Diverse Firewall Design

Alex X. Liu Mohamed G. Gouda
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188, U.S.A.
{alex, gouda}@cs.utexas.edu

Abstract

Firewalls are safety-critical systems that secure most private networks. An error in a firewall either leaks secret information from its network or disrupts legitimate communication between its network and the rest of the Internet. How to design a correct firewall is therefore an important issue. In this paper, we propose the method of diverse firewall design, which is inspired by the well-known method of design diversity for building fault-tolerant software. Our method consists of two phases: a design phase and a comparison phase. In the design 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 comparison phase, the resulting multiple versions are compared with each other to find out all the discrepancies between them, then each discrepancy is further investigated and a correction is applied if necessary. The technical challenge in the method of diverse firewall design is how to discover all the discrepancies between two given firewalls. We present a series of three efficient algorithms for solving this problem: (1) a construction algorithm for constructing an equivalent ordered firewall decision diagram from a sequence of rules, (2) a shaping algorithm for transforming two ordered firewall decision diagrams to become semi-isomorphic without changing their semantics, and (3) a comparison algorithm for detecting all the discrepancies between two semi-isomorphic firewall decision diagrams.

1. Introduction

A firewall is a security guard placed at the point of entry between a private network and the outside Internet so 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; examples of these fields are source/destination IP address, source/destination port number, and protocol type. By examining the values of these fields for each incoming and outgoing packet, a firewall accepts legitimate packets and discards illegitimate ones according to its configuration. A firewall configuration defines which packets are legitimate and which are illegitimate. An error in a firewall configuration means a wrong definition of being legitimate or illegitimate for some packets, which will either allow unauthorized access from the outside Internet to the private network, or disable some legitimate communication between the private network and the outside Internet. Neither case is desirable. How to design a correct firewall configuration is therefore an important security issue. Since the correctness of a firewall configuration is the focus of this paper, we assume a firewall is correct *iff* (if and only if) its configuration is correct, and a firewall configuration is correct iff it satisfies its given requirement specification, which is usually written in a natural language. In the rest of this paper, we use "firewall" to mean "firewall configuration" if not 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. We observe that different designers may have different understandings of the same informal requirement specification, and different designers may exhibit different technical strengths and weaknesses. This observation motivates our method of diverse firewall design.

Our diverse firewall design method has two phases: a design phase and a comparison phase. In the design phase, the same requirement specification is given to multiple teams who proceed independently to design different versions of the firewall. Different teams preferably have different technical strengths and use different design methods. By maximizing diversity in the design phase, the coincident errors made by all teams are rare. In the comparison phase, the resulting multiple versions are compared with each other to discover all discrepancies. Then each discrepancy is further

investigated and a correction is applied if necessary. After these comparisons and corrections, all the versions become equivalent. Then any one of them can be deployed.

The technical challenge in this diverse firewall design method is that how to discover all the discrepancies between two given firewalls, where each is designed by either a sequence of rules or a firewall decision diagram. Our solution for comparing two given firewalls consists of the following three steps: (1) If either of the two firewalls is designed by a sequence of rules, we construct an equivalent ordered firewall decision diagram from the sequence of rules by the construction algorithm in Section 4. If either of the two firewalls is designed by a non-ordered firewall decision diagram, we at first generate an equivalent sequence of rules from the diagram, then construct an equivalent ordered firewall decision diagram from the sequence of rules. After this step, we get two ordered firewall decision diagrams. (2) If the two ordered firewall decision diagrams are not semi-isomorphic, we transform them to two semi-isomorphic firewall decision diagrams without changing their semantics by the shaping algorithm in Section 5. After this step, we get two semi-isomorphic firewall decision diagrams. (3) Given two semi-isomorphic firewall decision diagrams, all the discrepancies between them can be discovered by the comparison algorithm in Section 6.

