Author's personal copy

Computer Networks 53 (2009) 2800–2809

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Firewall policy verification and troubleshooting

Alex X. Liu

Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1266, United States

ARTICLE INFO

Article history:
Received 23 February 2008
Received in revised form 7 July 2009
Accepted 10 July 2009
Available online 15 July 2009
Responsible Editor: Chuang Lin

Keywords:
Firewall policy
Firewalls
Network security

ABSTRACT

Firewalls are important elements of enterprise security and have been the most widely adopted technology for protecting private networks. The quality of protection provided by a firewall mainly depends on the quality of its policy (i.e., configuration). However, due to the lack of tools for verifying and troubleshooting firewall policies, most firewalls on the Internet have policy errors. A firewall policy can error either create security holes that will allow malicious traffic to sneak into a private network or block legitimate traffic disrupting normal traffic, which in turn could lead to diestrous consequences.

We propose a firewall verification and troubleshooting tool in this paper. Our tool takes as input a firewall policy and a given property, then outputs whether the policy satisfies the property. Furthermore, in the case that a firewall policy does not satisfy the property, our tool outputs which rules cause the verification failure. This provides firewall administrators a basis for how to fix the policy errors.

Despite of the importance of verifying firewall policies and finding troublesome rules, they have not been explored in previous work. Due to the complex nature of firewall policies, designing algorithms for such a verification and troubleshooting tool is challenging. In this paper, we designed and implemented a verification and troubleshooting algorithm using decision diagrams, and tested it on both real-life firewall policies and synthetic firewall policies of large sizes. The performance of the algorithm is sufficiently high that they can practically be used in the iterative process of firewall policy design, verification, and maintenance. The firewall policy troubleshooting algorithm proposed in this paper is not limited to firewalls. Rather, they can be potentially applied to other rule-based systems as well.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Firewall serve as the first line of defense against malicious attacks and unauthorized traffic. They are cornerstones of network security and have been widely deployed in businesses and institutions. A firewall is placed at the point of entry between a private network and the outside Internet such that all incoming and outgoing packets have to pass through it. The function of a firewall is to examine every incoming or outgoing packet and decide whether to accept or discard it. This function is specified by a sequence (i.e., an ordered list) of rules, which is called the "policy", i.e., the configuration, of the firewall.

Each rule in a firewall policy is of the form ⟨*predicate*⟩ → ⟨*decision*⟩. The ⟨*predicate*⟩ of a rule is a boolean expression over some packet fields such as source IP address, destination IP address, source port number, destination port number, and protocol type. The ⟨*decision*⟩ of a rule can be *accept*, *discard*, or a combination of these decisions with other options such as a logging option. The rules in a firewall policy often conflict. To resolve such conflicts, the decision for each packet is the decision of the first (i.e., highest priority) rule that the packet matches. Table 1 shows an example firewall.

1.1. Motivations

In this paper, we consider the following two problems that are frequently raised in practice: *given a firewall policy*

E-mail address: alexliu@cse.msu.edu

1389-1286/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2009.07.003

Author's personal copy

Table 1
An example firewall.

Rule	Src IP	Dest. IP	Src Port	Dest. Port	Protocol	Action
r_1	*	192.168.0.1	*	25	TCP	accept
r_2	1.2.*.*	*	*	*	*	discard
r_3	*	*	*	*	*	accept

and a property, how can an administrator verify that the firewall policy satisfies the property? If the property is not satisfied, what are the troublesome rules in the policy that cause the verification failure? In this context, a property is a high-level policy that a firewall needs to enforce. An example property could be "the people in the development zone should not be able to access the database server in the accounting zone". Such a tool is helpful for firewall administrators to analyze and debug their firewalls as well as other routine duties such as demonstrating to their manager that the firewall does satisfy a set of necessary properties. This tool also can serve as a debugging tool in designing and analyzing firewall policies.

Due to the subtle and elusive nature of firewall rules, correctly verifying whether a firewall satisfies a property is by no means easy. First, the rules in a firewall policy are logically entangled because of conflicts among rules and the resulting order sensitivity. Second, a firewall policy may consist of a large number of rules. A firewall on the Internet may consist of hundreds or even a few thousand rules in extreme cases. Third, an enterprise firewall policy often consists of legacy rules that are written by different administrators, at different times, and for different reasons, which makes verifying firewall policies even more difficult. Verifying a large and complex sequence of logically related rules is certainly beyond human capability. Last but not least, an administrator can easily be deceived to believe that a property is satisfied by some rules in the middle of a firewall policy that seemingly implement the property.

Effective methods and tools for verifying and troubleshooting firewall policies, therefore, are crucial to the success of firewalls. However, firewall administrators are woefully under-assisted due to the lack of firewall policy verification and troubleshooting tools. Quantitative studies have shown that most firewalls on the Internet are plagued with policy errors [41]. Firewall policy errors can be dangerous and costly. On one hand, if a firewall policy error permits illegitimate communication, outside attackers may use these security holes to launch attacks. On the other hand, if a firewall policy error disallows legitimate communication, it may cause significant loss due to interrupted businesses. For example, if a firewall policy error prevents the communication between a web server and its supporting database server, all transactions that need such communication are interrupted.

1.2. Key contributions

Despite of the importance of verifying firewall policies and finding troublesome rules, they have not been well explored in previous work. In this paper, we propose an efficient algorithms for verifying firewall policies and finding troublesome rules. To our best knowledge, this paper represents the first study of firewall policy verification and troublesome rules identification. The input of our verification algorithm includes a firewall policy and a given property, and the output is whether the policy satisfies the property. In the case that a firewall policy does not satisfy a property, our algorithm outputs which rules cause the verification failure. This provides firewall administrators a basis for how to fix the policy.

1.3. Road map

The rest of this paper proceeds as follows. We start with some example applications of our firewall verification and troubleshooting algorithm in Section 2. We then formally define the terms that use throughout this paper in Section 2. In Section 4, we present our firewall verification and troubleshooting algorithm. In Section 5, we discuss some further issues for firewall verification and troubleshooting. In Section 6, we show our experimental results. In Section 7, we review previous related work. In Section 8, we give concluding remarks.

Because the focus of this paper is on security policies, we simply use the term "firewall" to mean "firewall policy", "firewall rule set", or "firewall configuration" unless otherwise specified.

2. Example

In this section, we show an example application of our firewall verification and troubleshooting tool. Consider the example firewall in Table 1. We suppose that the private network behind this firewall has a mail server, whose IP address is 192.168.0.1.

Here we briefly explain the meaning of the three rules in Table 1. Rule r_1 means that all email packets to the email server are accepted. Note that for a packet, if its destination port number is 25 and its protocol type is TCP, then the packet is an email (SMTP) packet. Rule r_2 means that all packets from the domain 1.2.*.* are discarded. Rule r_3 means that all packets are accepted. Note that whenever a packet arrives at a firewall, the decision of the first rule that the packet matches is executed.

