Systematic Structural Testing of Firewall Policies

JeeHyun Hwang[1] Tao Xie[1] Fei Chen[2] Alex X. Liu[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206

[2] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1266

jhwang4@ncsu.edu xie@csc.ncsu.edu {feichen,alexliu}@cse.msu.edu

Abstract

Firewalls are the mainstay of enterprise security and the most widely adopted technology for protecting private networks. As the quality of protection provided by a firewall directly depends on the quality of its policy (i.e., configuration), ensuring the correctness of firewall policies is important and yet difficult. To help ensure the correctness of a firewall policy, we propose a systematic structural testing approach for firewall policies. We define structural coverage (based on coverage criteria of rules, predicates, and clauses) on the policy under test. To achieve high structural coverage effectively, we have developed three automated packet generation techniques: the random packet generation, the one based on local constraint solving (considering individual rules locally in a policy), and the most sophisticated one based on global constraint solving (considering multiple rules globally in a policy).

We have conducted an experiment on a set of real policies and a set of faulty policies to detect faults with generated packet sets. Generally, our experimental results show that a packet set with higher structural coverage has higher fault-detection capability (i.e., detecting more injected faults). Our experimental results show that a reduced packet set (maintaining the same level of structural coverage with the corresponding original packet set) maintains similar fault-detection capability with the original set.

1 Introduction

Serving as the first line of defense against malicious attacks and unauthorized traffic, firewalls are crucial elements in securing the private networks of most businesses, institutions, and home networks. A firewall is typically placed at the point of entry between a private network and the outside Internet so that all network traffic has to pass through it. In a distributed system, messages are encapsulated into packets, which often pass through multiple access points in a network and firewalls are responsible for filtering, monitoring, and securing such packets as the parts of a distributed system [10]. Corruption or misconfiguration in firewalls may cause that the firewalls fail to filter malicious packets properly and affect the performance and security of a distributed system.

As security problems of firewalls are often caused by misconfiguration in firewall policies, correctly specifying firewall policies is a critical and yet challenging task for building reliable firewalls [14]. There are many factors for misconfiguring firewall policies. First, the rules in a firewall policy are logically entangled because of the conflicts among rules and the resulting order sensitivity. Second, a firewall policy may consist of a large number of rules. A firewall on the Internet may consist of hundreds or even a few thousands of rules in extreme cases. Last but not the least, an enterprise firewall policy often consists of legacy rules that are written by different operators, at different times, and for different reasons, which make maintaining firewall policies even more difficult.

To help ensure the correctness of firewall policies, researchers and practitioners have developed various firewall analysis and testing tools. The main function of these firewall analysis tools is to detect "bad smell" (i.e., "anomalies") in firewall policies based on some common patterns of firewall configuration mistakes [2, 15]. Such firewall analysis tools are certainly useful; however, the main drawback of such tools is that the "anomalies" may not be mistakes and the number of "anomalies" could be too large to be practically useful. Several firewall policy testing techniques have been proposed [7, 9, 11]. However, these firewall policy testing techniques are not based on well-established testing techniques in software engineering. For example, these techniques do not consider coverage criteria [16] for firewall policy testing

In this paper, we propose firewall policy testing based on the concept of *firewall policy coverage*, which helps test a firewall policy's structural entities (i.e., rules, predicates, and clauses) to check whether each entity is specified correctly. In firewall policy testing, test inputs and outputs are packets and their evaluated decisions (against the firewall

policy under test), respectively. Given test packets and the policy under test, when evaluating packets against the policy, our coverage measurement tool measures *firewall policy coverage* —- which entities of the policy are involved (called "covered") in the evaluation. Our systematic firewall policy testing helps detect faults, which often do not follow some configuration mistake patterns (e.g., anomalies [2, 15] and configuration errors [14]).

Policy testers shall generate a test suite to achieve high structural coverage, which helps investigate a large portion of policy entities for fault detection. We have developed an automated packet-generation tool (that can generate packets) for three packet-generation techniques: the random packet generation technique, the one based on local constraint solving (considering individual rules locally in a policy), and the most sophisticated one based on global constraint solving (considering multiple rules globally in a policy). As generated packets are often large and manual inspection of packet-decision pairs is tedious, we have developed an automated packet reduction tool to reduce the number of packets while keeping the same level of structural coverage.

We have conducted an experiment on a set of real firewall policies with mutation testing [4], which is a specific form of fault injection that creates faulty versions of a policy by making small syntactic changes. We generate six packet sets (for each policy): the first three are generated by the three packet generation techniques, respectively and the other three are their reduced packet sets, respectively. Our experimental results show that among the first three packet sets, a packet set with higher structural coverage (including rule, predicate, and clause coverage) often achieve higher fault-detection capability (i.e., detecting more injected faults), which is measured through the number of "killed mutants" (i.e., detected faults). Our experimental results also show that a reduced packet set achieves similar fault-detection capability with the original packet set.

The rest of the paper is organized as follows. Section 2 presents background information on firewall policies. Section 3 presents a firewall model. Section 4 describes structural coverage of a firewall policy. Sections 5 and 6 illustrate our proposed packet-generation techniques and test reduction technique, respectively. Sections 7 and 8 describe a mechanism of measuring fault-detection capability and an implementation of our framework, respectively. Section 9 illustrates the study of measuring policy coverage and fault-detection capability. Sections 10 and 11 discuss related work and issues, respectively. Section 12 concludes.