In conclusion, we make two main contributions in this paper: (1) we propose the method of diverse firewall design; (2) we present a series of three algorithms for discovering all the discrepancies between two given firewalls: a construction algorithm, a shaping algorithm, and the comparison algorithm. The experimental results shows that these three algorithms are very efficient.

2. Related Work

The idea of diverse firewall design is inspired by N-version programming [6, 7, 8, 23], and back-to-back testing [25, 26]. The basic idea of N-version programming is to give the same requirement specification to N teams to independently design and implement N programs using different algorithms, languages, or tools. Then the resulting N programs are executed in parallel. A decision mechanism is deployed to examine the N results for each input from the N programs and selects a correct or "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 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 [5], Airbus flight control [24], and nuclear reactor protection [12].

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 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 if necessary.

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 the 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. By contrast, the N-version programming method requires the deployment of all the N programs and executing them in parallel. Second, the algorithms in this paper can detect all the discrepancies between the multiple firewall versions. By contrast, back-to-back testing is not guaranteed to detect all the discrepancies among N programs.

A firewall is usually designed by a sequence of rules and the rules may overlap and conflict with each other. Two rules overlap iff there is at least one packet that matches both rules. Two rules conflict iff the two rules overlap and also have different decisions. The conflicts among rules make firewall design difficult and error prone. Detection of conflicts was discussed in [10, 13, 19, 21]. Similar to conflict detection, six types of so-called "anomalies" were defined in [1, 2, 3]. Examining each conflict or anomaly is helpful in reducing errors; however, the number of conflicts or anomalies in a firewall is usually 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 high level languages for describing firewall rules were proposed in [11, 18]. These languages are helpful in firewall design; however, high level rules may still conflict.

Some firewall rule analysis tools are discussed in [4, 20]. The analysis is basically answering queries from users. These tools are helpful in analyzing some suspicious behaviors of a firewall; however, the ad-hoc queries are not guaranteed to cover all aspects of a firewall.

There are some tools currently available for network vulnerability testing, such as Satan [14, 15] and Nessus [22]. 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.

3. Firewall Design Methods

We define a *packet* over the fields $F_1, \cdots, F_d$ as a d-tuple $(p_1, \cdots, p_d)$ where each p_i is an element in the domain $D(F_i)$ of field F_i, and each $D(F_i)$ is an interval of nonnegative integers. For example, one of the fields of an IP packet is the source address, and the domain of this field is $[0, 2^{32})$. For the brevity of presentation, we assume that all packets are over the d fields $F_1, \cdots, F_d$, and we use Σ to denote the set of all packets. It follows that Σ is a finite set of size $|D(F_1)| \times \cdots \times |D(F_d)|$.

3.1. Firewall

A firewall consists of a sequence of rules, where each rule is of the following format:

$$(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$$

where each S_i is a nonempty subset of $D(F_i)$, and the $\langle decision \rangle$ is either $accept$ or $discard$. 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 rule $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$ iff the condition $(p_1 \in S_1) \wedge \cdots \wedge (p_d \in S_d)$ holds. Since a packet may match more than one rule in a firewall, each packet is mapped to the decision of the first rule that the packet matches. The predicate of the last rule in a firewall is usually a tautology to ensure that every packet has at least one matching rule in the firewall.

3.2. Firewall Function

A *firewall function* is a many-to-one mapping: $\Sigma \rightarrow \{accept, discard\}$. This function can be defined by a sequence of rules or by a firewall decision diagram. Two firewalls are equivalent iff they implement the same firewall function.

