Systematic Structural Testing of Firewall Policies

JeeHyun Hwang, Tao Xie, Fei Chen, and Alex X. Liu

***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, 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 firewall policy under test. To achieve high structural coverage effectively, we have developed four automated packet generation techniques: the random packet generation, the one based on local constraint solving (considering individual rules locally in a policy), the one based on global constraint solving (considering multiple rules globally in a policy), and the one based on boundary values.**

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.

***Index Terms*—Firewall policy, validation, test packet generation, structural coverage, fault detection.**

I. 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 such 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 [1]. 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 [2]. 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. 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 [3], [4]. 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 [5]–[7]. 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 [8] 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. Moreover, our systematic firewall policy testing helps detect faults with the test packets, which often do not follow some configuration mistake patterns (e.g., anomalies [3], [4] and configuration errors [2]). Intuitively, policy testers shall generate test packets to achieve high structural coverage, which helps investigate a large portion of policy entities for fault detection.

As it is tedious for policy testers to manually generate test packets for firewall policies, we have developed an automated packet-generation tool (that can generate packets) for four packet-generation techniques: the random packet generation technique, the one based on local constraint solving (considering individual rules locally in a policy), the one based on global constraint solving (considering multiple rules globally in a policy), and the one based on boundary values. As gener-

Manuscript received December 4, 2010; revised May 23, 2011 and October 8, 2011. The associate editor coordinating the review of this paper and approving it for publication was E. Bertino.

The preliminary version of this paper titled "Systematic Structural Testing of Firewall Policies" was published in the Proceedings of the 2008 IEEE International Symposium on Reliable Distributed Systems (SRDS), pages 105–114.

J. Hwang and T. Xie are with the Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-8206 (e-mail: jhwang4@ncsu.edu, xie@csc.ncsu.edu).

F. Chen and A. X. Liu are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1266 (e-mail: {feichen, alexliu}@cse.msu.edu).

Digital Object Identifier 10.1109/TNSM.2012.012012.100092

1932-4537/12/$31.00 © 2012 IEEE

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*

Fig. 1. An example firewall policy.

ated 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 [9], which is a specific form of fault injection that creates faulty versions of a policy by making small syntactic and semantic changes. We generate packet sets (for each policy) with the packet generation techniques. Our experimental results show that 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). On the comparison of packet sets and their reduced packet sets, 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 II presents background information on firewall policies. Section III presents a firewall policy model. Section IV describes structural coverage criteria of a firewall policy. Section V illustrates a framework of our proposed packet-generation techniques, test reduction technique, and a mechanism of measuring fault-detection capability. Section VI describes an implementation of our framework. Section VII illustrates our experiments of measuring policy coverage and fault-detection capability. Section VIII discusses related work. Section IX concludes the paper.

II. 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 range of 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 whose destination IP address is $192.168.0.0$ and protocol type is UDP. When evaluating the packet, we find that the packet can match both r_1 and r_3. Among the two rules, as r_1 is the first-matching rule, the packet is evaluated to be accepted (with regards to the decision of r_1). If a packet matches no rules in a firewall policy, there exists an implicit last tautology rule to discard the packet.

III. 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.

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

A $\langle \mathit{predicate} \rangle$ in a rule is a boolean expression over *fields* on which a packet arrives. The $\langle \mathit{decision} \rangle$ of a rule can be *accept* or *discard* and returned as the evaluation result when the $\langle \mathit{predicate} \rangle$ is evaluated to be true.

The $\langle \mathit{predicate} \rangle$ is represented as a conjunction form as follows.

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

In a policy model, we represent a value in a *field* F_i (e.g., IP address) with its corresponding range S_i (e.g., $F_i \in$ [2,5]) to simplify the representation format. We refer to each $F_i \in S_i$ as a $\langle \mathit{clause} \rangle$, which can be evaluated to either true or false. Table I 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 $\langle \mathit{predicate} \rangle$ in a rule is evaluated true, then the corresponding decision is returned; otherwise, the next rule (if exists) is evaluated.

TABLE I
SUMMARY OF NOTATIONS

P	a set of predicates of the rules in a policy
C	a set of clauses of the predicates in a policy
r_i	a rule in a firewall policy
p_i	a predicate in a rule r_i
c_i	an ith clause in a predicate
F_i	a field (e.g., IP address)
D_i	domain of field F_i (e.g., $[0, 2^{32}-1]$ for the IP address)
S_i	a subset of domain D_i (e.g., [2,5])
$C_{p_i}(c_j)$	a constraint of a clause c_j in a predicate p_i
$C(p_i)$	a constraint of a predicate p_i
k	a packet

$$r_1 : F_1 \in [2,5] \wedge F_2 \in [5,10] \rightarrow accept$$
$$r_2 : F_1 \in [6,7] \wedge F_2 \in [5,10] \rightarrow discard$$

Fig. 2. Example firewall rules.

