

# Automated Repair By Example for Firewalls

William T. Hallahan, Ennan Zhai, Ruzica Piskac  
Yale University

{william.hallahan, ennan.zhai, ruzica.piskac}@yale.edu

**Abstract**—Firewalls are widely deployed to manage enterprise networks. Because enterprise-scale firewalls contain hundreds or thousands of rules, ensuring the correctness of firewalls – that the rules in the firewalls meet the specifications of their administrators – is an important but challenging problem. Although existing firewall diagnosis and verification techniques can identify potentially faulty rules, they offer administrators little or no help with automatically fixing faulty rules. This paper presents FireMason, the first effort that offers automated repair by example for firewalls. Once an administrator observes undesired behavior in a firewall, she may provide input/output examples that comply with the intended behaviors. Based on the examples, FireMason automatically synthesizes new firewall rules for the existing firewall. This new firewall correctly handles packets specified by the examples, while maintaining the rest of the behaviors of the original firewall. Through a conversion of the firewalls to SMT formulas, we offer formal guarantees that the change is correct. Our evaluation results from real-world case studies show that FireMason can efficiently find repairs.

## I. INTRODUCTION

Firewalls play an important role in today’s individual and enterprise-scale networks. A typical firewall is responsible for managing all incoming and outgoing traffic between an internal network and the rest of the Internet by accepting, forwarding, or dropping packets based on a set of rules specified by its administrators. Because of the central role firewalls play in networks, small changes can propagate unintended consequences throughout the networks. This is especially true in increasingly large and complex enterprise networks.

A single line in a firewall could, for example, allow anyone to access production services, and therefore it is critical to ensure the *correctness* of firewall rules. Broadly speaking, a firewall is correct if the rules of that firewall meet the specification of its administrator. There have been many efforts that aim to check the correctness of firewall rules through techniques such as firewall analysis [23], [30], verification [22], and root-cause troubleshooting [26], [31], [34]. For instance, systems like Margrave [26] and Fang [23] build an event tree recording states of an observed error, and backtrack through it to find the root causes.

While existing tools can identify the cause of an error, administrators still have to manually find an effective repair to the firewall so that it meets the specification. We propose the first framework, called FireMason, that not only detects errors in firewall behaviors, but also automatically repairs the firewall. Specifically, a user provides a list of examples of packet routing (e.g., all packets with a certain source IP address should be dropped) to describe what the firewall should do. The current firewall might or might not route the packets as specified in the examples. Given the complexity of enterprise-scale networks,

finding such a repair requires considerable expertise on the part of the administrator. To the best of our knowledge, there is no existing effort that automates firewall repair.

The main challenge of firewall repair is to show that a generated firewall is indeed repaired and that new rules do not change the routing of packets which are not described by the given examples. We employ an SMT solver for this task. In a nutshell, FireMason translates a given firewall into a sequence of first-order logic formulas falling into the EUF+LIA logic [25], thus allowing us to use an SMT solver for reasoning about the firewalls. By using SMT solvers, FireMason provides formal guarantees that the repaired firewalls satisfy two important properties:

- Those packets described in the examples will be routed in the repaired firewall, as specified.
- All other packets will be routed by the repaired firewall exactly as they were in the original firewall.

Taken together, these two properties allow administrators confidence that the repairs had the intended effect.

Previous work has modeled firewalls using less expressive logics. For example, Zhang *et al.* [34] use SAT and QBF formulas, while Margrave [26], uses first-order relational logic (specifically, through the use of KodKod [30]). By using our formalism we are able to check some important and widely used, but previously out-of-scope, properties. In particular, the ability to reason about linear integer arithmetic with an SMT solver is invaluable in handling *rate limits*. Rate limits, which are frequently used in all modern firewalls, put a restriction on the number of packets matched in a given amount of time. Using SMT solvers we are able to efficiently reason about limiting rules. Due to the complexity of modeling limits, no previous work has considered firewalls with such rules. Such rules say, for example, that we can only accept 6 packets per minute from a certain IP address. As before, the user provides a list of examples, but with relative times. This requires reasoning about the priorities and permissions of each firewall entry, as well as the temporal patterns of the incoming packets.