2.1. Verification example 1

Suppose that this firewall is required by its high-level security policies to discard all packets from a malicious domain 1.2.*.*. Given the policy in Table 1 and this property, our verification tool will report the following two things.

Author's personal copy

First, this property is not satisfied by this firewall. Second, it is rule r_1 that causes the verification failure.

2.2. Verification example 2

Suppose that the private network has a web server, whose IP address is 192.168.0.2, and the firewall is required by its high-level security policies to accept all HTTP traffic to and from the web server except the web traffic to and from the malicious domain 1.2.*.*. Given the policy in Table 1 and this property, our verification tool will report that this property is satisfied by the firewall.

3. Formal definitions

We now formally define the concepts of fields, packets, and firewalls. A *field* F_i is a variable of finite length (i.e., of a finite number of bits). The domain of field F_i of w bits, denoted $D(F_i)$, is $[0, 2^w - 1]$. A *packet* over the d fields $F_1, \ldots, F_d$ is a d-tuple $(p_1, \ldots, p_d)$ where each $p_i (1 \leqslant i \leqslant d)$ is an element of $D(F_i)$. Firewalls usually check the following five fields: source IP address, destination IP address, source port number, destination port number, and protocol type. The lengths of these packet fields are 32, 32, 16, 16, and 8, respectively. We use Σ to denote the set of all packets over fields $F_1, \ldots, F_d$. It follows that Σ is a finite set and $|\Sigma| = |D(F_1)| \times \cdots \times |D(F_d)|$, where $|\Sigma|$ denotes the number of elements in set Σ and $|D(F_i)|$ denotes the number of elements in set $D(F_i)$.

A *rule* has the form $\langle predicate \rangle \rightarrow \langle decision \rangle$. A $\langle predicate \rangle$ defines a set of packets over the fields F_1 through F_d, and is specified as $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ where each S_i is a subset of $D(F_i)$. A rule $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ is called an *atomic rule* if and only if each S_i is specified as either a prefix or a nonnegative integer interval. A *prefix* $\{0,1\}^k\{*\}^{w-k}$ with k leading 0 s or 1 s for a packet field of length w denotes the integer interval $[\{0,1\}^k\{0\}^{w-k}, \{0,1\}^k\{1\}^{w-k}]$. For example, prefix 01** denotes the interval [0100, 0111]. As another example, IP prefix 1.2.*.* denotes the range of IP addresses from 1.2.0.0 to 1.2.255.255. Note that an IP address of 4 bytes can be deemed as a nonnegative integer of 4 bytes. As a non-atomic rule can be easily converted into multiple atomic rules, in this paper, we assume all rules are atomic; therefore, we simply use the term "rules" to mean "atomic rules" if not otherwise specified.

A packet matches a rule if and only if the packet matches the predicate of the rule. A packet $(p_1, \ldots, p_d)$ *matches* a predicate $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ if and only if the condition $p_1 \in S_1 \wedge \cdots \wedge p_d \in S_d$ holds. We use *DS* to denote the set of possible values that $\langle decision \rangle$ can be. Typical elements of *DS* include accept, discard, accept with logging, and discard with logging.

A sequence of rules $\langle r_1, \ldots, r_n \rangle$ is *complete* if and only if for any packet p, there is at least one rule in the sequence that p matches. To ensure that a sequence of rules is complete and thus a firewall, the predicate of the last rule is usually specified as $F_1 \in D(F_1) \wedge \cdots F_d \in \wedge D(F_d)$. A *firewall* f is a sequence of rules that is complete. The size of f, denoted $|f|$, is the number of rules in f.

Two rules in a firewall may *overlap*; that is, there is at least one packet that match both rules. Furthermore, two rules in a firewall may *conflict*; that is, the two rules not only overlap but also have different decisions. Firewalls typically resolve such conflicts by employing a first-match resolution strategy where the decision for a packet p is the decision of the first (i.e., highest priority) rule that p matches in a firewall. The decision that firewall f makes for packet p is denoted $f(p)$.

We can think of a firewall f as defining a many-to-one mapping function from Σ to *DS*. Two firewalls f_1 and f_2 are *equivalent*, denoted $f_1 \equiv f_2$, if and only if they define the same mapping function from Σ to *DS*; that is, for any packet $p \in \Sigma$, we have $f_1(p) = f_2(p)$. A rule is *redundant* in a classifier if and only if removing the rule does not change the semantics of the classifier.

4. Firewall verification

In this section, we present a method for verifying firewall properties and finding troublesome rules.

4.1. Property representation

To verify whether a firewall satisfies a given property, we need to translate the property to a set of non-overlapping rules, which we call *property rules* in this context to distinguish them from firewall rules. For example, the property of discarding all packets from a malicious domain 1.2.*.* can be converted to the property rule in Tables 2.

How to convert high-level descriptions of a property to a set of property rules is out of the scope of this paper. We assume that property rules are available for verification purposes.

Given a firewall and a set of property rules, the verification is successful if and only if every property rule is satisfied by the firewall. Next, we focus on how to verify whether a firewall satisfies one property rule.

4.2. Verification of non-overlapping firewalls

To make our firewall verification algorithm easy to understand, we first assume that the given firewalls are non-overlapping. A firewall is non-overlapping if and only if no rules in the firewall overlap. Two rules overlap if and only if there exists at least one packet that can match both rules. Note that real-life firewalls are most likely overlapping ones. The above unrealistic assumption is only for the purpose of introducing our verification algorithm that does not require this assumption.

Fig. 1 shows an example non-overlapping firewall. In this firewall, for simplicity, we assume each packet has only two fields, F_1 and F_2, and the domain of each field is [1,100].

Table 2
Example property rule.

Src IP	Dest. IP	Src Port	Dest. Port	Protocol	Action
1.2.*.*	*	*	*	*	discard

Author's personal copy

$$
\begin{array}{lllll}
R_1: & F_1 \in [20, & 50] & \wedge F_2 \in [20, & 70] & \rightarrow accept \\
R_2: & F_1 \in [20, & 50] & \wedge F_2 \in [1, & 19] & \rightarrow accept \\
R_3: & F_1 \in [20, & 50] & \wedge F_2 \in [71, & 100] & \rightarrow discard \\
R_4: & F_1 \in [1, & 19] & \wedge F_2 \in [1, & 39] & \rightarrow accept \\
R_5: & F_1 \in [51, & 60] & \wedge F_2 \in [1, & 39] & \rightarrow accept \\
R_6: & F_1 \in [1, & 19] & \wedge F_2 \in [40, & 100] & \rightarrow discard \\
R_7: & F_1 \in [51, & 60] & \wedge F_2 \in [1, & 100] & \rightarrow discard \\
R_8: & F_1 \in [61, & 100] & \wedge F_2 \in [1, & 100] & \rightarrow discard
\end{array}
$$

Fig. 1. A non-overlapping firewall.

