FIREMAN: A Toolkit for FIREwall Modeling and ANalysis

Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah and Prasant Mohapatra
{lyuan@ece, jnmai@ece, su@cs, hchen@cs, chuah@ece, prasant@cs}.ucdavis.edu
University of California, Davis

***Abstract*— Security concerns are becoming increasingly critical in networked systems. Firewalls provide important defense mechanisms for network security. However, misconfigurations in firewalls are very common and significantly weaken the desired security. This paper introduces FIREMAN, a static analysis toolkit for firewall modeling and analysis. By treating firewall configurations as specialized programs, FIREMAN applies static analysis techniques to check misconfigurations, such as policy violations, inconsistencies, and inefficiencies, in individual firewalls as well as among distributed firewalls. FIREMAN performs symbolic model checking of the firewall configurations for all possible IP packets and along all possible data paths. It is both sound and complete because of the finite state nature of firewall configurations. FIREMAN is implemented by modeling firewall rules using binary decision diagrams (BDDs), which have been used successfully in hardware verification and model checking. We have experimented with FIREMAN and used it to uncover several real misconfigurations in enterprise networks, some of which have been subsequently confirmed by these networks' administrators.**

I. Introduction

Firewall is a critical, widely deployed mechanism for improving the security of enterprise networks. However, configuring a firewall is daunting and error-prone even for an experienced administrator. As a result, misconfigurations in firewalls are common and serious. In examining 37 firewalls in production enterprise networks in 2004, Wool found that all the firewalls were misconfigured and vulnerable, and that all but one firewall were misconfigured at multiple places [1]. As another evidence, Firewall Wizards Security Mailing List has discussed many real firewall misconfigurations [2]. The wide and prolonged spread of worms, such as Blaster and Sapphire, demonstrated that many firewalls were misconfigured, because "well-configured firewalls could have easily blocked them" [1].

The following scripts illustrates how easy firewall misconfigurations can happen:

```
permit tcp 192.168.0.0/16 any
deny tcp 192.168.1.0/24 any 3127
```

The second rule is configured to deny all the outbound traffic to a known backdoor TCP port for the *MyDoom.A* worm, and is correct by itself. However, since a firewall examines each rule sequentially and accepts (or rejects) a packet immediately when the packet is matched to a rule, a preceding rule may *shadow* subsequent rules matching some common packets. The first rule, which accepts all the outbound traffic from the local network 192.168.0.0/16, shadows the second rule and leaves the hole wide open.

Correctly configuring firewall rules has never been an easy task. In 1992, Chapman [3] discussed many problems that make securely configuring packet filtering a daunting task. Some of them, e.g., omission of port numbers in filtering rules, have been addressed by firewall vendors. However, many others are yet to be addressed successfully. Since firewall rules are written in platform-specific, low-level languages, it is difficult to analyze whether these rules have implemented a network's high-level security policies accurately. Particularly, it is difficult to analyze the interactions among a large number of rules. Moreover, when large enterprises deploy firewalls on multiple network components, due to dynamic routing, a packet from the same source to the same destination may be examined by a different set of firewalls at different times. It is even more difficult to reason whether all these sets of firewalls satisfy the end-to-end security policies of the enterprise.

We propose to use static analysis to discover firewall misconfigurations. Static analysis has been applied successfully to discover security and reliability bugs in large programs [4, 5], where it examines the control-flow and/or the data-flow to determine if a program satisfies user-specified properties without running the program. A firewall configuration is a specialized program, so it is natural to apply static analysis to check firewall rules. Compared to testing, static analysis has three major advantages: (1) it can proactively discover and remove serious vulnerabilities before firewalls are deployed; (2) it can discover all the instances of many known types of misconfigurations, because it can exhaustively examine every path in the firewall efficiently; (3) when multiple firewalls are deployed in a complex network topology and are subject to dynamic routing configurations, static analysis can discover vulnerabilities resulting from the interaction among these firewalls without the need to configure these routers.

Testing has been proposed to discover firewall misconfigurations [6–9], where a tool generates packets and examines whether a firewall processes these packets as intended. However, due to the enormous address space of packets, one cannot test all possible packets practically. Al-Shaer and Hamed describe common pairwise inconsistencies in firewall rules and propose an algorithm to detect these inconsistencise [10, 11]. Our work is inspired by them, but our tool can detect a much wider class of misconfigurations, such as inconsistencies and inefficiencies among multiple rules and security policy violations, and misconfigurations due to the interaction among multiple firewalls. To the best of our knowledge, our work is the first to apply rigorous static analysis techniques to real firewalls and to have found real misconfigurations.

We have implemented our approach in the tool FIREMAN[1]. FIREMAN discovers two classes of misconfigurations: (1) *violations of user-specified security policies*—For example, allowing incoming packets to reach the TCP port 80 on an internal host violates the security policies of most networks; (2) *inconsistencies and inefficiency among firewall rules*, which indicate errors or warnings regardless of the security policies—For example, a rule intended to reject a packet is shadowed by a preceding rule that accepts the packet. FIREMAN can discover problems not only in individual firewalls but also in a distributed set of firewalls that collectively violate a security policy.

We summarize our major contributions as follows:

1) We give a comprehensive classification of firewall misconfigurations for both single firewalls and distributed firewalls (Section III);
2) We present a static analysis algorithm to examine firewall rules for policy violations and inconsistencies at different levels: intra-firewall, inter-firewall, and cross-path (Section IV);
3) We provide an implementation of our algorithm in the tool FIREMAN based on binary decision diagrams (BDDs). Using FIREMAN, we have discovered previously unknown misconfigurations in production firewalls. (Section V)

The rest of this paper is organized as follows. Section II describes the operational model of firewalls, which lays the foundation for static analysis and error detection. Section III classifies misconfigurations into policy violations, inconsistencies, and inefficiencies. Section IV presents our static analysis algorithm for checking firewall misconfigurations. Section V describes our implementation and evaluation of FIREMAN, and the previously unknown misconfigurations that FIREMAN discovered in production firewalls. Section VI reviews related work and Section VII concludes this paper.

II. Modeling Firewalls

A. Model of Individual Firewall

Firewalls from different vendors may vary significantly in terms of configuration languages, rule organizations and interaction between lists or chains. However, a firewall generally consists of a few interfaces and can be configured with several access control lists (ACLs). Both the ingress and egress of an interface can be associated with an ACL. If an ACL is associated to the ingress, filtering is performed when packets arrive at the interface. Similarly, if an ACL is associated to the egress, filtering will be performed before packets leave the interface.

Each ACL consists of a list of rules. Individual rules can be interpreted in the form $\langle P,\ action\rangle$, where P is a predicate describing what packets are matched by this rule and $action$ describing the corresponding action performed on the matched packets. Packets not matched by the current rule will be forwarded to the next rule until a match is found or the end of the ACL is reached. At the end of an ACL, the default action will be applied. This is similar to an "if-elif-else" construct in most programming languages. Implicit rules vary on different firewall products. On Cisco PIX firewall and routers, the implicit rule at the end of an ACL denies everything. On Linux Netfilter, the implicit rule is defined by the *policy* of the chain.

Traditional *stateless* firewalls treat each packet in isolation and check every packet against the ACL, which is computation intensive and often the performance bottleneck. Modern *stateful* firewalls can monitor TCP 3-way handshake and build an entry in the state table. If the firewall matches a packet to an *ESTABLISHED* flow, it can permit it without checking the ACL, thus significantly reduce the computation overhead. However, the ACLs still determine whether a state can be established in the first place. Therefore, the correct configuration of ACLs is important even for stateful firewalls.

Format	Action for Matched Packets
$\langle P,$ accept$\rangle$	permit the packet
$\langle P,$ drop$\rangle$	drop the packet
$\langle P,$ chain Y$\rangle$	goto user chain "Y"
$\langle P,$ return$\rangle$	resume calling chain

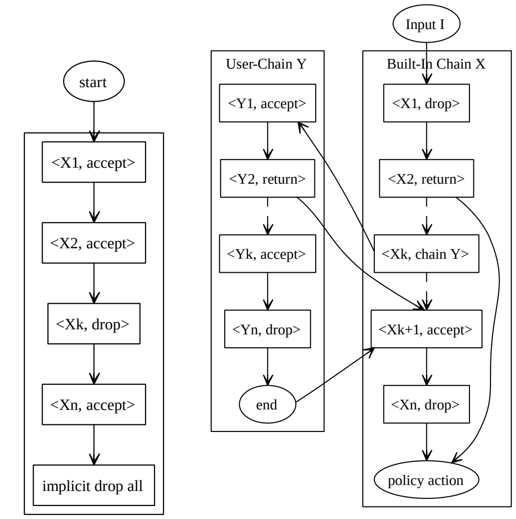
TABLE I: Firewall rule formats.

