First Step Towards Automatic Correction of Firewall Policy Faults

FEI CHEN and ALEX X. LIU
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824-1266, U.S.A.
{feichen, alexliu}@cse.msu.edu
and
JEEHYUN HWANG and TAO XIE
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695, U.S.A.
jhwang4@ncsu.edu, xie@csc.ncsu.edu

Firewalls are critical components of network security and have been widely deployed for protecting private networks. A firewall determines whether to accept or discard a packet that passes through it based on its policy. However, most real-life firewalls have been plagued with policy faults, which either allow malicious traffic or block legitimate traffic. Due to the complexity of firewall policies, manually locating the faults of a firewall policy and further correcting them are difficult. Automatically correcting the faults of a firewall policy is an important and challenging problem. In this paper, we first propose a fault model for firewall policies including five types of faults. For each type of fault, we present an automatic correction technique. Second, we propose the first systematic approach that employs these five techniques to automatically correct all or part of the misclassified packets of a faulty firewall policy. Third, we conducted extensive experiments to evaluate the effectiveness of our approach. Experimental results show that our approach is effective to correct a faulty firewall policy with three of these types of faults.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms: Algorithm, Performance, Security

Additional Key Words and Phrases: Automatic Fault Fixing, Firewall Faults, Firewall Policy

The preliminary version of this paper titled "First Step Towards Automatic Correction of Firewall Policy Faults" was published in USENIX LISA 2010 [Chen et al. 2010]. This work is supported in part by NSF grant CNS-0716579, NSF grant CNS-0716407, an MSU IRGP Grant, and an NIST grant. Alex X. Liu is the corresponding author. Tel: 517-353-5152.

1. INTRODUCTION

1.1 Motivation

Firewalls serve as critical components for securing the private networks of business, institutions, and home networks. A firewall is often placed at the entrance between a private network and the outside Internet so that it can check all incoming and outgoing packets and decide whether to accept or discard a packet based on its policy. A firewall policy is usually specified as a sequence of rules that follow the first-match semantics where the decision for a packet is the decision of the first rule that the packet matches. However, most real-life firewall policies are poorly configured and contain faults (*i.e.*, misconfigurations) [Wool 2004]. A policy fault either creates security holes that allow malicious traffic to sneak into a private network or blocks legitimate traffic and disrupts normal business processes. In other words, a faulty firewall policy evaluates some packets to unexpected decisions. We call such packets *misclassified packets* of a faulty firewall policy. Therefore, it is important to develop an approach that can assist firewall administrators to automatically correct firewall faults. Besides, the automatic fault fixing techniques are under-investigated in many areas of computer and networking, and these techniques are very important for building self-organizing systems and autonomic computing systems. Because in reality, we cannot avoid errors or faults in any computer or networking systems. Without automatic fault fixing techniques, one fault may cause a system to fail. Thus, we hope to attract attention from the research community on this important, yet challenging, problem.

1.2 Technical Challenges

There are three key challenges for automatic correction of firewall policy faults. First, it is difficult to determine the number of policy faults and the type of each fault in a faulty firewall. The reason is that a set of misclassified packets can be caused by different types of faults and different number of faults. Second, it is difficult to correct a firewall fault. A firewall policy may consist of a large number of rules (*e.g.*, thousands of rules) and each rule has a predicate over multi-dimensional fields. Locating a fault in a large number of rules and further correcting it by checking the field of each dimension are two difficult tasks. Third, it is difficult to correct a fault without introducing other faults Due to the first-match semantics of firewall policies, correcting a fault in a firewall rule affects the functionality of all the subsequent rules, and hence may introduce other faults into the firewall policy.

1.3 Our Approach

To correct a faulty firewall policy, essentially we need to correct all *misclassified packets* of the policy such that all these packets will be evaluated to expected decisions. However, it is not practical to manually find every misclassified packet and then correct it due to the large number of misclassified packets of the faulty policy.

The idea of our approach is that we first find some samples of all the misclassified packets and then use these samples to correct all or part of the misclassified packets of the faulty policy. We propose the first comprehensive fault model for firewall policies. The proposed fault model includes five types of faults, *wrong order*, *missing rules*, *wrong decisions*, *wrong predicates*, and *wrong extra rules*. For each type of

fault, we propose a correction technique based on the passed and failed tests of a firewall policy. Passed tests are packets that are evaluated to expected decisions. Failed tests are packets that are evaluated to unexpected decisions. Note that the failed tests are samples of all misclassified packets.

To generate passed and failed tests, we first employ automated packet generation techniques [Hwang et al. 2008] to generate test packets for a faulty firewall policy. The generated packets can achieve high structural coverage, *i.e.*, covering all or most rules [Hwang et al. 2008]. Second, administrators classify these packets into passed and failed tests by checking whether their evaluated decisions are correct. Identifying passed/failed tests can be automated in some situations, *e.g.*, when policy properties are written, or multiple implementations of the policy are available. Even if this operation cannot be done automatically, manual inspection of passed/failed tests is also common practice for ensuring network security in industry. For example, applying some existing vulnerability testing tools, such as Nessus [Nessus 2004] and Satan [Satan 1995], does need manual inspection. In this paper, our goal is to automatically correct policies after we have passed/failed packets. Identifying passed/failed tests is out of scope of this paper.

Given passed and failed tests, correcting a faulty firewall policy is still difficult because it is hard to identify the number of faults and the type and the location of each fault in the firewall policy. To address this problem, we propose a greedy algorithm. In each step of the greedy algorithm, we try every correction technique and choose one technique that can maximize the number of passed tests. We then repeat this step until there are no failed tests.

Our proposed approach is not guaranteed to correct all faults in a firewall policy because it is practically impossible unless the formal representation of the policy is available. However, in practice, most administrators do not have such formal representations of their firewall policies. To correct a faulty firewall policy without its formal representation, administrators need to examine the decisions of all 2^{104} packets[1] and manually correct each of misclassified packets; doing so is practically impossible. This paper represents the first step towards automatic correction of firewall policy faults. We hope to attract more attention from the research community on this important and challenging problem.

1.4 Key Contributions

Our major contributions can be summarized as below:

(1) We propose the first systematic approach that can automatically correct all or part of the misclassified packets of a faulty firewall policy.
(2) We conduct extensive experiments on real-life firewall policies to evaluate the effectiveness of our approach.

1.5 Summary of Experimental Results

We generated a large number of faulty firewall policies from 40 real-life firewalls, and then applied our approach over each faulty policy and produced the fixed policy.

[1]A packet typically includes five fields, source IP (32 bits), destination IP (32 bits), source port (16 bits), destination port (16 bits), and protocol type (8 bits). Thus, the number of possible packets is $2^{32+32+16+16+8} = 2^{104}$.

Faulty policies with k faults ($1 \leq k \leq 5$) were tested. These faults in a faulty policy were of the same type. The experimental results show that for three types of faults, wrong order, wrong decisions, and wrong extra rules, our approach can effectively correct misclassified packets. When $k \leq 4$, our approach can correct all misclassified packets for over 53.2% faulty policies. This result is certainly encouraging and we hope that this paper will attract more attention from the research community to this important problem. For two other types of faults, missing rules and wrong predicates, our approach does not achieve satisfactory results, deserving further study.

2. RELATED WORK

2.1 Firewall Policy Fault Localization

Fault localization for firewall policies has drawn attention recently [Hwang et al. 2009; Marmorstein and Kearns 2007]. Marmorstein *et al.* proposed a technique to find failed tests that violate the security requirement of a firewall policy and further use the failed tests to locate two or three faulty rules in a firewall policy [Marmorstein and Kearns 2007]. However, they did not provide a systematic methodology to identify faulty rules according to different types of firewall faults, *e.g.*, wrong order of firewall rules. Furthermore, they applied their approach only to a simple firewall policy (with 5 rules), which cannot strongly demonstrate the effectiveness of their approach.

Our previous work proposed a technique to locate a fault in a firewall policy [Hwang et al. 2009]. The approach first analyzes a faulty firewall policy and its failed tests and then finds the potential faulty rules based on structural coverage metrics[2]. However, this work has three limitations: (1) it considers only two types of faults, which are wrong decisions and wrong predicates, while a firewall policy may contain other types of faults; (2) it considers only a firewall policy with a single fault, while a firewall policy may contain multiple faults; (3) it does not propose a technique to correct the faults in a firewall policy.

2.2 Firewall Policy Analysis and Testing

Firewall policy analysis tools, *i.e.*, conflict detection, anomaly detection, and change impact analysis, have been proposed in prior work (*e.g.*, [Al-Shaer and Hamed 2004; Baboescu and Varghese 2002; Hari et al. 2000; Tang et al. 2008; Liu 2007; Yuan et al. 2006]). Tools for detecting potential firewall policy faults by conflict detection were proposed in [Baboescu and Varghese 2002; Hari et al. 2000; Tang et al. 2008]. Similar to conflict detection, some other tools were proposed for detecting anomalies in a firewall policy [Al-Shaer and Hamed 2004; Yuan et al. 2006]. Detecting conflicts or anomalies is helpful for finding faults in a firewall policy. However, the number of conflicts or anomalies could be too large to be manually inspected. Therefore, correcting a faulty policy is difficult by using these firewall policy analysis tools. Change impact analysis of firewall policies has also been studied [Liu 2007]. Such tools are helpful to analyze the impact after changing a firewall policy, but no algorithm has been presented for correcting a faulty firewall policy.

[2]Firewall policy coverage is measured based on which entities (e.g., rules or fields) are involved (called "covered") during packet evaluation.

Firewall policy testing tools have also been explored in prior work (*e.g.*, [CERT 2001; Jürjens and Wimmel 2001; Lyu and Lau 2000; Hoffman and Yoo 2005; Al-Shaer et al. 2009]). Such tools focus on injecting packets as tests into a firewall to detect faults in the firewall policy. If the evaluated decision of a packet is not as expected, faults in the firewall policy are exposed. However, because a firewall policy may have a large number of rules and the rules often conflict, it is difficult to manually locate faults and correct them based on the passed and failed tests.

2.3 Software Fault Localization and Fixing

Fault localization and fixing have been studied for years in the software engineering and programming language communities. Such research focuses on locating and fixing a fault in a software program. For example, Zeller et al. proposed a technique based on delta debugging, which isolates causes based on difference of program runs [Zeller 2002]. While their approach finds likely fault locations (*e.g.*, fault-inducing variables among set of variables) based on difference of program runs, our approach finds and fixes likely fault locations based on an increase in the number of passed tests. Furthermore, using delta debugging in the context of firewall policies is not effective. Delta debugging relies on dependency information (cause-effect chain) of variables. However, in our context, values assigned for fields do not change over evaluation and there is no dependency (i.e., data dependency) among the fields of the firewall policy.

