Diverse Firewall Design

Alex X. Liu, *Member*, *IEEE*, and Mohamed G. Gouda, *Member*, *IEEE*

Abstract—Firewalls are the mainstay of enterprise security and the most widely adopted technology for protecting private networks. An error in a firewall policy either creates security holes that will allow malicious traffic to sneak into a private network or blocks legitimate traffic and disrupts normal business processes, which, in turn, could lead to irreparable, if not tragic, consequences. It has been observed that most firewall policies on the Internet are poorly designed and have many errors. Therefore, how one can design firewall policies correctly is an important issue. In this paper, we propose the method of diverse firewall design, which consists of three phases: a design phase, a comparison phase, and a resolution phase. In the design phase, the same requirement specification of a firewall policy is given to multiple teams who proceed independently to design different versions of the firewall policy. In the comparison phase, the resulting multiple versions are compared with each other to detect all functional discrepancies between them. In the resolution phase, all discrepancies are resolved, and a firewall that is agreed upon by all teams is generated. The major technical challenge in the method of diverse firewall design is how one can discover all functional discrepancies between two given firewall policies. We present a series of three efficient algorithms for solving this problem: a construction algorithm, a shaping algorithm, and a comparison algorithm. The algorithms for discovering all functional discrepancies between two given firewall policies can be used to perform firewall policy change impact analysis as well. Firewall policies often need to be changed, as networks evolve, and new threats emerge. Many firewall policy errors are caused by the unintended side effects of policy changes. Our algorithms can be used directly to compute the impact of firewall policy changes by computing the functional discrepancies between the policy before changes and the policy after changes.

Index Terms—Firewall policy, policy design, design diversity, change impact analysis, network security.

1 Introduction

Firewalls are crucial elements in network security, and they have been widely deployed to secure private networks in businesses and institutions. A firewall is a security guard 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. A packet can be viewed as a tuple with a finite number of fields such as source IP address, destination IP address, source port number, destination port number, and protocol type. By examining the values of these fields for incoming and outgoing packets, a firewall accepts legitimate packets and discards illegitimate ones according to its "policy," that is, "configuration."

A firewall policy consists of a sequence (that is, an ordered list) of rules, where each rule is of the form $\langle predicate \rangle \rightarrow \langle decision \rangle$. The $\langle predicate \rangle$ 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 $\langle decision \rangle$ 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 (that is, the highest priority) rule that the packet matches.

- *A.X. Liu is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1226. E-mail: alexliu@cse.msu.edu.*
- *M.G. Gouda is with the Department of Computer Sciences, The University of Texas at Austin, 1 University Station (C0500), Austin, TX 78712-0233. E-mail: gouda@cs.utexas.edu.*

Manuscript received 12 Apr. 2007; revised 9 Aug. 2007; accepted 22 Oct. 2007; published online 1 Nov. 2007.
Recommended for acceptance by T. Abdelzaher.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-04-0110.
Digital Object Identifier no. 10.1109/TPDS.2007.70802.

1.1 Motivation

Although a firewall policy is a mere sequence of rules, correctly designing one is, by no means, easy. The rules in a firewall policy are logically entangled because of conflicts among rules and the resulting order sensitivity [26]. Ordering the rules correctly in a firewall is critical yet difficult. The implication of any rule in a firewall cannot be understood correctly without examining all the rules listed above that rule. Furthermore, 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. One can imagine the complexity of the logic underlying so many conflicting rules.

An error in a firewall policy, that is, a wrong definition of being legitimate or illegitimate for some packets, means that the firewall either accepts some malicious packets, which consequently creates security holes in the firewall, or discards some legitimate packets, which consequently disrupts normal business. Either case could cause irreparable, if not tragic, consequences. Given the importance of firewalls, such errors are not acceptable. Unfortunately, it has been observed that most firewalls on the Internet are poorly designed and have many errors in their policies [26]. Therefore, how one can design firewall policies correctly is an important issue.

Since the correctness of a firewall policy is the focus of this paper, we assume that a firewall is correct if and only if its policy is correct and a firewall policy is correct if and only if it satisfies its given requirement specification, which is usually written in a natural language. In the rest of this paper, we use the term "firewall" to mean "firewall policy" or "firewall configuration," unless otherwise specified.

We categorize firewall errors into specification-induced errors and design-induced errors. Specification-induced

errors are caused by the inherent ambiguities of informal requirement specifications, especially if the requirement specification is written in a natural language. Design-induced errors are caused by the technical incapacity of individual firewall designers. Different designers may have different understandings of the same informal requirement specification, and different designers may exhibit different technical strengths and weaknesses. Note that in this paper, we assume that the given requirement specification of a firewall is informal. Automatically converting a formal firewall specification to a deployable firewall policy has been addressed in [12]. However, the formal specification of a firewall policy is still difficult to specify correctly. The above observations motivate our method of diverse firewall design.

1.2 Our Solution

Our diverse firewall design method has the following phases:

1. *Design phase.* In this phase, the same requirement specification of a firewall is given to multiple teams who proceed independently to design different versions of the firewall. In the industry, firewalls are typically designed and maintained by a group of people rather than just one person. To apply the method of diverse firewall design, we can divide one group into several teams.
2. *Comparison phase.* In this phase, the resulting multiple versions are compared with each other to determine all functional discrepancies among them. The functional discrepancies need to be presented in human readable format in order to be used in the next step.
3. *Resolution phase.* In this phase, first, every discrepancy is discussed and resolved by all teams. Second, a firewall that is unanimously agreed upon by all teams is generated.

The major technical challenge in the method of diverse firewall design is how one can discover all functional discrepancies between two given firewalls in human readable format. Our solution to this problem consists of a series of three efficient algorithms for solving this problem: a construction algorithm, a shaping algorithm, and a comparison algorithm.

After all functional discrepancies are computed, the teams need to discuss the correct decision for each discrepancy. After all discrepancies are resolved, the technical question that we need to answer is: How do we generate the final firewall that reflects the resolved functional discrepancies? We present two methods for this purpose in Section 6.

1.3 Other Applications: Firewall Change Impact Analysis

The algorithms presented in this paper can be used in other applications as well, such as firewall change impact analysis. Firewall policies are always subject to change due to a variety of reasons. Making policy changes is a major routine task for firewall administrators. For example, new network threats such as worms and viruses may emerge. To protect a private network from new attacks, firewall policies need to be changed accordingly. Modern organizations also continually transform their network infrastructure to maintain their competitive edge by adding new servers, installing new software and services, expanding connectivity, etc. In accordance with network changes, firewall policies need to be changed as well to provide necessary protection.

Unfortunately, making changes is a major source of firewall policy errors. Making correct firewall policy changes is remarkably difficult due to the interleaving nature of firewall rules. For example, when a firewall administrator inserts a new rule to a firewall policy, the meaning of the rules listed under this rule could be incorrectly changed, without the administrator noticing. Furthermore, firewall policy changes are made by human administrators, and it is common that human administrators make mistakes. It has been shown that administrator errors are the largest cause of failure for Internet services, and policy errors are the largest category of administrator errors [21].

The algorithms for discovering all functional discrepancies between two given firewalls can be directly used to perform firewall change impact analysis. The impact of the changes can literally be defined as the functional discrepancies between the firewall before changes and the firewall after changes.

1.4 Relationship to Prior Art

Some firewall design and analysis methods have been proposed previously [1], [5], [11], [12], [15], [19], [20], [29]. However, none of them has ever explored design diversity. Furthermore, none of them has ever tackled the problem of change impact analysis for firewall policies. The proposed diverse firewall design method is complementary to the previous work, because these methods can assist each individual team to design and analyze their firewall in the design phase before cross comparison.

Note that the scope of this paper is on firewalls and not Intrusion Detection Systems/Prevention Systems (IDSs/IPSs). Although the distinction between IDSs/IPSs and firewalls is blurry sometimes in the commercial world, IDSs/IPSs fundamentally differ from firewalls in that IDSs/IPSs check packet payloads, whereas firewalls do not.

1.5 Key Contributions

We make four key contributions in this paper:

1. We propose the method of diverse firewall design. This paper represents the first effort to apply the well-known principle of diverse design to firewalls.
2. We present a method that can compare two given firewalls and output all functional discrepancies between them in human readable format. This is the first method created for this purpose.
3. We present a method to compute firewall change impacts by computing all functional discrepancies between the firewalls before and after changes. This is the first method for doing firewall change impact analysis.
4. We implemented our algorithms in Java, and we evaluated their performance on both real-life and synthetic firewalls of large sizes. The experimental results show that our algorithms only use a few seconds to compare two different firewalls, where each firewall has up to 3,000 rules.

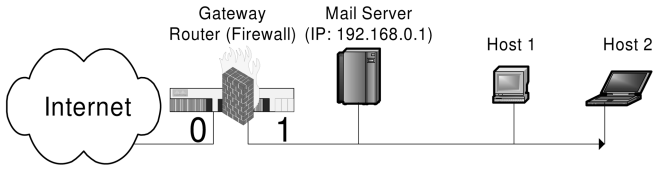

Fig. 1. A firewall.

The rest of this paper is organized as follows: We start with an overview of our diverse firewall design method in Section 2. In Sections 3, 4, and 5, we present a series of three algorithms for discovering all functional discrepancies between two firewalls. In Section 6, we discuss how we can generate a firewall that is agreed upon by all teams after all discrepancies are resolved. We discuss some further issues in Section 7. In Section 8, we present the experimental results that show the effectiveness and efficiency of our diverse firewall design method. Our conclusions are given in Section 10.