Depending on the available "actions," we classify firewalls into two typical models: (1) the *simple list model*, which is represented by Cisco PIX firewall and router ACLs and (2) the *complex chain model*, which is represented by Linux Netfilter. Firewalls using the simple list model allow only *"accept"* and *"drop"* actions. The complex chain model, in addition to *"accept"* and *"drop"*, also supports calling upon another user-defined chain or *"return."* We use *rule graphs* to model the control-flow of ACLs. As can be seen in Section IV-A, the rule graph of ACLs using the simple list model is just itself. The rule graphs for ACLs using the complex chain model are similar to control-flow graphs in programming languages.

1) Simple List Model: Figure 1(a) depicts the simple list model of an ACL. Since only *"accept"* or *"drop"* actions (first two forms shown in Table I) are allowed, any packet will traverse the list in order until a decision is made on each packet. An implicit rule at the end of the list will match the rest of the packets and apply the default action to these remaining packets. We make the implicit rule explicit by appending them to the end of the list.

2) Complex Chain Model: The Linux-based firewall implementation, Netfilter [12], has a more complicated grammar for the rules, which may take any of the four forms shown in Table I. In addition to *"accept"* or *"drop,"* the action field can call upon another user-defined chain for further processing. The user-defined chain can also choose to *"return"* to the next rule of the calling chain. One can view the action of calling a user-defined chain as a function call and the corresponding *"return"* as a function return. This feature, similar to the use

[1] FIREMAN stands for FIREwall Modeling and ANalysis.



(a) Simple List Model

(b) Complex Chain Model

Fig. 1: Model of individual firewalls

of functions in programming languages, facilitates reusable configurations and improves firewall efficiency.

Figure 1(b) depicts a typical firewall using the chain-based model. The built-in chain "X", which is the starting point, can call upon a user-defined chain "Y" for further processing. Chain "Y" can either explicitly return to the calling chain "X" when certain predicate is satisfied or the end of chain "Y" is reached. Other chains may call chain "Y" as well.

B. Network of Firewalls

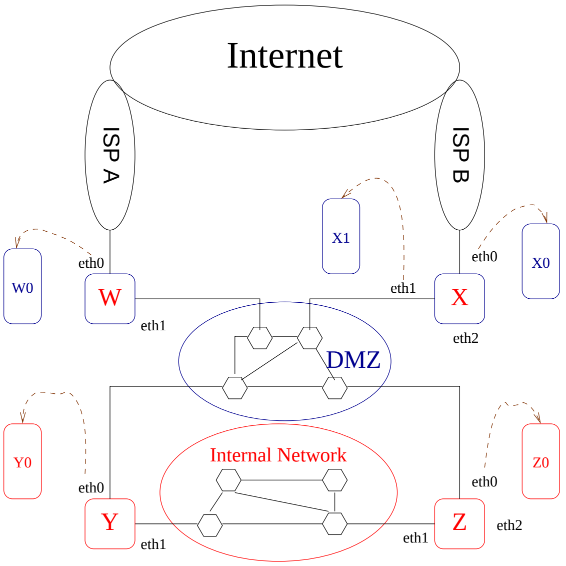


Fig. 2: Network of firewalls

In a typical network environment, multiple firewalls are often deployed across the network in a distributed fashion. Although firewalls are configured independently, a network depends on the correct configuration of *all* related firewalls to achieve the desired *end-to-end security behavior*. By "end-to-end security behavior," we refer to the decision on whether a packet should be allowed to reach a protected network. It can be from one side of a Virtual Private Network (VPN) to another side of the VPN. It can also be from the wild untrusted Internet to the trusted secured intranet.

Take Figure 2 for example, an enterprise network is connected to the Internet through two different ISPs and firewalls W and X are deployed to guard the Demilitarized Zone (DMZ). Services such as Web and email that must allow public access are more vulnerable (hence less trustworthy) and normally put in the DMZ. Further inside, the internal network is guarded by additional firewalls Y and Z. In general, firewalls Y and Z will have a tighter security policy. Important applications and sensitive data are often running inside the internal trusted networks, and only limited accesses are allowed.

Since there exists multiple paths from the Internet to the internal network. The filtering action taken depends on the path a packet actually traverses. A packet does not actually choose its data path. But the dynamics of the underlying routing plane may assign different paths for the same set of packets at different time. Ideally, firewalls should perform consistently regardless of the underlying routing decisions. To guarantee reachability of desired packets, the administrator must ensure that none of the firewalls on the path drops them. On the other hand, the administrator must ensure that no possible paths permits any malicious packets from accessing the protected network.

III. Misconfigurations

A firewall does not provide security in its own right. The way the firewall is configured determines the overall security effectiveness. In this section, we discuss firewall misconfigurations and classify them. Section III-A discusses policy violations, which can be checked against well-defined policies. Not all misconfigurations can be caught by policy definitions. In Section III-B, we discuss inconsistent configurations and how to use these to infer misconfigurations. Section III-C discusses some inefficient configurations that are not errors, but may still adversely affect the firewall performance.

The sample scripts used in this paper are written in the format of: *<action, protocol, src ip, src port, dst ip, dst port>* where *src ip* and *src port* denote respectively source IP address and source port number respectively. Similarly, *dst ip* and *dst port* refer to destination IP address and port number respectively. Both source and destination ports are optional. The IP addresses used in this paper are written in private IP address blocks on purpose only to avoid exposing address information.

A. Policy Violation

Administrators often have a high-level policy describing what should be prohibited (*blacklists*) or ensured (*whitelist*) access to the network. It is crucial that firewall configurations exactly reflect the security policy. Any nonconforming configurations may result in undesired blocking, unauthorized access, or even the potential for an unauthorized person to alter security configurations. Therefore, a firewall must be verified against any policy violations.

Although policy definition is subjective to individual institutions, the network security community has some well-understood guidelines on firewall configurations. From an external auditor's point of view, Wool [1] studied 37 configurations of Check Point's FireWall-1 product and noticed 12 common firewall configuration errors. Among them, allowing "any" destination on outbound rules, "any" service on inbound rules happens to 90% of the configurations. Allowing NetBIOS and Portmapper/Remote Procedure Call service is also a common class of errors that exposes the network to very insecure services. A major number of firewalls are not configured correctly to provide proper protection. Approximately 46% of the firewalls are not configured with a *stealth rule* to hide itself, and above 70% of them are open to insecure management protocols or external management machines. All these "errors" affect the security of the entire network and must be carefully checked.

Another source of input for the blacklist is the *bogon* list [13], which describes IP blocks or port numbers not currently allocated by IANA and RIRs plus those reserved for private or special use. Attackers often use these IP blocks or ports for DoS attacks, spamming or hacking activities. Most firewall administrators would want to ensure that traffic from/to these IP blocks or port numbers are neither explicitly nor implicitly permitted to reach their networks.

B. Inconsistency

Firewall configurations represent the administrator's intention, which should be consistent. Therefore, inconsistencies are often good indicator of misconfigurations. Unlike policy violations, for which there are well-defined references (*blacklists* and *whitelists*) to compare against, checking for inconsistencies is solely based on the configuration files and does not need external input. Inconsistencies happen at three levels: *intra-firewall*, *inter-firewall*, and *cross-path*.