Suppose that we want to verify whether this firewall satisfies the following property rule:

$$F_1 \in [30, 40] \wedge F_2 \in [80, 90] \rightarrow discard$$

Comparing this property rule with each rule in the firewall, we can find that this property rule does not conflict with any rule. Two rules conflict if and only if they overlap and they have different decisions. Therefore, this property rule is satisfied by the firewall.

As another example, suppose that we want to verify whether this firewall satisfies the following property rule:

$$F_1 \in [1, 100] \wedge F_2 \in [20, 30] \rightarrow discard$$

Comparing this property rule with each rule in the firewall, we can find that this property rule conflicts with both rule R_4 and R_5. Therefore, this property rule is not satisfied, and rule R_4 and R_5 are the cause of the failure.

The algorithm for verifying non-overlapping firewalls can be directly derived from Theorem 1.

Theorem 1. *A non-overlapping firewall satisfies a given property rule if and only if the property rule does not conflict with any rule in the firewall.*

Proof. We first prove that if a property rule does not conflict with any rule in a non-overlapping firewall then the firewall satisfies the property rule. Consider a non-overlapping firewall $f = \langle r_1, \ldots, r_n \rangle$. For any rule r_i, $1 \leqslant i \leqslant n$, we use $\mathbb{M}_{r_i}$ to denote the set of packets that match r_i and D_{r_i} to denote the decision of r_i. Because f is a non-overlapping firewall, we have $\Sigma = \mathbb{M}_{r_1} \cup \cdots \cup \mathbb{M}_{r_n}$. Therefore, for any property rule r, $\mathbb{M}_r = \mathbb{M}_r \cap \Sigma = \mathbb{M}_r \cap (\mathbb{M}_{r_1} \cup \cdots \cup \mathbb{M}_{r_n}) = (\mathbb{M}_r \cap \mathbb{M}_{r_1}) \cup \cdots \cup (\mathbb{M}_r \cap \mathbb{M}_{r_n})$. For any $1 \leqslant i, j \leqslant n$, because $M_{r_i} \cap \mathbb{M}_{r_j} = \emptyset$, we have $(\mathbb{M}_r \cap \mathbb{M}_{r_1}) \cap (\mathbb{M}_r \cap \mathbb{M}_{r_n}) = \emptyset$. Thus, for any packet $p \in \mathbb{M}_r$, there exists one and only one $1 \leqslant i \leqslant n$ such that $p \in \mathbb{M}_r \cap \mathbb{M}_{r_i}$. Because r does not conflict with r_i, we have $D_r = D_{r_i} = f(p)$. Therefore, firewall f satisfies the property rule r.

Second, we prove that if a non-overlapping firewall satisfies a given property rule then the property rule does not conflict with any rule in the firewall. Consider a non-overlapping firewall $f = \langle r_1, \ldots, r_n \rangle$ that satisfies a property rule r. As shown above, $\mathbb{M}_r = \mathbb{M}_r \cap \Sigma = \mathbb{M}_r \cap (\mathbb{M}_{r_1} \cup \cdots \cup \mathbb{M}_{r_n}) = (\mathbb{M}_r \cap \mathbb{M}_{r_1}) \cup \cdots \cup (\mathbb{M}_r \cap \mathbb{M}_{r_n})$. For any $1 \leqslant i \leqslant n$ that $\mathbb{M}_r \cap \mathbb{M}_{r_i} \neq \emptyset$ and for any packet $p \in \mathbb{M}_r \cap \mathbb{M}_{r_i}$, we have $f(p) = D_{r_i}$. Because f satisfies property rule r, we have $f(p) = D_r$. Therefore, $D_{r_i} = D_r$, which means that r_i does not conflict with r. Note that if $\mathbb{M}_r \cap \mathbb{M}_{r_i} = \emptyset$, r_i does not conflict with r because they do not overlap. □

4.3. Verification of generic firewalls

Here we consider the verification of generic firewalls, where the given firewall can be either overlapping or non-overlapping. A firewall is called an overlapping firewall if and only if there are at least two rules in the firewall which are overlapping. Fig. 2 shows an example overlapping firewall.

4.3.1. Firewall decision diagrams

An important data structure used in firewall verification is firewall decision diagrams. A firewall decision diagram is a compact and conflict-free representation of firewalls [14,16]. A Firewall Decision Diagram (FDD) with a decision set DS and over fields $F_1, \ldots, F_d$ is an acyclic and directed graph that has the following five properties:

(1) There is exactly one node that has no incoming edges. This node is called the *root*. The nodes that have no outgoing edges are called *terminal* nodes.
(2) Each node v has a label, denoted $F(v)$, such that

$$F(v) \in \begin{cases} \{F_1, \ldots, F_d\} & \text{if } v \text{ is a nonterminal node,} \\ DS & \text{if } v \text{ is a terminal node.} \end{cases}$$

(3) Each edge $e: u \rightarrow v$ is labeled with a nonempty set of integers, denoted $I(e)$, where $I(e)$ is a subset of the domain of u's label (i.e., $I(e) \subseteq D(F(u))$).
(4) A directed path from the root to a terminal node is called a *decision path*. No two nodes on a decision path have the same label.
(5) The set of all outgoing edges of a node v, denoted $E(v)$, satisfies the following two conditions:
   (a) *Consistency*: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$.
   (b) *Completeness*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$. □

For example, we can convert the firewall in Fig. 2 to a firewall decision diagram as shown in Fig. 3. Note that we use "a" to represent "accept" and "d" to represent "discard".

The FDD data structure is similar to the interval decision diagrams (IDDs) in [5] as well as the GEM data structure in [37]. The detailed difference between FDDs and IDDs is discussed in [16].

We define a *full-length ordered FDD* as an FDD where in each decision path all fields appear exactly once and in the same order. For ease of presentation, as the rest of this paper only concerns full-length ordered FDDs, we use the term "FDD" to mean "full-length ordered FDD" if not otherwise specified.

An FDD is *reduced* if and only if no two nodes are isomorphic and no two nodes have more than one edge between them. Two nodes v and v' in an FDD are *isomorphic* if and only if v and v' satisfy one of the following

$$
\begin{array}{lllll}
r_1: & F_1 \in [20, & 50] & \wedge F_2 \in [20, & 70] & \rightarrow accept \\
r_2: & F_1 \in [1, & 60] & \wedge F_2 \in [40, & 100] & \rightarrow discard \\
r_3: & F_1 \in [1, & 100] & \wedge F_2 \in [1, & 100] & \rightarrow accept
\end{array}
$$

Fig. 2. An overlapping firewall.

