The Margrave Tool for Firewall Analysis

Timothy Nelson
Worcester Polytechnic Institute
tn@cs.wpi.edu

Christopher Barratt
Brown University
cbarratt@cs.brown.edu

Daniel J. Dougherty
Worcester Polytechnic Institute
dd@cs.wpi.edu

Kathi Fisler
Worcester Polytechnic Institute
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

Writing and maintaining firewall configurations can be challenging, even for experienced system administrators. Tools that uncover the consequences of configurations and edits to them can help sysadmins prevent subtle yet serious errors. Our tool, Margrave, offers powerful features for firewall analysis, including enumerating consequences of configuration edits, detecting overlaps and conflicts among rules, tracing firewall behavior to specific rules, and verification against security goals. Margrave differs from other firewall-analysis tools in supporting queries at multiple levels (rules, filters, firewalls, and networks of firewalls), comparing separate firewalls in a single query, supporting reflexive ACLs, and presenting exhaustive sets of concrete scenarios that embody queries. Margrave supports real-world firewall-configuration languages, decomposing them into multiple policies that capture different aspects of firewall functionality. We present evaluation on networking-forum posts and on an in-use enterprise firewall-configuration.

1 Introduction

Writing a sensible firewall policy from scratch can be difficult; maintaining existing policies can be terrifying. Oppenheimer, Ganapathi, and Patterson [31] have shown that operator errors, specifically configuration errors, are a major cause of online-service failure. Configuration errors can result in lost revenue, breached security, and even physical danger to co-workers or customers. The pressure on system administrators is increased by the frenetic nature of their work environment [6], the occasional need for urgent changes to network configurations, and the limited window in which maintenance can be performed on live systems.

Many questions arise in checking a firewall's behavior: Does it permit or block certain traffic? Does a collection of policies enforce security boundaries and goals? Does a specific rule control decisions on certain traffic? What prevents a particular rule from applying to a packet? Will a policy edit permit or block more traffic than intended? These questions demand flexibility from firewall-analysis tools: they cover various levels of granularity (from individual rules to networks of policies), as well as reasoning about multiple versions of policies (to check the impact of edits). Margrave handles all these and more, offering more functionality than other published firewall tools.

Margrave's flexibility comes from thinking about policy analysis from an end-user's perspective. The questions that users wish to ask about policies obviously affect modeling decisions, but so does our form of answer. Margrave's core paradigm is *scenario finding*: when a user poses a query, Margrave produces a (usually exhaustive) set of scenarios that witness the queried behavior. Whether a user is interested in the impact of changes or how one rule can override another, scenarios concretize a policy's behavior. Margrave also allows queries to be built incrementally, with new queries refining the results from previous ones.

Margrave's power comes from choosing an appropriate model. Embracing both scenario-finding and multi-level policy-reasoning leads us to model policies in first-order logic. While many firewall-analysis tools are grounded in logic, most use propositional models for which analysis questions are decidable and efficient. In general, one cannot compute an exhaustive and finite set of scenarios witnessing first-order logic formulas. Fortunately, the formulas corresponding to many common firewall-analysis problems do yield such sets. Margrave identifies such cases automatically, thus providing exhaustive analysis for richer policies and queries than other tools. Demonstrating that firewall analyzers can benefit from first-order logic without undue cost is a key contribution of this paper.

Our other key contribution lies in how we decompose IOS configurations into policies for analysis. Single firewall configurations cover many functions, such as access filtering, routing, and switching. Margrave's IOS compiler generates separate policies for each task, thus enabling analysis of either specific functionality or whole-firewall behavior. Task-specific policies aid in isolating causes of problematic behaviors. Our firewall models support standard and most extended ACLs, static NAT, ACL-based and map-based dynamic NAT, static routing, and policy-based routing. Our support for state is limited to reflexive access-lists; it does not include general dynamic NAT, deep packet inspection, routing via OSFP, or adaptive policies. Margrave has an iptables compiler in development; other types of firewalls, such as Juniper's JunOS, fit our model as well.

A reader primarily interested in a tool description can read Sections 2, 6, and 7 for a sense of Margrave and how it differs from other firewall-analysis tools. Section 2 illustrates Margrave's query language and scenario-based output using a multi-step example. Section 3 describes the underlying theory (based on first-order logic), including our notion of policies. Section 4 shows how firewall questions map into Margrave. Section 5 describes the implementation, including the compiler for firewall-configurations and a query-rewriting technique that often improves performance. Section 6 presents experimental evaluation on both network-forum posts and an in-use enterprise firewall. Section 7 describes related work. Section 8 concludes with perspective and future work.

2 Margrave in Action on Firewalls

Margrave presents scenarios that satisfy user-specified queries about firewall behavior. Queries state a behavior of interest and optional controls on which data to consider when computing scenarios. Scenarios contain attributes of packet contents that make the query hold. A separate command language controls how scenarios are displayed. The extended example in this section highlights Margrave's features; Table 1 summarizes which of these features are supported by other available (either free or commercial) firewall analyzers. The Margrave website [22] contains sources for all examples.

In this paper, a *firewall* encompasses filtering (via access-lists), NAT transformation, and routing; we reserve the term *router* for the latter component. The IOS configuration in Figure 1 defines a simple firewall with only filtering. This firewall controls two interfaces (`fe0` and `vlan1`). Each has an IP address and an access-list to filter traffic as it enters the interface; in lines 3 and 7, the number (101 or 102) is a label that associates access rules (lines 9-16) with each interface, while the `in` keyword specifies that the rules should apply on entry. Rules are checked in order from top to bottom; the first rule whose conditions apply determines the decision on a packet. This firewall allows inbound web and mail traffic to the corresponding servers (the `.10` and `.11` hosts), but denies a certain blacklisted IP address (the `10.1.1.2` host). All traffic arriving at the inside-facing interface `vlan1` is allowed. As this filter is only concerned with packets as they arrive at the firewall, our queries refer to the filter as `InboundACL`.

```
1  interface fe0
2  ip address 10.150.1.1 255.255.255.254
3  ip access-group 101 in
4  !
5  interface vlan1
6  ip address 192.128.5.1 255.255.255.0
7  ip access-group 102 in
8  !
9  access-list 101 deny ip host 10.1.1.2 any
10 access-list 101 permit tcp
11             any host 192.168.5.10 eq 80
12 access-list 101 permit tcp
13             any host 192.168.5.11 eq 25
14 access-list 101 deny any
15 !
16 access-list 102 permit any
```

Figure 1: Sample IOS configuration

Basic Queries: All firewall analyzers support basic queries about which packets traverse the firewall. The following Margrave query asks for an inbound packet that `InboundACL` permits:

```
EXPLORE InboundACL:Permit(<req>)

SHOW ONE
```

Query 1

`EXPLORE` clauses describe firewall behavior; here, the behavior is simply to permit packets. `<req>` is shorthand for a sequence of variables denoting the components of a request (detailed in Section 4):

$$\langle \textit{ahostname}, \textit{src-addr-in}, \textit{src-port-in}, \textit{protocol}, \ldots \rangle.$$

Users can manually define this shorthand within Margrave; details and instructions for passing queries into Margrave are in the tool distribution [22]. `SHOW ONE` is an output-configuration command that instructs Margrave to display only a single scenario. The resulting output indicates the packet contents:

```
********* SOLUTION FOUND at size = 15
src-addr-in: IPAddress
protocol: prot-tcp
dest-addr-in: 192.168.5.10
src-port-in: port
exit-interface: interface
entry-interface: fe0
dest-port-in: port-80
length: length
ahostname: hostname-router
src-addr-out: IPAddress
message: icmpmessage
```

Result

	ITVal	Fireman	Prometheus	ConfigChecker	Fang/AlgoSec	Vantage
Which packets	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
User-defined queries	$\checkmark$		?	$\checkmark$	$\checkmark$	$\checkmark^{\text{nip}}$
Rule Responsibility	$\checkmark$	?	$\checkmark^{-}$	$\sim$	$\checkmark$	$\checkmark$
Rule Relationships	$\sim$	$\checkmark^{-}$	$\checkmark$	$\checkmark^{-}$	$\checkmark^{\text{nip}}$	$\checkmark$
Change-impact	?			$\checkmark$	$\checkmark^{\text{nip}}$	$\checkmark^{-}$
First-order queries	?		?			?
Support NAT	$\checkmark$		$\checkmark$	$\checkmark$	$\checkmark$	
Support Routing	$\checkmark$		$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark^{\text{nip}}$
Firewall Networks	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark^{\text{nip}}$
Language integration						$\checkmark$
Commercial Tool?	no	no	yes	no	yes	yes

Table 1: Feature comparison between Margrave and other available firewall-analysis tools. In each cell, $\checkmark$ denotes included features; $\checkmark^{\text{nip}}$ denotes features reported by the authors in private communication but not described in published papers; $\checkmark^{-}$ denotes included features with more limited scope than in Margrave; $\sim$ denotes features that can be simulated, but aren't directly supported; ? denotes cases for which we aren't sure about support. Section 7 describes nuances across shared features and discusses additional research for which tools are not currently available.

This scenario shows a TCP packet (line 3) arriving on the fast-ethernet interface (line 7), bound for the web server (line 4, with line 11 of Figure 1) on port 80 (line 8). The generic `IPaddress` in lines 2 and 11 should be read as "any IP address not mentioned explicitly in the policy"; lines 5 and 6 are similarly generic. Section 5 explains the `size=15` report on line 1.