1.	deny tcp 10.1.1.0/25 any
2.	permit udp any 192.168.1.0/24
3.	drop tcp 10.1.1.128/25 any
4.	drop udp 172.16.1.0/24 192.168.1.0/24
5.	permit tcp 10.1.1.0/24 any
6.	drop udp 10.1.1.0/24 192.168.0.0/16
7.	permit udp 172.16.1.0/24 any

TABLE II: Sample script 1

1) Intra-firewall Inconsistency:

1) *Shadowing*: refers to the case where *all* the packets one rule intends to drop (accept) have been accepted (dropped) by preceding rules. This often reveals a misconfiguration and is considered an "error." A rule can be shadowed by one preceding rule that matches a superset of the packets. In Table II, rule 4 is shadowed by rule 2 because every UDP packet from 172.16.1.0/24 to 192.168.1.0/24 is permitted by rule 2, which matches any UDP packets destined to 192.168.1.0/24. Alternatively, a rule may also be shadowed by a set of rules *collectively*. For example, rule 5 is shadowed by the *combination* of rules 1 and 3. Rule 1 drops TCP packets from 10.1.1.0/25, and rule 3 drops TCP packets from 10.1.1.128/25. Collectively, they drop all TCP packets from 10.1.1.0/24, which are what rule 5 intends to accept.
2) *Generalization*: refers to the case where a subset of the packets matched to this rule has been excluded by preceding rules. It is the opposite of shadowing and happens when a preceding rule matches a subset of this rule but takes a different action. In Table II, rule 7 is a generalization of rule 4 because UDP packets from 172.16.1.0/24 *and* to 192.168.1.0/24 form a subset of UDP packets from 172.16.1.0/24 (rule 7), yet the decision for the former is different from the later.
3) *Correlation*: refers to the case where the current rule *intersects* with preceding rules but specifies a different action. The predicates[2] of these correlated rules intersect, but are not related by the superset or subset relations. The decision for packets in the intersection will rely on the order of the rules. Rules 2 and 6 are correlated with each other. The intersection of them is "udp 10.1.1.0/24 192.168.1.0/24," and the preceding rule determines the fate of these packets.

Generalization or correlation may not be an error but a commonly used technique to exclude part of a larger set from certain action. Proper use of these techniques could result in fewer number of rules. However, these techniques should be used very consciously. ACLs with generalizations or correlations can be ambiguous and difficult to maintain. If a preceding rule is deleted, the action for some packets in the intersection will change. On a large and evolving list of rules, it may be difficult to realize all the related generalizations and correlations manually. Without a priori knowledge about the administrators intention, we cannot concretely tell whether this is a misconfiguration. Therefore, we classify them as "warnings."

$X0$	1.	drop tcp any 10.1.0.0/16
	2.	permit tcp any any
$X1$	1	permit any any any
$Z0$	1.	drop tcp any 10.0.0.0/8
	2.	permit tcp any any
	3.	drop udp any 192.168.0.0/16
$W0$	1.	drop tcp any 10.0.0.0/8
	2.	permit tcp any any
	3.	drop udp any 192.168.0.0/16
$Y0$	1.	permit tcp any any
	2.	permit udp 172.16.0.0/16 192.168.0.0/16

TABLE III: Sample script 2

[2] In this context, we view a predicate as both a set of matching packets and a logical predicate specifying this particular set. We use these two interpretations interchangeably.

2) Inter-Firewall Inconsistency: Inconsistencies among different firewalls might not be errors. When a few firewalls are chained together, a packet has to survive the filtering action of *all* the firewalls on its path to reach its destination. Therefore, a downstream firewall can often rely on upstream firewall to achieve policy conformance and can be configured much loosely. On the other hand, a downstream firewall at the inner perimeter often needs a tighter security policy. Consider the topology in Figure 2 with the configuration scripts in Table III, packets destined to $10.0.0.0/8$ but not to $10.1.0.0/16$, e.g., $10.2.0.0/16$, will be permitted by $X0$ (rule 2) and therefore have access to the DMZ. However, they are dropped by $Z0$ (rule 1) to protect the internal network.

Without input from the administrator, the only inter-firewall inconsistency we, as tool writer, can classify as an "error" is *shadowed accept rules*. By explicitly allowing certain predicates, we infer that the administrator *intends* to receive these packets. For example, in Table III, rule 2 of $Y0$ permits UDP packets from 172.16.0.0/16 to 192.168.0.0/16, yet these packets are filtered by $W0$ (rule 3) at the upstream. To the downstream users, this may manifest as a connectivity problem.

3) Cross-Path Inconsistency: As discussed in Section II-B, it is likely to have multiple data paths from the Internet to the same protected network. Cross-path inconsistency refers to the case where some packets dropped on one path are accepted through another path. It depends on the underlying routing table whether these anomalies are exploitable. However, an attacker needs to succeed only once, and attacks that affect routing table do exists. Cross-path inconsistencies may also manifest as intermittently disruptive services. Packets originally reachable to the network may switch over to another path that drops such packets because of routing changes.

Consider again the topology in Figure 2 with the configuration scripts in Table III, paths $X \rightarrow dmz \rightarrow Z$ and $W \rightarrow dmz \rightarrow Y$ both drop "udp any $192.168.0.0/16$," which probably should not be allowed to reach the internal network. Yet one may also notice that these packets can leak into the internal network through the path $X \rightarrow dmz \rightarrow Y$. This path may not always be available since the actual path is determined by the underlying routing protocol. However, routing is designed to be adaptive to link failures and heavy load. In addition, it is relatively easy to inject false routing messages [14]. A safe firewall configuration should not rely on that, and should assume that all paths are topologically possible.

Checking cross-path inconsistencies based on active testing is very difficult. It may disrupt the production network since routing tables must be altered to test different scenarios. Manually auditing such anomalies is also difficult. Even for a network of moderate size, the number of possible paths between two nodes can be large.

C. Inefficiency

A firewall needs to inspect a huge number of packets. Therefore, it is difficult not to be concerned with firewall efficiency. Numerous work have been dedicated to improve the speed of firewall through better hardware and software designs and implementations. To administrators, the most practical way to improve firewall efficiency is through better configuration of the firewall.

An efficient firewall configuration should require the minimum number of rules, use the least amount of memory, and incur the least amount of computational load while achieving the same filtering goals. Although inefficiency does not directly expose a vulnerability, a faster and more efficient firewall will encourage firewall deployment and therefore makes the network safer. In addition, the efficiency of a firewall can determine a network's responsiveness to Denial-of-Service (DoS) attacks.

1.	permit tcp 192.168.1.1/32 172.16.1.1/32
2.	permit tcp 10.0.0.0/8 any
3.	permit tcp 10.2.1.0/24 any
4.	drop tcp any any
5.	drop udp 10.1.1.0/26 any
6.	drop udp 10.1.1.64/26 any
7.	drop udp 10.1.1.128/26 any
8.	drop udp 10.1.1.192/26 any
9.	drop udp any

TABLE IV: Sample script 3

1) Redundancy: Redundancy refers to the case where if a rule is removed, the firewall does not change its action on *any* packets. Reducing redundancy can reduce the total number of rules, and consequently reduce memory consumption and packet classification time [15].

A rule can be considered redundant if the preceding rules have matched a superset of this rule and specifies the same action. For example, in Table IV, rule 3 is redundant since rule 2 has already specified the same action for all packets that match rule 3. A rule can also be made redundant by the subsequent rules. Rules 5, 6, 7 and 8 are all redundant because if we remove them, these packets are still going to be dropped by rule 9. In fact, for firewalls with a "drop all" policy implicitly appended to the end of an ACL, we do not need rules $4 - 9$ altogether.

Redundant accept or drop rules are "errors" *within the same firewall*. This is, however, not true in distributed firewalls. A packet *must* be accepted on *all* the firewalls on its path to reach the destination. Redundant accept rules on different firewalls, such as the second rules of $X0$ and $Z0$ in Table III, are both necessary. Redundant drop rules *on different firewalls* are unnecessary, but are often considered good practice to enhance security. This redundancy provides an additional line of defense if the outer-perimeter is compromised.

2) Verbosity: Verbosity refers to the case where a set of rules may be summarized into a smaller number of rules. For example, rules 5, 6, 7, and 8 in Table IV can be summarized into a single rule "drop udp $10.1.1.0/24$ any." Verbosity often happens in practice when administrators build up the filter list over a period of time. Such cases are frequently observed in the real configurations we have collected.