IV. STRUCTURAL COVERAGE CRITERIA

In firewall testing, exhaustive testing (i.e., executing all possible test packets) is time consuming and inefficient. 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.

A. 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 [10]. In this paper, a test suite is a set of packet-decision pairs to check whether a packet is evaluated to its corresponding expected decision. We define rule, predicate, and clause coverage criteria as follows.

Definition 1: **Rule Coverage Criterion** (**RCC**) for a test suite requires that for each rule r in a policy, the evaluation of the test packets in the test suite needs to match r (i.e., make r's predicate p to be evaluated to true) at least once, respectively.

In other words, 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`. RCC is achieved by these two test packets. More specifically, k_1 evaluates p_1 to `true`, causing r_1's decision to be returned without further evaluating p_2. k_2 evaluates p_1 to `false` and next evaluates p_2 to `true`. Note that when a packet finds the first-matching rule r (i.e., evaluating a predicate to true), policy evaluation stops and returns r's decision as a final decision.

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

To achieve PCC, in addition to k_1 and k_2, we require one more packet such as k_3 (6, 11) that evaluates p_2 to `false`. Figure 3 illustrates these three test packets that evaluate

packet	p_1	p_2	matching rule
k_1: (3, 5)	True	N/A	r_1
k_2: (6, 10)	False	True	r_2
k_3: (6, 11)	False	False	N/A

Fig. 3. Sample packets for all combinations of `true` and `false` values of predicates p_1 and p_2.

packet	c_1	c_2	$p_1 = (c_1 \wedge c_2)$
k_1: (3, 5)	True	True	True
k_2: (6, 10)	False	True	False
k_3: (3, 11)	True	False	False
k_4: (6, 11)	False	False	False

Fig. 4. Sample packets for all combinations of `true` and `false` values of clauses c_1 and c_2.

all combinations of `true` and `false` (of p_1 and p_2). `N/A` represents a not-applicable predicate or rule during packet evaluation. For k_1, we mark `N/A` in p_2's evaluation because k_1's decision is determined without further evaluating p_2.

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 requires that for each clause c of the predicates in a policy, the evaluation of the test packets in the test suite 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). Note that the boolean value of p_1 is equal to $c_1 \wedge c_2$. Figure 4 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. 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.

B. Structural Coverage

We have developed three structural coverage measurements that monitor 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 a policy.

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 $2 \times |P|$.

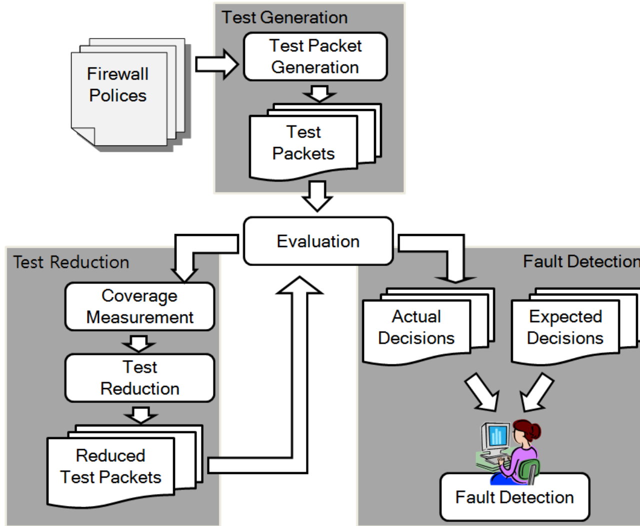


Fig. 5. Framework overview.

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 × $|C|$.

C. 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.

V. Framework

This section presents our framework for testing firewall policies. Figure 5 shows the overview of framework of our approach. Our framework includes three phases: test packet generation, test reduction, and fault detection. In the test packet generation phase, our test packet generation component analyzes a firewall policy and generates test packets to cover entities (e.g., predicates and clauses) in the policy. We propose four different test packet generation techniques described in Section V-A. In the test reduction phase, the test reduction component reduces the number of packets based on coverage criteria by including only packets that help increase policy coverage measurement during evaluation. In the fault detection phase, the policy authors manually inspect whether the actual decisions (i.e., evaluated decisions of the generated packet against the firewall policy) are consistent with expected decisions. If the authors find any inconsistent decisions, the authors determine that they detect a fault in the policy.