A user can ask for additional scenarios that illustrate the previous query via the command `SHOW NEXT`: Once Margrave has displayed all unique scenarios, it responds to `SHOW NEXT` queries with `no results`.

To check whether the filter accepts packets from the blacklisted server, we constrain `src-addr-in` to match the blacklisted IP address and examine only packets that arrive on the external interface. Both `src-addr-in` and `entry-interface` are variable names in `<req>`. The `IS POSSIBLE?` command instructs Margrave to display false or true, rather than detailed scenarios.

```
EXPLORE
InboundACL:Permit(<req>) AND
10.1.1.2 = src-addr-in AND
fe0 = entry-interface

IS POSSIBLE?
```
Query 2

In this case, Margrave returns false. Had it returned true, the user could have inspected the scenarios by issuing a `SHOW ONE` or `SHOW ALL` command.

Rule-level Reasoning: Tracing behavior back to the responsible rules in a firewall aids in both debugging and confirming that rules are fulfilling their intent. To support reasoning about rule effects, Margrave automatically defines two formulas for every rule in a policy (where R is a unique name for the rule):

- R`_matches(<req>)` is true when `<req>` satisfies the rule's conditions, and
- R`_applies(<req>)` is true when the rule both matches `<req>` and determines the decision on `<req>` (as the first matching rule within the policy).

Distinguishing these supports fine-grained queries about rule behavior. Margrave's IOS compiler constructs the R labels to uniquely reference rules across policies. For instance, ACL rules that govern an interface have labels of the form *hostname*-*interface*-`line`#, where *hostname* and *interface* specify the names of the host and interface to which the rule is attached and # is the line number at which the rule appears in the firewall configuration file.

The following query refines query 2 to ask for decision justification: the `EXPLORE` clause now asks for `Deny` packets, while the `INCLUDE` clause instructs Margrave to compute scenarios over the two `Deny` rules as well as the formulas in the `EXPLORE` clause:

```
EXPLORE
InboundACL:Deny(<req>) AND
10.1.1.2 = src-addr-in AND
fe0 = entry-interface
INCLUDE
InboundACL1:Router-fe0-line9_applies(<req>),
InboundACL1:Router-fe0-line14_applies(<req>)

SHOW REALIZED
InboundACL1:Router-fe0-line9_applies(<req>),
InboundACL1:Router-fe0-line14_applies(<req>)
```
Query 3

The `SHOW REALIZED` command asks Margrave to display the subset of listed facts that appear in some resulting scenario. The following results indicate that the rule at line 9 does (at least sometimes) apply. More telling, however, the absence of the rule at line 14 (the catch-all deny) indicates that that rule *never* applies to any packet from the blacklisted address. Accordingly, we conclude that line 9 processes all blacklisted packets.

```
< InboundACL:line9_applies(<req>) >
```

Result

The INCLUDE clause helps control Margrave's performance. Large policies induce many rule-matching formulas; enabling these formulas only as needed trims the scenario space. SHOW REALIZED (and its dual, SHOW UNREALIZED) controls the level of detail at which users view scenarios. The lists of facts that do (or do not) appear in scenarios often raise red flags about firewall behavior (such as an unexpected port being involved in processing a packet). Unlike many verification tools, Margrave does not expect users to have behavioral requirements or formal security goals on hand. Lightweight summaries such as SHOW REALIZED try to provide information that suggests further queries.

Computing Overshadowed Rules through Scripting: Query 3 checks the relationship between two rules on particular packets. A more general question asks which rules *never* apply to *any* packet; we call such rules *superfluous*. The following query computes superfluous rules:

```
EXPLORE true
UNDER InboundACL
INCLUDE
 InboundACL:router-fe0-line9_applies(<req>),
 InboundACL:router-fe0-line10_applies(<req>),
 InboundACL:router-fe0-line12_applies(<req>),
 InboundACL:router-fe0-line14_applies(<req>),
 InboundACL:router-vlan1-line16_applies(<req>)

SHOW UNREALIZED
 InboundACL:router-fe0-line9_applies(<req>),
 InboundACL:router-fe0-line10_applies(<req>),
 InboundACL:router-fe0-line12_applies(<req>),
 InboundACL:router-fe0-line14_applies(<req>),
 InboundACL:router-vlan1-line16_applies(<req>)
```

Query 4

As this computation doesn't care about request contents, the EXPLORE clause is simply true. The heart of this query lies in the INCLUDE clause and the SHOW UNREALIZED command: the first asks Margrave to consider all rules; the second asks for listed facts that are never true in any scenario. UNDER clauses load policies referenced in INCLUDE but not EXPLORE clauses.

While the results tell us which rules never apply, they don't indicate which rules overshadow each unused rule. Such information is useful, especially if an overshadowing rule ascribes the opposite decision. Writing queries to determine justification for each superfluous rule, however, is tedious. Margrave's query language is embedded in a host language (Racket [13], a descendent of Scheme) through which we can write scripts over query results. In this case, our script uses a Margrave command to obtain lists of rules that yield each of *Permit* and *Deny*, then issues queries to isolate overshadowing rules for each superfluous rule. These are similar to other queries in this section. Scripts could also compute hotspot rules that overshadow a large percentage of other rules.

Change-Impact: Sysadmins edit firewall configurations to provide new services and correct emergent problems. Edits are risky because they can have unexpected consequences such as allowing or restricting traffic that the edit should not have affected. Expecting sysadmins to have formal security requirements against which to test policy edits is unrealistic. In the spirit of lightweight analyses that demand less of users, Margrave computes scenarios illustrating packets whose decision or applicable rule changes in the face of edits.

For example, suppose we add the new boldface rule below to access-list 101 (the line numbers start with 14 to indicate that lines 1–13 are identical to those in Figure 1):

```
14 access-list 101 deny tcp
15        host 10.1.1.2 host 192.168.5.10 eq 80
```

If we call the modified filter InboundACL_new, the following query asks whether the original and new InboundACLs ever disagree on Permit decisions:

```
EXPLORE
(InboundACL:Permit(<req>) AND
 NOT InboundACL_new:Permit(<req>)) OR
(InboundACL_new:Permit(<req>) AND
 NOT InboundACL:Permit(<req>)))

IS POSSIBLE?
```

Query 5

Margrave returns false, since the rule at line 9 always overrides the new rule. If instead the new rule were:

```
14 access-list 101 deny tcp
15        host 10.1.1.3 host 192.168.5.10 eq 80
```

Margrave would return true on query 5. The corresponding scenarios show packet headers that the two firewalls treat differently, such as the following:

```
********* SOLUTION FOUND at size = 15
src-addr-in: 10.1.1.3
protocol: prot-tcp
dest-addr-in: 192.168.5.10
src-port-in: port
exit-interface: interface
entry-interface: fe0
dest-port-in: port-80
```

Result

As we might expect, this scenario involves packets from 10.1.1.3. A subsequent query could confirm that no other hosts are affected.

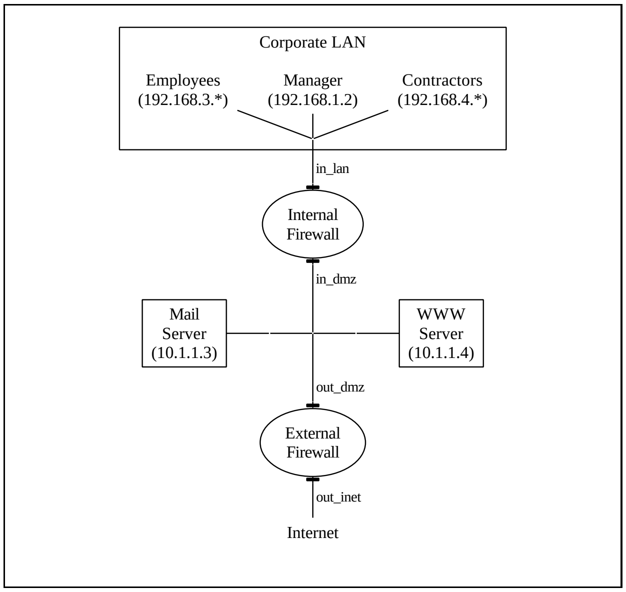

Figure 2: A small-business network-topology

Networks of Firewalls: So far, our examples have considered only single firewalls. Margrave also handles networks with multiple firewalls and NAT. Figure 2 shows a small network with web server, mail server, and two firewalls to establish a DMZ. The internal firewall performs both NAT and packet-filtering, while the external firewall only filters. The firewall distinguishes machines for employees (`192.168.3.*`), contractors (`192.168.4.*`), and a manager (`192.168.1.2`). This example captures the essence of a real problem posted to a networking help-forum.

```
1  hostname int
2  !
3  interface in_dmz
4  ip address 10.1.1.1 255.255.255.0
5  ip nat outside
6  !
7  interface in_lan
8  ip access-group 102 in
9  ip address 192.168.1.1 255.255.0.0
10 ip nat inside
11 !
12 access-list 102 permit tcp any any eq 80
13 access-list 102 deny any
14 !
15 ip nat inside source list 1 interface
16               in_dmz overload
17 access-list 1 permit 192.168.1.1 0.0.255.255
18 !
19 ip route 0.0.0.0 0.0.0.0 in_dmz
20
```