Author's personal copy

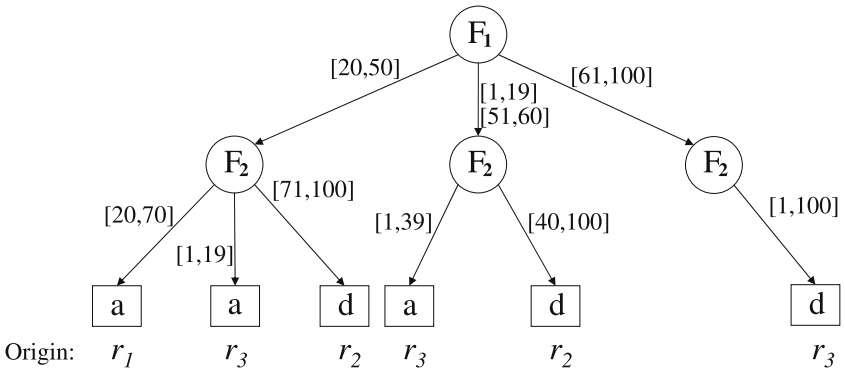

Fig. 3. A firewall decision diagram.

two conditions: (1) both v and v' are terminal nodes with identical labels; (2) both v and v' are nonterminal nodes and there is a one-to-one correspondence between the outgoing edges of v and the outgoing edges of v' such that every pair of corresponding edges have identical labels and they both point to the same node.

A decision path in a firewall decision diagram is a path from the root to a terminal node. Each decision path defines a rule. For example, the leftmost decision path in Fig. 3 defines the following rule:

$F_1 \in [20,50] \wedge F_2 \in [20,70] \rightarrow accept$

Fig. 1 shows the six rules defined by the six decision paths of the firewall decision diagram in Fig. 3.

4.3.2. List-based verification algorithm

To verify whether a given firewall satisfies a property rule, one solution is to first convert the firewall to an equivalent non-overlapping firewall, then conduct the verification on the non-overlapping firewall. For example, we can convert the firewall in Fig. 2 to an equivalent non-overlapping firewall as shown in Fig. 1. Recall that two firewalls are equivalent if and only if for any packet the two firewalls have the same decision.

Converting an overlapping firewall to a non-overlapping firewall is non-trivial. The best algorithm that we know of is to convert the given overlapping firewall to an equivalent reduced FDD and then generate a non-overlapping firewall from the FDD. Given an overlapping firewall, we can efficiently construct an equivalent FDD using the FDD construction algorithm in [27,29] and then reduce it. From the reduced FDD, we can easily generate a non-overlapping firewall by generating a rule from each decision path.

The complexity of the list-based verification algorithm is $O(dn^d)$. The above FDD-assisted algorithm converts a given overlapping firewall of n rules to a non-overlapping firewall of $O(n^d)$ rules in the worst case. Here d is the number of fields that a firewall examines. The typical value of d is 4 (for the four fields of source IP address, destination IP address, destination port number, and protocol type) or 5 (for the above 4 fields and the field of source port number). The complexity $O(n^d)$ is calculated as follows. Given a firewall of n rules, considering field F_i, the n ranges divide the domain of F_i into at most $2n-1$ ranges. Therefore, in the reduced FDD constructed from the firewall, for any nonterminal node labeled F_i, it has at most $2n-1$ outgoing edges. Therefore, the reduced FDD has at most $(2n-1)^d$ paths and the non-overlapping firewall generated from this FDD has $O(n^d)$ rules in the worst case.

Note that it is asymptotically inefficient to verify a property by directly comparing the property with every rule in the original overlapping firewall because of the conflicts among firewall rules. An overlapping firewall satisfying a property does not require at all that the property is consistent with every rule in the firewall.

Further note that the complexity of directly converting an overlapping firewall of n rules to a non-overlapping firewall using logical operations (without the help of FDDs) is $O(d^n)$ (i.e., exponential over the number of original rules) and therefore is computationally infeasible. This complexity is calculate as follows. Let the first rule in the original firewall be $(F_1 \in S_{11}) \wedge (F_2 \in S_{12}) \wedge \cdots \wedge (F_d \in S_{1d}) \rightarrow dec_1$ and the second rule be $(F_1 \in S_{21}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d}) \rightarrow dec_2$. We can break them into non-overlapping rules by rewriting the second rule as $((F_1 \in S_{21}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d})) - ((F_1 \in S_{11}) \wedge (F_2 \in S_{12}) \wedge \cdots \wedge (F_d \in S_{1d})) \rightarrow dec_2$. Using De Morgan's law, we have $((F_1 \in S_{21}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d})) - ((F_1 \in S_{11}) \wedge (F_2 \in S_{12}) \wedge \cdots \wedge (F_d \in S_{1d})) = ((F_1 \in S_{21}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d})) \wedge \overline{(F_1 \in S_{11}) \wedge (F_2 \in S_{12}) \wedge \cdots \wedge (F_d \in S_{1d})} = ((F_1 \in S_{21}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d})) \wedge ((F_1 \in \overline{S_{11}}) \vee (F_2 \in \overline{S_{12}}) \vee \cdots \vee (F_d \in \overline{S_{1d}})) = ((F_1 \in S_{21} \cap \overline{S_{11}}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d})) \vee ((F_1 \in S_{21}) \wedge (F_2 \in S_{22} \cap \overline{S_{12}}) \wedge \cdots \wedge (F_d \in S_{2d})) \vee \cdots \vee ((F_1 \in S_{21}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d} \cap \overline{S_{1d}}))$. Note that for each $1 \leqslant i \leqslant d$ $S_{2i} \cap \overline{S_{1i}}$ is one interval or the union of two non-adjacent intervals. Thus, the non-atomic rule $((F_1 \in S_{21} \cap \overline{S_{11}}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d})) \vee ((F_1 \in S_{21}) \wedge (F_2 \in S_{22} \cap \overline{S_{12}}) \wedge \cdots \wedge (F_d \in S_{2d})) \vee \cdots \vee ((F_1 \in S_{21}) \wedge (F_2 \in S_{22}) \wedge \cdots \wedge (F_d \in S_{2d} \cap \overline{S_{1d}})) \rightarrow dec_2$ can be decomposed into at most $2d$ atomic rules. Similarly, rewriting the jth rule will end up at most $(2d)^{j-1}$ atomic rules. Therefore, converting an overlapping firewall of n rules to an equivalent non-overlapping firewall directly using logical operations will result in $\frac{(2d)^n-1}{2d-1} = O(d^n)$ non-overlapping rules. Although d is a constant (of typically 4 or 5), n can be very large (in the order of thousands for enterprise firewalls). Of course, $O(d^n)$ is the worse case complexity. The Fang firewall analysis system [32,40,33] used a similar algorithm with the same $O(d^n)$ complexity and had shown that the algorithm performed reasonably well on their rulesets.