Furthermore, FireMason is also a stand-alone verification tool, that can either prove that a given specification holds, or produce counterexamples.

We evaluated our tool using real-world firewall issues, and observed that FireMason is able to efficiently generate correct firewalls meeting administrators’ examples, without introducing any new problems. In addition, our evaluations show that FireMason scales well to enterprise-scale networks.

In summary, this paper makes the following contributions. We developed a formalism to model firewalls and their behavior. This formalism allows us to use SMT solvers. By using them we can easily prove formal guarantees for verification and repair,

thus making FireMason the first work capable of automatically repairing firewalls based on easily specified examples. This allows administrators to conveniently specify their desired behaviors, and automate the repair process. Additionally, by using SMT solvers we can efficiently reason about limit rules, which were not considered by any of existing tools. Finally, we built a workable system that scales well with real-world examples and larger-scale datasets.

## II. PRELIMINARIES

**Repair by Example.** In this paper, we introduce the *repair by example* paradigm, which repairs faulty code so that it satisfies the given examples. In some ways, this resembles the programming by example paradigm [14], [21]. However, in programming by examples, the output is code which generalizes the given input examples. On the other hand, in the repairing by example paradigm the input is both an existing program and a set of examples. The goal is to adjust the input program to satisfy the examples, but otherwise to have a minimal effect on the programs behavior. This allows a user to easily specify instances of faulty behavior, but have confidence that the program will continue to function as it did before. With repair by example, it is important that the effect of the changes is constrained, whereas in programming by example there is no such restriction.

**ACL-Based Firewalls.** We focus on one of the most representative types of firewalls, Access Control List-based firewalls (or ACL-based firewalls). ACL-based firewalls, such as iptables [4], Juniper [18], and Cisco firewalls [13], are widely used in practice. A typical ACL-based firewall contains an ordered list of *rules*, each of which has *criteria* and an *action*. A criterion describes which preconditions need to hold for the action to take place (e.g., dropping or accepting a packet) [28]. When a network packet is received by an ACL-based firewall, the packet is evaluated against all the rules according to the order in which they appear. After the firewall finds the first rule whose criteria is satisfied by the packet, it performs the corresponding action. The criteria in a rule may refer to properties of the packet that is currently being processed, or to information tracked by the firewall. For instance

```
iptables -A INPUT -p 16 -s 123.23.12.1 -j DROP
```

has criteria denoting packets with a protocol of 16 and a source IP address of 123.23.12.1, and an action specifying those packets should be dropped.

Actions are either *terminating* or *non-terminating*. Terminating actions end the packet’s traversal (for example, once a packet is accepted or dropped, it no longer needs to check more rules in the ACL). Non-terminating actions (such as printing to a log file) allow a packet to continue traversing the ACL rules and finding a match for more rules. An action might also refer to another ACL, which then needs to be used to evaluate the packet. We refer to this as a *jump* to a different ACL.

The ACL jumps can not form a loop. That is, if ACL  $\mathcal{A}_1$  contains a jump to ACL  $\mathcal{A}_2$ , there can be no jumps from  $\mathcal{A}_2$  back to  $\mathcal{A}_1$ . However, suppose a packet is evaluated against all rules in an ACL  $\mathcal{A}_2$  and does not match any rule with

a terminating action. The packet will then continue being evaluated at the next rule in  $\mathcal{A}_1$ . If the packet started in  $\mathcal{A}_1$ , and the packet does not match any rule in  $\mathcal{A}_1$  with a terminating action, the packet will be routed according to the *policy* of  $\mathcal{A}_1$ . The policy is the default action on packets that start in a given ACL, and must be to either accept or drop the packet [9].