A. Test Packet Generation

As manually generating packets for testing policies is tedious, we have developed four techniques to automatically generate packets for the policy under test. The objective is to generate packets for achieving high structural coverage. This section describes four packet generation techniques (developed in our approach): the random packet generation technique, the packet generation technique based on local constraint solving, the packet generation technique based on global constraint solving, and the packet generation technique based on boundary values. The key difference between the second and third techniques is the scope (i.e, local or global) of constraints used in the packet generation. While the second and third techniques generate packets based on random values within values solved by each constraint solving, the fourth one generates test packets based on boundary values within values solved by local constraint solving.

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.

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_1, ..., D_n). Given the domains of the policy under test, the generator for the technique automatically generates a packet k by randomly selecting k_1, ..., k_n (within the domain D_1, ..., D_n, respectively). 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.

2) Packet Generation Technique based on Local Constraint Solving: In general, packet 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 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) for each rule. The generator generates a packet based on the concrete values to satisfy each constraint. As the generator generates packets based on satisfying constraints in predicates, the

generated packets may not be effective in covering 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 fails 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.

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 packet k to reach a rule r_i in a policy requires that k evaluates r_i's preceding rules' predicates to false and reaches the rule.

We may generate packets to reach and evaluate all the reachable rules in the policy. To cover entities in a rule r_i, we explore a (path) constraint $Path(r_i)$ that represents rule r_i reachability. $Path(r_i)$ is additionally used upon the preceding technique to cover target entities by taking into account the impact of overlapping predicates in the preceding rules.

More specifically, $Path(r_i)$ 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 $Path(r_i)$, to cover the predicate p_i in r_i, the generator constructs two constraints $Path(r_i) \wedge C(p_i)$ (for the `true` branch of p_i after reaching r_i) and $Path(r_i) \wedge \neg C(p_i)$ (for the `false` branch of p_i after reaching r_i). As the generator generates packets based on solutions of constraints $Path(r_i) \wedge C(p_i)$ and $Path(r_i) \wedge \neg C(p_i)$, 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 $Path(r_i)$ with the combinations of positive or negative constraints of clauses in r_i.

Given the constraints, 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.

Algorithm 1: Packet Generation Technique based on Boundary Values

```
Input: Firewall policy P where n is the number of rules, r_1, r_2, ...,
       r_n. C_p(c_1), ..., C_p(c_m) where each C_p(c_j) is the jth clause
       constraint of constraints C_p of a rule. D_1, D_2, ..., D_m where
       D_j is a domain of the jth field.
Output: A set of packets.
output = ⟨⟩;
for i := 1 to n do
    C_p = constraints of r_i;
    for j := 1 to m do
        k_j = MinValue (C_p(c_j));
    k = (k_1, k_2, ..., k_j);
    output = output ∪ k;
    for j := 1 to m do
        l = MinValue (C_p(c_j));
        if l ≤ MinValue(D_j) then
            k_j = l ;
        else
            k_j = l − 1 ;
    k = (k_1, k_2, ..., k_j);
    output = output ∪ k;
    for j := 1 to m do
        k_j = MaxValue (C_p(c_j));
    k = (k_1, k_2, ..., k_j);
    output = output ∪ k;
    for j := 1 to m do
        r = MaxValue (C_p(c_j));
        if r ≥ MaxValue(D_j) then
            k_j = r ;
        else
            k_j = r + 1 ;
    k = (k_1, k_2, ..., k_j);
    output = output ∪ k;
return output
```

4) Packet Generation Technique based on Boundary Values: To better generate packets to detect a fault in a firewall policy, our generator (based on boundary values) analyzes the policy under test and generates packets based on boundary values by solving local constraints (collected from the policy). The generated packets include boundary values, which are on the range boundaries (i.e., the smallest value and the largest value) of each field. Intuitively, when a fault is injected to a firewall policy, it is likely that the policy includes a faulty policy behavior on range boundaries of a field instead of other values.

The technique selects boundary values instead of random values from values satisfying rule constraints. Boundary values are the values around the smallest and largest values of a clause in a rule. For example, Figure 2 has a rule r_1 that includes two fields $F_1 \in [2, 5]$ and $F_2 \in [5, 10]$. For the smallest value 2 of F_1, boundary values are 1 and 2 that evaluate F_1 to be false and true, respectively. For the largest value 5 of F_1, boundary values are 5 and 6 that evaluate F_1 to be true and false, respectively. Similarly, we can select boundary values 4, 5, 10, and 11 for $F_2 \in [5, 10]$. Given boundary values of F_1 and F_2, we can generate four packets $(1, 4), (2, 5), (5, 10)$, and $(6, 11)$ to cover true and false branches of clauses in r_1.