Internal Firewall

Lines 15–17 in the internal firewall apply NAT to traffic from the corporate LAN.[1] Line 11 in the external firewall blacklists a specific external host (`10.200.200.200`). Despite the explicit rule on lines 19–20 in the external firewall, the manager cannot access the web. We have edited the configurations to show only those lines relevant to the manager and web traffic.

```
1  hostname ext
2  !
3  interface out_dmz
4  ip access-group 103 in
5  ip address 10.1.1.2 255.255.255.0
6  !
7  interface out_inet
8  ip access-group 104 in
9  ip address 10.200.1.1 255.255.0.0
10 !
11 access-list 104 deny 10.200.200.200
12 access-list 104 permit tcp any host 10.1.1.4
13                 eq 80
14 access-list 104 deny any
15 !
16 access-list 103 deny ip any
17                 host 10.200.200.200
18 access-list 103 deny tcp any any eq 23
19 access-list 103 permit tcp host 192.168.1.2
20                 any eq 80
21 access-list 103 deny any
22
```

External Firewall

The following query asks "What rules deny a connection from the manager's PC (line 2) to port 80 (line 10) somewhere outside our network (line 8) other than the blacklisted host (line 9)?"

```
1  EXPLORE prot-TCP = protocol AND
2  192.168.1.2 = fw1-src-addr-in AND
3  in_lan = fw1-entry-interface AND
4  out_dmz = fw2-entry-interface AND
5  hostname-int = fw1 AND
6  hostname-ext = fw2 AND
7
8  fw1-dest-addr-in IN 10.200.0.0/255.255.0.0
9  NOT 10.200.200.200 = fw1-dest-addr-in AND
10 port-80 = fw1-dest-port-in AND
11
12 internal-result(<reqfull-1>) AND
13
14 (NOT passes-firewall(<reqpol-1>) OR
15  internal-result(<reqfull-2>) AND
16  NOT passes-firewall(<reqpol-2>))
17
18 UNDER InboundACL
19 INCLUDE
20 InboundACL:int-in_lan-line-12_applies
21   (<reqpol-1>),
22 InboundACL:int-in_lan-line-17_applies
23   (<reqpol-1>),
24 InboundACL:ext-out_dmz-line-19_applies
25   (<reqpol-2>),
26 InboundACL:ext-out_dmz-line-21_applies
27   (<reqpol-2>),
28 InboundACL:ext-out_dmz-line-24_applies
29   (<reqpol-2>)
```

Query 6

Lines 12–16 capture both network topology and the effects of NAT. The `internal-result` and `passes-firewall` formulas capture routing in the face

[1] In this example, we use the 10.200.* private address space to represent the public IP addresses.

of NAT and passing through the complete firewall (including routing, NAT and ACLs) whose hostname appears in the request, respectively; Section 4 describes them in detail. The variables sent to the two `passes-firewall` formulas through `<reqpol-1>` and `<reqpol-2>` encode the topology: for example, these shorthands use the same variable name for *dest-addr-out* in the internal firewall and *src-addr-in* in the external firewall. The `fw1-entry-interface` and `fw2-entry-interface` variables (bound to specific interfaces in lines 3–4) appear as the entry interfaces in `<reqpol-1>` and `<reqpol-2>`, respectively.

A `SHOW REALIZED` command over the `INCLUDE` terms (as in query 3) indicates that line 21 of the external firewall configuration is denying the manager's connection. Asking Margrave for a scenario for the query (using the `SHOW ONE` command) reveals that the internal firewall's NAT is changing the packet's source address:

```
...
fw1-src-addr-out=fw2-src-addr_=
  fw2-src-addr-out: 10.1.1.1
fw1-src-addr_=fw1-src-addr-in: 192.168.1.2
```

Result

The external firewall rule (supposedly) allowing the manager to access the Internet (line 19) uses the internal pre-NAT source address; it never matches the post-NAT packet. Naïvely editing the NAT policy, however, can leak privileges to contractors and employees. Change-impact queries are extremely useful for confirming that the manager, and *only* the manager, gain new privileges from an edit. An extended version of this example with multiple fixes and the change-impact queries, is provided in the Margrave distribution.

Summary: These examples illustrate Margrave's ability to reason about both combinations of policies and policies at multiple granularities. The supported query types include asking which packets satisfy a condition (query 1), verification (query 2), rule responsibility (query 3), rule relationships (query 4) and change-impact (query 5). A formal summary of the query language and its semantics is provided with the Margrave distribution.

3 Defining Scenarios

Margrave views a *policy* as a mapping from requests to decisions. In a firewall, requests contain packet data and some routing data, while decisions include *Permit* and *Deny* (for ACLs), *Drop* (for routing), and a few others. Policies often refer to relationships between objects, such as "permit access by machines *on the same subnet*". Queries over policies often require quantification: "*Every* host on the local subnet can access *some* gateway router". First-order logic extends propositional logic with relational formulas (such as `SameSubnet(121.34.42.133,121.34.42.166)`) and quantifiers (∀ and ∃). For firewall policies, the available relations include the decisions, R`_matches` and R`_applies` (as shown in Section 2) and unary relations capturing sets of IP addresses, ports, and protocols.

Margrave maps both policies and queries into first-order logic formulas. To answer a query, Margrave first conjoins the query formula with the formulas for all policies referenced in the query, then computes solutions to the combined formula. A *solution* to a first-order formula contains a set of elements to quantify over (the *universe*) and two mappings under which the formula is true: one maps each relation to a set of tuples over the universe, and another maps each unquantified variable in the query to an element of the universe.[2] For example, the formula

$$\forall x\ host(x) \implies \exists y\ (router(y) \land CanAccess(x, y))$$

says that "every host can access some router". One solution has a universe of $\{h1, r1, r2\}$ and relation tuples *host(h1)*, *router(r1)*, *router(r2)*, and *CanAccess(h1,r2)* (the formula has no unquantified variables). Other solutions could include more hosts and routers, with more access connections between them. Solutions may map multiple variables to the same universe element. This is extremely useful for detecting corner cases in policy analysis; while humans often assume that different variables refer to different objects, many policy errors lurk in overlaps (such as a host being used a both web server and mail server). *Scenarios* are simply solutions to the formula formed of a query and the policies it references.

In general, checking whether a first-order formula has a solution (much less computing them all) is undecidable. Intuitively, the problem lies in determining a sufficient universe size that covers all possible solutions. This problem is disconcerting for policy analysis: we would like to show users an exhaustive set of scenarios to help them ensure that their policies are behaving as intended in all cases. Fortunately, Margrave can address this problem in most cases; Section 5 presents the details.

4 Mapping Firewalls to the Theory

There is a sizeable gap between the theory in Section 3 and a policy in a real-world language, such as the example in Figure 1. To represent policies in the theory, we must describe the shapes of requests, the available decisions, what relations can appear in formulas, and how policy rules translate into formulas. Section 2 used several relations relevant to firewalls, such

[2] In logical terms, a solution combines a first-order model and an environment binding free variables to universe elements.

```
(Policy InboundACL uses IOS-vocab
 (Rules
  ...
  (Router-fe0-line10 =
   (Permit hostname, ...) :-
   (hostname-Router hostname)
   (fe0 entry-interface)
   (IPAddress src-addr-in)
   (prot-tcp protocol)
   (Port src-port-in)
   (192.168.5.10 dest-addr-in)
   (port-80 dest-port-in))
  ...)
 (RComb FAC))
```

Figure 3: A Margrave policy specification

as `passed-firewall`. Margrave defines these relations and other details automatically via several mechanisms.

Policies: Figure 3 shows part of the result of compiling the IOS configuration in Figure 1 to Margrave's intermediate policy language. The fragment captures the IOS rule on line 10. `(Permit hostname, ...)` specifies the decision and states a sequence of variable names corresponding to a request. The `:-` symbol separates the decision from the conditions of the rule. Formula `(prot-tcp protocol)`, for example, captures that TCP is the expected protocol for this rule. Margrave represents constants (such as decisions, IP addresses, and protocols) as elements of singleton unary relations. A scenario that satisfies this rule will map the *protocol* variable to some element of the universe that populates the `prot-tcp` relation. The other conditions of the original rule are captured similarly. The `(RComb FAC)` at the end of the policy tells Margrave to check the policy rules in order (`FAC` stands for "first applicable"). The first line of the policy ascribes the name `InboundACL`.

Decomposing IOS into policies: Figure 4 shows our high-level model of IOS configurations. Firewalls perform packet filtering, packet transformation, and internal routing; the first two may occur at both entry to and exit from the firewall. Specifically, packets pass through the inbound ACL filter, inside NAT transformation, internal routing, outside NAT transformation, and finally the outbound ACL filter on their way through the firewall. The intermediate stages define additional information about a packet (as shown under the stage names): inside NAT may yield new address and port values; internal routing determines the next-hop and exit interface; outside NAT may yield further address and port values.

Internal routing involves five substages, as shown in Figure 6. Margrave creates policies (à la Figure 3) for each of the five substages. The `-Switching` policies determine whether a destination is directly connected to the firewall; the `-Routing` policies bind the `next-hop` IP address for routing. In addition, Margrave generates four policies called `InboundACL`, `OutboundACL`, `InsideNAT`, and `OutsideNat`. The two `-ACL` policies contain filtering rules for all interfaces.

