfactory. Consider, for example, a 32-bit IP address that is being used to make firewall decisions. Enumerating a list of all IP addresses is extremely space-consuming.

For these reasons, we are currently using an *internal* policy representation based on binary decision diagrams (BDDs) [1]. Input policies are initially imported, and then translated into their canonical BDD representations.



(a) Firewall policy fw



(b) SSH policy ssh

Figure 3: Sample policy decision diagrams

SPAN-QL queries are automatically translated into operations on BDDs.

In each BDD, the terminal nodes (true and false) can be interpreted to indicate whether the particular request and decision obtained from traversing the diagram from root to terminal is valid in the policy. For clarity, we will use the following naming convention for nodes in the decision diagrams: $PolicyName : AttrName : ValueRange$. For example, a node named $Firewall1 : SPort : [80, 80]$ refers to the policy called *Firewall1*, variable *SPort*, with value 80. For example, Figure 3(a) shows a decision diagram for the firewall policy in Figure 2(a). Notice that packets from source IP $192.168.*.*$ and decision = $accept$ is valid, while all other requests with decision = $drop$ are valid. Due to space constraints, we omit the details of how input policies are translated to BDDs of this form; however, the process is similar to that used by Margrave [2].

We implement the following set of operations on BDDs: logical operations (i.e., $P_1 \wedge P_2$, $P_1 \vee P2$, $\neg P$); the restriction operation $restrict(P, C)$, which removes nodes from the diagram that do not satisfy condition C; $rename(P, P1, < F_1', ..., F_n' >)$, which renames the the policy P and the variables contained therein; finally, the canonicalization operator $reduce()$.

Query Expression	Decision Diagram Operation
$P_1 \cup P_2$	$rename(P_1, P, < F_1, ...F_n >)$ $\vee rename(P_2, P, < F_1, ..., F_n >)$
$P_1 \cap P_2$	$rename(P_1, P, < F_1, ...F_n >)$ $\wedge rename(P_2, P, < F_1, ..., F_n >)$
$P_1 - P_2$	$rename(P_1, P, < F_1, ...F_n >)$ $\wedge\neg rename(P_2, P, < F_1, ..., F_n >)$
$P_1 \times P_2$	$P_1 \wedge P_2$
$P_1 \bowtie_C P_2$	$P_1 \wedge P_2 \wedge C$
$\sigma_C(P)$	$P \wedge C$

Figure 4: Implementing query operators using operations on decision diagrams

6.1 SPAN-QL Operator Implementation

The core set of decision diagram operations is sufficient to implement most of the query algebra operations we have defined on presentation policies[4]. For each query operator, Figure 4 shows one corresponding sequence of operations on the decision diagram representation. Notice that before performing a set operation on policies P_1 and P_2, we need to rename both policies in terms of the same new policy name, and same set of variable names.

Example: Consider again the example combining a firewall and an ssh policy, shown in Figure 2 and the corresponding decision diagram representations in Figure 3, and consider again the following query, where H is the named constraint set $\{ssh.Username \in \{root\} \rightarrow fw.Source \in \{192.168.2.4\}\}$.

SELECT * FROM fw, ssh
WHERE $fw.Action = ACCEPT$
AND $ssh.Action = ACCEPT$ AND H

This query can be translated to the following algebra expression:

$$\sigma_{fw.action=ACCEPT}(fw)$$
$$\bowtie_{H=\{ssh.uname\in\{root\}\rightarrow fw.Source\in\{192.168.2.4\}\}}$$
$$\sigma_{ssh.action=ACCEPT}(ssh)$$

The selection conditions S_1 and S_2 and the constraint H are converted to decision diagrams (omitted for space), and the algebra query is translated into the following expression involving operations on decision diagrams fw, ssh, S_1, S_2 and H:

$$(fw \wedge S_1) \wedge (ssh \wedge S_2) \wedge H$$

The resulting (reduced) decision diagram is shown in Figure 5. Notice that the result can be converted to the presentation policy model, provided that the resulting tabular form is not too large. Alternatively, the diagram itself can be returned to the user as an alternate expression of the query result.

[4]Unlike selections and joins, mapping projection to BDDs is problematic, but it can often be ignored except as a last step in the presentation of the results.