2 Overview

In this section, we present an overview of our diverse firewall design method by using an illustrative example, which will be used throughout this paper.

In our example, for simplicity, we assume that a firewall maps every packet to either decision: accept or discard. Most firewall software supports more than two decisions such as accept, accept and log, discard, and discard and log. Our diverse firewall design method can support any number of decisions.

2.1 Design Multiple Firewalls

Consider the simple network in Fig. 1. This network has a gateway router with two interfaces: interface 0, which connects the gateway router to the outside Internet, and interface 1, which connects the gateway router to the inside local network. The firewall for this local network resides in the gateway router.

Suppose that the requirement specification for this firewall is given as follows: *The mail server with IP address 192.168.0.1 can receive e-mail packets. The packets from an outside malicious domain 224.168.0.0/16 should be blocked. Other packets should be accepted and allowed to proceed.*

Suppose that we give this specification to two teams —Team A and Team B—which design the firewalls, as shown in Tables 1 and 2, respectively.

2.2 Compare Multiple Firewalls

Next, we briefly show our method for computing the functional discrepancies between two given firewalls. For example, given the two firewalls in Tables 1 and 2, our method produces all the functional discrepancies, as shown in Table 3.

The core data structure used in this paper for comparing multiple firewalls is *Firewall Decision Diagrams* (FDDs). FDDs were introduced in [10] as a notation for specifying firewalls. An FDD with a decision set DS and over fields $F_1, \cdots, F_d$ is an acyclic and directed graph that has the following 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

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

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 (that is, $I(e) \subseteq D(F(u))$).
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 conditions:
   - *Consistency.* $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$.
   - *Completeness.* $\bigcup_{e \in E(v)} I(e) = D(F(v))$.

A decision path in an FDD f is represented by $v_1 e_1 \cdots v_k e_k v_{k+1}$, where v_1 is the root, v_{k+1} is a terminal

TABLE 1
Firewall Designed by Team A

Rule #	Interface	Source IP	Destination IP	Destination Port	Protocol	Decision
r_1	0	*	192.168.0.1	25	TCP	accept
r_2	0	224.168.0.0/16	*	*	*	discard
r_3	*	*	*	*	*	accept

TABLE 2
Firewall Designed by Team B

Rule #	Interface	Source IP	Destination IP	Destination Port	Protocol	Decision
r'_1	0	224.168.0.0/16	*	*	*	discard
r'_2	0	*	192.168.0.1	25	TCP	accept
r'_3	0	*	192.168.0.1	*	*	discard
r'_4	*	*	*	*	*	accept

TABLE 3
Functional Discrepancies between the Two Firewalls Designed by Teams A and B

Discrepancy #	Interface	Source IP	Destination IP	Destination Port	Protocol	Team A Decision	Team B Decision
1	0	224.168.0.0/16	192.168.0.1	25	TCP	accept	discard
2	0	!224.168.0.0/16	192.168.0.1	25	!TCP	accept	discard
3	0	!224.168.0.0/16	192.168.0.1	!25	*	accept	discard

node, and each e_i is a directed edge from node v_i to node v_{i+1}. A decision path $(v_1 e_1 \cdots v_k e_k v_{k+1})$ in an FDD defines the following rule:

$$F_1 \in S_1 \wedge \cdots \wedge F_n \in S_n \rightarrow F(v_{k+1}),$$

where

$$S_i = \begin{cases} I(e_j), & \text{if there is a node } v_j \text{ in the decision path that is labeled with field } F_i, \\ D(F_i), & \text{if no node in the decision path is labeled with field } F_i. \end{cases}$$

For an FDD f, we use $f.rules$ to denote the set of all rules that are defined by all the decision paths of f. For any packet p, there is only one rule in $f.rules$ that p matches because of the consistency and completeness properties of an FDD.

Our method for computing the functional discrepancies between two given firewalls consists of the following steps:

Step 1: conversion. In this step, we convert each firewall to an equivalent FDD. Figs. 2 and 3 show the two FDDs that are converted from the two firewalls in Tables 1 and 2, respectively. Note that the example FDDs used in this paper are presented as trees for ease of understanding. The algorithm for constructing an equivalent FDD from a sequence of rules is presented in Section 3.

In this example, we suppose that each packet has the following fields:

1. interface,
2. source IP address,
3. destination IP address,
4. destination port, and
5. protocol type.

For ease of presentation, we assume that each packet has a field called "interface," whose value is the identification of the network interface on which a packet arrives. The shorthand for the five packet fields is listed in the following table, and for simplicity, we assume that the protocol type value in a packet is either 0 (TCP) or 1 (UDP):

shorthand	meaning	domain
I	Interface	$[0, 1]$
S	Source IP address	$[0, 2^{32})$
D	Destination IP address	$[0, 2^{32})$
N	Destination Port	$[0, 2^{16})$
P	Protocol Type	$[0, 1]$

In our examples, we also use the following shorthand. Note that α denotes the integer formed by 4 bytes of the IP address 224.168.0.0. This applies similarly for β and γ:

shorthand	meaning
a	accept
d	discard
α	224.168.0.0
β	224.168.255.255
γ	192.168.0.1

Step 2: shaping. In this step, we transform each FDD into another FDD without changing its semantics such that the two resulting FDDs are semi-isomorphic. Two FDDs are semi-isomorphic if and only if they are exactly the same, except for the labels of their terminal nodes. Figs. 4 and 5 show the two semi-isomorphic FDDs converted from the FDDs in Figs. 2 and 3, respectively. The algorithm for making two FDDs semi-isomorphic without changing their semantics is presented in Section 4.

Step 3: comparison. In this step, we compare the two semi-isomorphic FDDs in Figs. 4 and 5 for functional discrepancies. Table 3 shows all the functional discrepancies between the two semi-isomorphic FDDs in Figs. 4 and 5, which are also the functional discrepancies between the two firewalls in Tables 1 and 2. The algorithm for

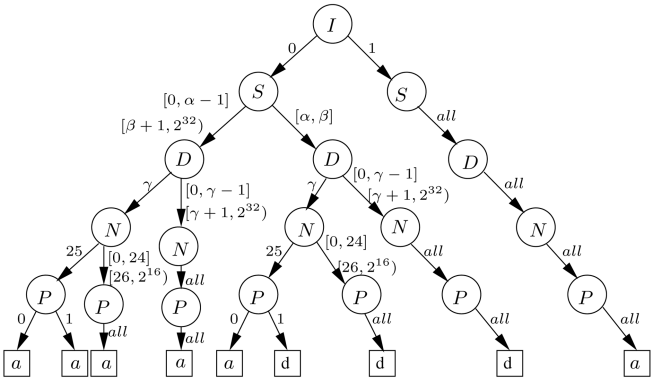

Fig. 2. The FDD constructed from the firewall designed by Team A in Table 1.

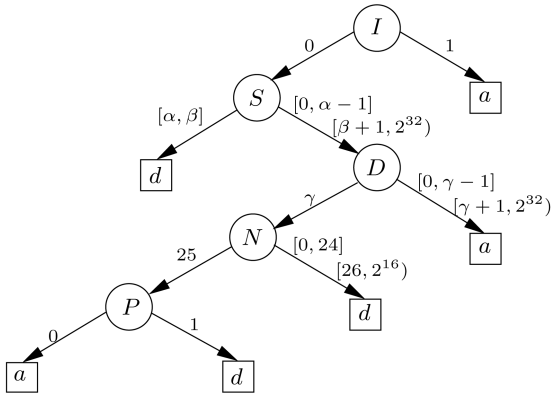

Fig. 3. The FDD constructed from the firewall designed by Team B in Table 2.

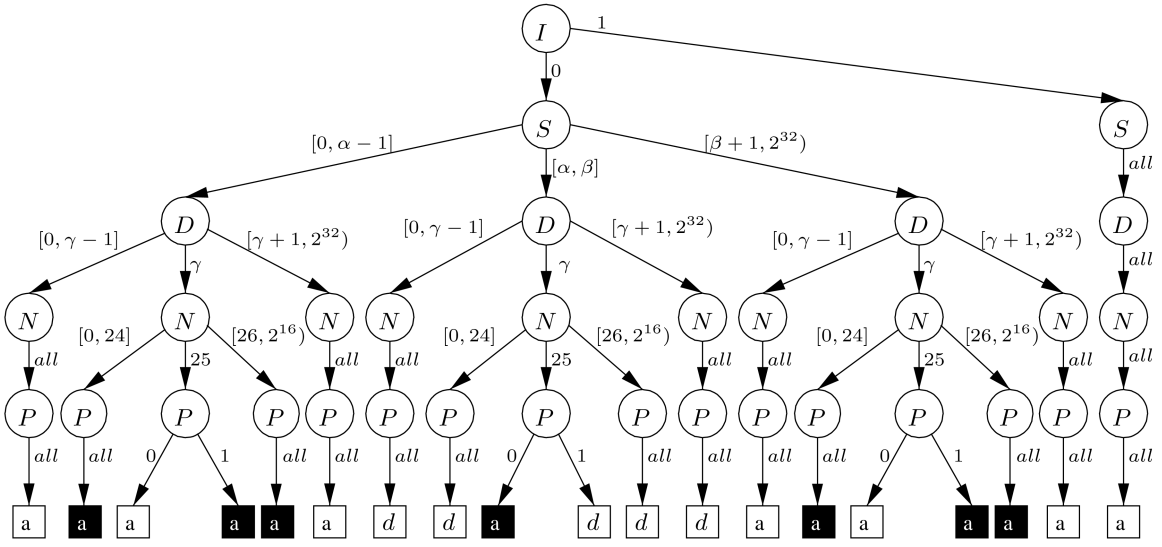

Fig. 4. The FDD transformed from the one in Fig. 2.