**Rate Limiting Rules.** When an administrator wants to restrict the amount of packets matching a certain rule, a rate limit can be specified for that rule. We call a firewall with such rules a *rate limiting firewall*. In many firewalls, including iptables [9], Juniper [18], and Cisco [13] firewalls, a *limit* is a criterion that specifies how frequently a rule can be matched. A limit is implemented as a counter  $l$ , and allows a match of a rule only if  $l > 0$ . A limit has two parameters: an average rate of packets per some time unit,  $ra$ , and a burst limit,  $b$ . Whereas other criteria are based solely on evaluating a single packet, a limit requires the firewall to maintain its counter, and hence warrants special consideration.

Firewalls use the token bucket algorithm [29] to determine if a packet should be dropped or further processed. The counter  $l$  decrements when a packet matches the rule, and increments every  $1/ra$  time units. The counter can never exceed the burst limit  $b$ . The next example shows how limits work in practice.

*Example.* Suppose that we set a limit on incoming packets, with  $ra = 1$  packet / second and  $b = 5$  packets. The firewall is initialized with  $l = b = 5$ . If we do not exceed the limit, we will accept incoming packets. If we do exceed it, we will drop them. Suppose in the first, third, and fourth seconds after initialization, we receive 1 packet, 3 packets, and 4 packets, respectively. We receive no packets during the second second.

At the end of the first second,  $l = 5 - 1 = 4$ , since 1 packet arrives. At the beginning of the second second,  $l$  will be incremented back to 5. The counter  $l$  will not be incremented at the beginning of the third second, since if it was incremented it would exceed the burst limit. However, since 3 packets arrive during the third second, by the end of the third second  $l = 2$ . At the beginning of the fourth second,  $l$  is incremented again to 3. During the fourth second, 4 packets arrive. The first 3 will be accepted, but will result in the counter being decremented, to  $l = 3 - 3 = 0$ . Since  $l = 0$  when the fourth packet arrives, that packet can not match the limit. Therefore, it is dropped.

## III. MOTIVATING EXAMPLES

**Stateless Example.** We start to demonstrate the functionality of FireMason with a problem inspired by a StackExchange post [1], shown in Figure 1. An administrator is maintaining firewall rules written in iptables [4], one of the most representative firewall script languages. The firewall initially contained rules labeled R1 to R5.

If the administrator wants to block TCP requests coming from the IP address 172.168.14.6, she may try expressing that as a rule and putting it at the end of the current firewall, cf. rule R6 in Figure 1. Such an action is very common in enterprise-scale firewall management, because administrators prefer appending a new rule to the existing rules [20].

FireMason can be used as a standard firewall analysis tool. To test her changes, the administrator can execute the query:

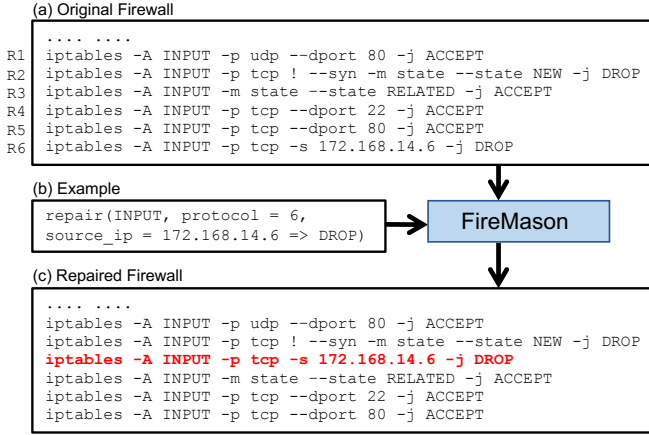


Figure 1: An example of a firewall repair problem.

```
verify(INPUT, protocol = tcp,
source_ip = 172.168.14.6 => DROP)
```

FireMason reports to the administrator that the specification is violated, and gives an example of a packet that will be incorrectly routed (For example, a TCP packet with the SYN flag set, a source ip address of 172.168.14.6, and a destination port of 22. Such a packet would be accepted by R3 or R4).