3.3. Firewall Decision Diagram

In [16], Gouda and Liu presented Firewall Decision Diagrams as a useful notation for specifying firewalls. A *Firewall Decision Diagram* (FDD) f over fields $F_1, \cdots, F_d$ is an acyclic and directed graph that has the following five properties:

1. There is exactly one node in f that has no incoming edges. This node is called the *root* of f. The nodes in f that have no outgoing edges are called *terminal* nodes of f.
2. Each node v in f has a label, denoted $F(v)$, such that
$$F(v) \in \begin{cases} \{F_1, \cdots, F_d\} & \text{if } v \text{ is nonterminal,} \\ \{accept, discard\} & \text{if } v \text{ is terminal.} \end{cases}$$
3. Each edge e in f has a label, denoted $I(e)$, such that if e is an outgoing edge of node v, then
$$I(e) \subseteq D(F(v)).$$
4. A directed path in f from the root to a terminal node is called a *decision path* of f. No two nodes on a decision path have the same label.
5. The set of all outgoing edges of a node v in f, 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*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$ □

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 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

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

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

Given an FDD f, any sequence of rules that consists of all the rules in S_f is equivalent to f. The order of the rules in such a firewall is immaterial because the rules in S_f are non-overlapping.

Given a sequence of rules, how to construct an equivalent FDD is discussed in Section 4.

3.4. A Running Example

In this paper, we use the following running example. Consider the simple network in Figure 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. The requirement specification for this firewall is depicted in Figure 2.

Suppose we give this specification to two teams: Team A and Team B. Team A designs the firewall by the FDD

in Figure 3 and Team B designs the firewall by the sequence of rules in Figure 4. In this paper, we use the following shorthand: a (Accept), d (Discard), I (Interface), S (Source IP), D (Destination IP), N (Destination Port), P (Protocol Type). We use α to denote the integer formed by the four bytes of the IP address 192.168.0.0, and similarly β for 192.168.255.255, and γ for 192.1.2.3. We assume the protocol type value in a packet is either 0 (TCP) or 1 (UDP). For ease of presentation, we assume that each packet has a field containing the information of the network interface on which a packet arrives.

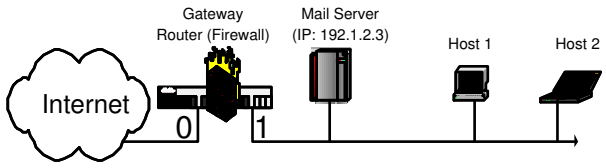

Figure 1. A firewall

The mail server with IP address 192.1.2.3 can receive emails. The packets from an outside malicious domain 192.168.0.0/16 should be blocked. Other packets should be accepted and allowed to proceed.

Figure 2. The requirement specification

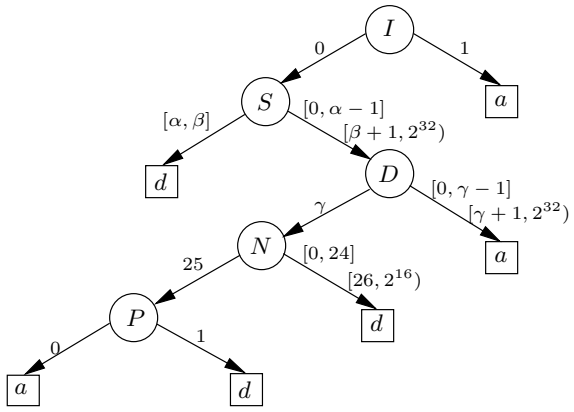

Figure 3. The FDD by Team A

Given these two firewalls, one in Figure 3 and the other in in Figure 4, we use the following three steps to discover all the discrepancies between them: (1) construct an equivalent ordered FDD (in Figure 6) from the sequence of rules

1. $(I \in \{0\}) \wedge (S \in all) \wedge (D \in \{\gamma\}) \wedge (N \in \{25\}) \wedge (P \in \{0\}) \rightarrow a$
2. $(I \in \{0\}) \wedge (S \in [\alpha, \beta]) \wedge (D \in all) \wedge (N \in all) \wedge (P \in all) \rightarrow d$
3. $(I \in all) \wedge (S \in all) \wedge (D \in all) \wedge (N \in all) \wedge (P \in all) \rightarrow a$

Figure 4. The firewall by Team B

in Figure 4 by the construction algorithm in Section 4; (2) transform the two ordered FDDs, one in Figure 3 and the other in Figure 6, to two semi-isomorphic FDDs (where one is in Figure 9) by the shaping algorithm in Section 5; (3) discover all the discrepancies between the two semi-isomorphic FDDs by the comparison algorithm in Section 6.