Requests and Decisions: Margrave automatically defines a relation for each decision rendered by each of the 9 subpolicies (e.g., `InboundACL:Permit` in query 1). Each relation is defined over requests, which contain packet headers, packet attributes, and values generated in the intermediate stages; the boxes in Figure 4 collectively list the request contents. As Margrave is not stateful, it cannot update packet headers with data from intermediate stages. The contents of a request reflect the intermediate stages' actions: for example, if the values of `src-addr_` and `src-addr-out` are equal, then `OutsideNAT` did not transform the request's packet. Currently, Margrave shares the same request shape across all 9 subpolicies (even though `InboundACL`, for example, only examines the packet header portion).

Flows between subpolicies: Margrave encodes flows among the 9 subpolicies through three relations (over requests) that capture the subflows marked in Figure 4.

- Internal routing either assigns an exit interface and a next-hop to a packet or drops the packet internally. Margrave uses a special exit-interface value to mark dropped packets; the `int-dropped` relation contains requests with this special exit-interface value. Any request that is not in `int-dropped` successfully passes through internal routing.
- Unlike internal routing, NAT never drops packets. At most, it transforms source and destination ports and addresses. Put differently, NAT is a function on packets. `internal-result` captures this function: it contains all requests whose `next-hop`, `exit-interface`, and `OutsideNAT` components are consistent with the packet header and `InsideNAT` components (as if the latter were inputs to a NAT function).
- ACLs permit or deny packets. The relation `passes-firewall` contains requests that the two ACLs permit, are in `internal-result` (i.e., are consistent with NAT), and are not in `int-dropped` (i.e., are not dropped in internal routing).

Our IOS compiler automatically defines each of these relations as a query in terms of the 9 IOS subpolicies (capturing topology as in query 6). Margrave provides a `RENAME` command that saves query results under a user-specific name for use in later queries. Users can name any set of resulting scenarios in this manner.

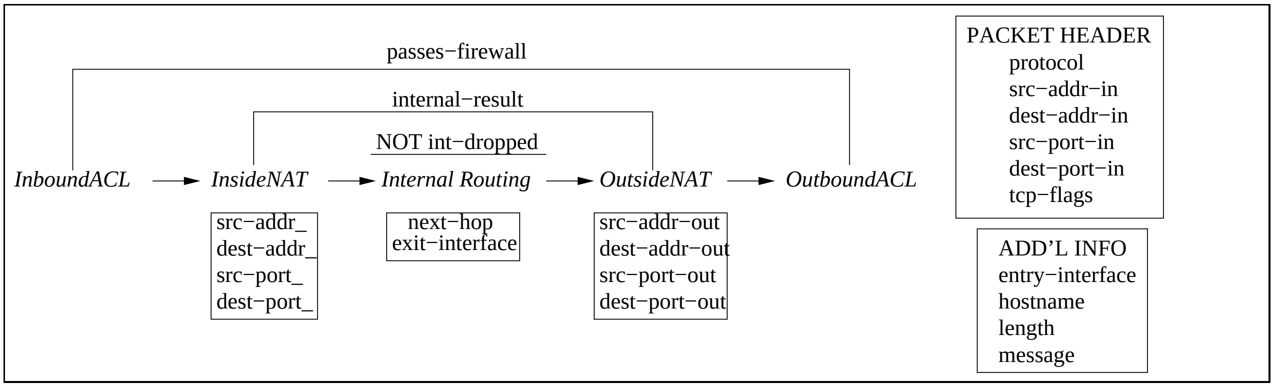

Figure 4: Margrave's decomposition of firewall configurations

```
(PolicyVocab IOS-vocab
 (Types
  (Interface : interf-drop
               (interf-real vlan1 fe0))
  (IPAddress :
    192.128.5.0/255.255.255.0
    10.1.1.0/255.255.255.254
    192.168.5.11
    192.168.5.10
    10.1.1.2)
  (Protocol : prot-ICMP prot-TCP prot-UDP)
  (Port: port-25 port-80)
  (Decisions Permit Deny ...)
  ...
  (disjoint-all Protocol)
  (nonempty Port)
  ...
)
```

Figure 5: A Margrave vocabulary specification

Vocabularies: The 9 subpolicies share ontology about ports and IP addresses. Margrave puts domain-knowledge common to multiple policies in a *vocabulary* specification; the first line of a policy specification references its vocabulary through the `uses` keyword. Figure 5 shows a fragment of the vocabulary for IOS policies: it defines datatypes (such as `Protocol`) and their elements (correspondingly, `prot-ICMP`, `prot-TCP`, `prot-UDP`).

Vocabularies also capture domain constraints such as "all protocols are distinct" or "there must be at least one port" (both shown in Figure 5). While these constraints may seem odd, they support Margrave's scenario-finding model. Some potential "solutions" (as described in Section 3) are nonsensical, such as one which assigns two distinct numbers to the same physical port. Domain constraints rule out nonsensical scenarios.

Generalizing Beyond Firewalls

The policy- and vocabulary-specifications in Figures 3 and 5 show how to map specific domains into Margrave. Datatypes, constraints, and rules capture many other kinds of policies, including access-control policies, hypervisor configurations, and product-line specifications. Indeed, this general-purpose infrastructure is another advantage of Margrave over other firewall-analysis tools: Margrave can reason about interactions between policies from multiple languages for different configuration concerns. For example, if data security depends on a particular interaction between a firewall and an access-control policy, both policies and their interaction can be explored using Margrave. We expect this feature to become increasingly important as enterprise applications move onto the cloud and are protected through the interplay of multiple policies from different sources.

5 Implementation

Margrave consists of a frontend read-eval-print loop (REPL) written in Racket [13] and a backend written in Java. The frontend handles parsing (of queries, commands, policies, and vocabularies) and output presentation. The actual analysis and scenario generation occurs in the backend.

5.1 The Scenario-Finding Engine

Margrave's backend must produce sets of solutions to first-order logic formulas. We currently use a tool called Kodkod [32] that produces solutions to first-order formulas using SAT solving.[3] SAT solvers handle propositional formulas. Kodkod bridges the gap from first-order to propositional formulas by asking users for a finite universe-size; under a finite universe-size, first-order formulas translate easily to propositional ones. Figure 7 shows an example of the rewriting process. Every solution produced using a bounded size is legitimate (in logical terms, our analysis is *sound*). However, analysis will miss solutions that require a universe larger than the given size (in logical terms, it is not *complete*).

[3] Within Kodkod, we use a SAT-solver called SAT4J [8].

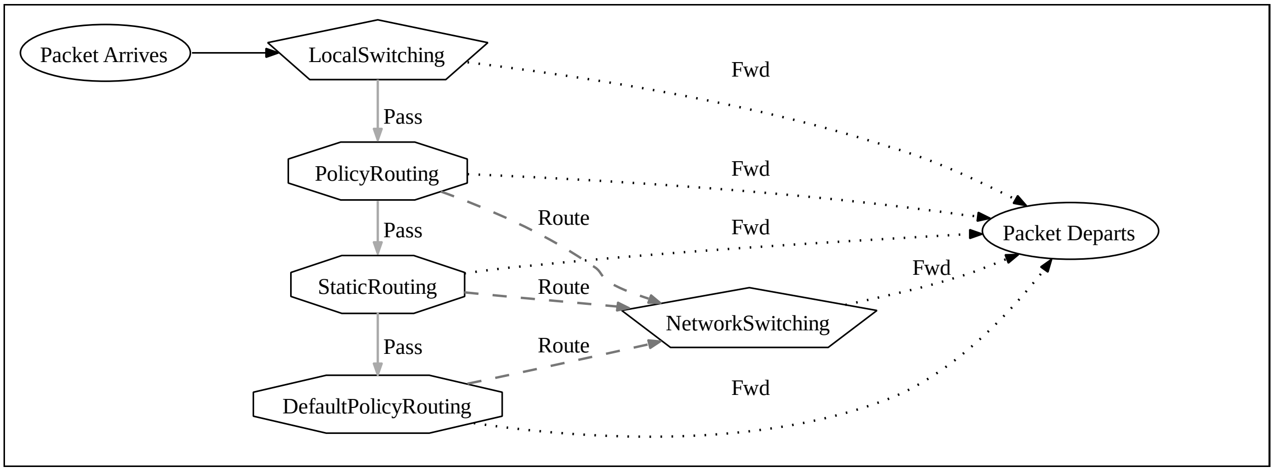

Figure 6: Internal flow of packets within a router. Edges are labeled with decisions rendered by the policies at the source of the edge. Routing policies determine the next-hop IP address, while switching policies send traffic to directly to a connected device.

Fortunately, most firewall queries (including those in this paper) correspond to formulas with no universal ($\forall$) quantifiers. For such formulas, the number of existentially-quantified variables provides a sufficient universe size to represent all solutions. Margrave automatically supplies Kodkod with the universe bound for such formulas. For queries that do not have this form, such as "can *every* host reach some other machine on the network", either Margrave or the user must supply a universe size for the analysis. The query language has an optional `CEILING` clause whose single argument is the desired universe size. If `CEILING` is omitted, Margrave uses a default of 6. Experience with Kodkod in other domains suggests that small universe sizes can yield useful scenarios [15]. If Margrave can compute a sufficient bound but the user provides a lower `CEILING`, Margrave will only check up to the `CEILING` value. Whenever Margrave cannot guarantee that scenario analysis is complete, it issues a warning to the user. The `size=15` statement in the first line of scenarios shown in Section 2 report the universe-size under which Margrave generated the scenario.