Author's personal copy

4.3.3. FDD-based verification algorithm

A more efficient algorithm is to conduct the verification on the firewall decision diagram directly. Verifying whether a firewall decision diagram satisfies a property rule is based on the following theorem.

Theorem 2 (*Firewall verification theorem*). *A firewall decision diagram satisfies a property rule if and only if the property rule does not conflict with any rule defined by a decision path of the firewall decision diagram.*

Proof. First, each path in a firewall decision diagram corresponds to a rule. Second, a firewall decision diagram is equivalent to the non-overlapping firewall where each rule is generated from the decision diagram and the firewall contains all such rules generated from the decision diagram. Because of such one-to-one correspondence, this theorem directly follows from Theorem 1. □

The firewall verification algorithm is in Fig. 4. This algorithm first converts the given firewall to an equivalent firewall decision diagram. Then it begins to traverse the diagram from its root. Let the property rule be $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$. For any edge e in a firewall decision diagram, we use $I(e)$ to denote the label of e. For any node v in a firewall decision diagram, we use $F(v)$ to denote the label of v. Let F_1 be the label of the root. For each outgoing edge e of the root, we compute $I(e) \cap S_1$. If $I(e) \cap S_1 = \emptyset$, we skip edge e and do not traverse the subgraph that e points to. If $I(e) \cap S_1 \neq \emptyset$, then we continue to traverse the subgraph that e points to in a similar fashion. Whenever a terminal node is encountered, we compare the label of the terminal node and $\langle dec \rangle$. If they are different, we then terminate the traversal and report that the given firewall does not satisfy the given property. We use $e.t$ to denote the (target) node that edge e points to.

The FDD-based firewall verification algorithm is essentially a more efficient version of the list-based firewall verification algorithm because all repetitive comparisons are eliminated in the FDD-based algorithm. For example, considering the non-overlapping firewall in Fig. 1 and the equivalent FDD in Fig. 3, verifying the example property $F_1 \in [30, 40] \wedge F_2 \in [80, 90] \rightarrow a$ on the list involves three times comparison of [30,40] and [20,50] while verifying the same property on the FDD involves only one time comparison of [30,40] and [20,50].

The complexity of the FDD-based firewall verification algorithm is $O(kd \log n)$, where k be the total number of paths that a property overlaps on the FDD. This complexity is calculated as follows. As every nonterminal node in a reduced FDD cannot have more than $2n - 1$ outgoing edges, finding the right outgoing edge to traverse takes $O(\log n)$ time using binary search. Thus, the processing time for the query is $O(kd \log n)$.

Note that the complexity of FDD construction is $O(n^d)$ as it has at most $O(n^d)$ paths. Further note that the FDD construction time is a one time cost, not the cost for each property verification.

FDD – based Firewall Verification Algorithm
Input : (1)A firewall
(2)A property rule $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$
Output : *true* if the firewall satisfies the property rule;
false otherwise.
Steps:
1. Convert the given firewall to a firewall decision diagram;
2. **return**(Verify($root, (F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$));

Verify(v, $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$)
1. **if** (v is a terminal node) and ($F(v) = \langle dec \rangle$)
then return true;
if (v is a terminal node) and ($F(v) \neq \langle dec \rangle$)
then return false;
2. **if** (v is a nonterminal node) **then**
/*Let F_i be the label of v*/
for each edge e in $E(v)$ **do**
if ($I(e) \cap S_i \neq \emptyset \wedge$
$\sim Verify(e.t, (F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$))
then return false;
3. **return true**;

Fig. 4. FDD-based firewall verification algorithm.

4.4. Finding "troublesome" rules

In some applications such as firewall troubleshooting, if a firewall fails to satisfy a given property, the firewall administrator also needs to know which rules caused the verification failure. Such information is useful because the administrator can further investigate which rules need to change.

Our firewall verification algorithm can be extended as follows to report the "troublesome" rules when verification fails. First, when we convert a firewall to an equivalent firewall decision diagram, for every decision path (i.e., the path from the root to a terminal node), we need to record the *origin rule* of the path. The origin rule of a path is defined as follows.

Definition 1. For any decision path ρ, assuming the rule defined by ρ is $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$, the first rule $(F_1 \in S'_1) \wedge \cdots \wedge (F_d \in S'_d) \rightarrow \langle dec \rangle$ in the firewall that satisfies the condition $S_1 \subseteq S'_1 \wedge \cdots S_d \subseteq S'_d$ is called the *origin rule* of the decision path ρ.

To convert a firewall of a sequence of rules $\langle r_1, \ldots, r_n \rangle$ to an equivalent firewall decision diagram, we first construct a partial firewall decision diagram from r_1, which contains only one decision path that defines r_1. The origin rule of this path is basically r_1. Second, we append rule r_2 to the partial firewall decision diagram making the resulting partial firewall decision diagram equivalent to partial firewall $\langle r_1, r_2 \rangle$ based on first-match semantics. In appending rule r_2, the origin rule of each new decision path created in the process is r_2. Similarly, we keep appending rules $r_3, \ldots, r_n$. In this process, we keep the firewall decision diagram in a tree shape, where for every terminal node there is one and only one decision path that contains the node. Therefore, we can store the identifier of the origin rule of each decision path at the terminal node of the path. Note that the same firewall rule can appear on multiple terminals.

The pseudocode of the FDD construction algorithm that also outputs the origin rule of each decision path is in Fig. 5.

Author's personal copy

Construction Algorithm With Origin Rules Marked
Input : A firewall f of a sequence of rules $\langle r_1, \cdots, r_n \rangle$
Output : (1) An FDD f' such that f and f' are equivalent;
(2) The origin rule of each decision path is computed
Steps:
1. build a decision path with root v from rule r_1; mark r_1 as the origin rule of this path;
2. **for** $i := 2$ **to** n **do** APPEND(v, i, r_i);
End
APPEND(v, i, $(F_m \in S_m) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$)
/*$F(v) = F_m$ and $E(v) = \{e_1, \cdots, e_k\}$*/
1. **if** ($S_m - (I(e_1) \cup \cdots \cup I(e_k))) \neq \emptyset$ **then**
(a) add an outgoing edge e_{k+1} with label $S_m - (I(e_1) \cup \cdots \cup I(e_k))$ to v;
(b) build a decision path from rule $(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$, and make e_{k+1} point to the first node in this path;
(c) mark r_i as the origin rule of this path;
2. **if** $m < d$ **then**
for $j := 1$ **to** k **do**
if $I(e_j) \subseteq S_m$ **then**
APPEND($e_j.t$,
i,
$(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$);
else if $I(e_j) \cap S_m \neq \emptyset$ **then**
(a) add one outgoing edge e to v, and label e with $I(e_j) \cap S_m$;
(b) make a copy of the subgraph rooted at $e_j.t$, and make e points to the root of the copy;
(a) replace the label of e_j by $I(e_j) - S_m$;
(d) APPEND($e.t$,
i,
$(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$);

Fig. 5. Firewall decision diagram construction algorithm with origin rules marked.