2 Firewall Policy

A firewall policy is composed of a sequence of rules that specify under what conditions a packet is accepted and discarded while passing between a private network and the outside Internet. In other words, the policy describes a sequence of rules to decide whether packets are accepted (i.e., being legitimate) or discarded (i.e., being illegitimate). A rule is composed of a set of fields (generally including source/destination IP addresses, source/destination port numbers, and protocol type) and a decision. Each field represents the possible values (to match the corresponding value of a packet), which are either a single value or a finite interval of non-negative integers.

A packet matches a rule if and only if each value of the packet satisfies the corresponding values in the rule. Upon finding a matching rule, the corresponding decision of that rule is derived. When evaluating a packet, the firewall policy follows the first-match semantic: the first matching rule is given the highest priority among all the matching rules.

Figure 1 shows an example of a firewall policy. The symbol "*" denotes that the corresponding field's range (in a rule) is equal to the domain of the field and is satisfied by any packet. An IP address is a 32 bit value (e.g., 192.168.0.0), which is represented as a four-part dotted-decimal address. Classless Inter-domain Routing (CIDR) notation is used to represent IP ranges over an IP address with a subnet mask (e.g., /16 or /24). For example, the range of 192.168.0.0/24 implies IP addresses from 192.168.0.0 to 192.168.0.255. This range consists of all possible IP addresses starting with the same left-most 24 bits (i.e., 192.168.0) on the given IP address. Each of the remaining 8 bits (which do not have fixed values) is either 0 or 1.

The example has three firewall rules r_1, r_2, and r_3. Rule r_1 accepts any packet whose destination IP address is the network $192.168.0.0/16$ (which indicates the range [192.168.0.0, 192.168.255.255]). Rule r_2 discards any packet whose source IP address is the network $1.2.3.0/24$ (which indicates the range [1.2.3.0, 1.2.3.255]) and port is the range $[1, 2^8 - 1]$ with the TCP protocol type. Rule r_3 is a tautology rule to discard all packets. Consider that a packet k whose destination IP address is 192.168.0.0 and protocol type is UDP. When evaluating k, we find that k can match both r_1 and r_3. Among the two rules, as r_1 is the first-matching rule, k is evaluated to be accepted (with regards to the decision of r_1).

3 Firewall Policy Model

This section illustrates a model of a firewall policy based on common generic features. A firewall policy is composed of a sequence of rules, each of which has the form (called the generic representation) as follows.

$$\langle \textit{predicate} \rangle \rightarrow \langle \textit{decision} \rangle \quad (1)$$

A $\langle \textit{predicate} \rangle$ in a rule is a boolean expression over *fields* on which a packet arrives. The $\langle \textit{decision} \rangle$ of a rule can be

Rule	Source IP	Source Port	Destination IP	Destination Port	Protocol	Decision
r_1	*	*	192.168.0.0/16	*	*	accept
r_2	1.2.3.0/24	*	*	$[1, 2^8 - 1]$	TCP	*discard*
r_3	*	*	*	*	*	*discard*

Figure 1. An example firewall policy

accept or *discard* and returned as the evaluation result when the ⟨*predicate*⟩ is evaluated to be true.

A packet k can be viewed as a tuple (fv_1, ..., fv_n) over a finite number of *fields* F_1, ..., F_n, where fv_i is a variable whose values are within a domain, denoted by $D(F_i)$. For example, $D(F_i)$ of the source/destination IP address is $[0, 2^{32} - 1]$, (which indicates $[0.0.0.0, 255.255.255.255]$). In a policy model, we convert values in *fields* (e.g., IP address) to corresponding integer values to simplify a representation format. The ⟨*predicate*⟩ is represented as a conjunction form as follows.

$$F_1 \in S_1 \wedge ... \wedge F_n \in S_n \quad (2)$$

We refer to each $F_i \in S_i$ as a ⟨*clause*⟩, which can be either evaluated to true or false. Table 1 summarizes the notations used in this paper.

The first-match semantic (of a firewall policy) shows the same behavior with the execution of a series of `IF-THEN-ELSE` statements in program code. Given a sequence of rules, the following process is iterated until reaching the last rule: if a ⟨*predicate*⟩ in a rule is evaluated true, then the corresponding decision is returned; otherwise, the next rule (if exists) is evaluated.

T	a test suite
P	a set of predicates of the rules in a policy
C	a set of clauses of the predicates in a policy
K	a set of all valid packets such that $k_i \in D(F_i)$
p_i	a predicate in a rule r_i
c_i	a clause in a predicate
r_i	a rule in a typical firewall representation (e.g., Figure 1)
R_i	a rule in a policy model representation (e.g., Figure 2)
F_i	a field
S_i	a subset of domain $D(F_i)$
$D(F_i)$	domain of field F_i
$C_{p_i}(c_i)$	a constraint of a clause c_i in a predicate p_i
$C(p_i)$	a constraint of a predicate p_i
k_i	a packet
fv_i	a field value in a packet
$P(r_i)$	a path constraint to reach a rule r_i in a policy

Table 1. Summary of notations

$$R_1 : F_1 \in [2, 5] \wedge F_2 \in [5, 10] \rightarrow accept$$
$$R_2 : F_1 \in [6, 7] \wedge F_2 \in [5, 10] \rightarrow discard$$

Figure 2. Example firewall rules

4 Structural Coverage Criteria

In firewall testing, exhaustive testing (i.e., executing all possible test packets) is time consuming, inefficient, and often infeasible. Instead of exhaustive testing, we focus on testing to cover only specific entities (i.e., a predicate tested to be false or true) based on a set of defined coverage criteria.