3. BACKGROUND

3.1 Firewall Policies

A firewall policy is a sequence of *rules* $\langle r_1, \cdots, r_n \rangle$ and each *rule* is composed of a *predicate* over d *fields*, $F_1, \cdots, F_d$ and a *decision* for the packets that match the predicate. Figure 1 shows a firewall policy, whose format follows Cisco Access Control Lists [Cisco Reflexive ACLs].

A *field* F_i is a variable of finite length (*i.e.*, of a finite number of bits). The domain of field F_i of w bits, denoted as $D(F_i)$, is $[0, 2^w - 1]$. Firewalls usually check five fields, source IP (32 bits), destination IP (32 bits), source port (16 bits), destination port (16 bits), and protocol type (8 bits). For example, the domain of the source IP is $[0, 2^{32} - 1]$. Note that in this paper we only consider the stateless firewalls which make decisions by checking the packet itself. Stateful firewalls, which make decisions by checking not only the packet but also the packets that were accepted previously, are out of the scope of this paper.

A *packet* p over the d fields $F_1, \cdots, F_d$ is a d-tuple $(x_1, \cdots, x_d)$ where each x_i $(1 \leq i \leq d)$ is an element of $D(F_i)$. An example packet over these five fields is (1.2.3.5, 192.168.1.1, 78, 25, TCP).

A *predicate* defines a set of packets over the fields $F_1, \cdots, F_d$, and is specified as $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$. Each S_i is a subset of $D(F_i)$ and is specified as either a prefix or a range. A *prefix* $\{0,1\}^k\{*\}^{w-k}$ (with k leading 0s or 1s) denotes the range $[\{0,1\}^k\{0\}^{w-k}, \{0,1\}^k\{1\}^{w-k}]$. For example, prefix 01** denotes the range $[0100, 0111]$.

A *decision* is an action for the packets that match the predicate of the rule. For firewalls, the typical decisions include *accept*, *discard*, *accept with logging*, and *discard with logging*.

Note that some firewall policies may not follow the above firewall model. For example, a Linux firewall policy may consist of multiple chains and each chain consists of a sequence of rules with first-match semantics. The decision of a rule could be *jump to other chain*. The stateless firewall policy in Linux can also be converted to our firewall policy model. For the rules in each chain of a Linux firewall policy, it is trivial to convert their predicates into our firewall model. Without loss of generality, we assume that the decision of a rule r_i in chain c_1 is *jump to chain* c_2 and chain c_2 consists of 2 rules r_{j_1}, r_{j_2} with the default decision *discard*. Rule r_i in chain c_1 and chain c_2 are shown as follows.

Rule r_i in chain c_1
$r_i : F_1 \in S_1^i \wedge \cdots \wedge F_d \in S_d^i \rightarrow$ `jump to` c_2

Chain c_2 with the default decision *discard*
$r_{j_1} : F_1 \in S_1^{j_1} \wedge \cdots \wedge F_d \in S_d^{j_1} \rightarrow decision^{j_1}$
$r_{j_2} : F_1 \in S_1^{j_2} \wedge \cdots \wedge F_d \in S_d^{j_2} \rightarrow decision^{j_2}$

Then we can convert rule r_i to three rules $r_{i_1}, r_{i_2}, r_{i_3}$ as follows:

$$\begin{array}{lll} r_{i_1} : F_1 \in (S_1^i \cap S_1^{j_1}) & \wedge \cdots \wedge F_d \in (S_d^i \cap S_d^{j_1}) & \rightarrow \langle decision^{j_1} \rangle \\ r_{i_2} : F_1 \in (S_1^i \cap S_1^{j_2}) & \wedge \cdots \wedge F_d \in (S_d^i \cap S_d^{j_2}) & \rightarrow \langle decision^{j_2} \rangle \\ r_{i_3} : F_1 \in S_1^i & \wedge \cdots \wedge F_d \in S_d^i & \rightarrow \langle discard \rangle \end{array}$$

It is trivial to prove that the functionality of the three rules $r_{i_1}, r_{i_2}, r_{i_3}$ is equivalent to that of rule r_i with the decision *jump to* c_2. Thus, we can replace r_i with $r_{i_1}, r_{i_2}, r_{i_3}$ in chain c_1. Repeat the above steps for rules with the decision *jump to other chain*. Finally we can convert a stateless Linux firewall policy into our firewall model.

A packet $(x_1, \cdots, x_d)$ *matches* a rule $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ if and only if the condition $x_1 \in S_1 \wedge \cdots \wedge x_d \in S_d$ holds. For example, the packet (1.2.3.5, 192.168.1.1, 78, 25, TCP) matches the rule r_1 in Figure 1.

A rule $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ is called a *singleton rule* if and only if each S_i has only one element.

Rule	Src. IP	Dest. IP	Src. Port	Dest. Port	Prot.	Dec.
r_1	1.2.3.*	192.168.1.1	*	25	TCP	accept
r_2	*	*	*	*	*	discard

Fig. 1. An example firewall

A firewall policy $\langle r_1, \cdots, r_n \rangle$ is *complete* if and only if for any packet p, there is at least one rule that p matches. To ensure that a firewall policy is complete, the predicate of the last rule is usually specified as $F_1 \in D(F_1) \wedge \cdots \wedge F_d \in D(F_d)$, *i.e.*, the last rule r_2 in Figure 1.

Two rules in a firewall policy may *overlap*; that is, there exists at least one packet that matches both rules. Two rules may *conflict*; that is, the two rules not only overlap but also have different decisions. For example, in Figure 1, two rules r_1, r_2 overlap and conflict because the packet (1.2.3.5, 192.168.1.1, 78, 25, TCP) matches r_1 and r_2, and the decisions of r_1 and r_2 are different.

Firewalls typically resolve conflicts by employing the *first-match semantics* where the decision for a packet p is the decision of the first (i.e., highest priority) rule that p matches in the firewall policy. Thus, for the packet (1.2.3.5, 192.168.1.1, 78, 25, TCP), the decision of the firewall policy in Figure 1 is accept.

3.2 Packet Generation

To check the correctness or detect faults in a firewall policy, administrators need to generate test packets to evaluate that the policy is correct. In our previous work [Hwang et al. 2008], we developed automated packet generation techniques to achieve high structural coverage. One cost-effective technique is packet generation based on local constraint solving. In this paper, we use this technique to generate packets for firewall policies. This technique statically analyzes rules to generate test packets. Given a policy, the packet generator analyzes the predicate in an individual rule and generates packets to evaluate the constraints (i.e., rule fields) to be true or false. The generator first constructs constraints to evaluate each field in a rule to be either false or true, and then it generates a packet based on the concrete values derived by constraint solving. For example, given rule r_1 in Figure 1, the generator analyzes r_1 and generates a packet (e.g., packet (1.2.3.5, 192.168.1.1, 23447, 25, TCP)) to cover r_1; this packet evaluates each of r_1's fields to be true during evaluation. Then, the generator analyzes r_2 and generates a packet (e.g., packet (2.2.3.5, 192.168.1.1, 23447, 26, UDP)) to cover r_2; this packet evaluates each of r_2's fields to be true during evaluation. When firewall policies do not include many conflicts, this technique can effectively generate packets to achieve high structural coverage.

4. A FAULT MODEL OF FIREWALL POLICES

A fault model of firewall policies is an explicit hypothesis about potential faults in firewall policies. Our proposed fault model includes five types of faults.

(1) *Wrong order.* This type of fault indicates that the order of rules is wrong. Recall that the rules in a firewall policy follow the first-match semantics due to conflicts between rules. Misordering firewall rules can misconfigure a firewall policy. Wrong order of rules is a common fault caused by adding a new rule at the beginning of a firewall policy without carefully considering the order between the new rule and the original rules. For example, if we swap r_1 and r_2 in Figure 1, all packets will be discarded.

(2) *Missing rules.* This type of fault indicates that administrators need to add new rules to the original policy. Usually, administrators add a new rule regarding a new security concern. However, sometimes they may forget to add the rule to the original firewall policy.

(3) *Wrong predicates.* This type of fault indicates that predicates of some rules are wrong. When configuring a firewall policy, administrators define the predicates of rules based on security requirements. However, some special cases may be overlooked.

(4) *Wrong decisions.* This type of fault indicates that the decisions of some rules are wrong.

(5) *Wrong extra rules*. This type of fault indicates that administrators need to delete some rules from the original policy. When administrators make some changes to a firewall policy, they may add a new rule but sometimes forget to delete old rules that filter a similar set of packets as the new rule does.

In this paper, we consider faults in a firewall policy that can be represented as a set of misclassified packets. Under this assumption, given a set of misclassified packets, we can always find one or multiple faults in our fault model that can generate the same set of misclassified packets. One simple way to find such faults is that for each misclassified packet, we consider that the faulty policy misses a singleton rule for this misclassified packet. Therefore, we can always find multiple *missing rules* faults that can generate the same set of misclassified packets.

The correction techniques for these five types of faults are called *wrong-order correction*, *missing-rule correction*, *wrong-predicate correction*, *wrong-decision correction*, and *wrong-extra-rule correction*, respectively. Each operation in these five techniques is called a *modification*.

5. AUTOMATIC CORRECTION OF FIREWALL POLICY FAULTS

Normally, a faulty firewall policy is detected when administrators find that the policy allows some malicious packets or blocks some legitimate packets. Because the number of these observed malicious packets or legitimate packets is typically small, these packets may not provide enough information about the faults in the firewall policy, and hence correcting the policy with these packets is difficult. Therefore, after finding a faulty firewall policy, we first employ the automated packet generation techniques [Hwang et al. 2009], which can achieve high structural coverage, to generate test packets for the faulty policy. Second, administrators identify passed/failed tests automatically or manually. According to security requirements for the firewall policy, if the decision of a packet is correct, administrators classify it as a passed test; otherwise, administrators classify it as a failed test. In some situations, *e.g.*, when some policy properties are written, classifying some test packets can be automated. Manual inspection is also a common practice for ensuring network security in industry. For example, applying some existing vulnerability testing tools, such as Nessus [Nessus 2004] and Satan [Satan 1995], does need manual inspection. Our goal is to automatically correct policies after we have passed/failed packets. Identifying passed/failed tests is out of the scope of this paper.