Second, we need to modify the firewall verification algorithm as follows. For every decision path in the firewall decision diagram, we need to compare the rule defined by the decision path and the given property rule; if they conflict, then the verification fails and the algorithm reports the origin of the decision path as one of the "troublesome" rules.

4.5. Optimizing firewall verification algorithm

To further improve the efficiency of our firewall verification algorithm, after we convert a firewall to an equivalent FDD, we need to reduce the size of the FDD.

A brute force deep comparison algorithm for FDD reduction was proposed in [16]. A more efficient FDD reduction algorithm that processes the nodes level by level from the terminal nodes to the root node using signatures to speed up comparisons is proposed in [30]. Below we briefly introduce this signature-based FDD reduction algorithm. Starting from the bottom level, at each level, we compute a signature for each node at that level. For a terminal node v, set v's signature to be its label. For a nonterminal node v, suppose v has k children $v_1, v_2, \ldots, v_k$, in increasing order of signature ($Sig(v_i) < Sig(v_{i+1})$ for $1 \leqslant i \leqslant k-1$), and the edge between v and its child v_i is labeled with E_i, a sequence of non-overlapping prefixes in increasing order. Set the signature of node v as follows: $Sig(v) = h(Sig(v_1), E_1, \ldots, Sig(v_k), E_k)$, where h is a one-way and collision resistant hash function such as MD5 [36] and SHA-1 [6]. For any such hash function h, given two different inputs x and y, the probability of $h(x) = h(y)$ is extremely small. After we have assigned signatures to all nodes at a given level, we search for isomorphic subgraphs as follows. For every pair of nodes v_i and $v_j (1 \leqslant i \neq j \leqslant k)$ at this level, if $Sig(v_i) \neq Sig(v_j)$, then we can conclude that v_i and v_j are not isomorphic; otherwise, we explicitly determine if v_i and v_j are isomorphic. If v_i and v_j are isomorphic, we delete node v_j and its outgoing edges, and redirect all the edges that point to v_j to point to v_i. Further, we eliminate double edges between node v_i and its parents.

5. Discussion

5.1. Prefix and intervals

Real-life firewalls usually check five packet fields: source IP address, destination IP address, protocol type, source port number, and destination port number. Of these five fields, source and destination IP addresses are usually represented using prefix formats, protocol type is typically a concrete value, and the fields of source and destination ports are represented by integer intervals. Note that prefix formats and interval formats are interchangeable. For example, IP prefix 192.168.0.0/16 can be converted to the interval from 192.168.0.0 to 192.168.255.255, where an IP address can be regarded as a 32-bit integer. As another example, the interval [2,8] can be converted to 3 prefixes: 001*, 01*, 1000.

In doing firewall property verification, we first convert the source and destination IP addresses from prefix formats to integer intervals. Note that every prefix can be converted to only one integer interval. Second, we run our firewall verification algorithm on the converted rules.

5.2. Making corrections

When a firewall fails to satisfy a property, there are two possibilities. One is that the property is specified incorrectly. In this case, the "troublesome" rules reported by our firewall verification algorithm could give the firewall administrator some insights on how to further revise the specification of the property. The other possibility is that the firewall was designed incorrectly. In this case, the firewall administrator has two ways to correct the firewall. The first way is to examine the "troublesome" rules reported by our firewall verification algorithm, and make corrections on these rules. The second way is to simply add the property rules on to the beginning of the firewall. This will guarantee that the property rules are satisfied by the revised firewall.

6. Experimental results

We implemented our firewall verification and troubleshooting algorithm using Visual Basic on Microsoft .Net framework 2.0. Given any firewall and any property, our algorithm verifies whether the firewall satisfies the prop-

Author's personal copy

erty; if not, our algorithm outputs all the troublesome rules. To evaluate the performance of this algorithm, first, we ran our algorithm on two real-life firewalls. Second, we stress tested our algorithms on a large number of synthetic firewalls. These experiments were carried out on a desktop PC running Windows XP with 1G memory and a single 2.2 GHz AMD Opteron 148 processor. In both cases, the experimental results show that our FDD-based firewall verification algorithm performs and scales very well.

We obtained two real-life firewalls for this study. One firewall, with 129 rules, is from a university. The other, with 661 rules, is from a private company. Table 3 shows the performance of our algorithms on these two firewalls.

Firewall configurations are considered confidential due to security concerns. To further evaluate the performance of our algorithms on large firewalls, we run our algorithms on synthetic firewalls of large sizes. Every predicate of a rule in our synthetic packet classifiers has five fields: source IP, destination IP, source port, destination port, and protocol. We based our generation method upon Rovniagin and Wool's [37] model of synthetic rules.

In our experiments, we generate 100 firewalls of each fixed size. For each firewall, we run our algorithm 100 times. In each run of the algorithm we generate the property rule randomly. Fig. 6 shows the average running time for converting a firewall to a firewall decision diagram. Note that the time for constructing a firewall decision is a one time cost. Once a firewall decision diagram is constructed from a firewall, we can use it to verify multiple properties.

Fig. 7 shows the average running time of both the list-based firewall verification algorithm and the FDD-based firewall verification algorithm. The experimental results show that the FDD-based firewall verification algorithm is 4 orders of magnitude faster than the list-based firewall verification algorithm.

Table 3
Performance of firewall verification algorithms on two real-life firewalls.

	# Rules	FDD Construction	Verification
Firewall I	129	15 ms	0.03 ms
Firewall II	661	812 ms	0.374 ms

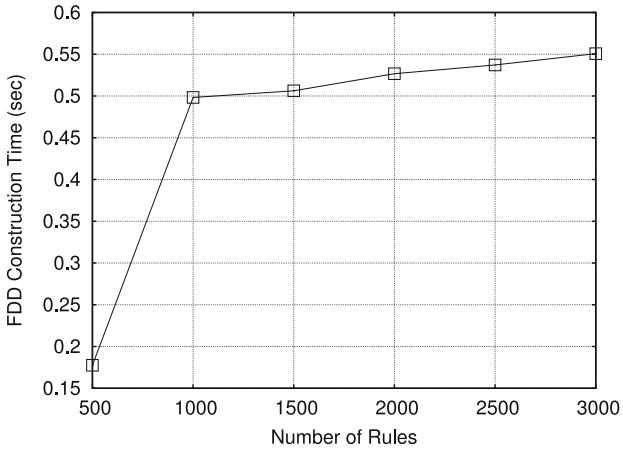

Fig. 6. Performance of firewall decision diagram construction.