4.1 Definition

Treating the firewall policy under test as program code (i.e., `IF-THEN-ELSE` statements), we apply structural coverage criteria similar to the ones defined by Ammann et al [3]. In this paper, we define rule, predicate, and clause coverage criteria as follows. Note that P denotes a set of predicates of the rules in the policy under test and C denotes the set of clauses in the predicates in P. T denotes a test suite, which is a set of packet-decision pairs.

Definition 1 ***Rule Coverage Criterion*** *(RCC) for a test suite T requires that for each $r \in R$, the evaluation of the test packets in T needs to match r (i.e, make $p \in P$ to be evaluated to true) at least once, respectively.*

RCC requires that for each predicate p, p is evaluated to true at least once. Figure 2 shows example firewall rules where only two *fields* F_1 and F_2 are used. For example, given two test packets, k_1 (3, 5) and k_2 (6, 10) over two fields F_1 and F_2, both predicates p_1 and p_2 (of R_1 and R_2, respectively) are evaluated to `true` and RCC is achieved by these two test packets.

Definition 2 ***Predicate Coverage Criterion*** *(PCC) for a test suite T requires that for each $p \in P$, the evaluation of the test packets in T needs to make p to be evaluated to true and false at least once, respectively.*

PCC requires that for each predicate p, p is evaluated to true and false at least once. We find that k_1 and k_2 do not evaluate p_2 to `false`. Therefore, we require one more

packet such as k_3 (6, 11) that evaluates p_2 to be `false` to achieve PCC.

Covering every predicate in a firewall requires at most $2n$ test packets, where n is the number of rules. However, the minimal number of test packets (for PCC) could be less than $2n$ because a single test packet can satisfy multiple true or false branches of predicates. As RCC and PCC do not require each clause to be covered, we then define clause coverage criterion (CCC), which specifically targets at covering each clause in a predicate.

Definition 3 ***Clause Coverage Criterion (CCC)*** *for a test suite T requires that for each $c \in C$, the evaluation of the test packets in T needs to make c to be evaluated to true and false at least once, respectively.*

In CCC, each clause is required to be evaluated to `true` and `false` at least once independently from other clauses. Consider that p_1 includes two clauses c_1 and c_2 (with regards to F_1 and F_2, respectively). Figure 3 illustrates four test packets that evaluate all combinations of `true` and `false` (of c_1 and c_2) and the corresponding boolean value of p_1 (in Column 5). There are several ways to cover clauses in p_1: (1) select k_2 and k_3 or (2) select k_1 and k_4. However, instead of the first selection, the second selection has an advantage to increase the coverage in terms of RCC and PCC.

	packets	c_1	c_2	$p_1 = (c_1 \wedge c_2)$
k_1	(3,5)	T	T	T
k_2	(6,10)	F	T	F
k_3	(3,11)	T	F	F
k_4	(6,11)	F	F	F

Figure 3. Sample packets for all combinations of `true` and `false` values of clauses c_1 and c_2

4.2 Measurement

We have developed a coverage measurement tool that monitors whether rules, predicates, or clauses are covered when evaluating packets against the policy under test. For each structural coverage criterion, we define coverage measurements as follows.

Rule coverage measurements. For the rule coverage criterion, *rule coverage* is the percentage of the number of covered rules (i.e., predicates being evaluated to true) in R over $|R|$.

Predicate coverage measurements. For the predicate coverage criterion, *predicate coverage* is the percentage of the number of covered predicates (i.e., predicates being evaluated to true or false) in P over $2 \times |P|$.

Clause coverage measurements. For the clause coverage criterion, *clause coverage* is the percentage of the number of covered true or false values of clauses in C over $2 \times |C|$.

4.3 Structural Coverage and Fault Detection

Policy testers may generate and select a test suite to achieve a certain (high) level of coverage. However, our main objective, through testing, is to detect faults in the firewall policy while reaching a certain level of coverage. Coverage analysis helps investigate a larger portion of entities for fault detection using a test suite that achieves higher structural coverage.

Consider that a fault in entities (i.e., rules, predicate, or clause) may cause to output incorrect decisions when evaluating some packets. A fault in a rule's decision (e.g., using `accept` by mistake instead of `discard`) is discovered if and only if the rule is covered and the derived decision is verified. A test suite with high rule coverage may detect such faults easily and increase our confidence on the correctness of the policy against such faults. Similarly, a test suite with high predicate/clause coverage may have a high chance to detect faults in predicates/clauses. Therefore, we are interested in covering each entity at least once to exercise a wide range of the policy's behavior.

5 Test Packet Generation

As manually generating packets for testing policies is tedious, we have developed three techniques to automatically generate packets for a policy under test. The objective is to generate packets for achieving high structural coverage.