Figure 2 shows a faulty firewall policy and its passed and failed tests. This policy includes 5 rules over two fields F_1 and F_2, where the domain of each field is [1,10].

We use a as a shorthand for "accept" and d as a shorthand for "discard". For the passed and failed tests, we use a and d to denote expected decisions. We assign each test a distinct ID p_i ($1 \leq i \leq 8$).

Given passed and failed tests, it is difficult to automatically correct a faulty firewall policy for three reasons. First, it is difficult to locate the faults because a firewall policy may consist of a large number of rules, and the rules often conflict. Second, before correcting a fault, we need to first determine the type of the fault and then use the corresponding correction technique to fix this fault. However, it is difficult to determine the type of a fault because the same misbehavior of a firewall policy, *i.e.*, the same set of misclassified packets, can be caused by different

$r_1 : F_1 \in [1,\ 5] \wedge F_2 \in [1, 10] \rightarrow a$
$r_2 : F_1 \in [1,\ 6] \wedge F_2 \in [3, 10] \rightarrow a$
$r_3 : F_1 \in [6, 10] \wedge F_2 \in [1,\ 3] \rightarrow d$
$r_4 : F_1 \in [7, 10] \wedge F_2 \in [4,\ 8] \rightarrow a$
$r_5 : F_1 \in [1, 10] \wedge F_2 \in [1, 10] \rightarrow d$

(a) An example faulty firewall policy

$p_1 : (3,\ 2) \rightarrow a$
$p_2 : (5,\ 7) \rightarrow a$
$p_3 : (6,\ 7) \rightarrow a$
$p_4 : (7,\ 2) \rightarrow d$
$p_5 : (8, 10) \rightarrow d$

(b) A set of passed tests

$p_6 : (6,\ 3) \rightarrow d$
$p_7 : (7,\ 9) \rightarrow a$
$p_8 : (8,\ 5) \rightarrow d$

(c) A set of failed tests

Fig. 2. An example faulty firewall policy with its failed and passed tests

types of faults. Third, it is difficult to correct a fault. Due to the first-match semantics, changing a rule can affect the functionality of all the subsequent rules. Without thorough consideration, correcting a fault may introduce a new fault into the firewall policy.

In this paper, we formalize the problem of correcting a faulty firewall policy as follows.

5.1 Problem Statement

Given a faulty firewall policy FW, a set of passed tests PT, and a set of failed tests FT, where $|PT| \geq 0$ and $|FT| > 0$, find a sequence of modifications $\langle M_1, \cdots, M_m \rangle$, where M_j $(1 \leq j \leq m)$ denotes one modification, such that the following two conditions hold:

(1) After applying $\langle M_1, \cdots, M_m \rangle$ to FW, all tests in $PT \cup FT$ become passed tests.
(2) No other sequence that satisfies the first condition has a smaller number of modifications than m.

Correcting a faulty firewall policy with the minimum number of modifications is a global optimization problem and hard to solve because the policy may consist of a large number of rules, and different combinations of modifications can be made. To address this problem, we propose two algorithms: a greedy algorithm and its improved algorithm that considers the preference of different modifications.

5.2 Greedy Algorithm

In the greedy algorithm, for each step, we correct one fault in the policy such that the number of passed tests increases (in other words, the number of failed tests decreases). To determine which correction technique should be used at each step, we try the five correction techniques. Then, we calculate the number of passed tests for each type of modifications and choose the correction technique that corresponds to the maximum number of passed tests. We then repeat the preceding step until there are no failed tests. Figure 3 illustrates this greedy algorithm for automatic correction of firewall policy faults.

Our greedy algorithm can guarantee to find a sequence of modifications that satisfies the first condition in the problem statement. For each step, the greedy algorithm can increase at least one passed test because of the *missing-rule correction*

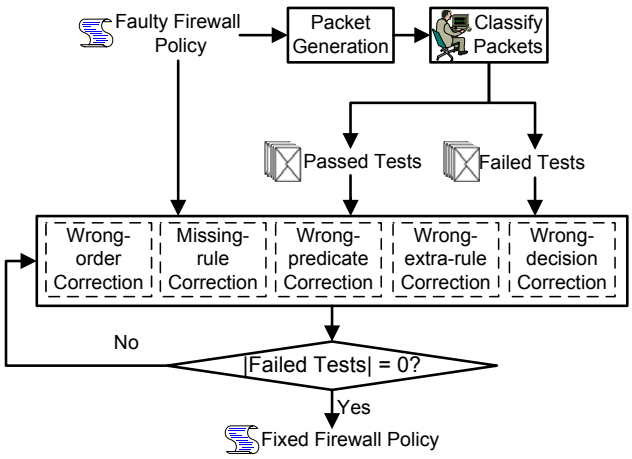


Fig. 3. The greedy algorithm for automatically correcting a faulty firewall policy

technique. Using this technique, we can convert each failed test to a singleton rule and then add these singleton rules at the beginning of the faulty firewall policy. For example, convert the failed test $(6,\ 3) \rightarrow d$ in Figure 2(c) to a singleton rule $F_1 \in [6,6] \wedge F_2 \in [3,3] \rightarrow d$. Note that adding a singleton rule for each failed test is the worst case of our greedy algorithm, *i.e.*, the maximum number of modifications, while in most cases our greedy algorithm rarely generates singleton rules. According to our missing-rule correction technique in Section 7, the worst case happens if and only if in each round of our greedy algorithm two conditions hold: (1) only missing-rule correction technique can increase the number of passed tests while other four fixing techniques cannot; (2) missing-rule correction technique cannot add a firewall rule, which is not a singleton rule, to correct multiple failed tests without changing passed tests to failed tests. These two conditions are very difficult to achieve in reality. However, the greedy algorithm cannot guarantee to find the global optimization solution that satisfies the second condition in the problem statement.

5.3 Improved Algorithm

In the improved algorithm, we consider correction technique preferences over five correction techniques. We prefer to correct the faulty firewall policy using three techniques, wrong-order correction, wrong-decision correction, and wrong-extra-rule correction. Figure 4 illustrates the improved algorithm for automatic correction of firewall policy faults. In this algorithm, for each step, we first try three techniques, wrong-order correction, wrong-decision correction, and wrong-extra-rule correction, and then find out which one can decrease the maximum number of failed tests. If none of them can decrease the number of failed tests, we try two other correction techniques, missing-rule correction and wrong-predicate correction.

Note that administrators can supervise this process. For each step, administrators can choose their preferred technique for correcting a fault in the policy. If administrators do not want to supervise the process, our greedy algorithm can automatically produce the fixed policy.

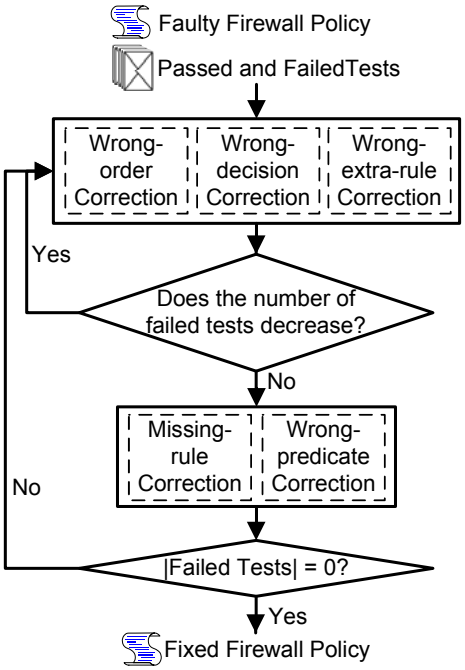


Fig. 4. The improved algorithm for automatically correcting a faulty firewall policy

Further note that without any restriction, our automatic approach for correcting firewall policy faults could introduce potential faults in the firewall policy. However, an administrator typically has some critical requirements when he/she designs the firewall policy. These critical requirements define that some packets should be accepted or discarded. The administrator can restrict the proposed approach not to violate the critical requirements. Consider a critical requirement that a data server in an organization should not be accessed by any outside connection. For each step of our greedy algorithm, if the modification generated in this step violates the requirement, the approach can simply choose the next modification that does not violate the requirement.

In the next five sections, we discuss our scheme for each correction technique, respectively. Recall that the last rule of a firewall policy is usually specified as $F_1 \in D(F_1) \wedge \cdots \wedge F_d \in D(F_d) \rightarrow \langle decision \rangle$. Checking whether the last rule is correct is trivial. Therefore, we assume that the last rule of a firewall policy is correct in our discussion.

6. WRONG-ORDER CORRECTION

Due to the first-match semantics, changing the order of two rules in a firewall policy (*i.e.*, swapping two rules) affects its functionality. Therefore, after swapping two rules of a firewall policy, we need to test and reclassify all passed tests and failed tests. It is computationally expensive to directly swap every two rules in a faulty firewall policy and then find the two rules such that swapping them can maximize colorredthe increase in the number of passed tests. Given a firewall policy with n

rules, without considering the last rule, there are $(n-1)(n-2)/2$ pairs of rules that can be swapped. Furthermore, for each swapping, we need to reclassify all passed and failed tests. Assume that the number of passed tests is m_1 and the number of failed tests is m_2. The computational cost of this brute-force way is $(n-1)(n-2)(m_1+m_2)/2$.

To address this challenge, we use all-match firewall decision diagrams (all-match FDDs) [Liu et al. 2008] as the core data structure. An all-match FDD is a canonical representation of a firewall policy such that any firewall policy can be converted to an equivalent all-match FDD. Figure 5 shows the all-match FDD converted from the faulty firewall policy in Figure 2. An all-match FDD for a firewall policy $FW{:}\langle r_1, \cdots, r_n\rangle$ over attributes $F_1, \cdots, F_d$ is an acyclic and directed graph that has the following five properties:

(1) There is exactly one node that has no incoming edges. This node is called the *root*. The nodes that have no outgoing edges are called *terminal* nodes.

(2) Each node v has a label, denoted as $F(v)$. If v is a nonterminal node, then $F(v) \in \{F_1, \cdots, F_d\}$. If v is a terminal node, then $F(v)$ is a list of integer values $\langle i_1, \cdots, i_k\rangle$ where $1 \leq i_1 < \cdots < i_k \leq n$.

(3) Each edge $e{:}u \rightarrow v$ is labeled with a nonempty set of integers, denoted as $I(e)$, where $I(e)$ is a subset of the domain of u's label (*i.e.*, $I(e) \subseteq D(F(u))$). The set of all outgoing edges of a node v, denoted as $E(v)$, satisfies two conditions: (1) *consistency*: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$; (2) *completeness*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$.