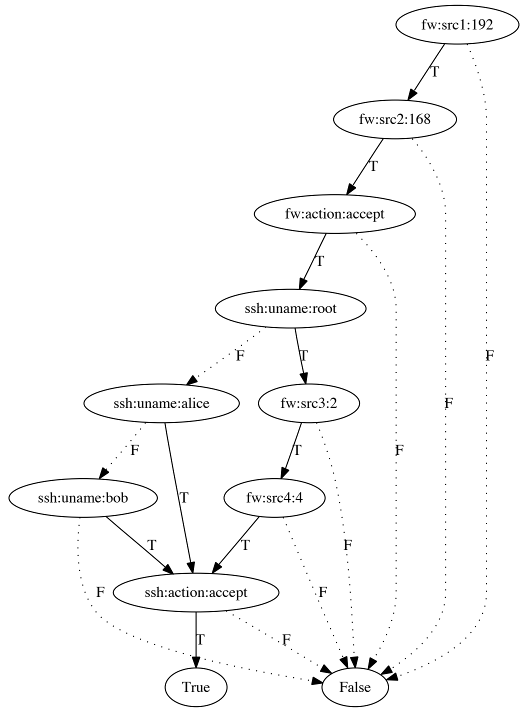


Figure 5: Query result, expressed as a decision diagram

7 Implementation Status

The current prototype of SPAN is based on a Python implementation of range-oriented binary decision diagrams (6,000 lines of code), which exposes the operations described in Section 6. Our system architecture supports the easy addition of policy parsers using wrappers. Currently, we have implemented a useful collection of policy types, each of which is obtained from the configuration files of services running on Unix machines: iptables firewall policies (/etc/sysconfig/iptables); sshd policy configuration (/etc/ssh/sshd_config); nfs policy configuration (/etc/exports).

The SPAN-SQL query engine and optimizer is currently under construction. We have observed, through several examples, that like relational database queries [8], selecting a good (equivalent) operation *plan* is important for providing good performance. (The tradeoffs are, of course, different because the operators are evaluated in decision diagrams.) We are considering how we might apply cost-based to this problem.

8 Future Research Directions

Cross-system policy configuration is a difficult problem in modern security. In this paper, we described our progress to date in designing and building SPAN, a cross-platform policy verification and query tool. We are currently working to build out the SPAN infrastructure and to evaluate a corpus of real policies for possible misconfigurations.

The SPAN work bridges ideas from databases and the work on decision diagram representations of security policies and suggests that users can analyze many aspects of security policies using a familiar and well-understood relational model. We were able to show that policies can be naturally mapped to relational schemas and their instances. We were also able to show that queries across different policies, even heterogenous ones, can be expressed as queries on multiple tables in the relational view. This also suggests that some performance optimizations may be possible by reordering algebra operations [8]. Of course, the performance tradeoffs are different from traditional query optimization because the underlying representation of policies are not tables, but decision diagrams.

While SPAN simplifies the task of expressing queries on poicies by providing a familiar model and query language, there is still an open challenge in how best to present the results of queries to the user. Presenting the results as tables is difficult as a fully-instantiated table (e.g., Figure 2) may be too large. Currently, SPAN generates decision diagrams and also can generate examples.

Another area of investigation is using queries as part of organization-level checks as to whether policies satisfy certain restrictions. For example, an organization may allow users to modify policies on their individual machines, but apply checks to make sure that the individual policies do not collectively violate corporate policies. These checks can be potentially expressed as comparisons of queries across policies and the expected result. This also helps address the user-interface problem. Users would focus on specifying the checks and making sure that their policies are compliant with those checks.

Finally, we are investigating extensions that would allow SPAN to be applied to more complex input policies (e.g., firewalls with state, or policies like file system ACLs that involve resource containment).

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[2] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M.Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE*, 2005.

[3] M. Gouda and X. Liu. Firewall design: Consistency, completeness, and compactness. In *ICDCS*, 2004.

[4] A. Liu, M. Gouda, H. Ma, and A. Ngu. Firewall queries. In *OPODIS*, 2005.

[5] R. Marmorstein and P. Kearns. An open source solution for testing NAT'd and nested iptables firewalls. In *LISA*, 2005.

[6] R. Marmorstein and P. Kearns. A tool for automated iptables firewall analysis. In *USENIX*, 2005.

[7] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Oakland*, 2000.

[8] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T.Price. Access path selection in a relational database system. In *SIGMOD*, 1979.

[9] A. Wool. Architecting the lumeta firewall analyzer. In *USENIX Security*, 2001.

[10] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *Oakland*, 2006.