4. Construction Algorithm

In this section, we discuss how to 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 from the first i rules, r_1 through r_i, we have constructed a partial FDD, whose root v is labelled F_1, and suppose 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 to 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 three 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 $I(e_j)$ doesn't 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}. So 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 that e_j points to.

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 that e_j points to, and let e' and e'' point to one copy each. We then deal with e' by the first case, and e'' by the second case.

In the following pseudocode of the construction algorithm, we use $e.t$ to denote the (target) node that the edge e points to.

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, $(F_m \in S_m) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$)
/*$F(v) = F_m$ 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;
2. **if** $m < d$ **then**
   for $j := 1$ **to** k **do**
     if $I(e_j) \subseteq S_m$ **then**
       APPEND($e_j.t$, $(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$);
     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($e.t$, $(F_{m+1} \in S_{m+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$);

As an example, consider the sequence of rules in Figure 4. Figure 5 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 Figure 6.

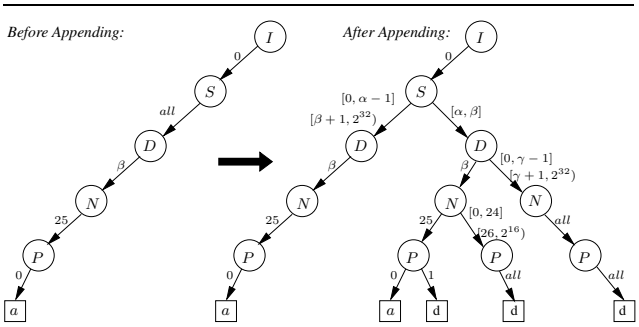

Figure 5. Appending rule $(I \in \{0\}) \wedge (S \in [\alpha, \beta]) \wedge (D \in all) \wedge (N \in all) \wedge (P \in all) \rightarrow d$

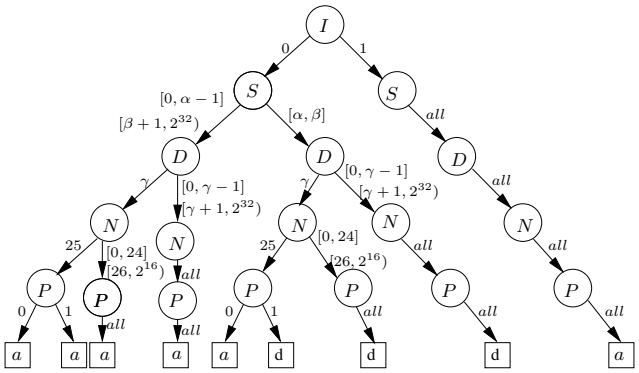

Figure 6. The FDD constructed from Figure 4

5. Shaping Algorithm

In this section we discuss how to transform two ordered, but not semi-isomorphic FDDs f_a and f_b to 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. We define ordered FDDs and semi-isomorphic FDDs as follows.

Definition 5.1 (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 iff for each decision path $(v_1 e_1 \cdots v_k e_k v_{k+1})$, we have $F(v_1) \prec \cdots \prec F(v_k)$.* □

From this definition, the FDDs constructed by the construction algorithm in Section 4 are ordered. Therefore, if a firewall f designed by a team is a non-ordered FDD f, we first generate a sequence of rules that consists of all the rules in S_f, where S_f is the set of all the rules defined by the decision paths of f; second, we construct an equivalent ordered FDD f' from the sequence of rules. Then use f', instead of f, to compare with other firewalls.