In this section, p and $C(p)$ denote a predicate and its constraint, respectively. To evaluate p to be `true` (`false`), a packet should satisfy the constraint $C(p)$ ($\neg\ C(p)$) (for the `true` (`false`) branch of p). $C(p)$ is represented in the form of $C_p(c_1) \wedge$ .... $\wedge\ C_p(c_n)$, where $C_p(c_1)$, ..., $C_p(c_n)$ are the constraints of the clause c_1, ..., c_n in p, respectively.

5.1 Random Packet Generation Technique

The random packet generation technique is straightforward. A packet k is in the form of (k_1, ..., k_n), where k_1, ..., k_n are numeric values over *fields* (such as source addresses), whose domains are denoted by $D(F_1)$, ..., $D(F_n)$. Given the domains of the policy under test, the generator for the technique automatically generates a packet k by randomly selecting k_i (within the domain $D(F_i)$). While the

technique does not require the policy itself in test generation and can quickly generate a large number of test packets, the technique often lacks effectiveness to achieve high structural coverage with the generated packets. Due to randomness, the number of the entities (i.e., predicates or clauses) being covered is often small in comparison to the total number of the entities in the policy under test.

5.2 Packet Generation Technique based on Local Constraint Solving

In general, test generation should focus on generating packets to cover those entities (i.e., predicates and clauses) that have not been covered previously. Different from the preceding technique, the technique based on local constraint solving statically analyzes rules to generate test packets. Given a policy, the packet generator for the technique analyzes the entities in an individual rule and generates packets to evaluate the constraints (i.e., conditions) of the entities to be true or false. The technique takes into account local constraints (given by a rule) without considering the impact of other rules in the policy.

More specifically, the generator constructs constraints $C(p)$ and $\neg C(p)$ (for both `true` and `false` branches of p). The generator generates a packet based on the concrete values to satisfy each constraint. However, the generated packets may not cover each clause (to be `true` and `false`). To target at covering many clauses, the generator constructs combinations of $C_p(c_i)$ and $\neg\ C_p(c_i)$. For example, combinations $C_p(c_1) \wedge .... \wedge C_p(c_n)$ (for `true` branches of all clauses) and $\neg C_p(c_1) \wedge .... \wedge \neg C_p(c_n)$ (for `false` branches of all clauses) can be considered.

There are two major limitations of the technique. First, the generated packets may fail to cover target entities due to overlapping predicates (i.e., predicates that can be satisfied by the same packet) across multiple rules. As shown in Figure 1, a packet k (whose destination IP address is $192.168.0.0$ and protocol type is UDP) satisfies the predicates of both r_1 and r_3 but fail to be evaluated against r_3, which can be k's potential target entities. Second, the technique cannot determine whether a structural entity could be covered in advance. Some rules may be completely shadowed by other rules and never evaluated. In such cases, there is no criterion to decide whether to generate additional packets (based on other more capable solutions to solve the same constraints) or stop testing.

5.3 Packet Generation Technique based on Global Constraint Solving

To better generate packets to cover target entities, our generator (based on global constraint solving) analyzes the policy under test and generates packets by solving global constraints (collected from the policy). The motivation of global constraint solving is to take into account the influence of overlapping predicates across rules. Covering entities in a rule requires that the predicates of all the preceding rules should be evaluated to false. To find such entities, we define rule reachability as follows.

Definition 4 ***Rule reachability*** *of a test packet k to reach a rule $r_i \in R$ (denoting the set of rules in the policy) requires that for each predicate $p_1, ..., p_i \in P$ (denoting the set of predicates in R), the evaluation of the test packet k needs to evaluate r_i's preceding predicates $p_1, ..., p_{i-1}$ to false and p_i to true or false.*

All the reachable rules are feasible to be evaluated by packets. Otherwise, infeasible rules could be detected and removed similar to dead code in programs.

More specifically, to cover entities in a rule r_i, we explore a (path) constraint $P(r_i)$ that represents rule r_i reachability. $P(r_i)$ is represented as the form of $\neg C(p_1) \wedge .... \wedge \neg C(p_{i-1})$ where $C(p_1)$, ..., $C(p_{i-1})$ are the predicate constraints in the preceding rules r_1, ..., r_{i-1}. Given the path constraint $P(r_i)$, to cover the predicate p_i in r_i, the generator constructs two constraints $P(r_i) \wedge C(p_i)$ and $P(r_i) \wedge \neg C(p_i)$. As the generator generates packets based on solutions of constraints $P(r_i) \wedge C(p_i)$ and $P(r_i) \wedge \neg C(p_i)$, the packets reach r_i and exercise r_i's `true` and `false` branches, respectively. To cover many clauses in a rule r_i, the generator constructs constraints as follows. The generator conjuncts $P(r_i)$ with the combinations of $C_p(c_i)$ and $\neg\ C_p(c_i)$ in r_i. In summary, $P(r_i)$ is additionally used upon the preceding technique to cover target entities by taking into account the impact of overlapping predicates in the preceding rules.

The generator generates packets based on solutions for the collected constraints. This technique is useful to generate packets with high structural coverage by taking into account the impact of the preceding rules of a target rule. However, this technique requires higher analysis time (e.g., constraint-solving cost) than the two preceding techniques.