(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. Given a decision path $\mathcal{P}{:}(v_1 e_1 \cdots v_d e_d v_{d+1})$, the matching set of $\mathcal{P}$ is defined as the set of all packets that satisfy $F(v_1) \in I(e_1) \wedge \cdots \wedge F(v_d) \in I(e_d)$. We use $C(\mathcal{P})$ to denote the matching set of $\mathcal{P}$.

(5) For any decision path $\mathcal{P} : (v_1 e_1 \cdots v_d e_d v_{d+1})$ where $F(v_{d+1}) = \langle i_1, \cdots, i_k\rangle$, if $C(\mathcal{P}) \cap C(r_j) \neq \emptyset$, $C(\mathcal{P}) \subseteq C(r_j)$ and $j \in \{i_1, \cdots, i_k\}$.

For ease of presentation, we use $\{\mathcal{P}_1, \cdots, \mathcal{P}_h\}$ to denote the all-match FDD of the firewall policy FW. The following theorem is based on this definition, the proof of which is in Appendix A.

THEOREM 6.1. *Given two firewall policies $FW_1{:}\langle r_1^1, \cdots, r_n^1\rangle$ and $FW_2{:}\langle r_1^2, \cdots, r_n^2\rangle$, and their all-match FDDs $\{\mathcal{P}_1^1, \cdots, \mathcal{P}_{h_1}^1\}$ and $\{\mathcal{P}_1^2, \cdots, \mathcal{P}_{h_2}^2\}$, if $\{r_1^1, \cdots, r_n^1\} = \{r_1^2, \cdots, r_n^2\}$, without considering terminal nodes, $\{\mathcal{P}_1^1, \cdots, \mathcal{P}_{h_1}^1\} = \{\mathcal{P}_1^2, \cdots, \mathcal{P}_{h_2}^2\}$.*

According to Theorem 6.1, for swapping two rules, we only need to swap the sequence numbers of the two rules in the terminal nodes of the all-match FDD. For finding two rules such that swapping them maximizes the number of passed tests, our correction technique includes five steps:

(1) Convert the policy to an equivalent all-match FDD.

(2) For each failed test p, we find the decision path $\mathcal{P}$: $(v_1 e_1 \cdots v_d e_d v_{d+1})$ that matches p (*i.e.*, $p \in C(\mathcal{P})$). Let $\langle i_1, \cdots, i_k\rangle$ $(1 \leq i_1 < \cdots < i_k \leq n)$ denote $F(v_{d+1})$. Note that the decision of r_{i_1} is not the expected decision for the failed test p; otherwise, p should be a passed test.

(3) Find the rules in $\{r_{i_2}, \cdots, r_{i_k}\}$ whose decisions are the expected decision of p. Suppose $\{r_{j_1}, \cdots, r_{j_g}\}$ are those rules that we find for p, where $\{r_{j_1}, \cdots, r_{j_g}\} \subseteq \{r_{i_2}, \cdots, r_{i_k}\}$. Because the decision of rules in $\{r_{j_1}, \cdots, r_{j_g}\}$ is the expected decision for p, swapping r_{i_1} with any rule in $\{r_{j_1}, \cdots, r_{j_g}\}$ changes p to a passed test. Note that because the last rule of a firewall is a default rule, colorredit is pointless to swap it with any preceding rule. If r_{j_g} is the last rule of the faulty firewall (*i.e.*, $j_g = n$), we delete r_{j_g} from $\{r_{j_1}, \cdots, r_{j_g}\}$.

(4) For all failed tests, we find out all rule pairs such that swapping two rules in a rule pair may increase the number of passed tests. Then we swap two rules in each rule pair. Note that swapping two rules in a rule pair changes the corresponding failed test to a passed test. However, this modification may change some passed tests to failed tests. Therefore, after swapping two rules in each rule pair, we reclassify all tests and calculate the number of passed tests.

(5) Find a rule pair such that swapping the two rules in this pair can maximize the number of passed tests.

Note that if there are more than one rule pair such that swapping two rules in each pair can maximize colorredthe increase in the number of passed tests, we choose the rule pair that affects the functionality of the minimum number of original firewall rules. Let $(r_{i_1}, r_{j_1}), \cdots, (r_{i_g}, r_{j_g})$ denote these rule pairs, where $i_k \leq j_k$ $(1 \leq k \leq g)$. Due to the first-match semantics, we choose the rule pair (r_i, r_j) where i is the maximum integer in $\{i_1, \cdots, i_g\}$.

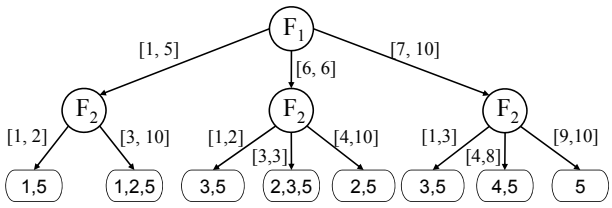


Fig. 5. All-match FDD converted from the faulty firewall policy in Figure 2

For the faulty firewall policy in Figure 2, we first convert the faulty firewall policy to an all-match FDD, which is shown in Figure 5. Second, for each failed test, we find the corresponding rule pairs. In the example, we find only one rule pair (r_2, r_3) for the failed test $(6,\ 3) \rightarrow d$. Third, after swapping r_2 and r_3, $(6,\ 2) \rightarrow d$ becomes a passed test and no passed test changes to a failed test. Therefore, swapping r_2 and r_3 increases the number of passed tests by 1.

7. MISSING-RULE CORRECTION

There are two challenges for adding a rule to a faulty firewall policy. First, given a faulty firewall policy with n rules, there are n positions where we can add a rule. Determining which position is the best for adding a rule is a challenge. Second, because the predicate of a firewall rule is composed of multiple fields and the number of possible values in each field is typically large, brute-force addition of every possible rule for each position is computationally expensive. Considering a firewall rule with five fields (*i.e.*, 32-bit source IP, 32-bit destination IP, 16-bit source port, 16-bit destination port, and 8-bit protocol type) and two possible decisions (*i.e.*,

accept and *discard*), the number of possible firewall rules that we can add for each position is $O(2^{204})$, because for each field with d-bit length, the number of possible ranges is $\binom{2}{2^d}$=$O(2^{2d-1})$. Furthermore, after adding a rule, we still need to reclassify all passed and failed tests.

The basic idea of our solution is that for each position, we first find all possible failed tests that can be corrected by adding a rule at this position, and then compute a rule that matches the maximum number of failed tests. To avoid changing a passed test to a failed test, the rule that we compute does not match any possible passed test. More formally, given a faulty firewall policy with n rules $\langle r_1, \cdots, r_n \rangle$, let position i ($1 \leq i \leq n$) denote the position between r_{i-1} and r_i. Note that we cannot add a rule after r_n because r_n is the default rule. Our correction technique for adding a rule includes five steps:

(1) For each position i, find a set of passed tests $PT(i)$ and a set of failed tests $FT(i)$ such that colorredno test p in $PT(i) \cup FT(i)$ matches any rule r_j ($1 \leq j \leq i-1$). Note that when $i = 1$, r_j does not exist. In such case, $PT(1) = PT$ and $FT(1) = FT$. Due to the first-match semantics, if a failed test p matches a rule r_i, adding a rule after rule r_i cannot change the decision of p and hence cannot correct p. Therefore, the set $FT(i)$ includes all possible failed tests that we can correct by adding a rule at position i.

(2) Based on the expected decisions of tests, divide $PT(i)$ into two sets $PT(i)_a$ and $PT(i)_d$ where $PT(i)_a$ consists of all passed tests with expected decision *accept* and $PT(i)_d$ consists of all passed tests with *discard*. Similarly, we divide $FT(i)$ into two sets $FT(i)_a$ and $FT(i)_d$. The purpose is that adding a rule cannot correct two failed tests with different expected decisions.

(3) For set $FT(i)_a$, compute a rule with decision *accept*, denoted as $r'_{i,a}$, that satisfies two conditions:
   (a) No passed test in $PT(i)_d$ matches $r'_{i,a}$.
   (b) Under Condition (a), $r'_{i,a}$ matches the maximum number of failed tests in $FT(i)_a$.

   The algorithm for computing rule $r'_{i,a}$ based on $FT(i)_a$ and $PT(i)_d$ is discussed in Section 7.1.

(4) Similar to Step 3, for set $FT(i)_d$, compute a rule with decision *discard*, denoted as $r'_{i,d}$, that satisfies two conditions:
   (a) No passed test in $PT(i)_a$ matches $r'_{i,d}$.
   (b) Under Condition (a), $r'_{i,d}$ matches the maximum number of failed tests in $FT(i)_d$.

(5) Find a rule $r'_{j,\ decision}$ ($1 \leq j \leq n$) that corrects the maximum number of failed tests and then add $r'_{j,\ decision}$ to position j.

Note that if there is more than one rule that can correct the maximum number of failed tests, we choose rule $r'_{j,\ decision}$ where j is the maximum integer among these rules such that adding this rule affects the functionality of the smallest number of original rules in a firewall policy.

For the faulty policy in Figure 2, Figure 6 shows the four sets $PT(i)_a$, $PT(i)_d$, $FT(i)_a$, and $FT(i)_d$ for each rule of the policy.