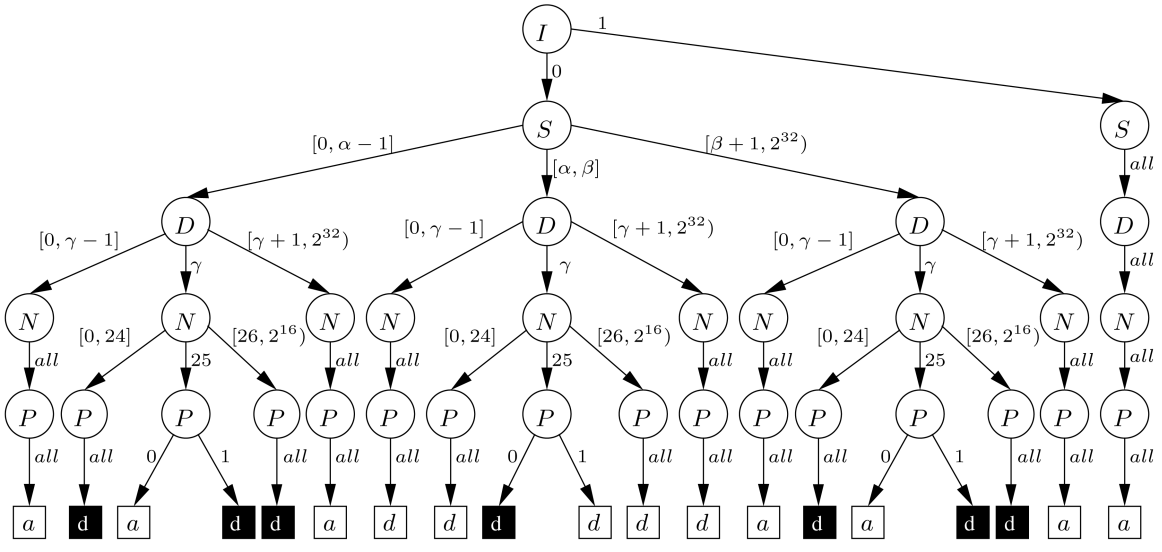

Fig. 5. The FDD transformed from the one in Fig. 3.

discovering all functional discrepancies between two semi-isomorphic FDDs is presented in Section 5.

3 Construction Algorithm

In this section, we discuss how we can construct an equivalent FDD from a sequence of rules.

3.1 Firewalls

We first formally define the concepts of fields, packets, and firewalls. A *field* F_i is a variable whose domain, denoted $D(F_i)$, is a finite interval of nonnegative integers. For example, the domain of the source address in an IP packet is $[0, 2^{32} - 1]$. A *packet* over the d fields $F_1, \cdots, F_d$ is a d-tuple $(p_1, \cdots, p_d)$, where each p_i $(1 \leq i \leq d)$ is an element of $D(F_i)$. We use Σ to denote the set of all packets over fields $F_1, \cdots, F_d$. 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 $D(F_i)$ for each i.

A firewall 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 specified as $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$, where each S_i is a nonempty interval that is a subset of $D(F_i)$. If $S_i = D(F_i)$, we can replace $F_i \in S_i$ by $F_i \in all$ or remove the conjunct $F_i \in D(F_i)$ altogether. A packet $p_1, \cdots, p_d$ *matches* a predicate $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ and the corresponding rule if and only if the condition $p_1 \in S_1 \wedge \cdots \wedge p_d \in S_d$ holds. We use α to denote the set of possible values that $\langle decision \rangle$ can be. Typical elements of α include accept, discard, accept with logging, and discard with logging. A firewall rule $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ is *simple* if and only if every S_i $(1 \leq i \leq d)$ is an interval of consecutive nonnegative integers.

A *firewall* f over the d fields $F_1, \cdots, F_d$ is a sequence of firewall rules. The size of f, denoted $|f|$, is the number of rules in F. A sequence of rules $\langle r_1, \cdots, r_n \rangle$ is *comprehensive* if and only if for any packet p, there is at least one rule in the sequence that p matches. A sequence of rules needs to be comprehensive for it to serve as a firewall. To ensure that a firewall is comprehensive, the

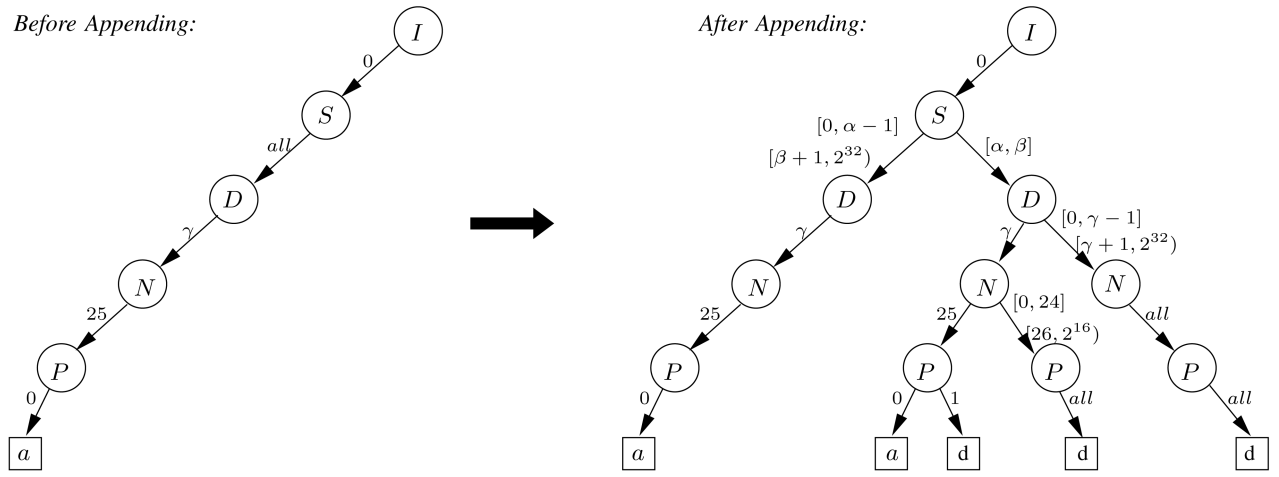

Fig. 6. Appending rule $(I \in \{0\}) \wedge (S \in [\alpha, \beta]) \wedge (D \in all) \wedge (N \in all) \wedge (P \in all) \rightarrow d$.

predicate of the last rule in a firewall is specified as $F_1 \in D(F_1) \wedge \cdots F_d \in \wedge D\,(F_d)$.

Two rules in a firewall may overlap; that is, a single packet may match both rules. Furthermore, two rules in a firewall may conflict; that is, the two rules not only overlap but also have different decisions. To resolve such conflicts, firewalls typically employ a first-match resolution strategy, where the decision for a packet p is the decision of the first (that is, the highest priority) rule that p matches in f. 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 α. 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 α; that is, for any packet $p \in \Sigma$, we have $f_1(p) = f_2(p)$. For any firewall f, we use $\{f\}$ to denote the set of firewalls that are semantically equivalent to f.

3.2 Construction of FDDs

Next, we discuss how we can construct an equivalent FDD from a sequence of rules $\langle r_1, \cdots, r_n \rangle$, where each rule is of the format $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$. Note that all the d packet fields appear in the predicate of each rule, and they appear in the same order.

We first construct a partial FDD from the first rule. A *partial FDD* is a diagram that has all the properties of an FDD, except the completeness property. The partial FDD constructed from a single rule contains only the decision path that defines the rule. Suppose that from the first i rules, r_1 through r_i, we have constructed a partial FDD, whose root v is labeled F_1. Suppose also that v has k outgoing edges $e_1, \cdots, e_k$. Let r_{i+1} be the rule $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$. Next, we consider how we can append rule r_{i+1} to this partial FDD.

At first, we examine whether we need to add another outgoing edge to v. If $S_1 - (I(e_1) \cup \cdots \cup I(e_k)) \neq \emptyset$, we need to add a new outgoing edge with label $S_1 - (I(e_1) \cup \cdots \cup I(e_k))$ to v, because any packet whose F_1 field is an element of $S_1 - (I(e_1) \cdots \cup I(e_k))$ does not match any of the first i rules but matches r_{i+1}, provided that the packet satisfies $(F_2 \in S_2) \wedge \cdots \wedge (F_d \in S_d)$. Then, we build a decision path from $(F_2 \in S_2) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$ and make the new edge of the node v point to the first node of this decision path.

Second, we compare S_1 and $I(e_j)$ for each j, where $1 \leq j \leq k$. This comparison leads to one of the following cases:

1. $S_1 \cap I(e_j) = \emptyset$. In this case, we skip edge e_j, because any packet whose value of field F_1 is in set $I(e_j)$ does not match r_{i+1}.
2. $S_1 \cap I(e_j) = I(e_j)$. In this case, for a packet whose value of field F_1 is in set $I(e_j)$, it may match one of the first i rules, and it also may match rule r_{i+1}. Thus, we append the rule $(F_2 \in S_2) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$ to the subgraph rooted at the node to which e_j points.
3. $S_1 \cap I(e_j) \neq \emptyset$, and $S_1 \cap I(e_j) \neq I(e_j)$. In this case, we split edge e into two edges: e' with label $I(e_j) - S_1$ and e'' with label $I(e_j) \cap S_1$. Then, we make two copies of the subgraph rooted at the node to which e_j points. Let e' and e'' point to one copy each. We then deal with e' by the first case and with e'' by the second case.