6 Test Reduction

The number of generated packets can be high. In such cases, it is tedious for the policy authors to manually inspect a test suite, which is a set of packet-decision pairs. Therefore, we should reduce the size of the test suite for inspection without incurring substantial loss in fault-detection capability. Since structural coverage is an important factor for reflecting fault-detection capability, we can reduce the size of the test suite while keeping its coverage level. In our test reduction, we apply the minimization technique used by Martin et al. [13] to reduce test packets.

Given a packet set, we evaluate a packet against the policy under test one by one. We use a greedy algorithm that removes a packet from the packet set if and only if evaluating the packet does not increase any of the coverage metrics that are achieved by previously evaluated packets in the packet set. Therefore, the reduced packet set is a set of packets where each packet contributes to cover (not-yet-covered) new entities and increase coverage. However, through such a greedy algorithm, this reduced set may not be a "minimum set", which includes the smallest possible number of packets that achieve the same coverage level as the original packet set.

7 Measuring Fault-Detection Capability

Fault detection is a central focus of any testing process. In this paper, we use mutation testing [4] to measure fault-detection capabilities of a test suite. In policy mutation testing, we inject a fault into the original policy and thereby create a mutant (faulty version). Injected faults can be of various types including simple mistakes (e.g., incorrect decision in a rule) and complex configuration errors involving multiple rules. The intuition behind mutation testing is that if a policy contains a fault, there will usually be a set of mutants that can be detected (killed) only by a test that also detects that fault. In other words, the ability to detect small, minor faults such as mutants implies the ability to detect complex faults.

When different decisions are produced by the evaluations of the same test packet on the original policy and its mutant, the test packet is adequate to detect the fault in the mutant and we say that the mutant is "killed". When various mutants are used, fault-detection capability of a test suite is measured through the mutant-killing ratio, which is the number of mutants killed by the test suite divided by the total number of mutants.

Table 2 shows the chosen mutation operators for firewall policies and their descriptions. Mutation operators may change predicates, clauses, or decisions of a policy. We classify mutation operators into two groups: (1) rule-level mutation operators including RPT, RPF, CRO, CRD, and RMR and (2) clause-level mutation operators including RCT, RCF, $CRSV$, $CREV$, $CRSO$, and $CREO$. The first group modifies a predicate, rule order, or a decision in a rule. The number of generated mutants with each mutation operator is equal to the number of rules of the policy. The second group modifies a clause in a rule. The number of generated mutants with each mutation operator is equal to the number of clauses. The total number of generated mutants is proportional to the number of rules, predicates, and clauses of the policy. Note that the mutant generator for each mutation operator may generate equivalent mutants, which are mutants with the same behaviors as the original policy; an equivalent mutant cannot be killed by any test packet.

In our approach, we apply mutation testing to investigate the relationship between firewall policy structural coverage achieved by a packet set and the packet set's fault-detection capability.

8 Implementation

Our implementation (written in Java) includes four components: test generation, test evaluation, packet reduction, and mutation generation. In the test generation component, for packet generation based on local constraint solving, our packet generator selects random values (that satisfy a given constraint) for each field value of a test packet. For packet generation based on global constraint solving, we leveraged a theorem prover called Z3[1]. The component statically analyzes the policy under test. Then the component automatically collects and converts each constraint to the SMT-LIB (Satisfiability Modulo Theories Library) format[2]. Z3 analyzes this constraint and finds concrete solutions (i.e., numeric values), each of which is transformed to a test packet. If no solution exists, Z3 outputs *unsolvable*.

In the test evaluation component, as no universal firewall evaluation engine is available, we developed a generic firewall evaluation engine to simulate evaluating packets against the policy under test. The engine parses and stores rules as a `List`. When evaluating a packet, the engine searches for the first-applicable rule and outputs the rule's decision. Through packet evaluation, the evaluation engine also monitors the evaluation process of all the packet/decision pairs and which predicates and clauses in the policy are covered. The engine also automatically compares the evaluated decisions (on the policy and the mutated policies) and log "killed" mutant information if the decisions are inconsistent.

In the packet reduction component, our packet reduction tool observes the details of covered entities and their covering packets as well as the details of uncovered entities when evaluating a packet set. In the mutation generation component, our mutator automatically generates mutant policies by modifying the policy under test using the selected mutation operator.

9 Experiments

We carried out our experiments on a laptop PC running Windows XP SP2 with 1G memory and dual 1.86GHz Intel Pentium processor. The packet generation tool generates

[1] `http://research.microsoft.com/projects/z3/`
[2] `http://www.smtlib.org/`

Table 2. Mutation operators for policy mutation testing.

Name	Description
Rule Predicate True (RPT)	A rule is applied to all packets by modifying every clause range to "*".
Rule Predicate False (RPF)	A rule is never applied to any packet by modifying every clause range to an invalid range (e.g., $[10, 5]$).
Rule Clause True (RCT)	A clause c_i is applied to the field value fv_i of all packets by modifying the clause range to "*".
Rule Clause False (RCF)	A clause c_i is never applied to the field value fv_i of all packets by modifying the clause range to an invalid range (e.g., $[10, 5]$).
Change Range Start point Value ($CRSV$)	The range in a clause is changed by modifying the start point value randomly.
Change Range End point Value ($CREV$)	The range in a clause is changed by modifying the end point value randomly.
Change Range Start point Operator ($CRSO$)	The range in a clause is changed by increasing the start point value by one.
Change Range End point Operator ($CREO$)	The range in a clause is changed by decreasing the end point value by one.
Change Rule Order (CRO)	Rule order is changed by exchanging the locations of two adjacent rules.
Change Rule Decision (CRD)	A rule's decision is inverted (i.e., `accept` to `discard` or `discard` to `accept`).
Remove Rule (RMR)	No packet can be applied to the rule simply by removing the rule in a policy.