	$PT(i)_a$	$PT(i)_d$	$FT(i)_a$	$FT(i)_d$
r_1	p_1, p_2, p_3	p_4, p_5	p_7	p_6, p_8
r_2	p_3	p_4, p_5	p_7	p_6, p_8
r_3	–	p_4, p_5	p_7	p_8
r_4	–	p_5	p_7	p_8
r_5	–	p_5	p_7	–

Fig. 6. $PT(i)_a$, $PT(i)_d$, $FT(i)_a$, and $FT(i)_d$ for each rule in Figure 2

7.1 Computing Rules $r'_{i,a}$ and $r'_{i,d}$

Without loss of generality, in this section, we discuss the algorithm for computing $r'_{i,a}$ based on a set of failed tests $FT(i)_a$ and a set of passed tests $PT(i)_d$. First, we generate a rule that can match all failed tests in $FT(i)_a$. Suppose that the predicate of a firewall rule is composed of d fields. For each field j ($1 \leq j \leq d$), assume that x_j is the minimum value of all failed tests in $FT(i)_a$ and y_j is the maximum value. Therefore, the rule r:$F_1 \in [x_1, y_1] \wedge \cdots \wedge F_d \in [x_d, y_d] \rightarrow a$ matches all failed tests in $FT(i)_a$. Second, we use the passed tests in $PT(i)_d$ to split the rule to multiple rules, each of which does not match any passed test. Let $(z_1, \cdots, z_d) \rightarrow d$ denote the first passed test p in $PT(i)_d$. If rule r matches p, for each field j, we generate two rules by using z_j to split $[x_j, y_j]$ into two ranges $[x_j, z_j - 1]$ and $[z_j + 1, y_j]$. The resulting two rules for field j are as follows.

$$F_1 \in [x_1, y_1] \wedge \cdots \wedge F_{j-1} \in [x_{j-1}, y_{j-1}] \wedge \boldsymbol{F_j \in [x_j, z_j - 1]} \wedge F_{j+1} \in [x_{j+1}, y_{j+1}] \wedge \cdots \wedge F_d \in [x_d, y_d] \rightarrow a$$

$$F_1 \in [x_1, y_1] \wedge \cdots \wedge F_{j-1} \in [x_{j-1}, y_{j-1}] \wedge \boldsymbol{F_j \in [z_j + 1, y_j]} \wedge F_{j+1} \in [x_{j+1}, y_{j+1}] \wedge \cdots \wedge F_d \in [x_d, y_d] \rightarrow a$$

Note that if $x_j > z_j - 1$ (or $z_j + 1 > y_j$), the rule that includes $[x_j, z_j - 1]$ (or $[z_j + 1, y_j]$) is meaningless and it should be deleted from the resulting rules. If rule r does not match p, p cannot split r. Then, we use the second test in $PT(i)_d$ to split the resulting rules generated from p. Repeat this step until we check all the passed tests in $PT(i)_d$. Finally, we choose one rule that matches the maximum number of failed tests.

Take two sets $PT(2)_a$ and $FT(2)_d$ in Figure 6 as an example, rule $r'_{2,d}$ can be computed as $F_1 \in [6, 8] \wedge F_2 \in [3, 5] \rightarrow d$, which can correct two failed tests p_6 and p_8.

8. WRONG-PREDICATE CORRECTION

There are two challenges for fixing a predicate in a faulty firewall policy. First, for a faulty firewall policy with n rules, there are $n-1$ possible predicates that we can correct. Note that the last rule r_n is the default rule. Second, similar to adding rules, brute-force fixing of the predicate for each rule is computationally expensive. The number of possible predicates for each rule is $O(2^{203})$.

The basic idea for wrong-predicate correction is similar to adding rules. We first find all possible failed tests that can be corrected by fixing a predicate, and then compute a rule that matches the maximum number of failed tests. However, there are two major differences. First, for fixing the predicate of r_i, we compute only a rule with the same decision of r_i. Second, after fixing the predicate of rule r_i, the original rule r_i does not exist in the firewall policy. Therefore, the passed tests

whose first-matching rule is r_i may become failed tests. The set of these passed tests for r_i can be computed as $PT(i)-PT(i+1)$ (shown in Figure 8). The passed tests whose first-matching rule is not r_i should be prevented from changing to failed tests. Therefore, the set of all possible failed tests that we can correct by fixing r_i's predicate is $FT(i)\cup(PT(i)-PT(i+1))$. Our correction technique for wrong-predicate correction includes five steps:

(1) For each position i ($1\leq i\leq n$), find a set of passed tests $PT(i)$ and a set of failed tests $FT(i)$ such that no test p in $PT(i)\cup FT(i)$ matches any rule r_j ($1\leq j\leq i-1$).

(2) For each rule r_i ($1\leq i\leq n-1$), compute the set of all possible failed tests $FT(i)\cup(PT(i)-PT(i+1))$ that we can correct by fixing r_i's predicate. Let $\widehat{FT(i)}$ denote $FT(i)\cup(PT(i)-PT(i+1))$. The complement of the set $FT(i)\cup(PT(i)-PT(i+1))$ is $PT(i+1)$, which is the set of passed tests that we cannot change to failed tests by fixing r_i's predicate.

(3) Based on the expected decisions of tests, divide $PT(i+1)$ into two sets $PT(i+1)_a$ and $PT(i+1)_d$, and divide $\widehat{FT(i)}$ into two sets $\widehat{FT(i)}_a$ and $\widehat{FT(i)}_d$.

(4) Without loss of generality, assume that r_i's decision is *accept*. For set $\widehat{FT(i)}_a$, we compute $r''_{i,a}$ that satisfies two conditions:
   (a) No passed test in $PT(i+1)_d$ matches $r''_{i,a}$.
   (b) Under condition (a), $r''_{i,a}$ matches the maximum number of failed tests in $\widehat{FT(i)}_a$.

   The algorithm for computing rule $r''_{i,a}$ based on $\widehat{FT(i)}_a$ and $PT(i+1)_d$ is the same as that in Section 7.1. Let r''_i denote the resulting rule.

(5) Find a rule r''_j ($1\leq j\leq n-1$) that can correct the maximum number of failed tests and then replace rule r_j.

Note that if there is more than one rule that can correct the maximum number of failed tests, we choose rule r''_j where j is the maximum integer among these rules.

For the faulty policy in Figure 2, Figure 7 shows the four sets $PT(i+1)_a$, $PT(i+1)_d$, $\widehat{FT(i)}_a$, and $\widehat{FT(i)}_d$ for each rule. Rule $r''_{2,a}$ can be computed as $F_1\in[6,7]\wedge F_2\in[7,9]\rightarrow a$, which can correct one failed test p_7.

	$PT(i+1)_a$	$PT(i+1)_d$	$\widehat{FT(i)}_a$	$\widehat{FT(i)}_d$
r_1	p_3	p_4, p_5	p_1, p_2, p_7	p_6, p_8
r_2	–	p_4, p_5	p_3, p_7	p_6, p_8
r_3	–	p_5	p_7	p_4, p_8
r_4	–	p_5	p_7	p_8

Fig. 7. $PT(i+1)_a$, $PT(i+1)_d$, $\widehat{FT(i)}_a$, and $\widehat{FT(i)}_d$ for each rule in Figure 2

9. WRONG-DECISION CORRECTION

The idea of fixing a decision is that for each rule r_i, we first find the passed tests and failed tests whose first-matching rule is r_i. The colorredset of passed tests for r_i can be computed as $PT(i)-PT(i+1)$ and the set of the failed tests for r_i can be computed as $FT(i)-FT(i+1)$. If we change the decision of r_i, the passed tests in $PT(i)-PT(i+1)$ become failed tests and the failed tests in $FT(i)-FT(i+1)$

become passed tests. Then, we can calculate colorredthe increase in the number of passed tests by fixing r_i's decision. Finally, we fix the decision of the rule that corresponds to the maximum increased number of passed tests. Our correction technique for fixing a decision includes three steps:

(1) For each rule r_i $(1 \leq i \leq n-1)$, compute two sets: $PT(i) - PT(i+1)$ and $FT(i) - FT(i+1)$.
(2) Calculate colorredthe increase in the number of passed tests by fixing r_i's decision, which is $|FT(i){-}FT(i{+}1)|{-}|PT(i){-}PT(i+1)|$.
(3) Fix the decision of a rule that can maximize colorredthe increase in the number of passed tests.

Note that if there is more than one rule such that fixing the decision of each of them can maximize colorredthe increase in the number of passed tests, we choose the rule with the largest sequence number.

For the faulty policy in Figure 2, Figure 8 shows the two sets $PT(i) - PT(i+1)$ and $FT(i) - FT(i+1)$ for each rule. Clearly, fixing the decision of r_4 can change the failed test p_8 to a passed test.

	$PT(i) - PT(i+1)$	$FT(i) - FT(i+1)$
r_1	p_1, p_2	–
r_2	p_3	p_6
r_3	p_4	–
r_4	–	p_8

Fig. 8. $PT(i){-}PT(i{+}1)$ and $FT(i){-}FT(i{+}1)$ for each rule in Figure 2

10. WRONG-EXTRA-RULE CORRECTION

The idea of deleting a firewall rule is that we use the all-match FDD to calculate colorredthe increase in the number of passed packets by deleting each rule, and then delete the rule that can maximize colorredthe increase in the number of passed packets. Given a faulty policy with n rules and its all-match FDD, our correction technique for deleting a rule includes three steps:

(1) For each rule r_i $(1{\leq}i{\leq}n{-}1)$, find every decision path $\mathcal{P}{:}(v_1e_1\cdots v_de_dv_{d+1})$ such that $C(\mathcal{P}){\subseteq}C(r_i)$ and i is the first rule id in $F(v_{d+1})$. Let $\{\mathcal{P}^i_1, \cdots, \mathcal{P}^i_h\}$ denote the set of such decision paths.
(2) For each decision path $\mathcal{P}^i_g{:}(v_1e_1\cdots v_de_dv_{d+1})$ $(1 \leq g \leq h)$, find the set of passed tests $PT(\mathcal{P}^i_g)$ and the set of failed tests $FT(\mathcal{P}^i_g)$, where any test in $PT(\mathcal{P}^i_g)$ or $FT(\mathcal{P}^i_g)$ matches $\mathcal{P}^i_g$. Let $\langle i_1, \cdots, i_k\rangle$ $(1{\leq}i_1{<}\cdots{<}i_k{\leq}n)$ denote $F(v_{d+1})$. Note that $i_1 = i$ because of the first-match semantics. Let l_g denote colorredthe increase in the number of passed tests that match $\mathcal{P}^i_g$ after deleting rule r_i. To calculate l_g, we need to check whether r_i and r_{i_2} have the same decision. If r_i and r_{i_2} have the same decision, deleting r_i does not change the two sets $PT(\mathcal{P}^i_g)$ and $FT(\mathcal{P}^i_g)$. In this case, $l_g{=}0$. Otherwise, the passed tests in $PT(\mathcal{P}^i_g)$ become failed tests and the failed tests in $FT(\mathcal{P}^i_g)$ become passed tests. In this case, $l_g = |FT(\mathcal{P}^i_g)| - |PT(\mathcal{P}^i_g)|$. Therefore, colorredthe increase in the number of passed packets after deleting rule r_i can be computed as $\sum_{g=1}^{h} l_g$.

(3) Delete the rule that can maximize the number of passed packets.