Informally, two FDDs are semi-isomorphic if their graphs are isomorphic, the labels of their corresponding nonterminal nodes match, and the labels of their corresponding edges match. In other words, only the labels of their terminal nodes may differ. Formally:

Definition 5.2 (Semi-isomorphic FDDs) *Two FDDs f and f' are semi-isomorphic iff there exists a one-to-one mapping σ from the nodes of f onto the nodes of f', such that the following two 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 to two semi-isomorphic FDDs uses the following three 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 labelled with a field F, then we can insert a node v' labelled 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.

5.1. FDD Simplifying

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

Definition 5.3 (Simple FDDs) *An FDD is simple iff each node in the FDD has at most one incoming edge and each edge in the FDD is labelled 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).

5.2. Node Shaping

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

Definition 5.4 (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 iff one of the following two conditions holds:*

1. *Both v_a and v_b have no parents, i.e., 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.* □

For example, the two nodes labelled F_1 in Figure 7 are shapable since they have no parents.

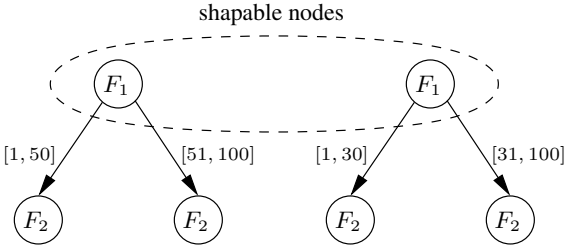

Figure 7. Two shapable nodes in two FDDs

Definition 5.5 (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 iff one of the following two 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.* □

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

1. Step I: 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 $F(v_a) \prec F(v_b)$. It

is straightforward to show that in this case along all the decision paths containing node v_b, no node is labelled $F(v_a)$. 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 have the same label as v'_b. (Recall that this node insertion operation leaves the semantics of the FDD unchanged.)

2. Step II: 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 its corresponding child of v_b are shapable.

   Suppose $D(F(v_a)) = D(F(v_b)) = [a, b]$. We know that each outgoing edge of v_a or v_b is labelled with a single interval. Suppose 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 $a_{i+1} = b_i + 1$. Also suppose 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 two cases: (1) $b_1 = b'_1$: In this case $I(e_1) = I(e'_1)$, therefore, node $e_1.t$ and node $e'_1.t$ are shapable. (Recall that we use $e.t$ to denote the node that edge e points to.) Then we can continue to compare 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 $b_1 < b'_1$. In this case, we split e'_1 into two edges e and e', where e is labelled $[a, b_1]$ and e' is labelled $[b_1 + 1, b'_1]$. 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 to compare 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 labelled 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 ends 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.

In the following 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 $I(e')$.

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 $\sim$(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

If we apply the above node shaping procedure to the two shapable nodes labelled F_1 in Figure 7, we make them semi-isomorphic as shown in Figure 8.

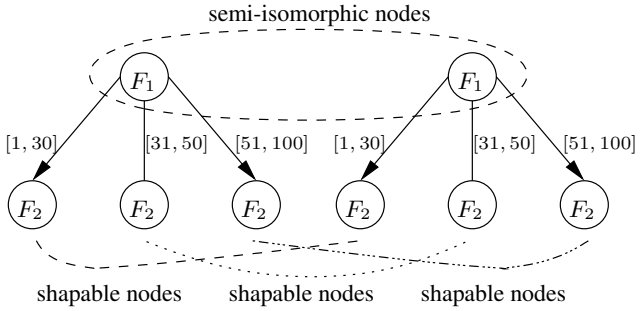

Figure 8. Two semi-isomorphic nodes

5.3. FDD Shaping

To make two ordered FDDs f_a and f_b semi-isomorphic, at first we make f_a and f_b simple, then we make f_a and f_b semi-isomorphic as follows. Suppose 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, 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 as follows:

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

As an example, if we apply the above shaping algorithm to the two FDDs in Figure 3 and 6, we obtain two semi-isomorphic FDDs. One of those FDDs is shown in Figure 9, and the other one is identical to the one in Figure 9 with one exception: the labels of the black terminal nodes are reversed.