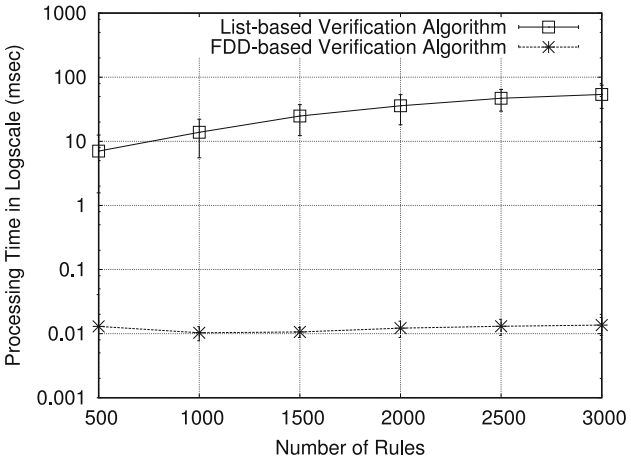

Fig. 7. Performance of firewall verification algorithm.

7. Related work

Previous work that is closest to ours is firewall testing [4,7,20,23,21,38,31]. In these testing methods, test cases, i.e., packets, are first generated based on a given firewall and the properties that the firewall needs to satisfy; then the generated packets are injected into the firewall to check whether the packet gets accepted or discarded. The focus of this paper is firewall verification rather than firewall testing. On one hand, our firewall verification technique outperforms these firewall testing techniques because we do not need to inject packets into a firewall or inspect the outcome of the firewall, which are often time and labor consuming. Although packet injection and inspection can be automated, such automation usually requires to take the firewall device offline, which is often not affordable. On the other hand, these firewall testing schemes are capable of detecting errors in firewall software or hardware, while our firewall verification technique can only detect errors in firewall policies. In practice, firewall policy verification and firewall testing are complementary and can be used together for better firewall security.

In [32,40,33], a query based firewall analysis system named *Fang* was presented. In Fang, a firewall query is described by a triple (a set of source addresses, a set of destination addresses, a set of services), where each service is a tuple (protocol type, source ports, destination ports). The meaning of such a query is "which IP addresses in the set of source addresses can use which services in the set of services to which IP addresses in the set of destination addresses". We make three contributions in comparison with Fang. First, although we can verify some accept properties using Fang by converting the properties to queries, we cannot verify discard properties using Fang. Second, we can identify troublesome rules while Fang cannot. Identifying troublesome rules is critical in firewall policy verification. Third, for verifying firewall policies, our algorithm is more efficient than Fang. While Fang processes queries by linearly scanning the rules in a firewall, we use the tree representation of firewall policies for verifying property rules. On the other hand, Fang can query

Author's personal copy

multiple firewalls, taking their network topology into account.

Some firewall analysis methods have been proposed in [25,26,13,28,19,9,2,8,18,34]. In [25], Liu presented algorithms for performing the change impact analysis of firewall policies.

In [28], Liu and Gouda studied the redundancy issues in firewall policies and gave an algorithm for removing all the redundant rules in a firewall policy. In [19], some ad-hoc "what if" questions that are similar to firewall queries were discussed. However, no algorithm was presented for processing the proposed "what if" questions. In [9], expert systems were proposed to analyze firewall rules. Clearly, building an expert system just for analyzing a firewall is overwrought and impractical. Detecting potential firewall policy errors by conflict detection was discussed in [2,8,18,34]. Similar to conflict detection, some anomalies are defined and techniques for detecting anomalies are presented in [1,44]. Examining each conflict or anomaly is helpful in reducing potential firewall policy errors; however, the number of conflicts or anomalies in a firewall is typically large, and the manual checking of each conflict or anomaly is unreliable because the meaning of each rule depends on the current order of the rules in the firewall, which may be incorrect.

Some firewall design methods have been proposed in [27,29,3,17,14,16,15]. These works aim at creating firewall rules, while we aim at analyzing firewall rules. Gouda and Liu proposed to use decision diagrams for designing firewalls in [14,16]. In [27,29], Liu and Gouda applied the technique of design diversity to firewall design. Gouda and Liu also proposed a model for specifying stateful firewall policies [15]. Guttman proposed a Lisp-like language for specifying high-level packet filtering policies in [17]. Bartal et al. proposed a UML-like language for specifying global filtering policies in [3].

Design of high-performance ATM firewalls was discussed in [43,42] with focus on firewall architectures. Firewall vulnerabilities were discussed and classified in [24,11]. However, the focus of [24,11] are the vulnerabilities of the packet filtering software and the supporting hardware part of a firewall, not the policy of a firewall.

There are some tools currently available for network vulnerability testing, such as Satan [10,12], Nessus [35], and MACE [39]. These vulnerability testing tools scan a private network based on the current publicly known attacks, rather than the requirement specification of a firewall. Although these tools can possibly catch errors that allow illegitimate access to the private network, they cannot find the errors that disable legitimate communication between the private network and the outside Internet. Firewall policy testing was studied in [22].

8. Conclusions

This paper presents a method for formally verifying firewall policies and finding troublesome rules. A tool with such functionalities is extremely useful in many ways. For example, it can be used in firewall debugging and troubleshooting. It also can be used iteratively in the process of designing a firewall. A firewall administrator can use this tool to unambiguously demonstrate to their manager that the firewall satisfies the organization's security policies.

We have implemented our firewall verification and troubleshooting algorithm and evaluated it on both real-life firewall policies and synthetic firewall policies of large sizes. The experimental results show that the performance of our algorithm is quite attractive. We believe that our algorithm can be practically used in designing and maintaining firewall policies, which will in turn help to eliminate the errors in firewall policies.

Acknowledgement

The author would like to thank the anonymous reviewers for their constructive comments and suggestions on improving the presentation of this work. This work is supported in part by the National Science Foundation under Grant No. CNS-0716407. The author also would like to thank Ke Shen for implementing the algorithm and conducting experiments.