The pseudocode of the FDD construction algorithm is shown in Fig. 7. Here, we use $e.t$ to denote the (target) node to which the edge e points.

As an example, consider the sequence of rules in Table 1. Fig. 6 shows the partial FDD that we construct from the first rule and the partial FDD after we append the second rule. The FDD after we append the third rule is shown in Fig. 2.

Theorem 1. *Given a firewall of n simple rules, the maximum number of paths in the FDD constructed using the FDD construction algorithm is $(2n-1)^d$, where d is the number of the fields in each rule.*

Proof. Let the n simple rules be $r_1, r_2, \cdots, r_n$, where each rule r_i is denoted

$$r_i = F_1 \in S_1^i \wedge F_2 \in S_2^i \wedge \cdots \wedge F_d \in S_d^i \rightarrow decision_i.$$

For each field F_i, S_1^i has two end points (the minimum and the maximum values of the range). Thus, there are

Construction Algorithm
Input : A firewall f of a sequence of rules $\langle r_1, \cdots, r_n \rangle$
Output : An FDD f' such that f and f' are equivalent
Steps:
1. build a decision path with root v from rule r_1;
2. **for** $i := 2$ **to** n **do** APPEND(v, r_i);
End

APPEND(v, $(F_m \in S_m) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$)
/*$F(v) = F_m$ 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 $(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$, and make e_{k+1} point to the first node in this path;
2. **if** $m < d$ **then**
for $j := 1$ **to** k **do**
if $I(e_j) \subseteq S_m$ **then**
APPEND($e_j.t$, $(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$);
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($e_j.t$, $(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle dec \rangle$);

Fig. 7. FDD construction algorithm.

at most $2n$ points in the range of F_i, and the total number of intervals separated by the $2n$ points is at most $2n-1$, which means that the number of outgoing edges of a node labeled F_i is at most $2n-1$. Because the total number of fields is d, the number of paths in the constructed FDD is at most $(2n-1)^d$. □

4 Shaping Algorithm

In this section, we discuss how we can transform two ordered but not semi-isomorphic FDDs f_a and f_b into two semi-isomorphic FDDs f'_a and f'_b such that f_a is equivalent to f'_a and f_b is equivalent to f'_b. Informally, an FDD is ordered if and only if along every path from the root to a terminal node, the labels of the nonterminal nodes obey the same order. Two FDDs are semi-isomorphic if and only if they are exactly the same, except for the labels of their terminal nodes. The formal definitions of ordered FDDs and semi-isomorphic FDDs are given as follows: Note that the FDDs constructed by the construction algorithm in Section 3 are ordered.

Definition 4.1 (ordered FDDs). *Let $\prec$ be the total order over the packet fields $F_1, \cdots, F_d$, where $F_1 \prec \cdots \prec F_d$ holds. An FDD is ordered if and only if for each decision path $(v_1 e_1 \cdots v_k e_k v_{k+1})$, we have $F(v_1) \prec \cdots \prec F(v_k)$.*

Definition 4.2 (semi-isomorphic FDDs). *Two FDDs f and f' are semi-isomorphic if and only if there exists a one-to-one mapping σ from the nodes of f onto the nodes of f' such that the following conditions hold:*

1. *For any node v in f, either both v and $\sigma(v)$ are nonterminal nodes with the same label or both of them are terminal nodes.*
2. *For each edge e in f, where e is from a node v_1 to a node v_2, there is an edge e' from $\sigma(v_1)$ to $\sigma(v_2)$ in f', and the two edges e and e' have the same label.*

The algorithm for transforming two ordered FDDs into two semi-isomorphic FDDs uses the following basic operations (note that none of these operations changes the semantics of the FDDs):

1. *Node insertion.* If along all the decision paths containing a node v, there is no node that is labeled with a field F, then we can insert a node v' labeled F above v as follows: Make all incoming edges of v point to v', create one edge from v' to v, and label this edge with the domain of F.
2. *Edge splitting.* For an edge e from v_1 to v_2, if $I(e) = S_1 \cup S_2$, where neither S_1 nor S_2 is empty, then we can split e into two edges as follows: Replace e by two edges from v_1 to v_2, label one edge with S_1, and label the other with S_2.
3. *Subgraph replication.* If a node v has m $(m \geq 2)$ incoming edges, we can make m copies of the subgraph rooted at v and make each incoming edge of v point to the root of one distinct copy.

4.1 FDD Simplifying

Before applying the shaping algorithm, presented in the following, to two ordered FDDs, we need to transform each of them into an equivalent simple FDD. A simple FDD is defined as follows:

Definition 4.3 (simple FDDs). *An FDD is simple if and only if each node in the FDD has at most one incoming edge and each edge in the FDD is labeled with a single interval.*

It is straightforward that the two operations of edge splitting and subgraph replication can be applied repetitively to an FDD in order to make this FDD simple. Note that the graph of a simple FDD is an outgoing directed tree. In other words, each node in a simple FDD, except the root, has only one parent node and has only one incoming edge (from the parent node).

4.2 Node Shaping

Next, we introduce the procedure for transforming two shapable nodes into two semi-isomorphic nodes, which is the basic building block in the shaping algorithm for transforming two ordered FDDs into two semi-isomorphic FDDs. Shapable nodes and semi-isomorphic nodes are defined as follows:

Definition 4.4 (shapable nodes). *Let f_a and f_b be two ordered simple FDDs, v_a be a node in f_a, and v_b be a node in f_b. Nodes v_a and v_b are shapable if and only if one of the following conditions holds:*

1. *Both v_a and v_b have no parents; that is, they are the roots of their respective FDDs.*
2. *Both v_a and v_b have parents, their parents have the same label, and their incoming edges have the same label.*

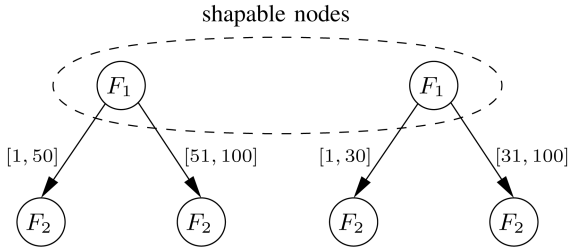

Fig. 8. Two shapable nodes in two FDDs.

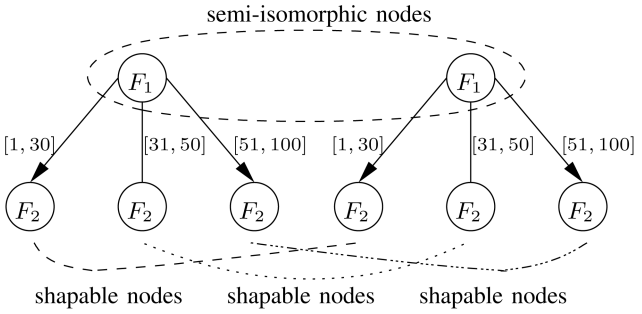

Fig. 9. Two semi-isomorphic nodes.

Definition 4.5 (semi-isomorphic nodes). *Let f_a and f_b be two ordered simple FDDs, v_a be a node in f_a, and v_b be a node in f_b. The two nodes v_a and v_b are semi-isomorphic if and only if one of the following conditions holds:*

1. *Both v_a and v_b are terminal nodes.*
2. *Both v_a and v_b are nonterminal nodes with the same label, and there exists a one-to-one mapping σ from the children of v_a to the children of v_b such that for each child v of v_a, v and $\sigma(v)$ are shapable.*

For example, the two nodes labeled F_1 in Fig. 8 are shapable, since they have no parents, and the two nodes labeled F_1 in Fig. 9 are semi-isomorphic nodes.

The algorithm for making two shapable nodes v_a and v_b semi-isomorphic consists of two steps:

Step 1. This step is skipped if v_a and v_b have the same label or both of them are terminal nodes. Otherwise, without loss of generality, assume that $F(v_a) \prec F(v_b)$. It is straightforward to show that in this case, along all the decision paths containing node v_b, no node is labeled $F(v_a)$. Therefore, we can create a new node v'_b with label $F(v_a)$, create a new edge with label $D(F(v_a))$ from v'_b to v_b, and make all incoming edges of v_b point to v'_b. Now, v_a has the same label as v'_b. (Recall that this node insertion operation leaves the semantics of the FDD unchanged.)

Step 2. From the previous step, we can assume that v_a and v_b have the same label. In the current step, we use the two operations of edge splitting and subgraph replication to build a one-to-one correspondence from the children of v_a to the children of v_b such that each child of v_a and the corresponding child of v_b are shapable.

Suppose that $D(F(v_a)) = D(F(v_b)) = [a, b]$. We know that each outgoing edge of v_a or v_b is labeled with a single interval. Suppose that v_a has m outgoing edges $\{e_1, \cdots, e_m\}$, where $I(e_i) = [a_i, b_i]$, $a_1 = a$, $b_m = b$, and every $a_{i+1} = b_i + 1$. In addition, suppose that v_b has n outgoing edges $\{e'_1, \cdots, e'_n\}$, where $I(e'_i) = [a'_i, b'_i]$, $a'_1 = a$, $b'_n = b$, and every $a'_{i+1} = b'_i + 1$.