`CEILING` settings may impact the results of commands. Margrave includes a `SHOW UNREALIZED` command that reports relations that are not used in any resulting scenario. However, a relation T might be unpopulated at one `CEILING` value yet populated at a higher value. For example, in the formula $\exists x \neg T(x)$, T is never used at `CEILING` 1, but can be realized at `CEILING` 2. Margrave users should only supply `CEILING` values if they appreciate such consequences.

Overall, we believe sacrificing exhaustiveness for the expressive power of first-order logic in policies and queries is worthwhile, especially given the large number of practical queries that can be checked exhaustively.

5.2 Rewriting Firewall Queries

Under large universe sizes, both the time to compute scenarios and the number of resulting scenarios increase. The latter puts a particular burden on the end-user who has to work through the scenarios. Query language constructs like `SHOW REALIZED` summarize details about the scenarios in an attempt to prevent the exhaustive from becoming exhausting. However, query optimizations that reduce universe sizes have more potential to target the core problem.

Most firewall queries have the form $\exists\, \overline{req}\ \alpha$, where α typically lacks quantifiers. Requests have 16 or 20 components (as shown in Figure 4), depending on whether they reference `internal-result`. Margrave therefore analyzes all-existential queries under a universe size of 16 or 20. However, these queries effectively reference a *single request* with attributes as detailed in α. This suggests that we could rewrite this query with a single quantified variable for a request and additional relations that encode the attributes. For example:

$$\exists\, pt\_in\ \exists\, pt\_out : route(pt\_in, pt\_out)$$

becomes

$$\exists\, pkt : is\_ptIn(pkt, i) \wedge is\_ptOut(pkt, o) \wedge route(i, o)$$

Effectively, these new relations lift attributes from the individual packet fields to the packet as a whole.

Formulas rewritten in this way require a universe size of only 1, for which scenario generation stands to be much faster and to yield fewer solutions. The tradeoff, however, lies in the extra relations that Margrave introduces to lift attributes to the packet level. Additional relations increase the time and yield of scenario computations, so the rewriting is not guaranteed to be a net win.

The original sentence:

$$\forall x\ host(x) \implies \exists y\ (router(y) \wedge CanAccess(x, y))$$

Assume a universe of size 2 with elements A and B. Expand the $\forall$-formula with a conjunction over each of A and B for x:

$$\begin{aligned} host(A) &\implies \exists y\ (router(y) \wedge CanAccess(A, y)) \wedge \\ host(B) &\implies \exists y\ (router(y) \wedge CanAccess(B, y)) \end{aligned}$$

Next, expand each $\exists$-formula with a disjunction over each of A and B for y:

$$\begin{aligned} host(A) &\implies (router(A) \wedge CanAccess(A, A)) \vee \\ &\quad\ (router(B) \wedge CanAccess(A, B)) \wedge \\ host(B) &\implies (router(A) \wedge CanAccess(B, A)) \vee \\ &\quad\ (router(B) \wedge CanAccess(B, B)) \end{aligned}$$

Replace each remaining formula with a propositional variable (e.g., $router(A)$ becomes p_2):

$$\begin{aligned} p_1 &\implies (p_2 \wedge p_3) \vee \\ &\quad\ (p_4 \wedge p_5) \wedge \\ p_6 &\implies (p_2 \wedge p_7) \vee \\ &\quad\ (p_4 \wedge p_8) \end{aligned}$$

Figure 7: Converting a first-order formula to a propositional one at a bounded universe size

Table 2 presents experimental results on original versus rewritten queries. In practice, we find performance improves when the query is unsatisfiable or the smallest model is large. A user who expects either of these conditions to hold can enable the rewriting through a query-language flag called `TUPLING`. All performance figures in this paper were computed using `TUPLING`.

6 Evaluation

We have two main goals in evaluating Margrave. First, we want to confirm that our query language and its results support debugging real firewall configuration-problems; in particular, the scenarios should accurately point to root causes of problems. We assume a user who knows enough firewall basics to ask the questions underlying a debugging process (Margrave does not, for example, pre-emptively try queries to automatically isolate a problem). Second, we want to check that Margrave has reasonable performance on large policies, given that we have traded efficient propositional models for richer first-order ones.

We targeted the first goal by applying Margrave to problems posted to network-configuration help-forums (Sections 6.1 and 6.2). Specifically, we phrased the poster's reported problem through Margrave queries and sought fixes based on the resulting scenarios. In addition, we used Margrave to check whether solutions suggested in follow-up posts actually fixed the problem without affecting other traffic. The diversity of firewall features that appear in forum posts demanded many compiler extensions, including reflexive access-lists and TCP flags. That we could do this purely at the compiler level attests to the flexibility of Margrave's intermediate policy- and vocabulary-languages (Section 4).

Rules	# Vars	Min Size	Not Tupled	Tupled
100	3	3	694ms	244ms
1000	14	6	7633ms	1221ms
1000	14	10	17659ms	1219ms
1000	14	14	32116ms	1205ms

Table 2: Run-time impact of `TUPLING` on ACL queries. The first column contains the number of rules in each ACL. The second column lists the number of existentially-quantified variables in the query; we include one 3-variable (non-firewall) query to illustrate the smaller gains on smaller variable counts. The 14-variable ACLs are older firewall examples with smaller request tuples. The "Min Size" column indicates the universe size for the smallest scenario that satisfied the query. Larger minimum sizes have a larger search space.

We targeted the second goal by applying Margrave to an in-use enterprise firewall-configuration containing several rule sets and over 1000 total rules (Section 6.3). Margrave revealed some surprising facts about redundancy in the configuration's behavior. Individual queries uniformly execute in seconds.

Notes on Benchmarking Our figures report Margrave's steady-state performance; they omit JVM warmup time. Policy-load times are measured by loading different copies of the policy to avoid caching bias. All performance tests were run on an Intel Core Duo E7200 at 2.53 Ghz with 2 GB of RAM, running Windows XP Home. Performance times are the mean over at least 100 individual runs; all reported times are ±200ms at the 95-percent confidence level. Memory figures report private (i.e., not including shared) consumption.

6.1 Forum Help: NAT and ACLs

"My servers cannot get access into the internet, even though I will be able to access the website, or even FTP... I don't really know what's wrong. Can you please help? Here is my current configuration..."

In our first forum example [4], the poster is having trouble connecting to the Internet from his server. He believes that NAT is responsible, and has identified the router as the source of the problem. The configuration included with the post appears in Figure 8 (with a slight semantics-preserving modification[4]).

A query (not shown) confirms that the firewall is blocking the connection. Our knowledge of firewalls indicates that packets are rejected either enroute to, or on return from, the webserver. Queries for these two cases are similar; the one checking for response packets is:

```
EXPLORE
NOT src-addr-in
  IN 192.168.2.0/255.255.255.0 AND
FastEthernet0 = entry-interface AND
prot-TCP = protocol AND
port-80 = src-port-in AND
internal-result(<reqfull>) AND
passes-firewall(<reqpol>)

IS POSSIBLE?
```

Query 7

Margrave reports that packets to the webserver are permitted, but responses are dropped. The resulting scenarios all involve source ports 20, 21, 23, and 80 (easily confirmed by re-running the query with a `SHOW REALIZED` command asking for only the port numbers). This is meaningful to a sysadmin: an outgoing web request is always made from an *ephemeral* port, which is never less than 1024. This points to the problem: the router is rejecting all returning packets. ACL 102 (Figure 8, lines 25–29) ensures that the server sees only incoming HTTP, FTP, and TELNET traffic, at the expense of rejecting the return traffic for any connections that the server initiates.

Enabling the server to access other webservers involves allowing packets *coming from* the proper destination ports. Methods for achieving this include:

1. Permit TCP traffic *from* port 80, via the edit:

```
access-list 102 permit tcp
             any host 209.172.108.16 eq 23
access-list 102 permit tcp any eq 80 any
access-list 102 deny tcp
             any host 209.172.108.16
```

2. Allow packets whose ack flags are set via the *established* keyword (or, in more recent versions, the *match-all +ack* option). This suggestion guards against spoofing a packet's source port field and allows servers to listen on unusual ports.

3. Use stateful monitoring of the TCP protocol via reflexive access-lists or the *inspect* command. This guards against spoofing of the TCP ACK flag.

Follow-up posts in the forum suggested options 1 and 3. Margrave can capture the first two options and the reflexive access-list approach in the third (it does not currently support *inspect* commands). For each of these, we can perform verification queries to establish that the `InboundACL` no longer blocks return packets, and we can determine the extent of the change through a change-impact query.

Space precludes showing the reflexive ACL query in detail. Reflexive ACLs allow return traffic from hosts to which prior packets were permitted. Margrave encodes prior traffic through a series of `connection-` relations over requests. Intuitively, a request is in a `connection-` relation only if the same request with the source- and destination-details reversed would pass through the firewall. Although the connection state is dynamic in practice, its stateless definition enables Margrave to handle it naturally through first-order relations.

Performance: Loading each version of the configuration took between 3 and 4 seconds. The final change-impact query took under 1 second. After loading, running the full suite of queries (including those not shown) required 2751ms. The memory footprint of the Java engine (including all component subpolicies) was 50 MB (19 MB JVM heap, 20 MB JVM non-heap).