packet sets using the three techniques (random packet generation, packet generation based on local constraint solving, and one based on global constraint solving). We use $Rand$, $Local$, and $Global$ to denote the packet sets generated by these three techniques, respectively. For each policy, we measured the structural coverage of each packet set and reduce the size of each packet set while keeping the same level of structural coverage. We use $R{-}Rand$, $R{-}Local$, and $R{-}Global$ to denote the reduced packet sets, respectively.

The mutator generates mutants (using the defined mutation operators) by seeding faults in each policy (with one mutant including one seeded fault). For each policy and its mutants, the evaluation engine checked if a mutant is "killed" and measured mutant-killing ratios of each packet set (i.e., the number of mutants killed by the packet set divided by the total number of mutants).

We compare our proposed three packet-generation techniques in terms of effectiveness to achieve structural coverage by the generated packet sets. In order to investigate the effect of structural coverage on fault-detection capability, we aim to demonstrate that packet sets with higher coverage can detect more faults than packet sets with lower coverage. We have also conducted the same experiment with reduced packet sets to further investigate whether this reduction significantly affects their fault-detection capability.

For each policy and packet set, the following metrics are measured: structural coverage percentage (rule coverage, predicate coverage, and clause coverage percentage), mutant-killing ratios, the number of packets in generated packet sets and reduced sets, and packet generation time.

9.1 Instrumentation

We conducted experiments on 14 real-life firewall policies collected from a variety of sources. We obtained 42 real-life firewall policies from distinct network services. As some firewall policies from the same network service provider have similar structure, we choose 14 real-life firewall policies with distinct structures. These policies' format is the same as described in Figure 1.

We generated about two packets per rule for each policy and packet-generation technique. The random packet-generation technique generated $n \times 2$ random packets, where n is the number of rules. For the local and global constraint-solving packet-generation techniques, we first generated the following two constraints for each rule: (1) a constraint for evaluating every clause in the rule to true and (2) a constraint for evaluating clauses, each of which is within (but not equal to) its domain, to false and the remaining clauses (which subsume their domains) to true. Because many clauses in firewall policies subsume their domains (e.g., clauses with "*" marks in Figure 1) and these clauses cannot be evaluated to false, we evaluated such clauses to true in the second constraint as described earlier. The lo-

Policy	# Rules	# Mutants	# Packets			# Reduced packets			Gen time (ms)
			Rand	Local	Global	R-Rand	R-Local	R-Global	Global
1 (Firewall1)	3	77	6	6	6	1	3	3	112
2 (BACKUP-IN-ACL)	5	135	10	10	9	1	5	4	172
3 (LAN-OUT-ACL)	28	655	56	56	43	1	17	17	1040
4 (MAIL1SLURM-OUT-ACL)	18	486	36	36	26	1	10	9	622
5 (MAILHOST-OUT-ACL)	26	689	52	52	44	2	18	19	896
6 (MAILHOST2-OUT-ACL)	26	703	52	52	39	2	14	14	919
7 (MAILHOST3-OUT-ACL)	27	724	54	54	41	3	15	15	982
8 (MAILHOST4-OUT-ACL)	28	730	56	56	53	3	24	27	969
9 (NEWS-OUT-ACL)	14	370	28	28	25	2	11	11	483
10 (NS3-OUT-ACL)	17	459	34	34	29	1	11	13	579
11 (RCPROTO1MO1-OUT-AC	23	640	46	46	34	3	14	11	810
12 (RCPROTO1MX1-OUT-AC	6	162	12	12	11	1	6	5	215
13 (SSH-OUT-ACL)	16	431	32	32	23	1	8	7	542
14 (WAN-IN-ACL)	24	658	48	48	40	2	19	17	843
Aerage	18.64	494.21	37.29	37.29	30.21	1.71	12.50	12.29	656.04

Figure 4. Experimental results on firewall policies

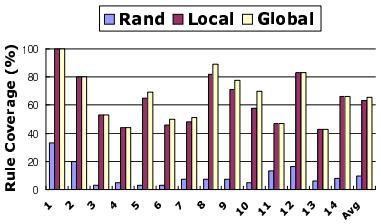

Figure 5. Rule coverage achieved by each packet set

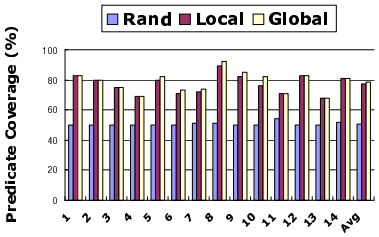

Figure 6. Predicate coverage achieved by each packet set

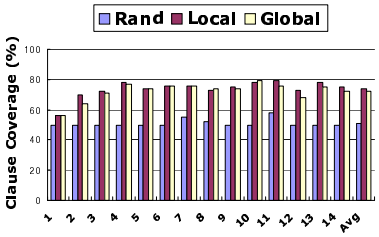

Figure 7. Clause coverage achieved by each packet set

cal constraint-solving packet-generation technique generated $n \times 2$ packets. The global constraint-solving packet-generation technique conjuncts the path constraint for a target rule with its two preceding constraints to form a new constraint for solving. If the new constraint is found to be infeasible (due to the impact of the path condition), this technique cannot generate packets to satisfy such constraints and may include fewer than $n \times 2$ packets.