Comparing edge e_1, whose label is $[a, b_1]$, and e'_1, whose label is $[a, b'_1]$, we have the following cases:

1. $b_1 = b'_1$. In this case, $I(e_1) = I(e'_1)$; therefore, nodes $e_1.t$ and $e'_1.t$ are shapable. (Recall that we use $e.t$ to denote the node to which edge e points.) Then, we can continue comparing e_2 and e'_2, since both $I(e_2)$ and $I(e'_2)$ begin with $b_1 + 1$.
2. $b_1 \neq b'_1$. Without loss of generality, we assume that $b_1 < b'_1$. In this case, we split e'_1 into two edges e and e', where e is labeled $[a, b_1]$, and e' is labeled $[b_1 + 1, b'_1]$. Then, we make two copies of the subgraph rooted at $e'_1.t$ and let e and e' point to one copy each. Thus, $I(e_1) = I(e)$, and the two nodes, $e_1.t$ and $e.t$ are shapable. Then, we can continue comparing the two edges e_2 and e', since both $I(e_2)$ and $I(e')$ begin with $b_1 + 1$.

The above process continues until we reach the last outgoing edge of v_a and the last outgoing edge of v_b. Note that each time that we compare an outgoing edge of v_a and an outgoing edge of v_b, the two intervals labeled on the two edges begin with the same value. Therefore, the last two edges that we compare must have the same label, because they both end with b. In other words, this edge splitting and subgraph replication process will terminate. When it terminates, v_a and v_b become semi-isomorphic.

Fig. 10 shows the pseudocode for making two shapable nodes in two ordered simple FDDs semi-isomorphic. We use $I(e) < I(e')$ to indicate that every integer in $I(e)$ is less than every integer in $I(e')$.

If we apply the above node shaping procedure to the two shapable nodes labeled F_1 in Fig. 8, we make them semi-isomorphic, as shown in Fig. 9.

4.3 FDD Shaping

To make two ordered FDDs f_a and f_b semi-isomorphic, we first make f_a and f_b simple and then make f_a and f_b semi-isomorphic as follows: Suppose that we have a queue Q, which is initially empty. At first, we put the pair of shapable nodes consisting of the root of f_a and the root of f_b into Q. As long as Q is not empty, we remove the head of Q, feed the two shapable nodes to the above *Node_Shaping* procedure, and then put all the pairs of shapable nodes returned by the *Node_Shaping* procedure into Q. When the algorithm finishes, f_a and f_b become semi-isomorphic. The pseudocode for this shaping algorithm is shown in Fig. 11.

As an example, if we apply the above shaping algorithm to the two FDDs in Figs. 2 and 3, we obtain two semi-isomorphic FDDs, as shown in Figs. 4 and 5.

5 Comparison Algorithm

In this section, we consider how we can compare two semi-isomorphic FDDs. Given two semi-isomorphic FDDs f_a and f_b with a one-to-one mapping σ, each decision path $(v_1 e_1 \cdots v_k e_k v_{k+1})$ in f_a has a corresponding decision path $(\sigma(v_1)\sigma(e_1) \cdots \sigma(v_k)\sigma(e_k)\sigma(v_{k+1}))$ in f_b. Similarly, each rule $(F(v_1) \in I(e_1)) \wedge \cdots \wedge (F(v_k) \in I(e_k)) \rightarrow F(v_{k+1}))$ in $f_a.rules$ has a corresponding rule $(F(\sigma(v_1)) \in I(\sigma(e_1))) \wedge \wedge \cdots \wedge (F(\sigma(v_k)) \in I(\sigma(e_k))) \rightarrow F(\sigma(v_{k+1}))$ in $f_b.rules$. Note that

Procedure Node_Shaping($f_a,\ f_b,\ v_a,\ v_b$)
Input : Two ordered simple FDDs f_a and f_b, and two shapable nodes v_a in f_a and v_b in f_b
Output: The two nodes v_a and v_b become semi-isomorphic, and the procedure returns a set S of node pairs of the form (w_a, w_b) where w_a is a child of v_a in f_a, w_b is a child of v_b in f_b, and the two nodes w_a and w_b are shapable.
Steps:
1. **if** (both v_a and v_b are terminal) **return(** $\emptyset$ **)**;
   else if ∼(both v_a and v_b are nonterminal and they have the same label)
   then /*Here either both v_a and v_b are nonterminal and they have different labels, or one node is terminal and the other is nonterminal. Without loss of generality, assume one of the following conditions holds: (1) both v_a and v_b are nonterminal and $F(v_a) \prec F(v_b)$, (2) v_a is nonterminal and v_b is terminal.*/
   insert a new node with label $F(v_a)$ above v_b, and call the new node v_b;
2. let $E(v_a)$ be $\{e_{a,1}, \cdots, e_{a,m}\}$ where $I(e_{a,1}) < \cdots < I(e_{a,m})$.
   let $E(v_b)$ be $\{e_{b,1}, \cdots, e_{b,n}\}$ where $I(e_{b,1}) < \cdots < I(e_{b,n})$.
3. $i := 1$; $j := 1$;
   while (($i < m$) or ($j < n$)) **do**{
      /*During this loop, the two intervals $I(e_{a,i})$ and $I(e_{b,j})$ always begin with the same integer.*/
      let $I(e_{a,i}) = [A, B]$ and $I(e_{b,j}) = [A, C]$, where A, B, C are three integers;
      if $B = C$ **then** $\{i := i + 1;\ j := j + 1;\ \}$
      else if $B < C$ **then**{
         (a) create an outgoing edge e of v_b, and label e with $[A, B]$;
         (b) make a copy of the subgraph rooted at $e_{b,j}.t$ and make e point to the root of the copy;
         (c) $I(e_{b,j}) := [B + 1, C]$;
         (d) $i := i + 1$;}
      else {/*$B > C$*/
         (a) create an outgoing edge e of v_a, and label e with $[A, C]$;
         (b) make a copy of the subgraph rooted at $e_{a,j}.t$ and make e point to the root of the copy;
         (c) $I(e_{a,i}) := [C + 1, B]$;
         (d) $j := j + 1$;}
   }
4. /*Now v_a and v_b become semi-isomorphic.*/
   let $E(v_a) = \{e_{a,1}, \cdots, e_{a,k}\}$ where $I(e_{a,1}) < \cdots < I(e_{a,k})$ and $k \geq 1$;
   let $E(v_b) = \{e_{b,1}, \cdots, e_{b,k}\}$ where $I(e_{b,1}) < \cdots < I(e_{b,k})$ and $k \geq 1$;
   $S := \emptyset$;
   for $i = 1$ **to** k **do**
      add the pair of shapable nodes ($e_{a,i}.t,\ e_{b,i}.t$) to S;
   return(S **)**;
End

Fig. 10. Node shaping algorithm.

$F(v_i) = F(\sigma(v_i))$ and $I(e_i) = I(\sigma(e_i))$ for each i, where $1 \leq i \leq k$. Therefore, for each rule $(F(v_1) \in I(e_1)) \wedge \cdots \wedge (F(v_k) \in I(e_k)) \rightarrow F(v_{k+1})$ in $f_a.rules$, the corresponding rule in $f_b.rules$ is $(F(v_1) \in I(e_1)) \wedge \cdots \wedge (F(v_k) \in I(e_k)) \rightarrow F(\sigma(v_{k+1}))$. Each of these rules is called the *companion* of the other.

This companionship implies a one-to-one mapping from the rules defined by the decision paths in f_a to the rules

Shaping Algorithm
Input : Two ordered FDDs f_a and f_b
Output : f_a and f_b become semi-isomorphic.
Steps:
1. make the two FDDs f_a and f_b simple;
2. $Q := \emptyset$;
3. add the shapable pair $(root\ of f_a, root\ of f_b)$ to Q;
4. **while** $Q \neq \emptyset$ **do**{
      remove the header pair (v_a, v_b) from Q;
      S :=Node_Shaping($f_a,\ f_b,\ v_a,\ v_b$);
      add every shapable pair from S into Q;
   }
End

Fig. 11. Shaping algorithm.

defined by the decision paths in f_b. Note that for each rule and its companion, either they are identical or they have the same predicate but different decisions. Therefore, $f_a.rules - f_b.rules$ is the set of all the rules in $f_a.rules$ that have different decisions from their companions. This applies similarly for $f_b.rules - f_a.rules$. Note that the set of all the companions of the rules in $f_a.rules - f_b.rules$ is $f_b.rules - f_a.rules$, and similarly, the set of all the companions of the rules in $f_b.rules - f_a.rules$ is $f_a.rules - f_b.rules$. Since these two sets manifest the functional discrepancies between the two FDDs, the two design teams can investigate them to resolve the discrepancies.