6.2 Forum Help: Routing

"there should be a way to let the network `10.232.104.0/22` *access the internet, kindly advise a solution for this..."*

In our second example [29], the poster is trying to create two logical networks: one "primary" (consisting of `10.232.0.0/22` and `10.232.100.0/22`) and one "secondary" (consisting of `10.232.4.0/22` and `10.232.104.0/22`). These logical networks are connected through a pair of routers (TAS and BAZ) which share a serial interface (Figure 9). Neither logical network should have access to the other, but both networks should have access to the Internet—the primary via `10.232.0.15` and the secondary via `10.232.4.10`.

The poster reports two problems: first, the two components of the primary network—`10.232.0.0/22` and `10.232.100.0/22`—cannot communicate with each other; second, the network `10.232.104.0/22` cannot access the Internet. The poster suspects errors in the TAS router configuration (omitted for sake of space).

We start with the first problem. The following query confirms that network `10.232.0.0/22` cannot reach `10.232.100.0/22` via the serial link. The `hostname` formulas introduce names for each individual router

[4] We replaced named interface references in static NAT statements with actual IP addresses; our compiler does not support the former.

```
name-server 207.47.4.2
name-server 207.47.2.178
!
interface FastEthernet0
ip address 209.172.108.16 255.255.255.224
ip access-group 102 in
ip nat outside
speed auto
full-duplex
!
interface Vlan1
ip address 192.168.2.1 255.255.255.0
ip nat inside
!
ip route 0.0.0.0 0.0.0.0 209.172.108.1
!
ip nat pool localnet 209.172.108.16 prefix-length 24
ip nat inside source list 1 pool localnet overload
ip nat inside source list 1 interface FastEthernet0
ip nat inside source static tcp 192.168.2.6 80 209.172.108.16 80
ip nat inside source static tcp 192.168.2.6 21 209.172.108.16 21
ip nat inside source static tcp 192.168.2.6 3389 209.172.108.16 3389
!
access-list 1 permit 192.168.2.0 0.0.0.255
access-list 102 permit tcp any host 209.172.108.16 eq 80
access-list 102 permit tcp any host 209.172.108.16 eq 21
access-list 102 permit tcp any host 209.172.108.16 eq 20
access-list 102 permit tcp any host 209.172.108.16 eq 23
access-list 102 deny tcp any host 209.172.108.16
```

Figure 8: The original configuration for the forum post for Section 6.1

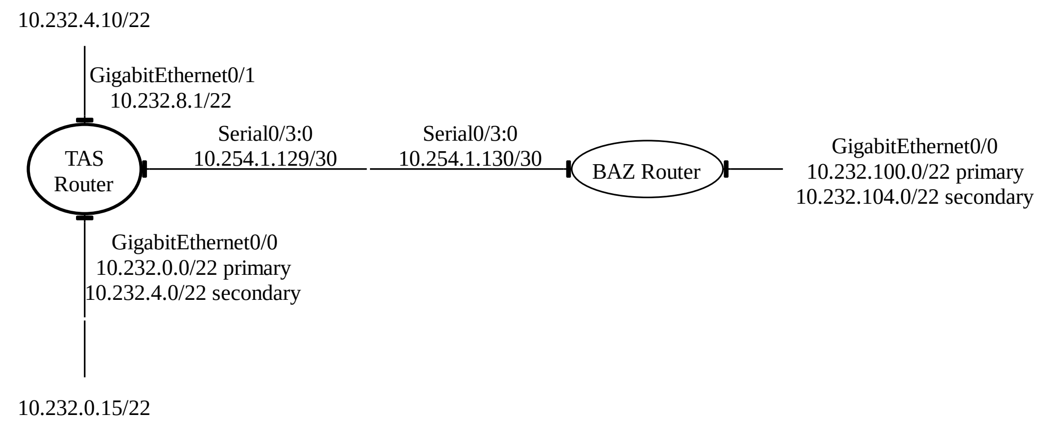

Figure 9: Structure of the network for the forum post for Section 6.2

based on the hostname specification in the IOS configuration; these names appear in the `tasvector-` and `bazvector-` requests. (The `-full-` requests extend the corresponding `-pol-` requests with additional variables needed for `internal-routing`.

```
1  EXPLORE hostname-tas = tas AND
2  hostname-baz = baz AND
3
4  internal-result(<tasvectorfull-fromtas>) AND
5  internal-result(<bazvectorfull-fromtas>) AND
6  passes-firewall(<tasvectorpol-fromtas>) AND
7  passes-firewall(<bazvectorpol-fromtas>) AND
8
9  GigabitEthernet0/0 = tas-entry-interface AND
10 tas-src-addr-in IN
11   10.232.0.0/255.255.252.0 AND
12 tas-dest-addr-in IN 10.232.100.0/255.255.252.0
13 AND "Serial0/3/0:0" = tas-exit-interface AND
14
15 "Serial0/3/0:0" = baz-entry-interface AND
16 GigabitEthernet0/0 = baz-exit-interface
17
18 IS POSSIBLE?
```

Query 8

Margrave returns false, which means that no packets from `10.232.0.0/22` reach `10.232.100.0/22` along this network topology.

By the topology in Figure 9, packets reach the TAS router first. We check whether packets pass through TAS by manually restricting query 8 to TAS (by removing lines 2, 5, 7, 14, and 15); Margrave still returns false. Firewall knowledge suggests three possible problems with the TAS configuration: (1) internal routing could be sending the packets to an incorrect interface, (2) internal routing could be dropping the packets, or (3) the ACLs could be filtering out the packets. Margrave's formulas for reasoning about internal firewall behavior help eliminate these cases: by negating `passed-firewall` on line 6, we determine that the packet does pass through the firewall, so the problem lies in the interface or next-hop assigned during routing. This example highlights the utility of not only having access to these formulas, but also having the ability to negate (or otherwise manipulate) them as any other subformula in a query.

To determine which interfaces the packets are sent on, we relax the query once again to remove the remaining reference to `Serial0/3/0:0` (on line 12) and execute the following `SHOW REALIZED` command:

```
SHOW REALIZED
     GigabitEthernet0/0 = exit-interface,
     "Serial0/3/0:0" = exit-interface,
     GigabitEthernet0/1 = exit-interface
```

Query 9

The output contains only one interface name:

```
{ GigabitEthernet0/0[exit-interface] }
```

Result

According to the topology diagram, packets from `10.232.0.0/22` to `10.232.100.0/22` should be using exit interface `Serial0/3/0:0`; the results, instead, indicate exit interface `GigabitEthernet0/0`. Firewall experience suggests that the router is either switching the correct next-hop address (`10.254.1.130`) to the wrong exit interface, or using the wrong next-hop address. The next query produces the next-hop address:

```
1  EXPLORE hostname-tas = tas AND
2  internal-result(<tasvectorfull-fromtas>) AND
3  passes-firewall(<tasvectorpol-fromtas>) AND
4  GigabitEthernet0/0 = tas-entry-interface AND
5  tas-src-addr-in IN
6    10.232.0.0/255.255.252.0 AND
7  tas-dest-addr-in IN 10.232.100.0/255.255.252.0
8
9  INCLUDE
10 10.232.0.15 = tas-next-hop,
11 10.232.4.10 = tas-next-hop,
12 tas-next-hop IN 10.254.1.128/255.255.255.252,
13 tas-next-hop IN 10.232.8.0/255.255.252.0
14
15 SHOW REALIZED
16 10.232.0.15 = tas-next-hop,
17 10.232.4.10 = tas-next-hop,
18 tas-next-hop IN 10.232.8.0/255.255.252.0,
19 tas-next-hop IN 10.254.1.128/255.255.255.252
```

Query 10

```
{ 10.232.0.15[tas-next-hop] }
```

Result

The next-hop address is clearly wrong for the given destination address. To determine the extent of the problem, we'd like to know whether *all* packets from the given source address are similarly misdirected. That question is too strong, however, as `LocalSwitching` may (rightfully) handle some packets. To ask Margrave for next-hops targeted by some source packet that `LocalSwitching` ignores, we replace line 7 in query 10 with:

```
NOT LocalSwitching:Forward(<routingpol-tas>)
```

Query 11

This once again highlights the value of exposing `LocalSwitching` as a separate relation. The revised query yields the same next-hop, indicating that all non-local packets are routing to `10.232.0.15`, despite the local routing policies. A simple change fixes the problem: insert the keyword **default** into the routing policy:

```
route-map internet permit 10
match ip address 10
set ip default next-hop 10.232.0.15
```

This change ensures that packets are routed to the Internet only as a last resort (i.e., when static destination-based routing fails). Running the original queries against the new specification confirms that the primary subnets now have connectivity to each other. Another query checks that this change does not suddenly enable the primary sub-network `10.232.0.0/22` to reach the secondary sub-network `10.232.4.0/22`.