Knowing that her repair does not work as intended, the administrator can also use FireMason as a repair tool. She provides an example of what should be changed in the firewall and invokes FireMason as shown in Figure 1 (b).

FireMason returns a repaired firewall, Figure 1 (c), to the administrator. The new rule is positioned close to similar rules, namely, those rules related to the TCP protocol. This positioning is very important. While one may argue that directly appending a rule to the top of firewall can also make the firewall behave correctly (in terms of functionality), such a way would, unfortunately, destroy the structure and organization of the firewall. Much like traditional code, keeping the firewall rules organized is important to facilitate later understanding and maintaining. Most importantly though, the rule is positioned so any packet matching the example is guaranteed to be dropped. R1 specifies a protocol other than TCP, and so never matches the example. The example could match R2, but R2 drops any matching packet anyway.

**Rate Limiting Example.** We next show how an administrator can use FireMason to add/repair rate limiting rules. To the best of our knowledge no existing firewall analysis tools can address this problem. Suppose an administrator wants to allow TCP connections with the SYN flag set once every 10 seconds (a task inspired by a forum post on StackExchange [6].) To do this, the administrator may provide a sequence of example packets and relative times, in seconds:

```
repair(INPUT, SYN, time = 0 => ACCEPT;
INPUT, SYN, time = 5 => DROP;
INPUT, SYN, time = 10 => ACCEPT)
```

As a result FireMason creates and inserts two new rules:

```
iptables -A INPUT -m limit --limit 6/minute \
--limit-burst 1 -p tcp --tcp-flags SYN SYN -j ACCEPT
iptables -A INPUT -p tcp --tcp-flags SYN SYN -j DROP
```

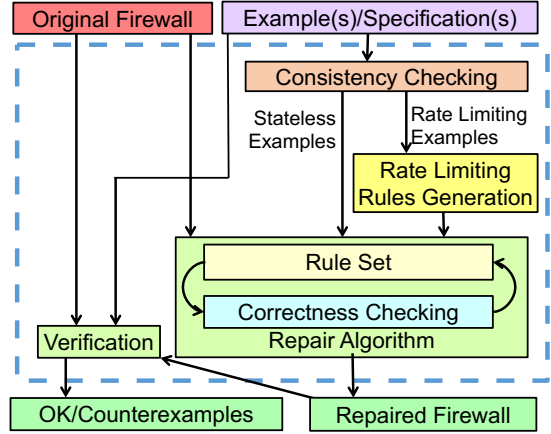


Figure 2: The workflow overview of FireMason.

This limit satisfies the administrator’s requirement. Only one TCP SYN packet can be received every 10 seconds.

#### IV. SYSTEM DESIGN

Figure 2 shows the overview of FireMason’s workflow. FireMason takes as input a firewall and a user command, which can be either a verification command or a repair command and contains a list of examples.

FireMason first translates the firewall and examples into a set of formulas belonging to a fragment of first-order logic. The translation (described in Sec. IV-A) produces two sets of EUF+LIA formulas [25], which means we can use an SMT solver to reason about firewalls.

The verification process (described in Sec. IV-B) checks if the rules specified in the examples are violated by the new firewall. If there are such rules, FireMason reports counterexamples to the user.

The repair process first checks consistency of the input examples and reports to the user if they are contradictory (Sec. IV-D). This also allows us to detect sets of examples that can be used to generate rate limiting rules. FireMason creates any needed rate limiting rules to handle provided examples. (Sec. IV-F). FireMason next runs the repair algorithm (Sec. IV-G). Finally, FireMason adds the rules to the firewall (Appendix, Sec. IV-C), checks if there are redundant rules in the newly generated firewall (Sec. IV-H), and outputs a correct firewall.