Let f_a be the FDD in Fig. 4 and f_b be the FDD in Fig. 5. Here, f_a is equivalent to the firewall in Table 1 designed by Team A, and f_b is equivalent to the firewall in Table 2 designed by Team B. By comparing f_a and f_b, we can discover all functional discrepancies between the firewalls designed by Teams A and B. The discrepancies are shown in Table 3, based on which the following questions need to be investigated:

1. Should we allow the computers from the malicious domain to send an e-mail to the mail server? Team A says yes, whereas Team B says no:

Interface	0
Source IP	224.168.0.0/16
Destination IP:	192.168.0.1
Destination Port:	25
Protocol Type:	TCP
Team A Decision:	accept
Team B Decision:	discard

2. Should we allow non-TCP packets with destination port number 25 to be sent from the hosts that are not in the malicious domain to the mail server? Team A says yes, whereas Team B says no:

TABLE 4
Resolved Functional Discrepancies

Discrepancy #	Interface	Source IP	Destination IP	Destination Port	Protocol	Resolved Decision
1	0	224.168.0.0/16	192.168.0.1	25	TCP	discard
2	0	!224.168.0.0/16	192.168.0.1	25	!TCP	accept
3	0	!224.168.0.0/16	192.168.0.1	!25	*	discard

TABLE 5
Firewall Generated from the Corrected FDD

Rule #	Interface	Source IP	Destination IP	Destination Port	Protocol	Decision
1	0	224.168.0.0/16	*	*	*	discard
2	0	*	192.168.0.1	25	*	accept
3	0	*	192.168.0.1	*	*	discard
4	*	*	*	*	*	accept

Interface	0
Source IP	!224.168.0.0/16
Destination IP:	192.168.0.1
Destination Port:	25
Protocol Type:	!TCP
Team A Decision:	accept
Team B Decision:	discard

3. Should we allow the packets with a destination port number other than 25 to be sent from the hosts who are not in the malicious domain to the mail server? Team A says yes, whereas Team B says no:

Interface	0
Source IP	!224.168.0.0/16
Destination IP:	192.168.0.1
Destination Port:	!25
Protocol Type:	*
Team A Decision:	accept
Team B Decision:	discard

6 Discrepancy Resolution

After all functional discrepancies are computed, the teams need to discuss correct decisions for each discrepancy. Consider the discrepancies shown in Table 3. Suppose that these discrepancies are resolved, as shown in Table 4.

The question that we want to answer in this section is: How do we generate the final firewall that reflects the resolved functional discrepancies? Of course, if one team made all the correct decisions according to the discrepancy resolution, we can simply deploy the firewall designed by that team. Next, we assume that no team makes all the correct decisions. In this paper, we propose two methods for this purpose. Then, we discuss which methods should be chosen in practice.

6.1 Method 1: Generate Rules from Corrected FDD

This method has two steps. First, correct one of the semi-isomorphic FDDs by using discrepancy resolution. Second, generate rules from the resulting FDD by using the algorithms presented in [12].

Step 1: FDD correction. We can pick either semi-isomorphic FDD generated by the FDD shaping algorithm and apply corrections on the labels of the terminal nodes. Note that after we apply fixes to two semi-isomorphic FDDs, they become exactly the same. Note that we cannot directly use the corrected FDD as the configuration of a firewall, because most existing firewall devices take a sequence of rules as their configuration.

Step 2: firewall generation. Given the corrected FDD, we can apply the algorithms in [12] for generating a compact firewall from an FDD. Table 5 shows the firewall generated from the corrected FDD. Interested readers can refer to [12] for more technical details.

6.2 Method 2: Combine Corrections with Original Firewalls

The second method is to create a new firewall by using the rules in the discrepancy resolution and one of the original firewalls. This method consists of the following steps:

Step 1: firewall composition. In this step, we first pick an original firewall, and then, we take all the rules in the discrepancy resolution in which the original firewall made incorrect decisions and add them to the beginning of the firewall.

Step 2: redundancy removal. In this step, we apply the firewall compaction algorithm in [19] to remove redundant rules from the resulting sequence of rules. A rule is redundant if and only if removing the rule does not change the semantics of the firewall.

For example, we can pick the firewall in Table 1 designed by Team A, and on top of that, we can add the first and third rules from the discrepancy resolution in Table 4. Note that Team A only made incorrect decisions for the packets that match the first and third rules in Table 4. By adding these two rules to the beginning of the original three rules designed by Team A, all packets are mapped to the correct decisions. After the above two steps, the resulting firewall is shown in Table 6. Similarly, we can pick the firewall in Table 2 designed by Team B and then add the second rule from the discrepancy resolution in Table 4 to the beginning of the firewall. After the above two steps, the resulting firewall is shown in Table 7.

TABLE 6
Firewall Generated by Combining the Rules in Table 4 and the Rules in Table 1

Rule #	Interface	Source IP	Destination IP	Destination Port	Protocol	Decision
1	0	224.168.0.0/16	192.168.0.1	25	TCP	discard
2	0	!224.168.0.0/16	192.168.0.1	!25	*	discard
3	0	*	192.168.0.1	25	TCP	accept
4	0	224.168.0.0/16	*	*	*	discard
5	*	*	*	*	*	accept

TABLE 7
Firewall Generated by Combining the Rules in Table 4 and the Rules in Table 2

Rule #	Interface	Source IP	Destination IP	Destination Port	Protocol	Decision
1	0	!224.168.0.0/16	192.168.0.1	25	!TCP	accept
2	0	224.168.0.0/16	*	*	*	discard
3	0	*	192.168.0.1	25	TCP	accept
4	0	*	192.168.0.1	*	*	discard
5	*	*	*	*	*	accept

7 Discussion

7.1 Prefix and Intervals

Real-life firewalls usually check five packet fields:

1. source IP address,
2. destination IP address,
3. source port number,
4. destination port number, and
5. protocol type.

Of these five fields, the first two fields are usually represented using prefix formats, and the last three fields are usually integer intervals. Note that prefix formats and interval formats are interconvertable. 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 three prefixes: $001*$, $01*$, and 1,000.

To use the algorithms presented in this paper, 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 the three algorithms described in this paper. Note that the functional discrepancies directly produced by our algorithms are in interval format. Third, for each functional discrepancy computed, we convert the source and destination IP addresses from intervals to prefixes. Thus, the format of outputs are similar to those of original firewall rules, which are easy to understand for firewall administrators. (A w-bit integer interval can be converted to at most $2w - 2$ prefixes [14].)

7.2 Design in FDDs

In our discussion so far, we have assumed that the two teams both design their firewalls by using a sequence of rules. In fact, a team can use the structured firewall design method in [12] to design the firewall by using an FDD. Such cases are easy to handle by using the FDD construction algorithm in this paper and the firewall generation algorithm in [12]. For example, if only one team designs the firewall by using a nonordered FDD, we can use the firewall generation algorithm in [12] to generate a sequence of rules from the FDD first and then apply the algorithms in this paper. As another example, if two teams design two ordered FDDs that are in a different order, we can first generate an equivalent sequence of rules from one diagram, and then, we can construct an equivalent ordered FDD from the sequence of rules by using the order of packet fields from the other FDD.

7.3 More Than Two Teams

In terms of firewall comparison, what we have discussed so far is how two firewalls can be compared. If we have N firewalls designed by N teams, where $N > 2$, there are two ways of comparing them: cross comparison and direct comparison. Cross comparison means comparing each of the $N * (N - 1)$ pairs, where each pair consists of two of the N firewalls. Direct comparison means extending the shaping algorithm and the comparison algorithm to handle N firewalls. This extension is considered fairly straightforward.

7.4 Complexity Analysis

Let n be the number of rules in a firewall and d be the total number of distinct packet fields that are examined by a firewall. Based on Theorem 1, the time and space complexity of the FDD construction algorithm is $O(n^d)$. Similarly, the time and space complexity of the FDD shaping algorithm and the FDD comparison algorithm is $O((n + m)^d)$, where n and m are the total number of rules in the two given firewalls, respectively. Despite such worst case complexities, our algorithms are practical for two reasons. First, d is typically small. Most real-life firewalls only examine four packet fields:

1. source IP address,
2. destination IP address,
3. destination port number, and
4. protocol type.

Second, the worst case of our algorithms is extremely unlikely to happen in practice. The experimental results in the next section confirm the above observations.

7.5 Why not BDDs?

Our solution uses FDDs as the basic data structure for computing the functional discrepancies between two given firewalls. One question that we need to answer is: Why not use Binary Decision Diagrams (BDDs) [6]? A BDD is a rooted directed acyclic graph that represents a Boolean function. In a BDD, each nonterminal node is labeled by a Boolean variable, and it has only two outgoing edges labeled 0 and 1, respectively. Each edge represents an assignment of 0 or 1. A BDD only has two terminal nodes labeled 0 and 1, respectively.

The answer is that the functional discrepancies computed by BDDs are not human readable. First, the BDD itself, that is, the one that represents the functional discrepancies between two firewalls, is not human readable, because every node in a BDD represents only a bit of a packet and not a field of a packet. Second, generating human readable discrepancies, which are similar to rules, from a BDD results in an exorbitant number of rules, which is in terms of millions. We have implemented BDD-based solutions using the CUDD package [23]. Unfortunately, comparing two small firewalls results in millions of rules. Although compressing millions of rules may not be impossible, it is, by no means, trivial. In contrast, using the data structure of FDDs, we can easily generate human readable functional discrepancies in rulelike format.

8 Experimental Results

In this section, we present the results of the experiments that we conducted to evaluate both the effectiveness and efficiency of our diverse firewall design method.

8.1 Effectiveness

To evaluate the effectiveness of the diverse firewall design method, we conducted a real experiment as follows: First, we obtained a real-life firewall used in a university. This firewall was maintained by a senior firewall administrator as a sequence of rules. This firewall, unfortunately, did not have a requirement specification. However, the rules in this firewall were well documented in that each rule had some detailed comments about why the rule was added. Taking the comments of the rules as the requirement specification, we let an undergraduate student of computer science design a firewall by using FDDs. Before the design started, we gave the student some training on designing firewalls by using FDDs.