Note that if rule r_i is not the first-matching rule for any failed test, $|FT(\mathcal{P}_g^i)|=0$ $(1\leq g\leq h)$ and hence $\sum_{g=1}^{h} l_g \leq 0$. In this case, deleting r_i cannot increase the number of passed packets. We can easily find such rules by computing the set $FT(i)-FT(i+1)$ for each rule r_i. Further note that if there is more than one rule such that deleting each of them can maximize colorredthe increase in the number of passed tests, we choose the rule with the maximum sequence number.

For the faulty firewall policy in Figure 2, by checking $FT(i)-FT(i+1)$ in Figure 8, we find that deleting rule r_1 or r_3 cannot increase the number of passed packets. In the all-match FDD of the faulty policy (shown in Figure 5), for rule r_2, there are two paths, $F_1\in[6,6]\wedge F_2\in[3,3]$ and $F_1\in[6,6]\wedge F_2\in[4,10]$, where 2 is the first integer in their terminal nodes. Because the failed test p_6 matches the first path, and r_2 and r_3 have different decisions, deleting r_2 changes p_6 to a passed test. Because the passed test p_3 matches the second path, and r_2 and r_5 have different decisions, deleting r_2 changes p_3 to a failed test. Therefore, deleting r_2 does not increase the number of passed tests. Similarly, deleting r_4 changes p_8 to a passed test, and hence increases the number of passed tests by 1.

11. EXPERIMENTAL RESULTS

11.1 Evaluation Setup

In our experiments, faulty firewall policies were generated from 40 real-life firewall policies that we collected from universities, ISPs, and network device manufacturers. The 40 real-life policies were considered as correct policies with respect to these faulty policies. Each firewall examines five fields: source IP, destination IP, source port, destination port, and protocol type. The number of rules for each policy ranges from dozens to thousands.

To evaluate the effectiveness and efficiency of the greedy algorithm, we first employed the technique of mutation testing [DeMillo et al. 1978] to create faulty firewall policies. The technique for injecting synthetic faults with mutation testing is a well-accepted mechanism for carrying out testing experiments in real practice. Particularly, each faulty policy contains one type of fault, and the number of faults in a faulty firewall policy ranges from 1 to 5. Given a real-life firewall with n rules, for each type of fault, we employed mutation testing [DeMillo et al. 1978] to randomly create $n-1$ faulty policies. In other words, we created $5\times(n-1)$ policies for each real-life firewall with n rules. Recall that it is trivial to check whether the last rule is correct. Thus, we did not change the last rule for generating faulty policies in our experiments. For example, to create a faulty firewall policy with k *wrong decisions* faults, we randomly changed the decisions of the k rules in a real-life firewall policy. Overall, we generated 35618 faulty firewall policies in our experiments. Second, for each faulty policy, we employed the firewall testing tool [Hwang et al. 2008] to generate test packets. Note that the test packets were generated based on the faulty policy rather than its corresponding real-life policy. For each faulty policy, on average, the total number of passed and failed tests is about $3n$, where n is the number of rules in the policy. Third, we classified the test packets into passed and failed tests. Recall that the real-life firewall policies were considered as correct policies in our experiments. Thus, to classify each test packet, we compared

two decisions generated by the faulty policy and its corresponding real-life policy. If the two decisions were the same, we classified the test packet as a passed test; otherwise, we classified it as a failed test. Note that in practice this step should be done by administrators. Finally, we implemented and applied our greedy algorithm over the faulty firewall policy, and then produced the fixed policy. For each step of the greedy algorithm, if different techniques increase the same number of passed tests, we randomly colorredchose one technique.

To evaluate the effectiveness and efficiency of the improved algorithm, we first created another set of $n-1$ faulty policies for each real-life firewall with n rules. Each faulty policy contains five faults and each fault is one distinct fault type defined in the fault model of firewall policies (Section 4). Note that these $n-1$ faulty policies are different from $5\times(n-1)$ faulty policies we generated for evaluating the greedy algorithm. Second, we employed the firewall testing tool [Hwang et al. 2008] to generate test packets and classified them into passed and failed tests. Third, we applied our improved algorithm over the faulty firewall policies and produced the fixed policies. To compare the improved algorithm with the greedy algorithm, we also applied our greedy algorithm over these $n-1$ faulty policies. The reason of creating anther set of $n-1$ faulty policies is that we can evaluate the effectiveness of the two algorithms over the faulty policies with different types of faults instead of the faulty policies with a single type of faults, which provides another perspective to understand the two algorithms.

11.2 Methodology

To measure the effectiveness of our approach, we need to first define the *difference* between two firewall policies. Given two policies FW_1 and FW_2, the *difference* between FW_1 and FW_2, denoted as $\Delta(FW_1, FW_2)$, is the total number of packets each of which has different decisions evaluated by FW_1 and FW_2. To compute $\Delta(FW_1, FW_2)$, we employed the firewall comparison algorithm [Liu and Gouda 2008]. This algorithm first finds the functional discrepancies between FW_1 and FW_2, where each discrepancy denotes a set of packets and each packet has different decisions evaluated by the two policies, and then compute the number of packets included by all the discrepancies.

Next, we define three metrics to measure the effectiveness of our approach. Let FW_{real} denote a real-life firewall policy and FW_{faulty} denote a faulty policy created from FW_{real}. Let FT denote the set of failed tests and $|FT|$ denote the number of failed tests. Let FW_{fixed} denote the fixed policy by correcting FW_{faulty} and $m(FW_{faulty})$ denote the number of modifications. Let $S(t, k)$ denote a set of faulty policies, where t denotes the type of fault and k denotes the number of faults in each faulty policy. We define the three metrics for evaluating the effectiveness of our approach as follows:

(1) The *difference ratio* over FW_{real}, FW_{faulty}, and FW_{fixed}:

$$\frac{\Delta(FW_{real}, FW_{fixed})}{\Delta(FW_{real}, FW_{faulty})}$$

(2) The *correction rate* over FT and FW_{fixed}:

$$\frac{\Delta(FW_{real}, FW_{faulty}) - \Delta(FW_{real}, FW_{fixed})}{|FT|}$$

(3) The *average number of modifications* over $S(t,k)$:

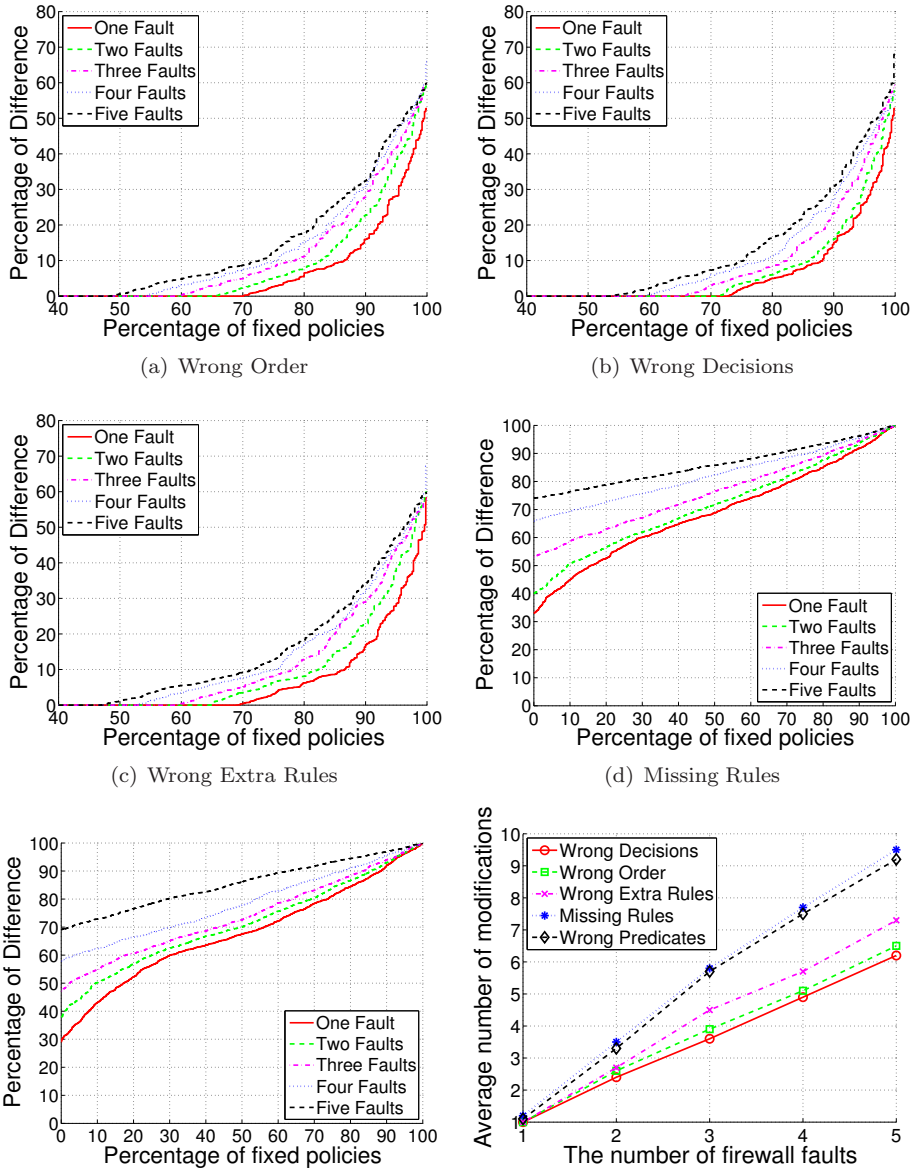
$$\frac{\sum_{FW_{faulty} \in S(t,k)} m(FW_{faulty})}{|S(t,k)|}$$

The intuition behind the difference ratio $\frac{\Delta(FW_{real}, FW_{fixed})}{\Delta(FW_{real}, FW_{faulty})}$ is to measure what percentage of misclassified packets were corrected after applying our approach. Because $\Delta(FW_{real}, FW_{faulty})$ denotes the total number of misclassified packets in the faulty firewall policy, and $\Delta(FW_{real}, FW_{faulty})$ denotes the total number of misclassified packets in the fixed firewall policy. If $\frac{\Delta(FW_{real}, FW_{fixed})}{\Delta(FW_{real}, FW_{faulty})} = 0$, FW_{fixed} corrects all misclassified packets, which means that FW_{fixed} is equivalent to FW_{real} in terms of functionality. For the example policy in Figure 1, if we generate a faulty firewall policy by changing r_1's decision to *discard*, the difference between the faulty policy and the example policy, $\Delta(FW_{real}, FW_{faulty})$, is $2^8 \times 2^{16} = 2^{24}$. Hence, the total number of packets that are misclassified by the faulty firewall policy is 2^{24}. If the fixed firewall policy corrects r_1's decision in the faulty firewall policy, the difference between the fixed policy and the example policy, $\Delta(FW_{real}, FW_{fixed})$, is 0. Thus, $\frac{\Delta(FW_{real}, FW_{fixed})}{\Delta(FW_{real}, FW_{faulty})} = 0$, which means that FW_{fixed} is equivalent to FW_{real} in terms of functionality.