IV. Problem Formulation and Checking Algorithm

In this section, we present the framework of FIREMAN that consists of two phases. First, FIREMAN parse a firewall configuration into an compact representation based on the operational semantics of a firewall. An ACL is translated into a rule graph and distributed firewalls, with additional information about network topology, are translated into an ACL-tree. We then check for anomalous configurations based on the rule graph and ACL-tree.

A. Parsing and Flow Graph Analysis

The purpose of this phase is twofold. First, a production network may consist of firewall products from different vendors, each with their own configuration languages and operation models. Our parser translates firewall configuration files originally written in their own languages into a uniform internal representation. Second, and more importantly, based on the configuration, network topology and routing information, we perform control-flow analysis to find all possible rule paths packets may go through. Each path represents a list of filtering operations packets may receive.

1) Rule Graph of Individual ACLs: For firewalls using the simple list model, there is no possibility of branching and the rule graph is the same list. For firewalls using the complex chain model, branching can be caused by calling “chain Y” and “return” from it. To handle such branching, we introduce $\langle P,\ \text{pass}\rangle$ to indicate that only packets matching this predicate will remain in this path. For a $\langle P,\ \text{chain Y}\rangle$ rule, we insert $\langle P,\ \text{pass}\rangle$ before going to “chain Y”. We also insert $\langle \neg P,\ \text{pass}\rangle$ for the path that does not jump to “chain Y”. Figure 3 visualizes all the four possible rule paths the ACL of Figure 1(b) could have.

Recursive function calls should be avoided since this could create loops. Loops can be easily prevented by ensuring that no rules appear twice on a rule path. Earlier versions of Netfilter drop a packet when it is found to be in a loop. But this problem is probably best to be avoided at configuration time. After eliminating loops, the rule graph can be constructed by linearization.

We denote the input to an ACL as I, which is the collection of packets that can possibly arrive at this access list. For an ACL using the complex chain model, the rule graph may give n rule paths from the input to the output. For each of the n rule paths, we traverse the path to collect information.

For the jth rule $\langle P_j, action_j\rangle$ in this rule path, we define the current state as $\langle A_j, D_j, F_j\rangle$, where A_j and D_j denote the network traffic accepted and denied before the jth rule, respectively; F_j denotes the set of packets that have been diverted to other data paths. We use R_j to denote the collection of the remaining traffic that can possibly arrive at the jth rule. R_j can always be found using the input I and the current state information, as shown in Equation 1.

$$R_j = I \cap \neg(A_j \cup D_j \cup F_j) \tag{1}$$

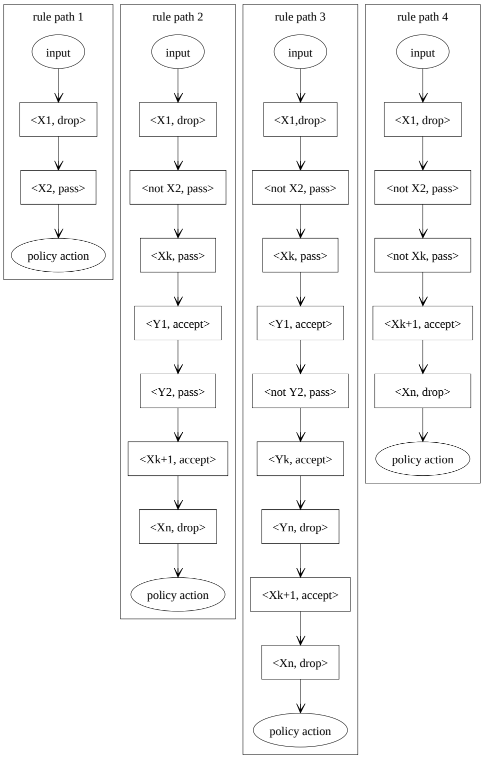


Fig. 3: Rule graph of the ACL in Figure 1(b)

For the first rule of an ACL, we have the initial value of $A_1 = D_1 = F_1 = \emptyset$ and $R_1 = I$. After reading each rule, we update the state according to the state transformation defined in Equation 2 until the end of each rule path. A state transform “$S_i, r \vdash S_{i+1}$” means if we read in rule r at state S_i, we will result in state S_{i+1}. Note that R is automatically updated when $< A, D, F >$ changes.

$$\begin{aligned}
\langle A, D, F\rangle,\quad \langle P,\ \text{accept}\rangle &\ \vdash \langle A \cup (R \cap P),\ D,\ F\rangle \\
\langle A, D, F\rangle,\quad \langle P,\ \text{drop}\rangle &\ \vdash \langle A,\ D \cup (R \cap P),\ F\rangle \\
\langle A, D, F\rangle,\quad \langle P,\ \text{pass}\rangle &\ \vdash \langle A,\ D,\ F \cup (R \cap \neg P)\rangle
\end{aligned} \tag{2}$$

At the end of rule path $path_i$, we can determine the packets accepted and dropped through this path to be A_{path_i} and D_{path_i} , respectively. Since any packet can take only one path, packets accepted by this ACL is the union of those accepted on all paths, as shown in Equation 3. In addition, since the default action of an ACL matches all packets, all packets will be either accepted or dropped.

$$\begin{aligned}
A_{ACL} &= \bigcup_{i \in path} A_{path_i} \qquad (3)\\
D_{ACL} &= \bigcup_{i \in path} D_{path_i} \\
A_{ACL} \cup D_{ACL} &= I_{ACL} \\
R_{ACL} &= \emptyset
\end{aligned}$$

2) ACL Graph of Distributed Firewalls: In the network of distributed firewalls, a packet will go through a series of ACLs to reach the destination. In this case, it needs to survive the filtering of *all* the ACLs on the path. On the other hand, a well-engineered network often has multiple paths and uses dynamic routing to improve performance and reliability. As a result, a packet could traverse different ACL paths at different times.

Given the topology as a directed graph, one can determine all the possible paths from one node to another. Since ACLs are associated with individual interface *and* a direction, one can build a tree of ACLs. Based on the information of network connectivity, one can compute the ACL tree rooted at a destination using either DFS or BFS algorithms. This tree graph reveals all the ACL paths packets may traverse to reach the destination. Note that we choose to be blind about the underlying routing and assume all the paths that are topologically feasible could be taken. This is because routing is designed to be dynamic and adaptive to link failures and loads. And firewall configuration should behave correctly and consistently regardless of the underlying routing dynamics.

For a large and well-connected graph, the number of paths can be large. For the portions of network that are not involved in packet filtering, and therefore do not interfere with the firewall configurations, we use abstract virtual nodes as representations. This approach can greatly reduce the complexity of the graph but can still keep the relevant information. For the network illustrated in Figure 2, we use three abstract virtual nodes "outside", "DMZ" and "inside" to indicate the untrusted Internet, DMZ and trusted internal network, respectively. Data paths between these three virtual nodes are often the primary concern of firewall administrators[3].

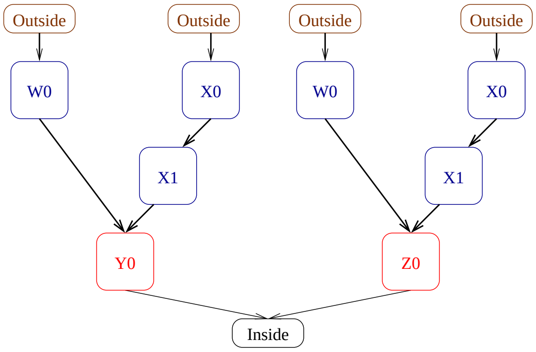


Fig. 4: ACL tree

Figure 4 shows the ACL tree built for Figure 2. In such ACL tree graph, ACLs are either in series, parallel, or a combination of them. For a set of n ACLs in series, packets must survive the filtering decision of *all* of them. Therefore, the accepted set of packets is the intersection of these ACLs accepted independently.

$$\begin{aligned} A &= \bigcap_{acl \in n} A_{acl} \\ D &= \bigcup_{acl \in n} D_{acl} \end{aligned} \tag{4}$$

For n ACLs in parallel, packets can come through either of the ACLs. Therefore, the accepted set of packets is the union of all ACLs accepted independently.

$$\begin{aligned} A &= \bigcup_{acl \in n} A_{acl} \\ D &= \bigcap_{acl \in n} D_{acl} \end{aligned} \tag{5}$$

Consequently, in the tree structure, the input to the parent node is the union of accepted packets of the children nodes.

$$I = \bigcup_{acl \in children} A_{acl} \tag{6}$$

Based on Equations 4 and 5, we can analyze firewall rules in the context of networks and distributed firewalls. Consider Figure 4 as an example, we assume the input from "outside" to be Ω, the entire set of possible packets, which is a very conservative approach. However, we believe this is justified for security reasons since "outside" is beyond the control of local administration.