The original firewall had 87 rules. The new firewall was designed as an FDD. Comparing the two firewalls using the algorithms presented in this paper, we discovered 84 functional discrepancies. Then, the senior firewall administrator and the undergraduate student discussed what the correct decision for each of the functional discrepancies should be. The conclusions of the discussion were that in 82 functional discrepancies, the original firewall made incorrect decisions, and in the other two functional discrepancies, the new firewall made incorrect decisions. The two functional discrepancies where the new firewall made incorrect decisions were cause by incorrect assumptions regarding the requirement specification of the firewall.

We learned some things from this experiment:

1. The method of diverse firewall design is effective in practice and can be used flexibly in a variety of scenarios. For example, it can be used to redesign an existing firewall as what we did while conducting the experiment. Many firewall administrators are afraid of redesigning their firewall due to the concerns of possible mistakes. Using the method of diverse firewall design, redesigning an existing firewall could be an effective way to find errors in the firewall.
2. A tool that can perform change impact analysis of firewalls is greatly needed in practice. Out of the 82 functional discrepancies, where the original firewall made incorrect decisions, 72 of them were caused by incorrect ordering of rules (and the rest were caused by missing rules). Most of the incorrect ordering of rules was caused by the firewall administrator incorrectly adding new rules to the beginning of the firewall when making changes. If the firewall administrator had a tool that could compute the impact of a change, such errors could be greatly reduced. The algorithms presented in this paper can be used to perform firewall change impact analysis by comparing the firewalls before and after changes.

8.2 Efficiency

We presented three algorithms in this paper, namely, the construction algorithm, the shaping algorithm, and the comparison algorithm, for detecting all functional discrepancies between two given firewalls. We implemented these algorithms in Java JDK 1.4. To evaluate the performance of these algorithms, first, we ran our algorithms on a real-life firewall of fairly large size (661 rules) and a real-life firewall of average size (42 rules). Second, we stress tested our algorithms on a large number of synthetic firewalls of large sizes. These experiments were carried out on a SunBlade 2000 machine running Solaris 9 with a 1-GHz CPU and 1-Gbyte memory. In both cases, the experimental results show that our algorithms perform and scale well.

8.2.1 Real-Life Firewalls

We first ran our algorithms on two real-life firewalls: one of large size with 661 rules and one of average size with 42 rules. (In real-life firewalls, only 0.7 percent have more than 1,000 rules, and the average number of rules is 50 [13].)

To simulate two design teams, we conducted the experiments on the two real-life firewalls as follows: For each firewall, in each experiment, we first randomly selected x percent of rules from the firewall. Let S be the set of selected rules. Second, we randomly picked a number y in the range from 0 to 100. Third, we randomly selected y percent of the rules in S to change their decisions. Last, for the remaining $1 - y$ percent of the rules in S, we deleted them from the original firewall. Thus, we obtained two firewalls: the original firewall and the resulting firewall after the above four steps

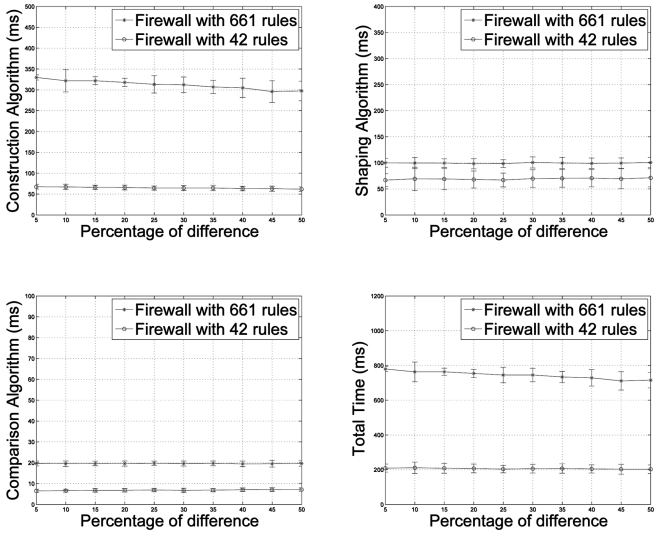

Fig. 12. Experimental results on real-life firewalls.

were applied. We used our algorithms to compute all functional differences between them. We let x range from 5 to 50. Note that the original firewall and the resulting firewall share $(1 - x \text{ percent}) \times \textit{the original firewall size}$ rules. For each firewall and each value of x, we ran the experiment 100 times, with a random value y each time.

The experimental results are shown in Fig. 12. The x-axis is the value of x, and the y-axis is the runtime of each algorithm in milliseconds. Note that the total time includes constructing two ordered FDDs from two sequences of rules, shaping the two ordered FDDs to be semi-isomorphic, and comparing the two semi-isomorphic FDDs.

8.2.2 Synthetic Firewalls

Firewall configurations are considered confidential due to security concerns. To further evaluate the performance of our algorithms on large firewalls, we generated synthetic firewalls based on the characteristics of real-life firewalls reported in [13]. Every rule in a synthetic firewall has five fields:

1. source IP address,
2. destination IP address,
3. source port number,
4. destination port number, and
5. protocol type.

In each experiment, we first generated two firewalls independently and then ran the three algorithms on them. Fig. 13 shows the average execution times for the construction algorithm, the shaping algorithm, and the comparison algorithm versus the total number of rules. In this figure, we see that it took less than 5 seconds to detect all discrepancies between two sequences of 3,000 rules.

In practice, our algorithms are expected to run faster because of two reasons. First, in real life, the two input firewalls are likely to be similar if they are from two design teams based on the same specification or if they are the two firewalls before and after firewall changes are applied. According to the shaping algorithm, comparing two firewalls with more common rules is faster. In comparison, the runtime data in Fig. 13 is for two firewalls that were generated independently. Second, in real life, the two input firewalls are likely to be smaller. In general, our algorithms run faster for smaller firewalls.

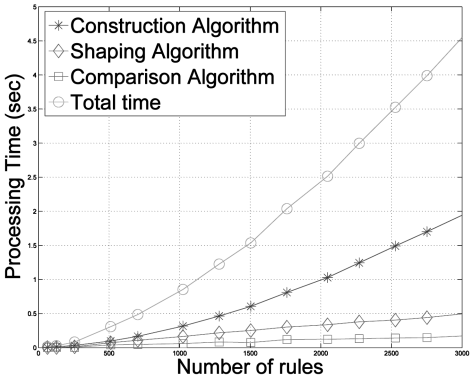

Fig. 13. Experimental results on synthetic firewalls of large sizes.

9 Related Work

Some firewall policy design and modeling methods have been proposed previously. We have proposed using FDDs for designing firewalls [12] and a model for specifying stateful firewall policies [11]. Guttman proposed a Lisp-like language for specifying high-level packet filtering policies [15]. Bartal et al. proposed a UML-like language for specifying global filtering policies [5]. Some firewall policy analysis methods have also been proposed before. We have proposed analyzing and testing firewall policies using queries [20] and a method for identifying all the redundant rules in a firewall policy [19]. In [1] and [29], some anomalies are defined, and techniques for detecting anomalies were presented. Those anomalies are subjectively defined and may not be deemed as errors by a firewall administrator. Using methods similar to firewall analysis, Hamed et al. studied the analysis and verification of IPsec policies [16]. The design of high-performance ATM firewalls were discussed in [27] and [28], with emphasis on firewall architectures. Firewall vulnerabilities were discussed and classified in [9] and [18], with emphasis on firewall software.

The relationship between our diverse firewall design method and previous firewall design and analysis methods are twofold. First, none of the previous work has ever explored design diversity. Furthermore, none of the studies has ever tackled the problem of change impact analysis for firewall policies. This paper represents the first attempt in this direction. Second, our diverse firewall design method is intended to complement, rather than to replace, the previous firewall design and analysis methods, as these methods can assist each individual team to design their firewall in the design phase before cross comparison.

Our idea of diverse firewall design is inspired by N-version programming [3], [4] and back-to-back testing [25]. The basic idea of N-version programming is to give the same requirement specification to N teams to independently design and implement N programs by using different algorithms, languages, or tools. Then, the resulting

N programs are executed in parallel. A decision selection mechanism is deployed to examine the N results for each input from the N programs and selects a correct or the "best" result. The key element of N-version programming is design diversity. The diversity in the N programs should be maximized such that coincident failure for the same input is rare. The effectiveness of the N-version programming method for building fault-tolerant software has been shown in a variety of safety-critical systems built since the 1970s, such as railway interlocking and train control [2], airbus flight control [24], and nuclear reactor protection [7].

Back-to-back testing is a complementary method to N-version programming. This method is used to test the resulting N versions before deploying them in parallel. The basic idea is given as follows: At first, create a suite of test cases. Second, for each test case, execute the N programs in parallel. Cross-compare the N results, then investigate each discrepancy discovered, and apply corrections.

Our diverse firewall design method has two unique properties that distinguish it from N-version programming and back-to-back testing. First, only one firewall version needs to be deployed and executed. This is because all discrepancies between the multiple firewall versions can be discovered by the algorithms presented in this paper, and corrections can be applied to make them equivalent. In contrast, the N-version programming method requires deploying all the N programs and executing them in parallel. Second, the algorithms in this paper can detect all functional discrepancies between the multiple firewall versions. In contrast, back-to-back testing is not guaranteed to detect all functional discrepancies among N programs.