More specifically, the generator generates packets based on boundary values to cover `true` and `false` branches of clauses

in a rule r_i. Algorithm 1 presents our technique to generate packets based on boundary values. Note that $C_p(c_j)$ is the jth clause constraint in a rule. In the algorithm, Lines 4-7 present that the generator generates a packet based on the smallest boundary values S to satisfy positive constraints (i.e., $C_p(c_1)$ $\wedge$ .... $\wedge$ $C_p(c_n)$ for `true` branches of all clauses) of each rule. Lines 8-15 present that the generator generates a packet based on boundary values (next to S) to satisfy negative constraints (i.e., $\neg C_p(c_1)$ $\wedge$ .... $\wedge$ $\neg C_p(c_n)$ for `false` branches of all clauses) of each rule. Lines 16-19 present that the generator generates a packet based on the largest boundary values L to satisfy positive constraints of each rule. Lines 20-27 present that the generator generates a packet based on boundary values (next to L) to satisfy negative constraints of each rule.

The generator generates packets based on boundary values within solutions for the collected constraints. This technique generates packets with high structural coverage (that can be achieved based on local constraint solving) and fault detection with using boundary values instead of any other values feasible to cover a target entity.

However, the packets generated based on boundary values of a rule's constraints could not reach the rule due to the impact of overlapping predicates in the preceding rules. To further generate packets based on correctly identified boundary values, we leverage an existing technique [11] to remove redundant overlapping predicates of firewall policies. In addition, this redundancy removal technique helps reduce the number of generated packets based on boundary values when redundant rules are removed and the number of rules is decreased.

B. Test Reduction

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.

Given a packet set, we evaluate each packet set against the policy. 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.

C. Measuring Fault-Detection Capability

Fault detection is a focus of any testing process. In this paper, we aim to investigate the relationship between firewall policy structural coverage achieved by a packet set and the packet set's fault-detection capability. We adopt mutation testing [9] to measure the fault-detection capability of the packet set.

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 packet that also detects that fault.

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 II 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, AR and RMR and (2) clause-level mutation operators including RCT, RCF, $CRSV$, $CREV$, $CRSO$, and $CREO$. The first group adds, removes, or modifies a rule in a policy. 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.

However, syntactic changes of firewall policies cannot guarantee semantic changes of the firewall policies. In other words, the mutant generator for each mutation operator may generate semantically equivalent mutants, which are mutants with the same behaviors as the original policy; an equivalent mutant cannot be killed by any test packet. In order to guarantee semantic changes of firewall policies after fault injection, we leverage an existing change-impact analysis tool [13] on firewall policies to determine whether the modifications incur any semantic changes. Given two policies p_1 (an original policy) and p_2 (its corresponding mutant), change-impact analysis is to analyze what would be different policy behaviors between p_1 and p_2.

VI. Implementation

Our implementation (written in Java) includes four components: packet generation, packet evaluation, packet reduction, and mutation generation. In the packet 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 and finds concrete solutions (i.e., numeric values), each of which is transformed to a test packet. If no solution exists, Z3 outputs *unsolvable*. For packet generation based on boundary values, our packet generator selects boundary values (that satisfy a given constraint) for each field value. In order to remove redundancy, we leverage an existing tool [13] to detect redundancy in a firewall policy.

In the packet evaluation component, 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. The engine also automatically compares the evaluated decisions

[1] http://research.microsoft.com/projects/z3/

TABLE II
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`).
AR (AR)	add a randomly generated rule in a policy.
Remove Rule (RMR)	remove the rule in a policy.

(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.

VII. EXPERIMENTS

We carried out our experiments on a laptop PC running Windows XP SP2 with 1G memory and dual 1.86GHz Intel Pentium processor. Our packet generation tool generates packet sets using the four techniques (random packet generation, packet generation based on local constraint solving, one based on global constraint solving, and one based on boundary values). We use $Rand$, $Local$, and $Global$ to denote the packet sets generated by these first three techniques, respectively. We use $Bound_1$ and $Bound_2$ to denote the packet sets generated by the fourth technique on an original policy and its redundancy-removed policy, 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 $Rand^-$, $Local^-$, $Global^-$, $Bound_1^-$, and $Bound_2^-$ 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 four 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.