##### A. Encoding Firewalls and Examples as FOL Formulas

**Translating Examples.** FireMason starts with a list of examples provided by the user, either for a verification or a repair process. Those examples are expressed using the grammar:

$$comm := verify(\{(acl, rule)\}^+ | repair(\{(acl, rule)\}^+)$$

$$rule := precon^+ \Rightarrow action$$

$$precon := protocol = INT | source\_ip = IP\_ADDRESS \\ | destination\_port = INT | \dots | not\ precon$$

$$action := ACCEPT | DROP$$

$$acl := STRING \quad \backslash\backslash \text{ ACL Name}$$



We represent every example by a tuple  $(a, r, t)$ , where  $a$  is the name of the ACL to which the rule  $r$  applies, and  $t$  is the time given in the example. If no time was given, we set  $t = \emptyset$ . This tuple is then used in FireMason’s algorithms. For instance, the example `repair(protocol = 16, time = 5 ⇒ ACCEPT)` is translated to `(INPUT, protocol = 16 ⇒ ACCEPT, 5)`.

**Translating Firewall scripts.** Broadly speaking, FireMason describes a firewalls behavior with a sequence of first-order logic formulas. The translation results in formulas that are amenable for reasoning with a SMT solver. Such encoding has two benefits: the computational burden of checking consistencies or finding redundant rules is done by a solver. In addition, we can easily formalize that the repaired firewall is indeed repaired and that only packets described by the examples will be treated differently and according to the specification.

While the majority of the rules could be easily translated to first-order formulas, one obstacle is when the firewall contains jumps. This becomes an issue especially when the ACL also uses limits. Consider, for example, an ACL that has at least two jumps to an ACL  $A_1$ . Let us assume that the ACL  $A_1$  has some limit rules. If a packet has to go through both the jumps, then when it reaches the limit in  $A_1$  the second time, the limit in  $A_1$  will have counted the packet twice.

We introduce a data structure, called a *FirewallMap*, which simplifies modeling of jumps and limits. A FirewallMap  $\mathcal{M}$  maps unique IDs (we use natural numbers) to tuples of ACL names and lists of the ACLs rules. A rule is modeled as an implication, where a set of criteria implies an action. Possible actions are ACCEPT, DROP, and GO( $a$ ). GO is parameterized by a natural number  $a$ , and represents a jump to the ACL with ID  $a$ . In the FirewallMap  $\mathcal{M}$  there is at most one GO referring to a particular ACL ID. Every rule in  $\mathcal{M}$  is assigned a tuple  $(a, r)$ , where  $a$  is an ID of the ACL where the rule appears and  $r$  is an ID of the rule in that ACL. This way there exists a single unique path through the FirewallMap to reach any individual rule. Without this property, it would be significantly more difficult to correctly model the order in which rules must be checked. Any ACL jumped to from more than one place in the original firewall is duplicated and assigned multiple IDs. The ACL mapped to by each of these IDs is identical, except any GOs in them must also have different IDs. We refer to these duplicated ACLs as *equivalent* to each other.

### Language for Encoding Firewall Behavior into Formulas.

We now describe a first-order language that we use to model firewalls and packets. Most of these predicates take a FirewallMap  $\mathcal{M}$  as an argument. One can think of  $\mathcal{M}$  as a firewall script.

Table I lists a selection of those predicates, functions, and their meanings. FireMason uses these functions and predicates to encode the firewall. For example, if rule  $r$  in ACL  $a$  in a FirewallMap  $\mathcal{M}$  had criteria specifying that it matched a packet  $p$  with protocol 17 and destination port 8, then FireMason translates that as follows:

$$\begin{aligned} &\text{matches\_criteria}(\mathcal{M}, p, a, r) \\ &\Leftrightarrow (\text{protocol}(p) = 17 \wedge \text{destination\_port}(p) = 8) \end{aligned}$$

Table II shows some axioms describing general relationships between the predicates and functions, and encoding actual firewall behavior. All formulas in the table are implicitly universally quantified, with additional guards  $0 \leq p < \text{max\_packets}$  and  $\text{valid\_rule}(\mathcal{M}, a, r)$ . Since the sets of values for  $\mathcal{M}, p, a$ , and  $r$  are finite, these formulas (as well as the definitions of `reaches_end`, `reaches_return`, `reaches_exit`, and `matches_rule` from Table I) can be finitely instantiated. Thus, no universal quantifiers are needed, and we encode the firewalls in the decidable EUF+LIA logic [25].