6. Comparison Algorithm

In this section, we consider how to compare two semi-isomorphic FDDs. Given two semi-isomorphic FDDs f and f' with a one-to-one mapping σ, each decision path $(v_1 e_1 \cdots v_k e_k v_{k+1})$ in f has a corresponding decision path $(\sigma(v_1)\sigma(e_1)\cdots\sigma(v_k)\sigma(e_k)\sigma(v_{k+1}))$ in f'. 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 S_f 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 $S_{f'}$. Note that $F(v_i) = F(\sigma(v_i))$ 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 S_f, the corresponding rule in $S_{f'}$ 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 two 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 to the rules defined by the decision paths in f'. Note that for each rule and its companion, either they are identical, or they have the same predicate but different decisions. Therefore, $S_{f_a} - S_{f_b}$ is the set of all the rules in S_{f_a} that have different decisions from their companions. Similarly for $S_{f_b} - S_{f_a}$. Note that the set of all the companions of the rules in $S_{f_a} - S_{f_b}$ is $S_{f_b} - S_{f_a}$; and similarly the set of all the companions of the rules in $S_{f_b} - S_{f_a}$ is $S_{f_a} - S_{f_b}$. Since these two sets manifest the discrepancies between the two FDDs, the two design teams can investigate them to resolve the discrepancies.

Let f_a be the FDD in Figure 9, and let f_b be the FDD that is identical to f_a with one exception: the labels of the black terminal nodes are reversed. Here f_a is equivalent to the firewall in Figure 3 designed by Team A, and f_b is equivalent to the firewall in Figure 4 designed by Team B. By comparing f_a and f_b, we discover the following discrepancies between the two firewalls designed by Team A and Team B:

1. $(I \in \{0\}) \wedge (S \in [\alpha, \beta]) \wedge (D \in \{\gamma\}) \wedge (N \in \{25\}) \wedge (P \in \{0\}) \rightarrow d$ in f_a / a in f_b
   Question to investigate: Should we allow the computers from the malicious domain send email to the mail server? Team A says no, while Team B says yes.
2. $(I \in \{0\}) \wedge (S \in [0, \alpha - 1] \cup [\beta + 1, 2^{32})) \wedge (D \in \{\gamma\}) \wedge (N \in \{25\}) \wedge (P \in \{1\}) \wedge \rightarrow d$ in f_a / a in f_b
   Question to investigate: Should we allow UDP packets with destination port number 25 sent from the hosts that are not in the malicious domain to the mail server? Team A says no, while Team B says yes.
3. $(I \in \{0\}) \wedge (S \in [0, \alpha - 1] \cup [\beta + 1, 2^{32})) \wedge (D \in \{\gamma\}) \wedge (N \in [0, 24] \cup [26, 2^{16})) \wedge (P \in all) \wedge \rightarrow d$ in f_a / a in f_b
   Question to investigate: Should we allow the packets with a destination port number other than 25 be sent