A. Instrumentation

We conducted experiments on 14 real-life firewall policies collected from a variety of sources. 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 local 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. The packet-generation technique based on boundary values generated at most $n \times 4$ packets. The technique removes duplicate packets to reduce the number of packets. Moreover, the technique removes redundancy to help reduce the number of rules, which reflects the number of packets.

When generating mutants, mutation operators may generate a mutated policy that is the same (syntactically or semantically) as the original policy. As such a mutant does not include any fault, we excluded the mutant. Moreover, we remove all of RPF and RCF mutants whose policy structure is detected by the mutator to be incorrect.

B. Comparison of Structural Coverage

Table III 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 group "# Packets" shows the size of the generated packet sets $Rand$,

TABLE III
EXPERIMENTAL RESULTS ON FIREWALL POLICIES

Policy	# Rules	# Mutants	# Packets					# Reduced packets					Gen time (ms)
			Rand	Local	Global	$Bound_1$	$Bound_2$	$Rand^-$	$Local^-$	$Global^-$	$Bound_1^-$	$Bound_2^-$	Global
1 (Firewall1)	3	51	6	6	6	10	10	1	3	3	3	3	112
2 (Firewall2)	5	26	10	10	9	12	6	1	5	4	6	3	172
3 (Firewall3)	28	280	56	56	43	104	29	1	17	17	18	12	1040
4 (Firewall4)	18	206	36	36	26	64	22	1	10	9	12	9	622
5 (Firewall5)	26	378	52	52	44	96	62	2	18	19	22	21	896
6 (Firewall6)	26	338	52	52	39	96	42	2	14	14	15	14	919
7 (Firewall7)	27	360	54	54	41	100	46	3	15	15	16	15	982
8 (Firewall8)	28	494	56	56	53	101	87	3	24	27	29	28	969
9 (Firewall9)	14	185	28	28	25	48	34	2	11	11	13	12	483
10 (Firewall10)	17	179	34	34	29	56	22	1	11	13	14	10	579
11 (Firewall11)	23	233	46	46	34	77	30	3	14	11	15	11	810
12 (Firewall12)	6	44	12	12	11	16	10	1	6	5	7	5	215
13 (Firewall13)	16	152	32	32	23	54	18	1	8	7	9	8	542
14 (Firewall14)	24	313	48	48	40	58	42	2	19	17	18	14	843
Average	18.64	231.36	37.29	37.29	30.21	63.71	32.86	1.71	12.50	12.29	14.07	11.79	656.04

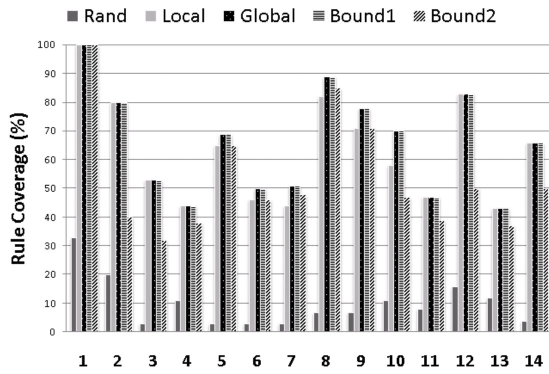


Fig. 6. Rule coverage achieved by each packet set.

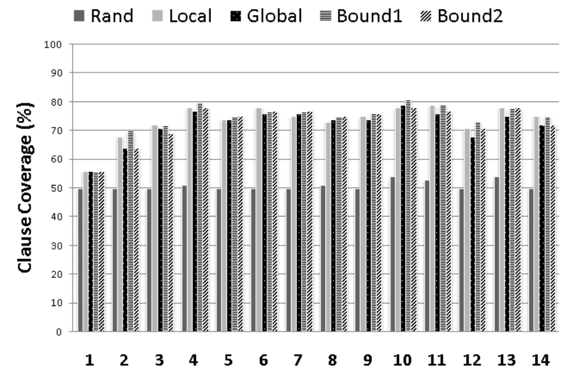


Fig. 8. Clause coverage achieved by each packet set.

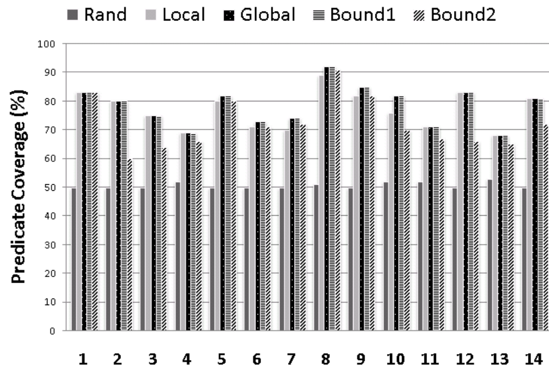


Fig. 7. Predicate coverage achieved by each packet set.