References

[1] E. Al-Shaer, H. Hamed, Discovery of policy anomalies in distributed firewalls, in: IEEE INFOCOM'04, March 2004.
[2] F. Baboescu, G. Varghese, Fast and scalable conflict detection for packet classifiers, in: Proceedings of the 10th IEEE International Conference on Network Protocols, 2002.
[3] Y. Bartal, A.J. Mayer, K. Nissim, A. Wool, Firmato: A novel firewall management toolkit, in: Proceedings of the IEEE Symposium on Security and Privacy, 1999, pp. 17–31.
[4] CERT. Test the firewall system <http://www.cert.org/security-improvement/practices/p060.html>.
[5] M. Christiansen, E. Fleury, Using interval decision diagrams for packet filtering. Technical Report, RS-02-43, University of Aarhus, Denmark, 2002.
[6] D. Eastlake, P. Jones, Us secure hash algorithm 1 (sha1), RFC 3174, 2001.
[7] A. El-Atawy, K. Ibrahim, H. Hamed, E. Al-Shaer, Policy segmentation for intelligent firewall testing, in: Proceedings of the First Workshop on Secure Network Protocols, November 2005.
[8] D. Eppstein, S. Muthukrishnan, Internet packet filter management and rectangle geometry, in: Symposium on Discrete Algorithms, 2001, pp. 827–835.
[9] P. Eronen, J. Zitting, An expert system for analyzing firewall rules, in: Proceedings of the Sixth Nordic Workshop on Secure IT Systems (NordSec 2001), 2001, pp. 100–107.
[10] D. Farmer, W. Venema, Improving the security of your site by breaking into it. <http://www.alw.nih.gov/Security/Docs/admin-guide-to-cracking.101.html>, 1993.
[11] M. Frantzen, F. Kerschbaum, E. Schultz, S. Fahmy, A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals, Computers and Security 20 (3) (2001) 263–270.
[12] M. Freiss, Protecting Networks with SATAN, O'Reilly and Associates, Inc., 1998.
[13] M. Gouda, A.X. Liu, M. Jafry, Verification of distributed firewalls, in: Proceedings of the IEEE Global Communications Conference (GLOBECOM), 2008.
[14] M.G. Gouda, A.X. Liu, Firewall design: consistency, completeness and compactness, in: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04), 2004, pp. 320–327.
[15] M.G. Gouda, A.X. Liu, A model of stateful firewalls and its properties, in: Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN-05), June 2005, pp. 320–327.
[16] M.G. Gouda, A.X. Liu, Structured firewall design, Computer Networks Journal (Elsevier) 51 (4) (2007) 1106–1120.
[17] J.D. Guttman, Filtering postures: Local enforcement for global policies, in: Proceedings of IEEE Symposium on Security and Privacy, 1997, pp. 120–129.

Author's personal copy

[18] A. Hari, S. Suri, G.M. Parulkar, Detecting and resolving packet filter conflicts, in: Proceedings of IEEE INFOCOM, 2000, pp. 1203–1212.
[19] S. Hazelhurst, A. Attar, R. Sinnappan, Algorithms for improving the dependability of firewall and filter rule lists, in: Proceedings of the International Conference on Dependable Systems and Networks (DSN'00), 2000, pp. 576–585.
[20] D. Hoffman, D. Prabhakar, P. Strooper, Testing iptables, in: Proceedings of the 2003 Conference of IBM Centre for Advanced Studies, 2003, pp. 80–91.
[21] D. Hoffman, K. Yoo, Blowtorch: a framework for firewall test automation, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 96–103.
[22] J. Hwang, T. Xie, F. Chen, A.X. Liu, Systematic structural testing of firewall policies, in: Proceedings 27th IEEE International Symposium on Reliable Distributed Systems (SRDS), 2008.
[23] J. Jürjens, G. Wimmel, Specification-based testing of firewalls, in: A. Ershov (Ed.), Proceedings of the Fourth International Conference Perspectives of System Informatics (PSI'01), LNCS, Springer.
[24] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, M. Frantzen, Analysis of vulnerabilities in internet firewalls, Computers and Security 22 (3) (2003) 214–232.
[25] A.X. Liu, Change-impact analysis of firewall policies, in: J. Biskup, J. Lopez (Ed.), Proceedings of the 12th European Symposium Research Computer Security (ESORICS), LNCS 4734, Springer-Verlag, September 2007, p. 155C170.
[26] A.X. Liu, Firewall policy verification and troubleshooting, in: Proceedings IEEE International Conference on Communications (ICC), May 2008.
[27] A.X. Liu, M.G. Gouda, Diverse firewall design, in: Proceedings of the International Conference on Dependable Systems and Networks (DSN), June 2004, pp. 595–604.
[28] A.X. Liu, M.G. Gouda, Complete redundancy detection in firewalls, in: Proceedings 19th Annual IFIP Conference on Data and Applications Security, LNCS 3654, August 2005, pp. 196–209.
[29] A.X. Liu, M.G. Gouda, Diverse firewall design, IEEE Transactions on Parallel and Distributed Systems (TPDS) 19 (8) (2008).
[30] A.X. Liu, C.R. Meiners, E. Torng, Tcam razor: a systematic approach towards minimizing packet classifiers in tcams, in: IEEE/ACM Transactions on Networking, accepted for publication.
[31] M.R. Lyu, L.K.Y. Lau, Algorithms for improving the dependability of firewall and filter rule lists, in: Proceedings of the 24th International Conference on Computer Systems and Applications (COMPSAC'2000), October 2000, pp. 116–121.
[32] A. Mayer, A. Wool, E. Ziskind, Fang: a firewall analysis engine, in: Proceedings of IEEE Symposium on Security and Privacy, 2000, pp. 177–187.
[33] A. Mayer, A. Wool, E. Ziskind, Offline firewall analysis, International Journal of Information Security 5 (3) (2005) 125–144.
[34] J.D. Moffett, M.S. Sloman, Policy conflict analysis in distributed system management, Journal of Organizational Computing 4 (1) (1994) 1–22.
[35] Nessus <http://www.nessus.org/>, March 2004.
[36] R. Rivest, The md5 message-digest algorithm. RFC 1321, 1992.
[37] D. Rovniagin, A. Wool, The geometric efficient matching algorithm for firewalls, in: Proceedings 23rd IEEE Convention of Electrical and Electronics Engineers in Israel (IEEEI). Technical Report available at <http://www.eng.tau.ac.il/~yash/ees2003-6.ps>, 2004, pp. 153–156.
[38] D. Senn, D. Basin, G. Caronni, Firewall conformance testing, Proceedings of the Testcom (Testing of Communicating Systems) (2005).
[39] J. Sommers, V. Yegneswaran, P. Barford, A framework for malicious workload generation, in: Proceedings of the Fourth ACM SIGCOMM Conference on Internet Measurement, ACM, New York, NY, USA, 2004, pp. 82–87.
[40] A. Wool, Architecting the lumeta firewall analyzer, in: Proceedings of the 10th USENIX Security Symposium, August 2001, pp. 85–97.
[41] A. Wool, A quantitative study of firewall configuration errors, IEEE Computer 37 (6) (2004) 62–67.
[42] J. Xu, M. Singhal, Design and evaluation of a high-performance atm firewall switch and its applications, IEEE Journal on Selected Areas in Communications (JSAC) 17 (6) (1999) 1190–1200.
[43] J. Xu, M. Singhal, Design of a high-performance atm firewall, ACM Transactions on Information and System Security 2 (3) (1999) 269–294.
[44] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, P. Mohapatra, Fireman: a toolkit for firewall modeling and analysis, in: IEEE Symposium on Security and Privacy, May 2006.

Alex X. Liu received his Ph.D. degree in computer science from the University of Texas at Austin in 2006. He is currently an assistant professor in the Department of Computer Science and Engineering at Michigan State University. He received the 2004 IEEE & IFIP William C. Carter Award. His research interests focus on networking, security, and dependable systems.