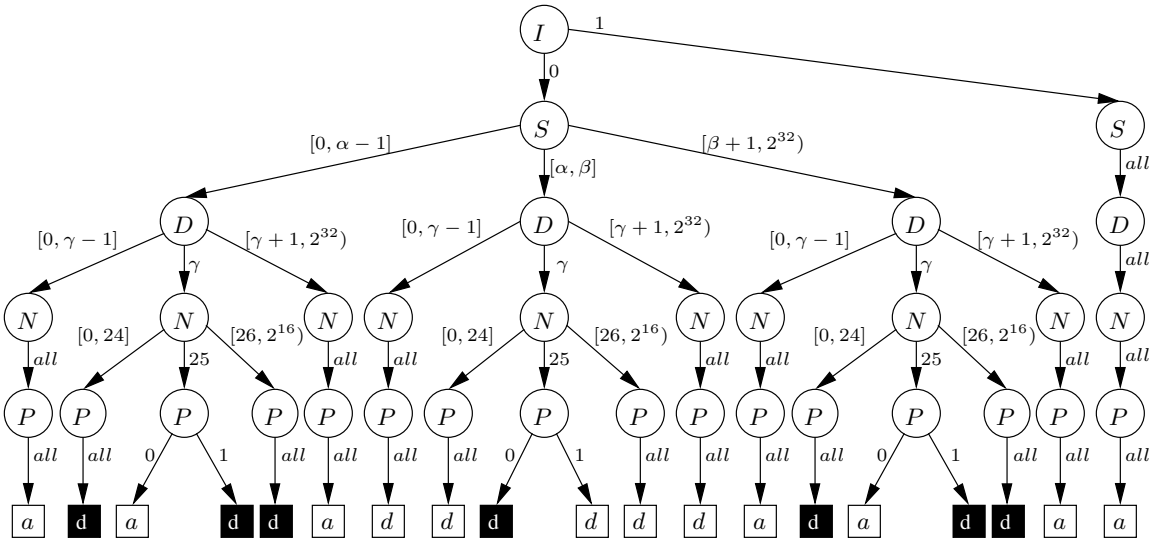

Figure 9. The FDD transformed from the FDD in Figure 3

from the hosts who are not in the malicious domain to the mail server? Team A says no, while Team B says yes.

7. Experimental Results

In this paper we presented three algorithms, a construction algorithm, a shaping algorithm and a comparison algorithm. These three algorithms can be used to detect all discrepancies between two given firewalls. In this section, we evaluate the efficiency of each of these three algorithms.

The construction algorithm is evaluated by the average time for constructing an FDD from a sequence of rules. The shaping algorithm is evaluated by the average time for shaping two FDDs where each is an FDD constructed from a sequence rules that we generate independently. The comparison algorithm is measured by the average time for detecting all the discrepancies between two semi-isomorphic FDDs that we get from the shaping algorithm. In the absence of publicly available firewalls, we create synthetic firewalls based on the characteristics of real-life packet classifiers discovered in [9, 17]. Each rule has the following five fields: interface, source IP address, destination IP address, destination port number and protocol type.

The programs are implemented in SUN Java JDK 1.4. The experiments were carried out on a SunBlade 2000 machine running Solaris 9 with 1Ghz CPU and 1 GB memory. Figure 10 shows the average execution times for the construction algorithm, for the shaping algorithm, and for the comparison algorithm versus the total number of rules. We also measured the average total time for detecting all the discrepancies between two sequences of rules, which includes the time for 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. From this figure, we see that it takes less than 5 seconds to detect all the discrepancies between two sequences of 3000 rules. In fact, it is very unlikely that a firewall can have this many rules (see the characteristics of real-life packet classifiers in [9, 17]). Clearly the efficiency of our three algorithms make them attractive to be used in practice for supporting our diverse firewall design method.

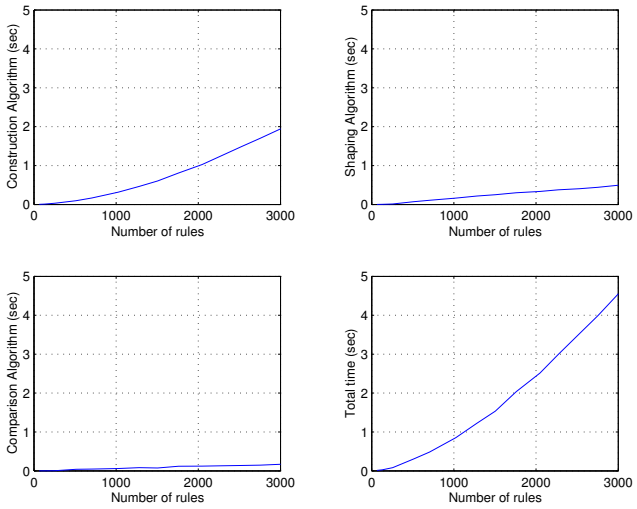

Figure 10. Experimental Results

8. Conclusions

In this paper, we propose the method of diverse firewall design, and present a series of three algorithms (a construction algorithm, a shaping algorithm and a comparison algorithm) for detecting all discrepancies between two given firewalls. The experimental results show that these algorithms are very efficient. It takes only about five seconds to detect all the discrepancies between two firewalls where each has 3000 rules. We believe that our method of diverse firewall design and the three algorithms will be used to design firewalls whose correctness is important.