Although numerous studies have been done on analyzing the change impact of general programs in software engineering communities [17], [22], this paper represents the first effort to analyze the change impact of firewall policies. Firewall policies and general programs are fundamentally different. Although accurately and completely computing the impact of software changes is nearly impossible in general, the algorithms presented in this paper can compute the accurate and complete impact of firewall policy changes.

Fisler et al. studied change impact analysis of access control policies in their seminal paper [8]. They proposed a solution using multiterminal BDDs to compute the impact of access control policy changes and verify whether an access control policy satisfies a given property. Their work is similar to ours in spirit; however, their solution cannot be applied to firewall policies, because the access control policies studied in [8] are quite different from firewall policies. In [8], every attribute-value pair is encoded as one variable in the MTBDD. This is natural for the access control policies studied in [8] but is not feasible for firewall policies because of the explosive number 2^{88} of attribute-value pairs.

10 Conclusions

In this paper, we make four major contributions. First, we proposed the method of diverse firewall design. This paper represents the first effort to apply the well-known principle of diverse design to firewalls. Second, we presented a method that can compare two given firewalls and output all functional discrepancies between them in human readable format. This is the first method for this purpose. Third, we presented a method to compute firewall change impacts by computing all functional discrepancies between the firewall before changes and the firewall after changes. This is the first method for performing firewall change impact analysis. Last, we implemented our algorithms and evaluated their performance on both real-life and synthetic firewalls of large sizes. Experimental results demonstrate that our algorithms are efficient in comparing two firewalls of large sizes. It is worth emphasizing that the methods and algorithms presented in this paper are not limited to the design and analysis of firewall policies. Rather, they can be applied to other rule based systems as well.

Acknowledgments

The authors would like to thank the editor Tarek Abdelzaher and the anonymous referees for their constructive comments and valuable suggestions on improving the presentation of this paper. The work of Alex X. Liu was supported in part by the US National Science Foundation under Grant CNS-0716407. The work of Mohamed G. Gouda was supported by the US National Science Foundation under Grant 0520250. A preliminary version of this paper was published in the Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN), pp. 595-604, Florence, Italy, June 2004. It won the William C. Carter Award.

References

[1] E. Al-Shaer and H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," *Proc. IEEE INFOCOM '04,* pp. 2605-2616, Mar. 2004.

[2] H. Anderson and G. Hagelin, "Computer Controlled Interlocking System," *Ericsson Rev.,* vol. 2, 1981.

[3] A. Avizienis, "The N-Version Approach to Fault Tolerant Software," *IEEE Trans. Software Eng.,* vol. 11, no. 12, pp. 1491-1501, 1985.

[4] A. Avizienis, "The Methodology of N-Version Programming," *Software Fault Tolerance,* Chapter 2, M.R. Lyu, ed. Wiley, pp. 23-46, 1995.

[5] Y. Bartal, A.J. Mayer, K. Nissim, and A. Wool, "Firmato: A Novel Firewall Management Toolkit," *Proc. IEEE Symp. Security and Privacy (S&P '99),* pp. 17-31, 1999.

[6] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers,* vol. 35, no. 8, pp. 677-691, 1986.

[7] A. Condor and G. Hinton, "Fault Tolerant and Fail-Safe Design of Candu Computerized Shutdown Systems," *IAEA Specialist Meeting on Microprocessors Important to the Safety of Nuclear Power Plants,* May 1988.

[8] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz, "Verification and Change Impact Analysis of Access-Control Policies," *Proc. 27th Int'l Conf. Software Eng. (ICSE '05),* May 2005.

[9] M. Frantzen, F. Kerschbaum, E. Schultz, and S. Fahmy, "A Framework for Understanding Vulnerabilities in Firewalls Using a Dataflow Model of Firewall Internals," *Computers and Security,* vol. 20, no. 3, pp. 263-270, 2001.

[10] M.G. Gouda and A.X. Liu, "Firewall Design: Consistency, Completeness and Compactness," *Proc. 24th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '04),* pp. 320-327, Mar. 2004.

[11] M.G. Gouda and A.X. Liu, "A Model of Stateful Firewalls and Its Properties," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 320-327, June 2005.

[12] M.G. Gouda and A.X. Liu, "Structured Firewall Design," *Computer Networks J.,* vol. 51, no. 4, pp. 1106-1120, Mar. 2007.

[13] P. Gupta, "Algorithms for Routing Lookups and Packet Classification," PhD dissertation, Stanford Univ., 2000.
[14] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network,* vol. 15, no. 2, pp. 24-32, 2001.
[15] J.D. Guttman, "Filtering Postures: Local Enforcement for Global Policies," *Proc. IEEE Symp. Security and Privacy (S&P '97),* pp. 120-129, 1997.
[16] H. Hamed, E. Al-Shaer, and W. Marrero, "Modeling and Verification of IPsec and VPN Security Policies," *Proc. 13th IEEE Int'l Conf. Network Protocols (ICNP '05),* pp. 259-278, Nov. 2005.
[17] S. Horwitz, "Identifying the Semantic and Textual Differences between Two Versions of a Program," *Proc. ACM Conf. Programming Language Design and Implementation (PLDI '90),* pp. 234-245, 1990.
[18] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen, "Analysis of Vulnerabilities in Internet Firewalls," *Computers and Security,* vol. 22, no. 3, pp. 214-232, 2003.
[19] A.X. Liu and M.G. Gouda, "Complete Redundancy Detection in Firewalls," *Proc. 19th Ann. IFIP Conf. Data and Applications Security,* pp. 196-209, Aug. 2005.
[20] A.X. Liu, M.G. Gouda, H.H. Ma, and A.H. Ngu, "Firewall Queries," *Proc. Eighth Int'l Conf. Principles of Distributed Systems (OPODIS '04),* pp. 124-139, Dec. 2004.
[21] D. Oppenheimer, A. Ganapathi, and D.A. Patterson, "Why Do Internet Services Fail, and What Can Be Done about It?" *Proc. Fourth Usenix Symp. Internet Technologies and Systems (USITS '03),* Mar. 2003.
[22] X. Ren, O.C. Chesley, and B.G. Ryder, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," *IEEE Trans. Software Eng.,* vol. 32, no. 9, pp. 718-732, 2006.
[23] F. Somenzi, *Cudd: Cu Decision Diagram Package Release 2.4.1,* http://vlsi.colorado.edu/fabio/cudd/, 2007.
[24] P. Traverse, "Airbus and ATR System Architecture and Specification," *Software Diversity in Computerized Control Systems,* U. Voges, ed. Springer Verlag, 1988.
[25] M.A. Vouk, "On Back-to-Back Testing," *Proc. Third Ann. Conf. Computer Assurance (COMPASS '88),* pp. 84-91, 1988.
[26] A. Wool, "A Quantitative Study of Firewall Configuration Errors," *Computer,* vol. 37, no. 6, pp. 62-67, 2004.
[27] J. Xu and M. Singhal, "Design and Evaluation of a High-Performance ATM Firewall Switch and Its Applications," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1190-1200, 1999.
[28] J. Xu and M. Singhal, "Design of a High-Performance ATM Firewall," *ACM Trans. Information and System Security,* vol. 2, no. 3, pp. 269-294, 1999.
[29] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "Fireman: A Toolkit for Firewall Modeling and Analysis," *Proc. IEEE Symp. Security and Privacy (S&P '06),* May 2006.

Alex X. Liu received the PhD 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, Michigan State University. His research interests include computer and network security, dependable and high-assurance computing, applied cryptography, computer networks, operating systems, and distributed computing. He is a member of the IEEE. He received the 2004 IEEE and IFIP William C. Carter Award, the 2004 National Outstanding Overseas Students Award sponsored by the Ministry of Education of China, the 2005 George H. Mitchell Award for Excellence in Graduate Research from the University of Texas at Austin, and the 2005 James C. Browne Outstanding Graduate Student Fellowship from the University of Texas at Austin.

Mohamed G. Gouda received the PhD degree in computer science from the University of Waterloo. From 1977 to 1980, he was with the Honeywell Corporate Technology Center, Minneapolis. In 1980, he joined the University of Texas at Austin, where he is currently with the Department of Computer Sciences as the Mike A. Myers Centennial Professor of Computer Sciences. He has supervised 19 PhD dissertations. He was the founding editor in chief (from 1985 to 1989) of the Springer-Verlag journal *Distributed Computing.* From 1996 to 1999, he served on the editorial board of *Information Sciences*, and he is currently on the editorial boards of *Distributed Computing* and the *Journal of High-Speed Networks.* His research interests include distributed and concurrent computing and network protocols. In these areas, he has been working on abstraction, formality, correctness, nondeterminism, atomicity, reliability, security, convergence, and stabilization. He has published more than 60 journal papers and more than 80 conference and workshop proceedings. He is the author of *Elements of Network Protocol Design* (John Wiley & Sons, 1998). He received the Kuwait Award in Basic Sciences in 1993, two IBM Faculty Partnership Awards for the academic years 2000-2001 and 2001-2002, and the IBM Austin Center for Advanced Studies Fellowship in 2002. He is a corecipient (with C.K. Wong and S.S. Lam) of the 2001 IEEE Communication Society William R. Bennet Best Paper Award for the paper *Secure Group Communications Using Key Graphs*, published in the February 2000 issue of the *IEEE/ACM Transactions on Networking.* He is also a corecipient (with Alex X. Liu) of the 2004 William C. Carter Award for the paper *Diverse Firewall Design*, published in the *Proceedings of the International Conference on Dependable Systems and Networks*. He is a member of the IEEE.