When generating mutants, mutation operators may generate a mutated policy that is the same (syntactically) as the original policy. As such a mutant does not include any fault, we excluded the mutant.

9.2 Comparison of Structural Coverage

Figure 4 shows the basic statistics of each firewall policy. Columns 1-3 show subject names, numbers of rules, and generated mutants for each firewall policy. Column "# Packets" shows the size of the generated packet sets $Rand$, $Local$, and $Global$, respectively for each packet generation technique. Columns 7-9 show the size of their reduced packet sets ($R{-}Rand$, $R{-}Local$, and $R{-}Global$), respectively. Column 10 shows the analysis time (in milliseconds) for generating $Global$ (the most costly one among the three techniques) and this analysis time also includes the time to generate and solve constraints.

$Global$ may contain fewer packets than $Rand$ and $Local$. The reason is that when solving a global constraint, the constraint can be infeasible to be solved and a constraint solver returns a decision of *unsolvable* — no packets are generated based on the decision. The analysis time for $Rand$ and $Local$ is not shown in Figure 4. The reason is that the time is too short to be measured in milliseconds and negligible (in comparison with that of $Global$).

Figures 5, 6, and 7 show the rule, predicate, clause coverage metrics, respectively, of each policy achieved by $Rand$, $Local$, and $Global$. We observe that $Rand$ achieved the lowest structural coverage. The reason is that randomly generated field values in generated packets have a low chance of satisfying constraints for a rule, predicate, or clause. We observe that $Global$ achieves higher rule/clause coverage than other packet sets. This observation is consistent with our expectation described in Section 5. On average, the $Global$ is approximately 2% and 56% higher than $Local$ and $Rand$ in terms of rule/clause coverage.

We also observe that for clause coverage, $Global$ achieves approximately similar (sometimes less) coverage with $Local$. As illustrated earlier, $Global$ may include fewer packets based on the constructed constraints. When a constraint is found to be infeasible, we did not take into account other clause-constraint combinations, which may be feasible to solve for covering some of uncovered clauses.

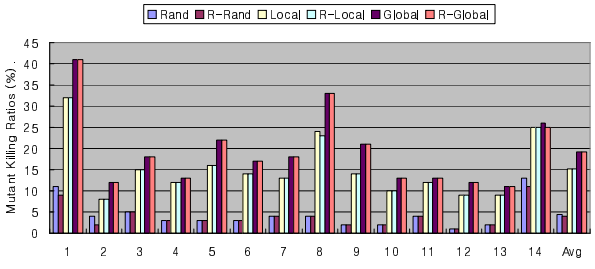

Figure 8. Mutant-killing ratios for all operators by subjects.

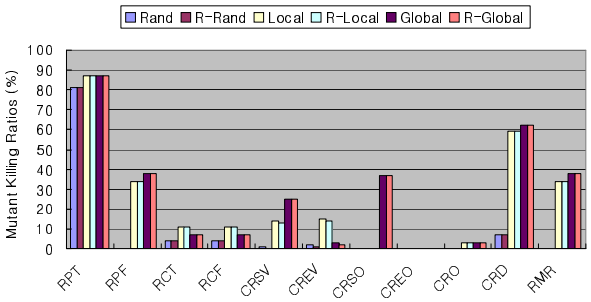

Figure 9. Mutant-killing ratios for all subjects by operators.

Instead, $Local$ may cover some (but not all) target clauses among such uncovered clauses. Furthermore, as our subjects have only a few or no overlapping predicates across rules, the packet-generation technique based on local constraint solving could generate a packet set with almost the highest structural coverage. If predicates are more complex, we expect that $Global$ shall perform better than $Local$.

9.3 Comparison of Fault-Detection Capability

To find correlation between each structural coverage and mutation-killing ratios, we classify mutation operations into two categories, rule-level and clause-level mutation operators (explained in Section 7).

Figure 8 shows the average mutant killing ratios for all operators by policies. We observe that the mutant killing ratios are similar over the generated packet sets and their reduced packet set. The largest ratio difference between the generated packet sets and their reduced packet set is less than 2%. $Rand$ and $R{-}Rand$ show the lowest mutant-killing ratios. As $Rand$ contains the largest number of packets and the lowest mutant-killing ratios, we observe that the size of a packet set is not highly correlated with fault-detection capability. We also observe that $Global$ and $R{-}Global$ achieve the highest mutant-killing ratios. This result is expected as the evaluation of these packet sets can involve more structural entities than the other packet sets.

We next present more details about mutants being killed. Figure 9 shows the average mutant killing ratios for all policies by operators. For rule-level mutation operators, we observe that $Global$ and $R{-}Global$ achieve highest mutant-killing ratios. The reason is that the highest rule/predicate coverage achieved by $Global$ and $R{-}Global$ helps exercise more rules and detect faults in rules.