One can determine that I_{Y0}, the input for $Y0$, is $I_{Y0} = A_{W0} \cup (A_{X0} \cap A_{X1})$. The entire set of packets that can reach the internal network from the Internet is

$$\begin{aligned} A =& A_{W0} \cup (A_{X0} \cap A_{X1}) \cap A_{Y0} \\ & \cup A_{W0} \cup (A_{X0} \cap A_{X1}) \cap A_{Z0} \\ =& A_{W0} \cup (A_{X0} \cap A_{X1}) \cap A_{Y0} \cup A_{Z0} \end{aligned} \tag{7}$$

B. Checking for Anomalies

Based on the rule graph, we perform local checks for individual firewall. Distributed firewall checks are based on both the ACL-tree and rule graph. We describes the algorithms below in detail.

1) ***Local Check for Individual Firewalls****:* FIREMAN performs local check for individual ACLs without considering the interaction with other firewalls in the network. Since a firewall can rely on the filtering action of other firewalls to achieve policy conformance, local checks focus on checking inconsistency and inefficiency. The local check is performed after parsing each rule, and just before updating the state as defined in Equation 2.

The input to an ACL is the entire set ($I = \Omega$), and $A_1 = D_1 = F_1 = \emptyset$.

- For $\langle P, \text{ accept}\rangle$ rules:
  1) $P_j \subseteq R_j \Rightarrow$ good: This is a good rule. It defines an action for a new set of packets, and it does not overlap with any preceding rules.
  2) $P_j \cap R_j = \emptyset \Rightarrow$ masked rule: This is an "error". This rule will not match any packets and action defined here will never be taken.

[3] Note that this paper uses the traffic from "outside" to "inside" for discussion. Our algorithm is general enough to consider traffic between any two points in the network.

a) $P_j \subseteq D_j \Rightarrow$ shadowing: This rule intended to accept some packets which have been dropped by preceding rules. This contradiction reveals a misconfiguration.
b) $P_j \cap D_j = \emptyset \Rightarrow$ redundancy: All the packets have been accepted by preceding rules or will not take this path.
c) else $\Rightarrow$ redundancy and correlation: Part of the packets for this rule have been dropped. Others are either accepted or will not take this path. Rule j itself is redundant since it will not match any packets. Some preceding rule has correlation with rule j also.

3) $P_j \not\subseteq R_j$ and $P_j \cap R_j \neq \emptyset \Rightarrow$ partially masked rule
a) $P_j \cap D_j \neq \emptyset \Rightarrow$ correlation: Part of the packets intend to be accepted by this rule have been dropped by preceding rules. This raises a "warning".
b) $\forall\ x < j, \exists\ \langle P_x,$ drop$\rangle$ s.t. $P_x \subseteq P_j \Rightarrow$ generalization: Rule j is a generalization of rule x since rule x matches a subset of the current rule j but defined a different action. This is a "warning".
c) $P_j \cap A_j \neq \emptyset$ and $\forall\ x < j, \exists \langle P_x,$ accept$\rangle$s.t.$P_x \subseteq P_j \Rightarrow$ redundancy: If rule $\langle P_x,$ accept$\rangle$ is removed, all the packets that match P_x can still be accepted to the current $\langle P_j,$ accept$\rangle$. Therefore, rule $\langle P_x,$ accept$\rangle$ is redundant. This is an "error".

- Similarly for $\langle P,$ drop$\rangle$ rules:
  1) $P_j \subseteq R_j \Rightarrow$ good:
  2) $P_j \cap R_j = \emptyset \Rightarrow$ masked rule:
  a) $P_j \subseteq A_j \Rightarrow$ shadowing: This rule intended to drop some packets which have been accept by preceding rules. This could be a serious security violation.
  b) $P_j \cap A_j = \emptyset \Rightarrow$ redundancy: All the packets have been dropped by preceding rules or will not take this path.
  c) else $\Rightarrow$ redundancy and correlation: Part of packets for this rule have been accepted. Others are dropped or will not take this path.
  3) $P_j \not\subseteq R_j$ and $P_j \cap R_j \neq \emptyset \Rightarrow$ partially masked rule
  a) $P_j \cap A_j \neq \emptyset \Rightarrow$ correlation: Part of the packets intend to be dropped by this rule have been accepted by earlier rules.
  b) $\forall\ x < j, \exists\ \langle P_x,$ accept$\rangle$ s.t. $P_x \subseteq P_j \Rightarrow$ generalization: Rule j is a generalization of rule x since rule x matches a subset of the current rule j but defined a different action.
  c) $P_j \cap A_j \neq \emptyset$ and $\forall\ x < j, \exists \langle P_x,$ drop$\rangle$s.t. $P_x \subseteq P_j \Rightarrow$ redundancy: If rule $\langle P_x,$ drop$\rangle$ is removed, all the packets that match P_x can still be dropped by the current rule. Therefore, rule $\langle P_x,$ drop$\rangle$ is redundant. This is an "error".

*2) **Checks for Distributed Firewalls**:* After passing the local checks, FIREMAN will perform distributed checks for network of firewalls. Such a check is performed based on the ACL-tree derived in Section IV-A.2. We start from the top level ACLs of the tree and go downwards level by level. At the top level, input to an ACL is the entire set ($I = \Omega$), and $A_1 = D_1 = F_1 = \emptyset$. Starting from the second level, we use Equations 4, 5 and 6 to derive the I set to the ACL. Based on the input I, we again traverse through rules in the ACL based on the same transformations defined in Equation 2.

- For $\langle P,$ accept$\rangle$ rules:
  1) $P \subseteq I \Rightarrow$ *good*: This is not a redundancy as in the case of local checks. A packet need to be permitted by *all* firewall on its path to reach destination.
  2) $P \subseteq \neg I \Rightarrow$ shadowing: This rule is shadowed by upstream ACLs. It tries to accept some packets that are blocked by *all* upstream firewalls. This kind of inconsistency can manifest as connectivity problems which are difficult to troubleshoot manually.
- For $\langle P,$ drop$\rangle$ rules:
  1) $P \subseteq I \Rightarrow$ raised security level?: This probably reveals a raised security level. In the case of Figure 2, certain packets might be allowed to access the DMZ but not the internal network. Therefore, ACLs $W0$, $X1$ and $X0$ will permit these packets but ACL $Y0$ will drop them.
  2) $P \subseteq \neg I \Rightarrow$ redundancy?: This is probably a redundancy since the packets to be dropped will not reach this ACL anyway. However, multiple lines of defense are often encouraged in practice to increase overall security level. This should be performed with caution by the administrator.

*3) **Checks at the Root of the ACL Tree**:* The root of the ACL tree is the destination, which is also the network we want to secure. Assume the root has m children, and child j gives input to the root as I_j. We want to ensure that all the inputs are the same. Otherwise, this is a "cross-path inconsistency" as discussed in Section III-B.3.

$$\forall j \in m, I_j = I \tag{8}$$

Policy conformance is checked by comparing the input I to the root of the ACL tree with the *blacklist* and *whitelist*. Since firewalls can rely on others to achieve policy conformance, checking at the root allows us to make the judgement based on the complete information of the entire ACL tree.

- $I \cap$blacklist $\neq \emptyset \Rightarrow$ policy violation: The firewalls permit some packets forbidden by the stated policy. This is a security violation.
- whitelist $\not\subseteq I \Rightarrow$ policy violation: The firewalls drop some protected packets. This causes disrupted service.

C. Formal Properties and Discussions

With respect to our definitions of misconfigurations in Section III, we have a soundness and completeness theorem for our analysis.

Theorem 1 (Soundness and Completeness): Our checking algorithm is both sound and complete:

- If the algorithm detects no misconfigurations, then there will not be any misconfigurations (*soundness*).
- Any misconfiguration detected by the algorithm is a real misconfiguration (*completeness*).