The intuition behind *correction rate* $\frac{\Delta(FW_{real}, FW_{faulty}) - \Delta(FW_{real}, FW_{fixed})}{|FT|}$ is to measure how much our algorithms outperform the algorithm of adding singleton rules to correct the failed tests. Note that FT is only a small portion of all misclassified packets in FW_{faulty}. The algorithm of adding singleton rules can only fix the failed tests in FT. For ease of presentation, we call the algorithm of adding singleton rules the *simple algorithm*. Based on the discussion of the difference ratio, we know that $\Delta(FW_{real}, FW_{faulty}) - \Delta(FW_{real}, FW_{fixed})$ denotes the number of misclassified packets that have been fixed by our algorithms. If $\frac{\Delta(FW_{real}, FW_{faulty}) - \Delta(FW_{real}, FW_{fixed})}{|FT|} >> 1$, our algorithms significantly outperform the simple algorithm. If $\frac{\Delta(FW_{real}, FW_{faulty}) - \Delta(FW_{real}, FW_{fixed})}{|FT|} = 1$, our algorithms have the same effectiveness as the simple algorithm. In other words, our algorithms are not better than the simple algorithm.

11.3 Effectiveness of the Greedy Algorithm

Figures 9(a)-9(e) show the cumulative distribution of difference ratios over FW_{real}, FW_{faulty}, and FW_{fixed} for each type of fault. In Figures 9(a)-9(e), we use "One Fault", $\cdots$, "Five Faults" to denote the number of faults in faulty firewall policies. A point (X, Y) in these figures describes that the difference ratios of $X\%$ fixed policies are less than or equal to $Y\%$. For example, the point (73.5, 0) in Figure 9(a) describes that the difference ratios of 73.5% fixed policies are equal to 0. In other words, 73.5% fixed policies are equivalent to the corresponding real-life policies. We observe that for three types of faults, wrong order, wrong decisions, and wrong extra rules, fixed policies can significantly reduce the number of misclassified packets. For



(a) Wrong Order (b) Wrong Decisions

(c) Wrong Extra Rules (d) Missing Rules

(e) Wrong Predicates (f) Avg. Number of Modifications

Fig. 9. Cumulative distribution of difference ratio and average number of modifications for each type of firewall policy faults

faulty policies with k faults, where k faults are one of these three types and $k \leq 4$, over 53.2% fixed policies are equivalent to their corresponding real-life policies. For faulty policies with 1 to 5 *wrong decisions* faults, the percentages of fixed policies that are equivalent to their corresponding real-life policies are 73.5%, 68.8%, 63.7%, 59.3%, and 53.8%, respectively. For faulty policies with 1 to 5 *wrong order* faults, the percentages of fixed policies that are equivalent to their corresponding real-life policies are 69.7%, 64.2%, 59.7%, 54.3%, and 48.9%, respectively. For faulty policies with 1 to 5 *wrong extra rules* faults, the percentages of fixed policies that are equivalent to their corresponding real-life policies are 68.3%, 63.5%, 59.3%, 53.2%, and 47.3%, respectively.

We also observe that fixed policies can reduce only a small number of misclassified packets for two types of faults, missing rules and wrong predicates. For faulty policies with 1 to 2 *missing rules* faults, the percentages of fixed policies that have 50% difference ratio with their corresponding real-life policies are 15.7% and 8.32%, respectively. For faulty policies with 1 to 2 *wrong predicates faults*, the percentages of fixed policies that have 50% difference ratio with their corresponding real-life policies are 17.3% and 9.1%, respectively. The reason is that, in most cases, the information provided by failed tests is not enough to recover the missing rule (or correct predicate). A firewall rule (or predicate) with 5 fields can be denoted as a hyperrectangle over a 5-dimensional space, and failed tests are only some points in the hyperrectangle. To recover the missing rule (or correct the wrong predicate), for each surface of the hyperrectangle, there should be at least one point on it. However, the chance of such a case is very small.

Figure 9(f) shows the average number of modifications for each type of faults. We observe that for faulty firewall policies with k faults ($k \leq 5$), the ratio between the average number of modifications and the number of faults is less than 2.

We also measured the *correction rates* for the fixed firewall policies. Our experimental results show that all the correction rates are larger than 158, and the average correction rate is 2268, which demonstrates that our algorithms significantly outperform the simple algorithm.

11.4 Effectiveness of the Improved Algorithm

Figure 10 shows the cumulative distribution of difference ratios over FW_{real}, FW_{faulty}, and FW_{fixed} by applying both the greedy and improved algorithms to the firewall policies with five different faults. We observe that the improved algorithm performs better than the greedy algorithm in terms of percentage of fixed policies. The percentage of fixed policies generated by the improved algorithm is 5% more than that of fixed policies generated by the greedy algorithm.

Similar as evaluating the greedy algorithm, we also measured the *correction rates* for the fixed firewall policies. Our experimental results show that all the correction rates are larger than 137, and the average correction rate is 2537, which demonstrates that our algorithms significantly outperform the simple algorithm.

11.5 Efficiency of Our Approach

We implemented our approach using Java 1.6.0. In our experiments, for a faulty firewall policy, we measure the total processing time of generating test packets, classifying packets into passed and failed tests, and fixing the policy to evaluate

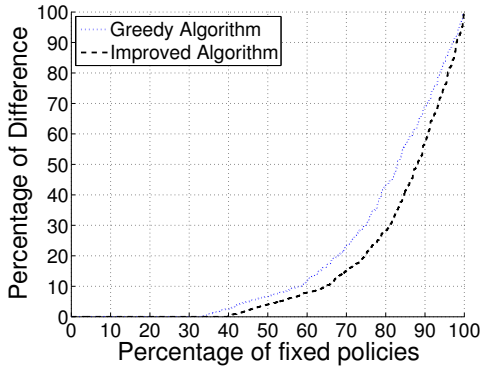


Fig. 10. Cumulative distribution of difference ratio for firewall policies with five different faults

the efficiency of our approach. Note that classifying test packets is automatically done in our experiments by comparing two decisions evaluated by the faulty firewall and its corresponding real-life firewall for test packets. In practice, this step should be done by administrators. Our experiments were carried out on a desktop PC running Linux with 2 quad-core Intel Xeon at 2.3GHz and 16GB of memory. Our experimental results show that for the faulty firewall policy with 7652 rules, the total processing time for fixing this faulty policy is less than 10 minutes.

12. CASE STUDY

In this section, we applied our automatic correction tool for firewall policy faults to a real-life faulty firewall policy with 87 rules and demonstrated that our tool can help the administrator to correct the misconfiguration in the firewall policy. The real-life firewall policy is shown in the Appendix B where the policy is anonymized due to the privacy and security concern.

We first employed the automated packet generation techniques [Hwang et al. 2009] to generate test packets for the firewall policy and then asked the administrator to identify passed/failed tests. The total number of generated tests is 162 and classifying all these tests takes less than 30 minutes. Among these test packets, we obtained seven failed tests, which are shown in Table I. Second, we applied our proposed solution to this firewall policy and generated a sequence of modifications to correct the seven failed tests in Table I. The resulting sequence includes four modifications: swapping rule 6 and rule 38, deleting rules 48, 49, and 50, which suggest that the firewall policy has one wrong-order fault and three wrong-extra-rule faults. We confirmed these faults with the administrator and he admitted that the resulting sequence of modifications generated by our tool can correct these faults automatically.

13. CONCLUSIONS

We make two key contributions in this paper. First, we propose the systematic approach that can automatically correct all or part of the misclassified packets of a faulty firewall policy. To the best of our knowledge, our paper is the first one for automatic correction of firewall policy faults. Second, we implemented our approach and evaluated its effectiveness on real-life firewalls. To measure the effectiveness

$p_1 : (157.96.252.36, 157.96.252.66, 13249, 25341, IP) \rightarrow a$
$p_2 : (67.48.121.156, 157.96.139.10, 4537, 109, TCP) \rightarrow a$
$p_3 : (35.121.47.232, 157.96.139.10, 21374, 109, TCP) \rightarrow a$
$p_4 : (25.35.113.153, 157.96.139.10, 7546, 110, TCP) \rightarrow a$
$p_5 : (154.182.56.79, 157.96.139.10, 16734, 110, TCP) \rightarrow a$
$p_6 : (193.21.135.85, 157.96.139.10, 19678, 143, TCP) \rightarrow a$
$p_7 : (213.174.191.25, 157.96.139.10, 24131, 143, TCP) \rightarrow a$

Table I. Seven failed tests for the real-life firewall policy

of our approach, we propose three metrics, which we believe are general metrics for measuring the effectiveness of firewall policy correction tools. The experimental results demonstrated that our approach is effective to correct a faulty firewall policy with three types of faults: wrong order, wrong decisions, and wrong extra rules.

REFERENCES

Al-Shaer, E., El-Atawy, A., and Samak, T. 2009. Automated pseudo-live testing of firewall configuration enforcement. *IEEE Journal on Selected Areas in Communications 27*, 302–314.

Al-Shaer, E. and Hamed, H. 2004. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*. 2605–2616.

Baboescu, F. and Varghese, G. 2002. Fast and scalable conflict detection for packet classifiers. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*. 717–735.

CERT. 2001. Test the firewall system. http://www.cert.org/security-improvement/practices/p060.html.

Chen, F., Liu, A. X., Hwang, J., and Xie, T. 2010. First step towards automatic correction of firewall policy faults. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*.

Cisco Reflexive ACLs. http://www.cisco.com/.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer 11,* 4 (April), 34–41.

Hari, A., Suri, S., and Parulkar, G. M. 2000. Detecting and resolving packet filter conflicts. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*. 1203–1212.

Hoffman, D. and Yoo, K. 2005. Blowtorch: a framework for firewall test automation. In *Proceedings of International Conference on Automated Software Engineering (AES)*. 96–103.

Hwang, J., Xie, T., Chen, F., and Liu, A. X. 2008. Systematic structural testing of firewall policies. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 105–114.

Hwang, J., Xie, T., Chen, F., and Liu, A. X. 2009. Fault localization for firewall policies. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 100–106.