Among clause-level mutation operations, $Local$ and $R{-}Local$ achieve the highest mutant-killing ratios over RCT, RCF, $CREV$, and $CREO$ mutated policies. As $Local$ and $R{-}Local$ evaluate more clauses to true or false, the packet sets are more effective to detect faults in a larger portion of clauses in the policy. $Global$ and $R{-}Global$ detect more faults in $CRSV$ and $CRSO$ mutated policies. The reason is that a packet in $Global$ is based on a solution for a given constraint, which is solved by the constraint solver Z3. Among all the possible solutions, Z3 tries to output solutions with lower-boundary values in the constraint. Therefore, $Global$ and $R - Global$ are effective to detect faults caused by the change of the start value of a clause over other packet sets.

10 Related Work

For testing access control policies such as XACML policies [1], Martin et al. [13] proposed policy testing approaches that define and measure policy structural coverage. They also proposed to mutate policies [12], and generate random requests automatically. Their proposed structural coverage criteria and mutation operators are not directly applicable to firewall policies due to the semantic and syntactic difference. Our approach includes more advanced test generation techniques (based on local and global constraint solving) than random techniques.

Some researchers proposed firewall testing with test cases generated based on their proposed criteria. Jürjens et al. [8] proposed specification-based testing, which generates test sequences to cover a state transition model of a firewall and its surrounding network. El-Atawy et al. [5] proposed policy criteria identified by interactions between rules. Their proposed criteria are used to test the correctness of a firewall implementation instead of that of the firewall policy itself, which our approach focuses on. In addition, we also use mutation testing to evaluate our approach.

Several firewall policy testing techniques [7, 9, 11] inject packets into a firewall and check whether the decisions of the firewall concerning the injected packets are correct.

These techniques lack rigorousness such as the use of coverage criteria and effective tools for generating covering tests. In contrast, our approach is based on solid foundations and advanced test-packet generation techniques.

11 Discussion and Future Work

We believe that our approach could be practical and effective to detect real faults in firewall policies. Real faults may consist of one or several simple faults described in our proposed mutation operators (in Table 2). Therefore, the detection of real faults often depends on that of the simple faults, which are shown to be detected more effectively by a packet set with higher structural coverage. In addition, the policy testers manually compare the evaluated decisions and their expected decisions to detect such faults. In such cases, our packet-reduction mechanism is useful to reduce the size of test packets.

Our current approach tests a (stateless) firewall policy. We plan to extend our approach to a stateful firewall. In a stateful firewall, the decision is not solely dependent on a packet itself, but also on other resources such as previously accepted packets and dynamic features in a firewall [6]. Testing such firewalls, which are represented as finite state machines (FSM), requires to generate a series of packets to cover states, edges, or paths in the FSM. We plan to extend our proposed test coverage criteria to address covering states, edges, or paths in the FSM.

12 Conclusion

We have developed a systematic structural testing approach for firewall policies. We defined three types of structural coverage for firewall policies: rule, predicate, and clause coverage criteria. Among the three proposed packet generation techniques, the global constraint solving technique often generated packet sets to achieve the highest structural coverage. Generally, our experimental results showed that a packet set with higher structural coverage has higher fault-detection capability (i.e., detecting more injected faults). Our experimental results showed that a reduced packet set (maintaining the same level of structural coverage with the corresponding original packet set) maintains similar fault-detection capability with the original set.

Acknowledgment

This work is supported in part by NSF grant CNS-0716579, NSF grant CNS-0716407, and MSU IRGP Grant.

References

[1] OASIS eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/xacml/`, 2007.

[2] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proc. 23rd Conf. IEEE Communications Soc.*, pages 2605–2616, 2004.

[3] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *Proc. 14th International Symposium on Software Reliability Engineering*, pages 99–107, 2003.

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

[5] A. El-Atawy, T. Samak, Z. Wali, E. Al-Shaer, F. Lin, C. Pham, and S. Li. An automated framework for validating firewall policy enforcement. In *Proc. 8th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 151–160, 2007.

[6] M. G. Gouda and A. X. Liu. A model of stateful firewalls and its properties. In *Proc. International Conference on Dependable Systems and Networks*, pages 128–137, 2005.

[7] D. Hoffman and K. Yoo. Blowtorch: a framework for firewall test automation. In *Proc. 20th IEEE/ACM international Conference on Automated software engineering*, pages 96 – 103, 2005.

[8] J. Jürjens and G. Wimmel. Specification-based testing of firewalls. In *Proc. 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 308–316, 2001.

[9] A. X. Liu, M. G. Gouda, H. H. Ma, and A. H. Ngu. Nonintrusive testing of firewalls. In *Proc. 1st International Computer Engineering Conference*, pages 196–201, 2004.

[10] S. W. Lodin and C. L. Schuba. Firewalls fend off invasions from the net. *IEEE Spectr.*, 35(2):26–34, 1998.

[11] M. R. Lyu and L. K. Y. Lau. Firewall security: Policies, testing and performance evaluation. In *Proc. 24th International Conference on Computer Systems and Applications*, pages 116–121, 2000.

[12] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. 16th International Conference on World Wide Web*, pages 667–676, 2007.

[13] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. 8th International Conference on Information and Communications Security*, pages 139–158, 2006.

[14] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[15] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for FIREwall Modeling and ANalysis. In *Proc. IEEE Symposium on Security and Privacy*, pages 199–213, 2006.

[16] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.