Now we turn to the poster's second problem: the secondary network 10.232.4.0/22 still cannot access the Internet. As before, we confirm this then compute the next-hop and exit interface that TAS assigns to traffic from the secondary network with an outside destination. The following query (with SHOW REALIZED over interfaces and potential next-hops) achieves this:

```
EXPLORE
tas = hostname-tas AND

internal-result2(<tasvectorfull-fromtas>) AND
firewall-passed2(<tasvectorpol-fromtas>) AND

GigabitEthernet0/0 = tas-entry-interface AND
tas-src-addr-in IN
  10.232.4.0/255.255.252.0 AND

NOT tas-dest-addr-in IN
  10.232.4.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.104.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.0.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.100.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.254.1.128/255.255.255.252 AND
NOT tas-dest-addr-in IN
  192.168.1.0/255.255.255.0 AND
NOT tas-dest-addr-in IN
  10.232.8.0/255.255.252.0
```

Query 12

```
{   gigabitethernet0/0[tas-exit-interface],
     10.232.4.10[tas-next-hop] }
```

Result

The next-hop for the secondary network's Internet gateway is as expected, but the exit-interface is unexpectedly GigabitEthernet0/0 (instead of GigabitEthernet0/1). In light of this scenario, the network diagram reveals a fundamental problem: the gateway 10.232.4.10 should be "on" the same network as the GigabitEthernet0/1 interface (address 10.232.8.1/22); otherwise LocalSwitching will send the packet to the wrong exit interface.

This problem can be resolved by changing the address of either the GigabitEthernet0/1 interface or the next-hop router (10.232.4.10). We chose the latter, selecting an arbitrary unused address in the 10.232.8.0/22 network:

```
route-map internet permit 20
match ip address 20
set ip default next-hop 10.232.8.10
```

Re-running the queries in this new configuration confirms that both goals are now satisfied.

Performance: Loading each version of the configuration took between 3 and 4 seconds. Query 12 took 351 ms. After loading, running the full suite of queries (including those not shown) finished in 8725ms. The memory footprint of the Java engine (including all component subpolicies) was 74 MB (49 MB JVM heap, 21 MB JVM non-heap).

Query	**Time (ms)**
Permit pkt from addr X on interface Y?	1587
Previous with rule responsibility	23317
Change-impact after 1 decision edit	3167
Previous with rule responsibility	24039
Detect all superfluous rules	22578
List overshadows per rule in previous	72178

Table 3: Run-time performance of various queries on the enterprise ACLs. For the change-impact query, we switched the decision from *deny* to *permit* on one non-superfluous rule. The overshadowing-rules computation asked only for overshadows with the opposite decision.

6.3 Enterprise Firewall Configuration

Our largest test case to date is an in-use enterprise iptables configuration. In order to stress-test our IOS compiler, we manually converted this configuration to IOS. The resulting configuration contains ACLs for 6 interfaces with a total of 1108 InboundACL rules (not counting routing subpolicies). The routing component of this firewall was fairly simple; we therefore focus our performance evaluation on InboundACL.

From a performance perspective, this paper has illustrated three fundamentally different types of queries: (1) computing over a single policy or network with just the default relations (which-packets and verification queries), (2) computing over a single policy or network while including additional relations (rule-responsibility and rule-relationship queries), and (3) computing over multiple, independent policies or networks (change-impact queries). The third type introduces more variables than the first two (to represent requests through multiple firewalls); it also introduces additional relations to capture the policies of multiple firewalls. The second type has the same number of variables, but more relations, than the first type. We therefore expect the best performance on the first type, even under TUPLING.

Table 3 reports run-time performance on each type of query over the enterprise firewall-configuration. Loading the policy's InboundACL component required 10694ms and consumed 51 MB of memory. Of that, 40 MB was JVM heap and 7 MB was JVM non-heap.

Section 2 described how we compute superfluous rules through scripting. For this example, these queries yielded surprising results: 900 of the 1108 rules in InboundACL were superfluous. Even more, 270 of the superfluous rules were (at least partially) overshadowed by a rule with a different decision. The sysadmins who provided the configuration found these figures shocking and subsequently expressed interest in Margrave.

7 Related Work

Studies of firewall-configuration errors point to the need for analysis tools. Oppenheimer, *et al.* [31] survey failures in three Internet services over a period of several months. For two of these services, operator error—predominately during configuration edits—was the leading cause of failure. Furthermore, conventional testing fails to detect many configuration problems. Wool [35] studies the prevalence of 12 common firewall-configuration errors. Larger rule-sets yield a much higher ratio of errors to rules than smaller ones; Wool concludes that complex rule sets are too difficult for a human administrator to manage unaided.

Mayer, Wool and Ziskind [26, 27] and Wool [34] describe a tool called Fang that has evolved into a commercial product called the AlgoSec Firewall Analyzer [3]. AlgoSec supports most of the same analyses as Margrave, covering NAT and routing, but it does not support first-order queries or integration with a programming language. AlgoSec captures packets that satisfy queries through sub-queries, which are a form of abstract scenarios.

Marmorstein and Kearns' [23, 24] ITVal tool uses Multi-way Decision Diagrams (MDDs) to execute SQL-like queries on firewall policies. ITVal supports NAT, routing, and chains of firewall policies. Later work [25] supports a useful query-free analysis: it generates an equivalence relation that relates two hosts if identical packets (modulo source address) from both are treated identically by the firewall. This can detect policy anomalies and help administrators understand their policies. Additional debugging aids in later work includes tracing decisions to rules and showing examples similar to scenarios. Margrave is richer in its support for change-impact and first-order queries.

Al-Shaer *et al.*'s ConfigChecker [1, 2] is a BDD-based tool that analyses networks of firewalls using CTL (temporal logic) queries. Rules responsible for decisions can be isolated manually through queries over sample packets. For performance reasons, the tool operates at the level of policies, rather than individual rules (other of the group's papers do consider rule-level reasoning); Margrave, in contrast, handles both levels.

Bhatt *et al.*'s Vantage tool [5, 9, 10] supports change-impact on rule-sets and other user-defined queries over combinations of ACLs and routing; it does not support NAT. Some of their evaluations [9] exploit change-impact to isolate configuration errors. This work also supports generating ACLs from specifications, which is not common in firewall-analysis tools.

Liu and Gouda [20, 21] introduce Firewall Decision Diagrams (FDDs) to answer SQL-like queries about firewall policies. FDDs are an efficient variant of BDDs for the firewall packet-filtering domain. Extensions of this work by Khakpour and Liu [17] present algorithms for many firewall analysis discussed in this paper, including user-defined queries, rule responsibility, and change-impact, generally in light of NAT and routing. A downloadable tool is under development.

Yuan, *et al.*'s Fireman tool [36] analyzes large networks of firewall ACLs using Binary Decision Diagrams (BDDs). Fireman supports a fixed set of analyses, including whitelist and blacklist violations and computing conflicting, redundant, or correlated rules between different ACLs. Fireman examines all paths between firewalls at once, but does not consider NAT or internal routing. Margrave's combination of user-defined queries and support for NAT and routing makes it much richer. Oliveira, *et al.* [30] extend Fireman with NAT and routing tables. Their tool, Prometheus, can also determine which ACL rules are responsible for a misconfiguration. It does not handle change-impact across firewalls, though it does determine when different paths through the same firewall render different decisions for the same packet. In certain cases, Prometheus suggests corrections to rule sets that guarantee desired behaviors. Margrave's query language is richer.

Verma and Prakash's FACE tool [33] aids both configuration of distributed firewalls and analyzing existing distributed firewalls expressed in iptables. It supports user-defined queries, as well as a form of change-impact over multiple firewalls. Its depth-first-search approach to propagating queries through a network resembles Mayer, Ziskind, and Wool's work. It does not handle routing or NAT. The tool is no longer available.

Gupta, LeFevre and Prakash [14] give a framework for the analysis of heterogeneous policies that is similar to ours. While both works provide a general policy-analysis language inspired by SQL, there are distinct differences. Their tool, SPAN, does not allow queries to directly reference rule applicability and the work does not discuss request-transformations such as NAT. However, SPAN provides tabular output that can potentially be more concise than Margrave's scenario-based output. SPAN is currently under development.

Lee, Wong, and Kim's NetPiler tool [18, 19] analyzes the flow graph of routing policies. It can be used to both simplify and detect potential errors in a network's routing configurations. The authors have primarily applied NetPiler to BGP configurations, which address the propagation of routes rather than the passage of packets. However, their methods could also be applied to firewall policies. Margrave does not currently support BGP, though its core engine is general enough to support them.

Jeffrey and Samak [16] present a formal model and algorithms for analyzing rule-reachability and cyclicity in iptables firewalls. This work does not address NAT or more general queries about firewall behavior.

Eronen and Zitting [11] perform policy analysis on Cisco router ACLs using a Prolog-based Constraint Logic Programming framework. Users are allowed to define their own custom predicates (as in Prolog), which enables analysis to incorporate expert knowledge. The Prolog queries are also first-order. This work is similar to ours in spirit, but is limited to ACLs and does not support NAT or routing information.

Youssef *et al.* [7] verify firewall configurations against security goals, checking both for configurations that violate goals and goals that configurations fail to cover. The work does not handle NAT or routing.

Margrave as described in this paper extends an earlier tool of the same name [12] developed by Tschantz, Meyerovich, Fisler and Krishnamurthi. The original Margrave targeted simple access-control policies, encoding them as propositional formulas that we analyzed using BDDs. Attempts to model enterprise access-control policies inspired the shift to first-order models embodied in the present tool. Not surprisingly, there is an extensive literature on logic-based tools for access-control policies; our other papers [12, 28] survey this literature.

8 Perspective and Future Work

Margrave is a general-purpose policy analyzer. Its most distinctive features lie in and arise from embracing scenario finding over first-order models. First-order languages provide the expressive power of quantifiers and relations for capturing both policies and queries. Expressive power generally induces performance cost. By automatically computing universe bounds for key queries, however, Margrave gets the best of both worlds: first-order logic's expressiveness with propositional logic's efficient analysis. Effectively, Margrave distinguishes between propositional *models* and propositional *implementations*. Most logic-based firewall-analysis tools conflate these choices.