$Local$, $Global$, $Bound_1$, and $Bound_2$, respectively for each packet generation technique. Columns 9-13 show the size of their reduced packet sets (denoted by $Rand^-$, $Local^-$, $Global^-$, $Bound_1^-$, and $Bound_2^-$), respectively. Column 14 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. Note that the last row shows the average.

We observe that $Global$ 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. We observe that $Bound_2$ contains fewer packets than $Bound_1$ since the number of rules in the policy under test is reduced after redundancy removal. The analysis time for $Rand$, $Local$, $Bound_1$, and $Bound_2$ is not shown in Table III. The reason is that the time is too short to be measured in milliseconds and negligible (in comparison with that of $Global$).

Figures 6, 7, and 8 show the rule, predicate, clause coverage metrics, respectively, of each policy achieved by $Rand$, $Local$, $Global$, $Bound_1$, and $Bound_2$. 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/predicate coverage than other packet sets. This observation is consistent with our expectation described in Section V-A. On average, $Global$ is approximately 2% (1.5%) and 56% (28%) higher than $Local$ and $Rand$ in terms of rule (predicate) coverage. $Bound_1$ achieves similar rule/predicate coverage with $Global$. $Bound_2$ achieves lower coverage than $Bound_1$ because packets (generated based on a redundancy-removed policy) are not suitable to achieve high structural coverage for its original policy due to structure change after redundancy removal.

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

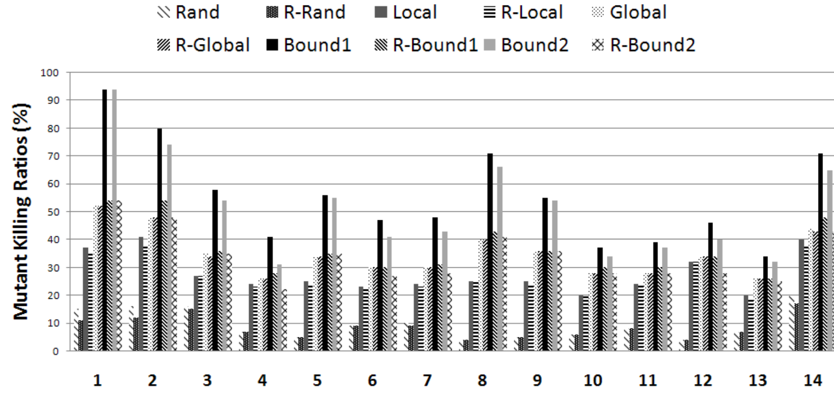


Fig. 9. Mutant-killing ratios for all operators by subjects.

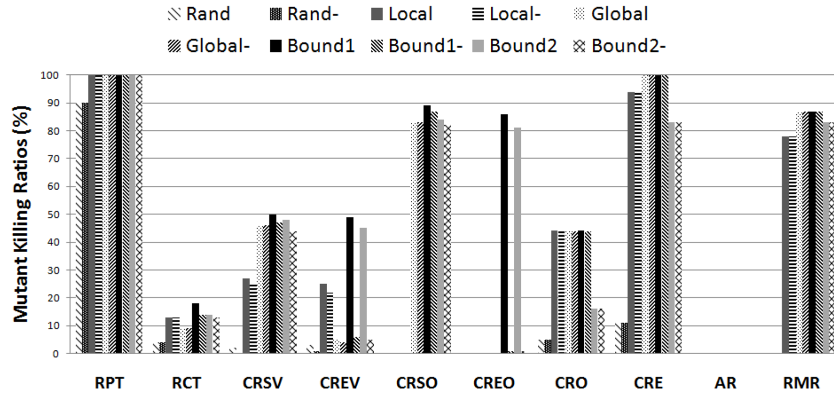


Fig. 10. Mutant-killing ratios for all subjects by operators.

covering some of uncovered clauses. Instead, $Local$, $Bound_1$, and $Bound_2$ 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 $Local$.

C. 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 V-C).

Figure 9 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. For $Rand$, $Local$, and $Global$, the largest ratio difference between the generated packet sets and their reduced packet set is less than 2%. $Rand$ and $Rand^-$ show the lowest mutant-killing ratios. As $Rand$ contains a relatively large 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 $Bound_1$ ($Bound_1^-$) achieves the highest mutant-killing ratios among the generated packet sets (the reduced packet sets). While $Local$, $Global$, and $Bound_1$ achieve similar structural coverage, $Bound_1$ achieves the highest mutant-killing ratios. This result is expected as the evaluation of these packet sets can involve more structural entities and boundary values than the other packet sets.