We can achieve both soundness and completeness (i.e., neither false negatives nor false positives) because firewalls are essentially finite-state systems. We perform symbolic model checking covering every path and every packet, that is we are doing exhaustive testing in an efficient manner.

Our algorithm is sound and complete with respect to our classification of misconfigurations. However, certain misconfigurations viewed as error by one administrator may not be viewed as errors by others. The concrete judgments for these cases depends on the intention of the particular administrator. There are cases that we cannot make concrete judgments and can only raise "warnings." These cases include, for example, *correlations* and *generalizations* for intra-firewall checks, and *raised security level* and *redundancy* for inter-firewall checks. This happens because we, as tool writers, do not know the intention of the administrator. This *intention gap*, however, does not affect our claim that the algorithm is sound and complete. Our tool raises "warnings" and leaves the decision to the administrator, who surely knows his/her own intention.

V. Implementation and Evaluation

A. BDD Representation of Firewall Rules

Updating state information for firewall rules and ACL graphs requires an efficient representation of the predicates of individual rules or any collection of the predicates. In addition, we must be able to implement efficient set operations with this representation. FIREMAN uses binary decision diagrams (BDDs) [16] to represent predicates and perform all the set operations. The BDD library used in FIREMAN is BuDDy [17], which provides efficient dynamic memory allocation and garbage collection.

BDD is an efficient data structure which is widely used in formal verification and simplification of digital circuits. A BDD is a directed acyclic graph that can compactly and canonically represent a set of boolean expressions [16]. The predicates of firewall rules describe constraints on certain fields of the packet header. We can represent them compactly and compute them efficiently using BDDs. For example, a source IP block $128.0.0.0/8$ can be represented as $x'_{32}x'_{31}x'_{30}x'_{29}x_{28}x'_{27}x_{26}x'_{25}$, whose corresponding BDD is shown in Figure 5(a). In such graph, the non-terminal vertices represent the variables of the boolean function, and the two terminal vertices represent the boolean values 0 and 1. To check if another source IP block is a subset of this IP block requires only a single *bdd_imp* (i.e., $\Rightarrow$, the logical implication) operation.

Performing set operations such as *intersection*, *union* and *not* on BDDs is also straightforward using BuDDy. Figure 5(c) presents the union of source IP $128.0.0.0/8$ (Figure 5(a)) and $192.0.0.0/8$ (Figure 5(b)). Note that BDDs can automatically summarize the two IP blocks and produce a canonical form for the union.

B. Building Blacklist

FIREMAN checks for policy violations based on given *blacklist* and *whitelist*. Although policy definitions are subjective to individual institution, there are some well-understood guidelines we believe that most administrators would want to observe.

The default blacklist of FIREMAN is built based on the bogon list [13] and 12 common mistakes pointed out by Wool [1]. For each IP blocks B listed on the bogon list, we read them as "drop any B any" and "drop any any B" rules to indicate that packets with either source IP in B or destination IP in B should be dropped.

Most rules listed in [1] can be encoded into the blacklist as well. Insecure or external access to firewall management can be encoded as "drop tcp any firewall telnet" which prevents telnet access to the firewall or "drop any external firewall" which prevents external access to the firewall. Insecure services like NetBios and Portmapper/Remote Procedure Call can be encoded as "drop any any any netbios" which prevents access to NetBios.

As discussed in Section IV-B.3, the blacklist describes prohibited behaviors and FIREMAN checks the firewall configurations against each item defined in the blacklist. The current implementation of FIREMAN does not define a default whitelist and this check is thus omitted. However, it is easy to write a list of predicates and FIREMAN can read them as "permit" rules and use it to compare against the input set I at the root of ACL tree.

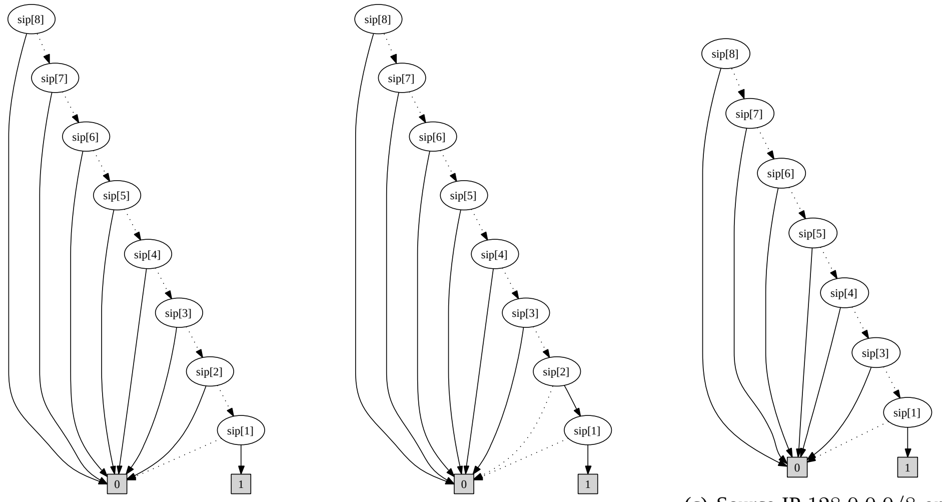
C. Misconfigurations Discovered

Obtaining production firewall configuration scripts is not easy because they contain sensitive security information. Table V lists the configuration files that we were able to obtain to test FIREMAN: $PIX1$ is for a Cisco PIX firewall used at an enterprise network; $BSD1$ is using OpenBSD packet filter at a campus network; and $PIX2$ is used by another enterprise network. Both $PIX1$ and $BSD1$ are actively used in production. All the script excerpts presented here have been modified to private IP address blocks. In Table V, the columns "P", "C", and "E" list respectively the number of policy violations, the number of inconsistencies, and the number of inefficiencies detected for each firewall configuration.

Firewall	Product	#ACLs	#rules	P	C	E
$PIX1$	PIX 6.03	7	249	3	16	2
$BSD1$	OpenBSD PF	2	94	3	0	0
$PIX2$	PIX 6.03	3	36	2	0	5

TABLE V: Configuration files and misconfigurations

1) Policy Violation (P): Policy violations are observed on all three configurations. $BSD1$ explicitly denied $10.0.0.0/8$, $172.16.0.0/12$ and $192.168.0.0/16$. In addition, the administrator commented that he/she wanted to deny all unroutable



(a) Source IP $128.0.0.0/8$

(b) Source IP $192.0.0.0/8$

(c) Source IP $128.0.0.0/8$ *or* $192.0.0.0/8$

Fig. 5: Using BDDs to represent and operate on firewall rules.

packets. Therefore, we infer that $BSD1$ is tasked to drop all unroutable packets. However, FIREMAN reveals that other unallocated IP address blocks such as $2.0.0.0/8$ and $5.0.0.0/8$, are implicitly allowed by rules like "permit udp any any port domain, ntp."

A similar problem is identified in $PIX1$. Only two of the three private IP blocks are explicitly dropped. It is interesting to note that while most administrators will not hesitate to block private IP address blocks, they may be reluctant to setup rules to filter unallocated IP address blocks as discussed in Section III-A. FIREMAN can be configured to read the latest bogon file every time it runs so that the bogon list is up to date. It can be used to enforce the policy to block all unroutable IP blocks.

Some of the 12 errors pointed by Wool [1] are observed on these three configurations. In particular, none of the three firewalls pays special attention to secure the firewall itself. They are not configured with a stealth rule to hide itself or limit the access to internal addresses and secure protocols. The default blacklist has a rule which denies any packets to the firewall itself not from internal network ("deny any !internal_IP firewall_IP"). FIREMAN diagnoses this problem by noting that the conjunction of input to the root of the ACL tree (I) and the blacklist is not empty.

Our results agree well with Wool's observation that firewalls are often not configured to well-understood security policy. In addition, FIREMAN is fully automated and does not require an experienced firewall/security expert to diagnose the problems.

2) Inconsistency (C): FIREMAN reported 8 correlations and 8 generalizations on one of the ACL in $PIX1$ which contains 141 rules. The rules causing the alarms are listed in Table VI. Rules 1–4 are permitting icmp access to individual hosts, and rules 5–8 are blocking icmp echo and traceroute to their networks. Therefore rules 1,3, and 4 are correlated with rules 5 and 6, and similarly, for rule 2 and rules 7, 8. Rule 9 is a generalization of rules 1–8.