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

Since a firewall is just one type of packet classifiers, the diverse firewall design method that we propose in this paper can be used for designing other packet classifiers. This generalization is straightforward.

References

[1] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly detection and rule editing. In *IEEE/IFIP Integrated Management IM'2003*, March 2003.

[2] E. Al-Shaer and H. Hamed. Management and translation of filtering security policies. In *IEEE International Conference on Communications*, May 2003.

[3] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE INFOCOM'04*, March 2004.

[4] A. W. Alain Mayer and E. Ziskind. Fang: A firewall analysis engine. In *Proc. of IEEE Symp. on Security and Privacy*, pages 177–187, 2000.

[5] H. Anderson and G. Hagelin. Computer controlled interlocking system. *Ericsson Review*, (2), 1981.

[6] A. Avizienis. The n-version approach to fault tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[7] A. Avizienis. The methodology of n-version programming. *Chapter 2 of Software Fault Tolerance, M. R. Lyu (ed.), Wiley, 23-46*, 1995.

[8] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. In *Proc. of Intl. Computer software and Appl. Conf.*, pages 145–155, 1977.

[9] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to cams? In *Proc. of IEEE INFOCOM*, 2003.

[10] F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. In *Proc. of the 10th IEEE International Conference on Network Protocols*, 2002.

[11] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *Proc. of IEEE Symp. on Security and Privacy*, pages 17–31, 1999.

[12] A. Condor and G. Hinton. Fault tolerant and fail-safe design of candu computerised shutdown systems. *IAEA Specialist Meeting on Microprocessors important to the Safety of Nuclear Power Plants,*, May 1988.

[13] D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Symp. on Discrete Algorithms*, pages 827–835, 2001.

[14] D. Farmer and 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.

[15] M. Freiss. *Protecting Networks with SATAN*. O'Reilly & Associates, Inc., 1998.

[16] M. G. Gouda and A. X. Liu. Firewall design: consistency, completeness and compactness. In *Proc. of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, pages 320–327, March 2004.

[17] P. Gupta. *Algorithms for Routing Lookups and Packet Classification*. PhD thesis, Stanford University, 2000.

[18] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *Proc. of IEEE Symp. on Security and Privacy*, pages 120–129, 1997.

[19] A. Hari, S. Suri, and G. M. Parulkar. Detecting and resolving packet filter conflicts. In *Proc. of IEEE INFOCOM*, pages 1203–1212, 2000.

[20] S. Hazelhurst, A. Attar, and R. Sinnappan. Algorithms for improving the dependability of firewall and filter rule lists. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'00)*, pages 576–585, 2000.

[21] J. D. Moffett and M. S. Sloman. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, 4(1):1–22, 1994.

[22] Nessus. http://www.nessus.org/. March 2004.

[23] X. Teng and H. Pham. A software-reliability growth model for n-version programming systems. *IEEE Transactions on Reliability*, 51(3):311–321, 2002.

[24] P. Traverse. Airbus and atr system architecture and specification. *Software Diversity in Computerised Control Systems, U. Voges (ed.), Springer Verlag*, 1988.

[25] M. A. Vouk. On back-to-back testing. In *Proc. of Annual Conference on Computer Assurance (COMPASS)*, pages 84–91, 1988.

[26] M. A. Vouk. On growing software reliability using back-to-back testing. In *Proc. 11th Minnowbrook Workshop on Software Reliability*, 1988.