We also observe that, in Figure 9, for most cases, mutant-killing ratios are below 60%. The reason for such low mutant-killing ratios is that a policy can include various types of faults denoted in Table II and our test packet generation could not find all possible changed behaviors of a given policy. For a CRO mutated policy, two rules swap locations. In order to detect such a fault, packets should match intersections of two packets. However, our test packet generation does not consider such intersections for test packet generation and cannot easily detect such a fault.

We next present more details about mutants being killed. Figure 10 shows the average mutant killing ratios for all policies by operators. For rule-level mutation operators, we observe that $Global$, $Global^-$, $Bound_1$, and $Bound_1^-$ achieve highest mutant-killing ratios. The reason is that the highest rule/predicate coverage achieved by $Global$, $Global^-$, $Bound_1$, and $Bound_1^-$ helps exercise more rules and detect faults in rules.

In Figure 10, we observe that our generated packet sets cannot detect any faults in the policies with AR faults. AR simulates a forgotten rule in a given policy. The reason for such low mutant-killing ratios is that our test packet generation is based on a set of rules in a given policy and does not have any information of a forgotten rule to help detect its fault. Moreover, randomly generating a packet for fault detection

is not trivial as well due to a large domain of a firewall policy representation. For example, an IP address field in a rule includes a subset of the IP address domain (i.e., $[0, 2^8 - 1]$), which is huge. There is a very low possibility that a randomly generated IP address field value in a packet could detect such a fault. In other words, in order to detect a fault in a rule, a packet matches not only an IP address field in the rule. The packet is required to match other fields in the rule as well. A randomly generated packet may match some of fields, especially when a field is a subset of a relatively small domain (e.g., Boolean). However, matching all of the fields in the rule with a randomly generated packet is not trivial.

Among clause-level mutation operations, $Bound_1$ and $Bound_1^-$ achieves the highest mutant-killing ratios over RCT, RCF, $CREV$, and $CREO$ mutated policies. As $Bound_1$ and $Bound_1^-$ 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. $Bound_1$ ($Bound_1^-$) and $Bound_2$ ($Bound_2^-$) detect more faults in $CRSV$ and $CRSO$ mutated policies. The reason is that a packet in $Bound_1$ ($Bound_1^-$) and $Bound_2$ ($Bound_2^-$) are based boundary values in the constraint. Therefore, $Bound_1$ ($Bound_1^-$) and $Bound_2$ ($Bound_2^-$) are effective to detect faults caused by the change of the boundary value of a clause over other packet sets.

VIII. Related Work

Coverage Criteria For Logical Expressions. A firewall policy is translated to program code (i.e., `IF-THEN-ELSE` statements) that includes a large number of conjunctive logical expressions to illustrate rules. Ammann et al [10] proposed coverage criteria for such logical expressions. For example, they proposed predicate and clause coverage criteria in notions of logical expressions. Although they proposed such criteria, they did not generate test suites for real program code to show the effectiveness of their coverage criteria. We not only propose logical coverage criteria, which are suitable for a firewall policy, but also developed test packet generation and mutation testing techniques to show the effectiveness in terms of fault-detection capability. To the best of our knowledge, there is no existing work targeting at test generation and mutation testing especially for a large number of logical expressions (in a firewall policy) as our work.

Mutation Testing of Specifications. Black et al. [14] and Wimmel et al. [15] proposed mutation testing for specifications. However, their mutation operators change operators (e.g., replacing an expression by its negation) and pre/post conditions of specifications. In our work, instead of changing operators and pre/post conditions, we mutate clauses and a rule's decision, where policy authors could make mistakes in specifying rules (e.g., specifying incorrect values).

Testing of Access Control Policies. For testing access control policies such as XACML policies [16], Martin et al. [12] proposed to mutate policies [17], 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 differences between access control policies and firewall policies. While firewall policies consist of a set of ranges (intervals) in rules, access control policies consist of structural elements such as policies, rules, subjects, objects, and actions. They do not use a well-established test generation technique to cover certain entities. Our approach uses more advanced technique: the local and global constraint solving mechanisms to generate packets covering certain structural entities effectively.