1.	permit icmp any 10.2.53.192/32
2.	permit icmp any 10.2.54.3/32
3.	permit icmp any 10.2.53.249/32
4.	permit icmp any 10.2.53.250/32
5.	deny icmp any 10.2.53.0/24 echo
6.	deny icmp any 10.2.53.0/24 traceroute
7.	deny icmp any 10.2.54.0/24 echo
8.	deny icmp any 10.2.54.0/24 traceroute
9.	permit icmp any any

TABLE VI: Inconsistencies found in $PIX1$

This script probably does not have any misconfigurations. As discussed, correlations and generalizations can often be tricks used by administrators to represent rules efficiently.

3) Inefficiency (E): FIREMAN noted 5 redundancies in $PIX2$. As shown in Table VII, rules 2 and 3 will not match any packets because they are matching a subset of those matched by rule 1. In addition, rule 4 is a generalization of rules 1, 2 and 3. One could keep only the rule 4 and achieve the same effect.

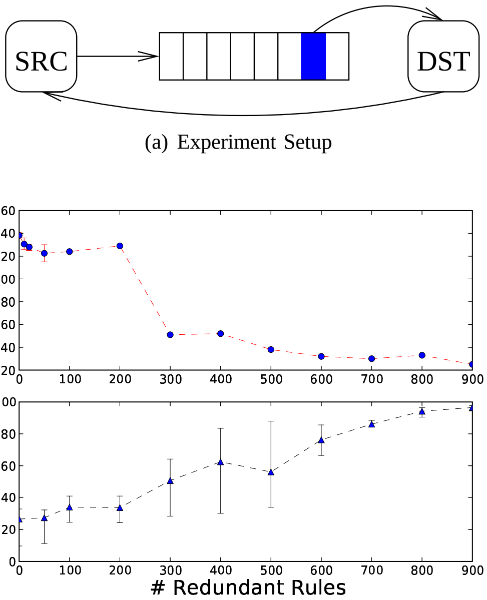
1.	permit ip 192.168.99.0/24 192.168.99.0/24
2.	permit ip 192.168.99.56/32 192.168.99.57/32
3.	permit ip 192.168.99.57/32 192.168.99.56/32
4.	permit ip 192.168.99.0/24 any

TABLE VII: Inefficiencies found in $PIX2$

Another redundancy FIREMAN caught is in $PIX1$, which explicitly denies $10.0.0.0/8$ and $192.168.0.0/16$ in some of its ACLs. However, since these two rules are the last two rules in the ACL, and the default action of PIX is to deny anything remaining, these two rules are unnecessary and reported as *redundancy*. Private communication with the administrator confirmed this observation, and the redundant rules will be removed.

D. Performance Impact on Firewall

Since inefficiency in firewall configurations does not violate an enterprise's network policies, administrators seldom notice or correct this problem. However, we will show that by using FIREMAN to identify redundant rules in a firewall configuration, we can improve the firewall throughput and reduce the CPU load significantly.



(a) Experiment Setup

(b) Impact of Redundant Rules on Throughput and CPU Load

Fig. 6: Performance impact of redundancies in firewalls

Figure 6(a) depicts our testbed consisting of two Linux machines, SRC and DST, both running Fedora Core 3 with kernel $2.6.1$ and directly connected via a Gigabit Ethernet. An *Iperf* client [18] on SRC sent packets to an *Iperf* server on DST. DST was also running a Netfilter firewall on the incoming interface. In our experiment, the firewall initially had a synthetically generated configuration where the rule that accepted all the packets from SRC was preceded by 900 rules that let these packets pass through (therefore, these 900 rules were redundant). We measured the throughput using *Iperf* over a 60-second period and the CPU load using the *top* command. We then graduately removed all the redundant rules and measured how it affected the throughput and CPU load.

Figure 6(b) presents the performance of the firewall under different levels of redundancies. The CPU load was close to 100% when the firewall had many redundant rules; however, the CPU load was reduced to only 30% when we removed all the redundancies. This indicates that removing redundancies in firewalls can reduce the CPU load significantly.

Removing redundancies can also improve the throughput. Figure 6(b) shows that FIREMAN improved the throughput from 620 Mbps to 740 Mbps by removing all the redundancies. We conjecture that removing redundancies increased throughput because: (1) it reduced the processing time on each packet and hence increased the throughput, and (2) it decreased the RTT(round-trip time) of the corresponding TCP session, and the achievable throughput of a TCP session is inversely proportional to its RTT [19].

The practical performance benefits of reducing redundancies in firewall configurations depend on the implementation of individual firewalls and the traffic profile at the firewalls. Particularly, we expect that large, complex firewalls would benefit more significantly than small, simple firewalls. But in all cases, since FIREMAN is a static analysis tool and therefore incurs no runtime cost, any firewall may benefit from FIREMAN with no performance penalty.

E. Performance and Scalability of FIREMAN

The complexity of intra-firewall checking is determined by the complexity of checking each rule and the number of rules in a configuration. Our algorithm performs the usual set operations, conjunction, disjunction, and complementation, on the A, D, and F sets for each rule. Our implementation (cf. Section V) uses binary decision diagrams (BDDs) to represent these sets canonically for efficient processing. On firewalls using the simple list model, our algorithm traverses each rule exactly once, so the total running time is $O(n)$, where n is the number of rules. This is witnessed in Figure 7, which shows that the average time required to check an ACL is proportional to its length for synthetically generated ACLs of different lengths. For example, it took FIREMAN less than 3 seconds to check an 800-rule ACL. Our algorithm scales better than Al-Shaer's, which compares two rules at a time and has an $O(n^2)$ complexity [11].

For firewalls using the complex chain model, we can achieve $O(n)$ time complexity with the following optimizations: (1) storing the state information and reusing it; and (2) merging the state information whenever possible. Next, we discuss in more detail these optimizations together with distributed firewall checking.

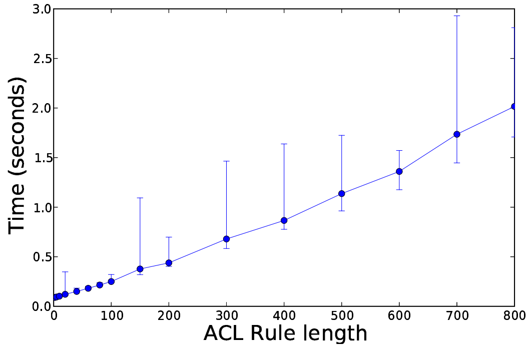


Fig. 7: FIREMAN's performance on checking individual firewalls

For distributed firewalls, the number of paths from "outside" to "inside" may be exponential. For example, for a graph with m nodes and an average outdegree k, there can be $O(k^m)$ simple paths in the worst-case. As firewalls often reside on normal routers, m and k may be large. Since checking each path separately would not scale, FIREMAN uses the following

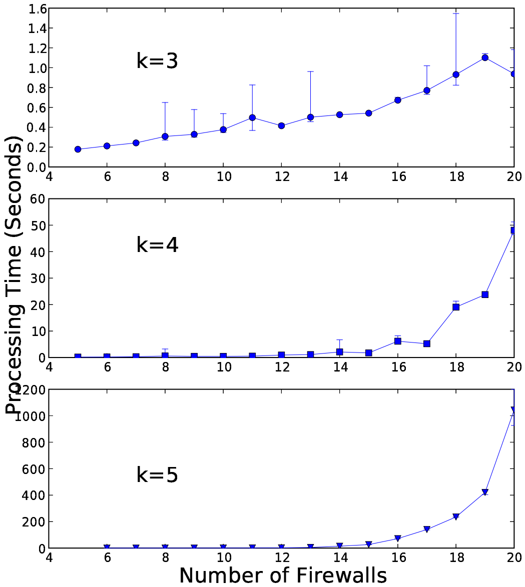


Fig. 8: FIREMAN's performance on checking distributed firewalls

techniques to improve its scalability. First, FIREMAN avoids unnecessary nodes and branching. Since firewalls and ACLs are rarely deployed in the network core, FIREMAN can reduce the number of nodes by combining a network of routers without ACLs into a single *abstract virtual node*. For instance, FIREMAN treats the DMZ in Figure 2 as a single node. Second, FIREMAN merges the paths where possible using Equations 4, 5 and 6 from Section IV. For example, consider the ACL tree in Figure 4. Instead of traversing

$$\text{outside} \rightarrow W0 \rightarrow Z0 \rightarrow \text{inside}$$

and

$$\text{outside} \rightarrow X0 \rightarrow X1 \rightarrow Z0 \rightarrow \text{inside}$$

separately, FIREMAN merges the two paths at $Z0$. Finally, FIREMAN saves intermediate results for reuse later. In Figure 4, FIREMAN checks the ACLs $W0$, $X0$ and $X1$ only once because the left and the right branches are symmetric. When an ACL appears multiple times in an ACL tree, FIREMAN rechecks the ACL while traversing the tree only if the ACL receives different input sets on different paths.