Jürjens, J. and Wimmel, G. 2001. Specification-based testing of firewalls. In *Proceedings of International Conference Perspectives of System Informatics (PSI)*. 308–316.

Liu, A. X. 2007. Change-impact analysis of firewall policies. In *Proceedings of European Symposium Research Computer Security (ESORICS)*. 155–170.

Liu, A. X. and Gouda, M. G. 2008. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 19,* 8.

Liu, A. X., Zhou, Y., and Meiners, C. R. 2008. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*. 574–582.

Lyu, M. R. and Lau, L. K. Y. 2000. Firewall security: Policies, testing and performance evaluation. In *Proceedings of International Conference on Computer Systems and Applications (COMPSAC)*. 116–121.

Marmorstein, R. and Kearns, P. 2007. Assisted firewall policy repair using examples and history. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*. 1–11.

Nessus. 2004. http://www.nessus.org/.

Satan. 1995. http://www.porcupine.org/satan/.

Tang, Y., Al-Shaer, E., and Boutaba, R. 2008. Efficient fault diagnosis using incremental alarm correlation and active investigation for internet and overlay networks. *IEEE Transactions on Network and Service Management 5*, 36–49.

Wool, A. 2004. A quantitative study of firewall configuration errors. *IEEE Computer 37,* 6, 62–67.

Yuan, L., Chen, H., Mai, J., Chuah, C.-N., Su, Z., and Mohapatra, P. 2006. Fireman: a toolkit for firewall modeling and analysis. In *Proceedings of IEEE Symposium on Security and Privacy (IEEE S&P)*. 199–213.

Zeller, A. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. 1–10.

Appendix A

Before we prove Theorem 6.1, we first prove the following two lemmas.

Lemma 13.1. *Given a firewall policy FW:*$\langle r_1,\cdots,r_n\rangle$ *and its all-match FDD* $\{\mathcal{P}_1,\cdots,\mathcal{P}_h\}$*, for any rule* r_i *in FW, if* $\mathcal{P}_{i_1},\cdots,\mathcal{P}_{i_m}$ *are all the decision paths whose terminal node contains* r_i*, the following condition holds:* $C(r_i) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t})$.

Proof: According to property 5 in the definition of all-match FDDs, we have $\cup_{t=1}^{m} C(\mathcal{P}_{i_t}) \subseteq C(r_i)$. Consider a packet p in $C(r_i)$. According to the consistency and completeness properties of all-match FDDs, there exists one and only one decision path that p matches. Let $\mathcal{P}$ denote this path. Thus, we have $p \in C(r_i) \cap C(\mathcal{P})$. According to property 5, i is in the label of $\mathcal{P}$'s terminal node. Thus, we have $\mathcal{P} \in \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$. Therefore, $p \in \cup_{t=1}^{m} C(\mathcal{P}_{i_t})$. Thus, we have $\cup_{t=1}^{m} C(\mathcal{P}_{i_t}) \supseteq C(r_i)$.

Lemma 13.2. *Given a firewall policy FW:*$\langle r_1,\cdots,r_n\rangle$ *and its all-match FDD* $\{\mathcal{P}_1,\cdots,\mathcal{P}_h\}$*, for any rule* r_i *in FW, there exists only one set of paths* $\mathcal{P} \in \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$ *such that* $C(r_i) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t})$.

Proof: Suppose there exists another one set of path $\{\mathcal{P}'_{i_1}, \cdots, \mathcal{P}'_{i_l}\}$, which is different from $\mathcal{P} \in \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$. Thus, there exists at least one $\mathcal{P}'_{i_s} \notin \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$ $(1 \leq s \leq l)$. According to the consistency and completeness properties of all-match FDDs, for any $\mathcal{P}_{i_t}$ $(1 \leq t \leq m)$, $\mathcal{P}_{i_t} \cap \mathcal{P}'_{i_s} = \emptyset$. Thus, $\cup_{t=1}^{m} C(\mathcal{P}_{i_t}) \neq \cup_{s=1}^{l} C(\mathcal{P}'_{i_s})$, which contradicts with our assumption.

Next we can prove Theorem 6.1 based on Lemma 13.1 and Lemma 13.2.

Proof of Theorem 6.1: Based on Lemma 13.2, for each rule $r_i^1 \in \{r_1^1, \cdots, r_n^1\}$ $(1 \leq i \leq n)$, we can find only one set of paths $\mathcal{P} \in \{\mathcal{P}_1^1, \cdots, \mathcal{P}_{h_1}^1\}$ such that $C(r_i^1) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t}^1)$. Because $\{r_1^1,\cdots,r_n^1\} = \{r_1^2,\cdots,r_n^2\}$, there exists r_j^2 $(1 \leq j \leq n)$ such that $r_j^2 = r_i^1$. Thus, for each rule r_i^1 and its corresponding rule r_j^2, we have

$$C(r_i^1) = C(r_j^2) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t}^1)$$

We also know that for the all-match FDD $\{\mathcal{P}_1^1,\cdots,\mathcal{P}_{h_1}^1\}$ generated from FW_1:$\langle r_1^1,\cdots,r_n^1\rangle$, the following condition holds:

$$\cup_{i=1}^{n}(\cup_{t=1}^{m}\mathcal{P}_{i_t}^1) = \{\mathcal{P}_1^1,\cdots,\mathcal{P}_{h_1}^1\}$$

Similarly, for the all-match FDD $\{\mathcal{P}_1^2,\cdots,\mathcal{P}_{h_2}^2\}$ generated from FW_2:$\langle r_1^2,\cdots,r_n^2\rangle$, we have

$$\cup_{i=1}^{n}(\cup_{t=1}^{m}\mathcal{P}_{i_t}^1) = \{\mathcal{P}_1^2,\cdots,\mathcal{P}_{h_2}^2\}$$

Thus, $\{\mathcal{P}_1^1,\cdots,\mathcal{P}_{h_1}^1\} = \{\mathcal{P}_1^2,\cdots,\mathcal{P}_{h_2}^2\}$.

Appendix B

The real-life firewall policy with 87 rules is shown as follows.

#	Src IP	Dest IP	Src Port	Dest Port	Protocol	Action
1	67.54.138.163	157.96.119.153	*	9100	TCP	accept
2	67.54.138.163	157.96.119.153	*	161	UDP	accept
3	*	*	*	*	53	deny
4	*	*	*	*	55	deny
5	*	*	*	*	77	deny
6	*	157.96.252.66	*	*	IP	accept
7	32.45.186.83	*	*	*	IP	deny
8	*	157.96.139.14	*	443	TCP	deny
9	231.49.182.251	*	*	*	IP	deny
10	*	*	*	3127	TCP	deny
11	*	*	*	2745	TCP	deny
12	*	*	4000	*	UDP	deny
13	*	*	*	111	UDP	deny
14	*	*	*	111	TCP	deny
15	*	*	*	2049	UDP	deny
16	*	*	*	2049	TCP	deny
17	*	*	*	7	UDP	deny
18	*	*	*	7	TCP	deny
19	*	*	*	6346	TCP	deny
20	*	*	*	7000	TCP	deny
21	*	*	*	161	UDP	deny
22	*	*	*	162	UDP	deny
23	*	*	*	1993	UDP	deny
24	*	*	*	67	UDP	deny
25	*	*	*	68	UDP	deny
26	*	*	*	49	UDP	deny
27	178.95.49.*	*	*	*	IP	deny
28	157.96.119.*	*	*	*	IP	deny
29	157.96.120.*	*	*	*	IP	deny
30	157.96.121.*	*	*	*	IP	deny
31	157.96.122.*	*	*	*	IP	deny
32	157.96.130.*	*	*	*	IP	deny
33	157.96.138.*	*	*	*	IP	deny
34	157.96.139.*	*	*	*	IP	deny
35	157.96.143.*	*	*	*	IP	deny
36	157.96.144.*	*	*	*	IP	deny
37	157.96.158.*	*	*	*	IP	deny
38	157.96.252.*	*	*	*	IP	deny
39	*	157.96.139.9	*	1949	UDP	accept
40	*	157.96.139.10	*	1949	UDP	accept
41	*	157.96.120.2	*	1949	UDP	accept
42	*	157.96.139.9	*	1949	TCP	accept
43	*	157.96.139.10	*	1949	TCP	accept
44	*	157.96.120.2	*	1949	TCP	accept
45	255.255.255.255	*	*	*	IP	deny
46	0.0.0.0	*	*	*	IP	deny
47	*	157.96.119.*	*	*	IP	deny

#	Src IP	Dest IP	Src Port	Dest Port	Protocol	Action
48	*	157.96.139.10	*	109	TCP	accept
49	*	157.96.139.10	*	110	TCP	accept
50	*	157.96.139.10	*	143	TCP	accept
51	62.78.103.*	*	*	*	IP	deny
52	*	*	*	6667	TCP	deny
53	*	*	*	6112	TCP	deny
54	*	*	*	109	TCP	deny
55	*	*	*	110	TCP	deny
56	*	*	*	1433	UDP	deny
57	*	*	*	1434	UDP	deny
58	*	*	*	135	TCP	deny
59	*	*	*	137	TCP	deny
60	*	*	*	138	TCP	deny
61	*	*	*	139	TCP	deny
62	*	*	*	445	TCP	deny
63	*	*	*	135	UDP	deny
64	*	*	*	137	UDP	deny
65	*	*	*	138	UDP	deny
66	*	*	*	139	UDP	deny
67	*	*	*	445	UDP	deny
68	*	*	*	143	TCP	deny
69	*	*	*	515	TCP	deny
70	*	*	*	512	TCP	deny
71	*	*	*	514	UDP	deny
72	*	*	*	69	UDP	deny
73	*	*	*	514	TCP	deny
74	*	157.96.138.138	*	5900	TCP	accept
75	*	157.96.138.138	*	5166	TCP	accept
76	*	157.96.138.138	*	*	IP	deny
77	*	157.96.138.101	*	5900	TCP	accept
78	*	157.96.138.101	*	5166	TCP	accept
79	*	157.96.138.101	*	*	IP	deny
80	*	157.96.138.80	*	*	IP	deny
81	*	157.96.138.82	*	*	IP	deny
82	*	157.96.138.234	*	*	IP	deny
83	*	157.96.138.235	*	*	IP	deny
84	*	157.96.138.236	*	*	IP	deny
85	*	157.96.128.*	*	*	IP	accept
86	*	157.96.140.*	*	*	IP	deny
87	*	*	*	*	IP	accept