Firewall Policy Test Criteria. Some researchers proposed firewall testing with test cases generated based on their proposed criteria. Jürjens et al. [18] 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. [19] proposed policy criteria identified by interactions between rules, called "policy segmentation" identified by interactions between rules. Different from their approaches, we use structural coverage criteria in each rule to help detect which entities are specified incorrectly. In addition, we also use mutation testing to evaluate our approach.

Firewall Policy Testing. Several firewall policy testing techniques [5]–[7] inject packets into a firewall and check whether the decisions of the firewall concerning the injected packets are correct. However, these techniques lack rigorousness in terms of the use of coverage criteria and effective mechanisms for generating covering packets. Furthermore, these testing techniques are inefficient when a tester needs to inject a large number of packets and examine their decisions. In contrast, our approach is based on solid foundations and advanced test-packet generation techniques.

IX. 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 four 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, a NIST grant, NSF grant CNS-0716407, and MSU IRGP Grant.

References

[1] S. W. Lodin and C. L. Schuba, "Firewalls fend off invasions from the net," *IEEE Spectrum*, vol. 35, no. 2, pp. 26–34, 1998.

[2] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, pp. 62–67, 2004.

[3] E. Al-Shaer and H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *Proc. 2004 IEEE Conf. on Communications*, pp. 2605–2616.

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

[5] M. R. Lyu and L. K. Y. Lau, "Firewall security: policies, testing and performance evaluation," in *Proc. 2000 International Conference on Computer Systems and Applications*, pp. 116–121.

[6] A. X. Liu, M. G. Gouda, H. H. Ma, and A. H. Ngu, "Non-intrusive testing of firewalls," in *Proc. 2004 International Computer Engineering Conference*, pp. 196–201.

[7] D. Hoffman and K. Yoo, "Blowtorch: a framework for firewall test automation," in *Proc. 2005 IEEE/ACM international Conference on Automated Software Engineering*, pp. 96–103.

[8] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[10] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *Proc. 2003 International Symposium on Software Reliability Engineering*, pp. 99–107.

[11] A. X. Liu and M. G. Gouda, "Complete redundancy detection in firewalls," in *Proc 2005 Annual IFIP Conference on Data and Applications Security*, pp. 196–209.

[12] E. Martin, T. Xie, and T. Yu, "Defining and measuring policy coverage in testing access control policies," in *Proc. 2006 International Conference on Information and Communications Security*, pp. 139–158.

[13] A. X. Liu, "Change-impact analysis of firewall policies," in *Proc 2007 European Symposium Research Computer Security*, pp. 155–170.

[14] P. E. Black, V. Okun, and Y. Yesha, "Mutation operators for specifications," in *Proc. 2000 IEEE International Conference on Automated Software Engineering*, pp. 81–88.

[15] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," in *Proc. 2002 International Conference on Formal Engineering Methods*, pp. 471–482.

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

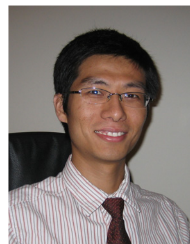
[17] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proc. 2007 International Conference on World Wide Web*, pp. 667–676.

[18] J. Jürjens and G. Wimmel, "Specification-based testing of firewalls," in *Proc. 2001 International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pp. 308–316.

[19] 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. 2007 IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 151–160.

JeeHyun Hwang received the BS degree in computer science from Korea University in Korea in 2003 and the MS degree in computer science from the State University of New York in Stony Brook in 2005. He is a Ph.D. student in the department of computer science at North Carolina State University. He is a member of the Automated Software Engineering Research Group led by Dr. Tao Xie. His research interests include testing and verifying access control policies.

Tao Xie received the Ph.D. degree from the University of Washington in 2005. He received the BS degree from Fudan University in 1997 and the MS degree from Peking University in 2000. He is an associate professor in the Department of Computer Science at North Carolina State University. His primary research interest is software engineering, with an emphasis on automated software testing and mining software engineering data. He is a member of the IEEE.



Fei Chen received the BS degree in Automation from Tsinghua University in 2005 and the MS degree in Automation from Tsinghua University in 2007. He is currently a Ph.D. student in the Department of Computer Science and Engineering at Michigan State University. His research interests include on networking, algorithms, and security.

Alex X. Liu received his Ph.D. degree in computer science from the University of Texas at Austin in 2006. He is currently an assistant professor in the Department of Computer Science and Engineering at Michigan State University. He received the IEEE & IFIP William C. Carter Award in 2004 and an NSF CAREER award in 2009. He received the MSU College of Engineering Withrow Distinguished Scholar Award in 2011. His research interests focus on networking, security, and dependable systems.