In Figure 8, we randomly generated a network of m firewalls, where each firewall connects to k other firewalls. When either m or k was small, distributed checking finished within seconds. When $m > 15$ and $k > 5$, distributed checking took several minutes. Even in the worst case with $m = 20$ and $k = 5$ (which we think is rare in enterprise networks), FIREMAN completed in under 20 minutes. Because FIREMAN runs offline, we believe that FIREMAN is scalable enough to check most distributed firewalls effectively.

VI. Related Work

There are numerous studies on network topology, IP connectivity [20–22], and router configurations [23–26]. Maltz et al. [27] reverse engineered routing design in operational networks from static analysis of dumps of the local configuration state of each router. These configuration files were automatically processed to abstract routing process graphs, route pathway graphs and address space structures. With these abstractions, network structure and routing operation in a global view were retrieved for further analysis. The subsequent work by the same authors [28] presented a unified modeling of packet filters and routing protocols to characterize *reachability* of a network. Our work makes another step towards understanding how distributed firewalls, as one of the "packet transformers" in the network, influence end-to-end behavior.

Firmato and Fang [6, 7] are a set of management and analysis tools that interact with users on queries about firewall rules. Lumeta [8] improved usability of Fang by automating the queries to check if firewalls are configured according to users' expectations. Both tools take a minimum network topology description and firewall configurations as input to build an internal representation of firewall rules which users can query. Such tools are essentially lightweight testing tools and do not offer the advantage of full coverage as static analysis tools do. Our goal is different and focus on checking for misconfigurations.

The work closest to ours is "Firewall Policy Advisor" by Al-Shaer and Hamed [10, 11]. Our classification of misconfigurations are inspired by them, but are more general and complete. The key distinction of FIREMAN is its capability to evaluate firewall configurations as a whole piece, not just limited to relation between *two* firewall rules. In addition to inconsistencies, FIREMAN also checks for policy violations and inefficiencies. Furthermore, FIREMAN works on any network topology and requires only a linear traversal through the rules. Our experiment running FIREMAN captured all misconfigurations in their sample scripts [11]

Gouda and Liu [29] used a firewall decision diagram to assist the design of firewalls, in order to compact configuration size while maintaining its consistency and completeness. Their focus is on the efficiency of a single firewall. Eronen and Zitting [30] described an expert system based on Eclipse, a constraint logic programming language to render Cisco router access lists.

Hazelhurst et al. [9] proposed to use BDD to represent firewall rules and access lists. Their goal was to achieve fast lookup through better hardware router implementation using BDDs. We chose BDD to represent not only individual rules but also the collective behavior of whole set of rules. Our focus is on checking the security properties of firewall rules on existing architectures.

VII. Conclusions

In this paper, we have presented a novel static analysis approach to check firewall configurations. First, we have proposed a framework for modeling individual and distributed firewalls. Second, we have designed a static analysis algorithm to discover various misconfigurations such as policy violations, inconsistencies and inefficiencies, at various levels including intra-firewall, inter-firewall, and cross-path. Our technique is based on symbolic model checking, using binary decision diagrams to compactly represent and efficient process firewall rules. Compared to other related research, our method is scalable and offers full-coverage of all possible IP packets and data paths. Our analysis algorithm is both sound and complete, thus has no false negatives and false positives.

We have implemented our approach in a toolkit called FIREMAN, which uses BDDs to represent firewall rules. Inspecting misconfigurations is fast, scalable and requires minimum amount of memory. In our experiments, FIREMAN was able to uncover misconfigurations on firewalls running in production environment. We believe FIREMAN is a useful and practical tool for network administrators as well as personal firewall users.

References

[1] A. Wool, "A quantitative study of firewall configuration errors," *IEEE Computer*, vol. 37, no. 6, 2004.

[2] F. Wizards, "Firewall wizards security mailing list," http://honor.icsalabs.com/mailman/listinfo/firewall-wizards.

[3] D. B. Chapman, "Network (in)security through IP packet filtering," in *Proceedings of the Third Usenix Unix Security Symposium*, Baltimore, MD, September 1992, pp. 63–76.

[4] H. Chen, D. Wagner, and D. Dean, "Setuid demystified," in *Proceedings of the Eleventh USENIX Security Symposium*, San Francisco, CA, 2002.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[6] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *Proc. 20th IEEE Symposium on Security and Privacy*, 1999.

[7] A. Mayer, A. Wool, and E. Ziskind, "Fang: A firewall analysis engine," in *Proc. IEEE Symposium on Security and Privacy*, 2000.

[8] A. Wool, "Architecting the Lumeta firewall analyzer," in *Proc. 10th USENIX Security Symposium*, Washington, D.C., 2001.

[9] S. Hazelhurst, A. Attar, and R. Sinnappan, "Algorithms for improving the dependability of firewall and filter rule lists," in *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks*, 2000.

[10] E. Al-Shaer and H. Hamed, "Firewall policy advisor for anomaly detection and rule editing," in *Proc. IEEE/IFIP Integrated Management Conference (IM'2003)*, 2003.

[11] ——, "Discovery of policy anomalies in distributed firewalls," in *Proc. IEEE Infocomm*, Hong Kong, 2004.

[12] Netfilter, "Linux netfilter," http://www.netfilter.org.

[13] T. Cymru, "The Team Cymru Bogon List v2.5 02 AUG 2004," http://www.cymru.com/Documents/bogon-list.html, 2004.

[14] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 2003.

[15] A. X. Liu and M. G. Gouda, "Removing redundancy from packet classifiers," University of Texas at Austin, Tech. Rep., 2004.

[16] H. R. Andersen, "An introduction to binary decision diagrams," http://www.itu.dk/people/hra/notes-index.html, 1997.

[17] J. Lind-Nielsen, "Buddy version 2.4," http://sourceforge.net/projects/buddy, 2004.

[18] DAST/NLANR, http://dast.nlanr.net/Projects/Iperf/, Mar 2005.

[19] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the congestion avoidance algorithm," *Computer Communications Review*, vol. 27, no. 3, 1997.

[20] CAIDA, "Skitter tool," http://www.caida.org/tools/measurement/skitter.

[21] R. Govindan and H. Tangmunarunkit, "Heuristics for Internet Map Discovery," in *IEEE INFOCOM*, 2000.

[22] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," in *ACM SIGCOMM*, 2002.

[23] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg,G. Hjalmtysson, and J. Rexford, "The cutting edge of IP router configuration," in *ACM HotNets*, 2003.

[24] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP Misconfiguration," in *ACM SIGCOMM*, 2002.

[25] N. Feamster, "Practical verification techniques for wide-area routing," in *ACM SIGCOMM HotNets-II*, 2003.

[26] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *Proc. NSDI*, 2005.

[27] D. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjalmtysson, and A. Greenberg, "Routing design in operational networks: A look from the inside," in *Proc. SIGCOMM'04*, 2004.

[28] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, and J. Rexford, "On static reachability analysis of IP networks," in *IEEE INFOCOM*, 2005.

[29] M. G. Gouda and X.-Y. A. Liu, "Firewall design: consistency, completeness and compactness," in *Proc. ICDCS 24*, 2004.

[30] P. Eronen and J. Zitting, "An expert system for analyzing firewall rules," in *Proc. 6th Nordic Worksh. Secure IT Systems*, 2001.