First-order modeling lets Margrave uniformly capture information about policies at various levels of granularity. This paper has illustrated relations capturing policy decisions, individual rule behavior, and the effects of NAT and internal routing. The real power of our first-order modeling, however, lies in building new relations from existing ones. Each of the relations capturing behavior internal to a firewall (`passes-firewall`, `internal-routing`, and `int-dropped`) is defined within Margrave's query language and exported to the user through standard Margrave commands. While our firewall compilers provide these three automatically, users can add their own relations in a similar manner. Technically, Margrave allows users to define their own named views (in a database sense) on collections of policies. Thus, Margrave embraces policy-analysis in the semantic spirit of databases, rather than just the syntactic level of SQL-style queries.

Useful views build on fine-grained atomic information about policies. Margrave's unique decomposition of IOS configurations into subpolicies for nine distinct firewall functions provides that foundation. Our pre-defined firewall views would have been prohibitively hard to write without a clean way to refer to components of firewall functionality. Margrave's intermediate languages for policies and vocabularies, in turn, were instrumental in developing the subpolicies. Both languages use general relational terms, rather than domain-specific ones. Vocabularies allow authors to specify decisions beyond those typically associated with policies (such as *Permit* and *Deny*). Our IOS compiler defines separate decisions for the different types of flows out of internal routing, such as whether packets are forwarded internally or translated to another interface. The routing views are defined in terms of formulas capturing these decisions. The policy language defines the formulas through rules that yield each decision (our rule language is effectively stratified Datalog). Had we defined Margrave as a firewall-specific analyzer, rather than a general-purpose one, we likely would have hardwired domain-specific concepts that did not inherently support this decomposition.

User-defined decisions and views support extending Margrave from within. Integrating Margrave into a programming language supports external extension via scripting over the results of commands. Margrave produces scenarios as structured (XML) objects that can be traversed and used to build further queries. `SHOW REALIZED` produces lists of results over which programs (such as superfluous rule detection in Section 2) can iterate to generate additional queries. Extending our integration with iterators over scenarios would yield a more policy-specific scripting environment.

In separate projects, we have applied Margrave to other kinds of policies, including access-control, simple hypervisors, and product-line configuration. Margrave's general-purpose flexibility supports reasoning about *interactions* between firewalls and other types of policies (increasingly relevant in cloud deployments). This is another exciting avenue for future work.

Margrave's performance is reasonable, but slower than other firewall analyzers. This likely stems partly from additional variables introduced during the encoding into propositional logic. In particular, we expect Margrave will scale poorly to large networks of firewalls, as our formulas grow linearly with the number of firewalls. Our use of SAT-solving instead of BDDs may be another factor, though Jeffrey and Samak's comparisons between these for firewall analysis [16] are inconclusive. Exploring alternative backends—whether based on BDDs or other first-order logic solvers—is one area for future work. However, we believe the more immediate questions lie at the modeling level. For example:

- Firewall languages include stateful constructs such as *inspect*. Existing firewall analysis tools, including Margrave, largely ignore state (we are limited to reflexive ACLs). How do we effectively model and reason about state without sacrificing performance?
- Modeling IP addresses efficiently is challenging. Many tools use one propositional variable per bit; Margrave instead uses one per IP address. This makes it harder to model arithmetic relationships on IP addresses (i.e., subranges), though it provides finer-grained control over which IP addresses are considered during analysis. Where is the sweet-spot in IP-address handling?

Margrave is in active development. We are extending our firewall compilers to support VPN and BGP. We would like to automatically generate queries for many common problems (such as overshadowing rule detection and change-impact). Section 2 also hinted at a problem with reusing queries in the face of policy edits: the compiler names rules by line-numbers, so edits may invalidate existing queries. We need to provide better support for policy-management including regression testing.

Acknowledgments:

Support for this research came from several National Science Foundation grants. Cisco supported an early phase of this project. We thank John Basik, Jeff Coady, Mark Dieterich, Jason Montville and Richard Silverman for sysadmins' perspectives on this project. Craig Wills explained how to report performance data. Our LISA shepherd, Matt Disney, provided useful suggestions. In compiling our related work, we contacted many authors with questions about their projects. We thank them for their prompt and cheerful responses and hope we have represented their work accurately; any errors are our own.

References

[1] Ehab S. Al-Shaer and Hazem H. Hamed. Firewall Policy Advisor for Anomaly Discovery and Rule Editing. In *Integrated Network Management*, pages 17–30, 2003.

[2] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *IEEE Conference on Computer Communications*, 2004.

[3] The AlgoSec Firewall Analzyer. `www.algosec.com`.

[4] azsquall. "ACL and NAT conflict each other. router stop working". `www.networking-forum.com/viewtopic.php?f=33&t=7635`, August 2008. Access Date: July 20, 2010.

[5] Sruthi Bandhakavi, Sandeep Bhatt, Cat Okita, and Prasad Rao. End-to-end network access analysis. Technical Report HPL-2008-28R1, HP Laboratories, November 2008.

[6] Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila Takayama, and Madhu Prabaker. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. In *ACM Conference on Computer Supported Cooperative Work*, pages 388–395, 2004.

[7] Nihel Ben Youssef, Adel Bouhoula, and Florent Jacquemard. Automatic Verification of Conformance of Firewall Configurations to Security Policies. In *IEEE Symposium on Computers and Communications*, pages 526 – 531, July 2009.

[8] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 2010. To appear.

[9] Sandeep Bhatt, Cat Okita, and Prasad Rao. Fast, Cheap, and in Control: A Step Towards Pain-Free Security! In *Large Installation System Administration Conference*, pages 75–90, 2008.

[10] Sandeep Bhatt and Prasad Rao. Enhancements to the Vantage Firewall Analyzer. Technical Report HPL-2007-154R1, HP Laboratories, June 2008.

[11] Pasi Eronen and Jukka Zitting. An expert system for analyzing firewall rules. In *Proceedings of the Nordic Workshop on Secure IT Systems*, pages 100–107, 2001.

[12] Kathi Fisler, Shriram Krishnamurthi, Leo Meyerovich, and Michael Tschantz. Verification and change impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, 2005.

[13] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `racket-lang.org/tr1/`.

[14] Swati Gutpa, Kristen LeFevre, and Atul Prakash. SPAN: A unified framework and toolkit for querying heterogeneous access policies. In *USENIX Workshop on Hot Topics in Security*, 2009.

[15] Daniel Jackson. *Software Abstractions*. MIT Press, 2006.

[16] Alan Jeffrey and Taghrid Samak. Model Checking Firewall Policy Configurations. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, 2009.

[17] Amir R. Khakpour and Alex X. Liu. Quantifying and querying network reachability. In *Proceedings of the International Conference on Distributed Computing Systems*, June 2010.

[18] Sihyung Lee, Tina Wong, and Hyong S. Kim. Improving Dependability of Network Configuration through Policy Classification. In *IEEE/IFIP Conference on Dependable Systems and Networks*, 2008.

[19] Sihyung Lee, Tina Wong, and Hyong S. Kim. NetPiler: Detection of Ineffective Router Configurations. *IEEE Journal on Selected Areas in Communications*, 27(3):291–301, 2009.

[20] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems*, 19(8), August 2008.

[21] Alex X. Liu and Mohamed G. Gouda. Firewall policy queries. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):766–777, June 2009.

[22] The Margrave Policy Analzyer. `www.margrave-tool.org/v3/`.

[23] Robert Marmorstein and Phil Kearns. A Tool for Automated iptables Firewall Analysis. In *USENIX Annual Technical Conference*, 2005.

[24] Robert Marmorstein and Phil Kearns. An Open Source Solution for Testing NAT'd and Nested iptables Firewalls. In *Large Installation System Administration Conference*, 2005.

[25] Robert Marmorstein and Phil Kearns. Firewall Analysis with Policy-Based Host Classification. In *Large Installation System Administration Conference*, 2006.

[26] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A Firewall Analysis Engine. In *IEEE Symposium on Security and Privacy*, pages 177–187, 2000.

[27] Alain Mayer, Avishai Wool, and Elisha Ziskind. Offline firewall analysis. *International Journal of Information Security*, 2005.

[28] Timothy Nelson. Margrave: An Improved Analyzer for Access-Control and Configuration Policies. Master's thesis, Worcester Polytechnic Institute, April 2010.

[29] oelolemy. "problem with policy based routing-urgent please !". `www.experts-exchange.com/Networking/Network_Management/Q_24113014.html`, February 2009. Access Date: July 20, 2010.

[30] Ricardo M. Oliveira, Sihyung Lee, and Hyong S. Kim. Automatic detection of firewall misconfigurations using firewall and network routing policies. In *DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance*, 2009.

[31] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[32] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[33] Pavan Verma and Atul Prakash. FACE: A Firewall Analysis and Configuration Engine. In *Proceedings of the Symposium on Applications and the Internet*, 2005.

[34] Avishai Wool. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the USENIX Security Symposium*, 2001.

[35] Avishai Wool. A Quantitative Study of Firewall Configuration Errors. *Computer*, 37(6):62–67, 2004.

[36] L. Yuan, J. Mai, Z. Su, H. Chen, C-N. Chuah, and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *IEEE Symposium on Security and Privacy*, 2006.