**Modeling Limits.** Limits have two attributes: an average rate  $ra$  in packets per time unit, and a burst limit of  $b$  packets. Each limit also uses a counter to decide if a packet can match the rule. Intuitively, it may seem one could easily model the behavior of a limit using linear integer arithmetic. However,  $ra$  might not be an integer when the units are converted to seconds. For example, 31 packets per minute is  $.51\bar{6}$  packets per second. Therefore, we introduce a new *sub* variable, which represents the time unit used by the limit, converted to seconds. For example, a limit with an average rate of 31 packets per minute and a burst of 10 will be assigned  $ra = 31$ ,  $sub = 60$ , and  $b = 600$  in the formula. Essentially, this corresponds to multiplying the whole formula by *sub*, to reduce the problem to integers.  $ra$  is now 31 *tokens* per second, we have a maximum of 600 tokens, and we require 60 tokens to send a single packet.

To have a correct counter of the number of packets, in our model we assign to each limit from the firewall two integer IDs, a main ID  $i$  and a secondary ID  $j$ . Limits for the same rule in equivalent ACLs all have the same main ID. The secondary IDs start from 0, and they increase every time a packet could meet that limit. We define two functions, `counter_pre`( $\mathcal{M}, i, j, p$ ) and `counter_post`( $\mathcal{M}, i, j, p$ ), parameterized by the limit’s main and secondary IDs, and the packet ids. They are used to track the value of the counter at any given point in time. `counter_pre`( $\mathcal{M}, i, j, p$ ) is the value of counter  $(i, j)$  immediately before packet  $p$  reaches the rule containing that limit. `counter_post`( $\mathcal{M}, i, j, p$ ) is the value of that counter immediately after. To check if a limit will allow a packet to match, we check if `counter_pre`( $\mathcal{M}, i, j, p$ )  $\geq sub$ .

The SMT formulas related to computation of limits are given in Table III. Note that, since we multiply  $ra$  and  $\Delta t(p)$ , we must know one of their values for this formula to be in LIA. Fortunately, when reading a limit from an existing firewall script we know  $ra$ . In Sec. IV-F we explain how  $\Delta t(p)$  is known in advance from the examples, so we can obtain  $ra$  from the SMT solver.

### B. Firewall Verification

Since firewalls are not annotated with standard specifications, systems for verifying firewalls, such as Margrave [26], verify firewalls against user provided queries. When performing the verification process, FireMason also checks if the given examples violate the firewall rules.

We first explain the verification process for examples without time (limit) constraints. Given an example,  $e = (n, c \Rightarrow act, \emptyset)$ , and a firewall  $\mathcal{M}$ , we verify  $e$  against  $\mathcal{M}$  by showing that the following formula  $\mathcal{F}$  is valid:

Table I: Partial list of predicates and functions used to model firewalls.

Predicate	Meaning of the predicate
$\text{valid\_acl}(\mathcal{M}, a)$	There exists an ACL with ID $a$ in FirewallMap $\mathcal{M}$
$\text{valid\_rule}(\mathcal{M}, a, r)$	$\text{valid\_acl}(\mathcal{M}, a)$ and there exists a rule with ID $r$ in $a$
$\text{matches\_criteria}(\mathcal{M}, p, a, r)$	Packet $p$ satisfies the criteria of rule $r$ in ACL $a$ in FirewallMap $\mathcal{M}$
$\text{reaches}(\mathcal{M}, p, a, r)$	Packet $p$ reaches rule $r$ in ACL $a$ in FirewallMap $\mathcal{M}$
$\text{starting\_acl}(\mathcal{M}, a)$	Returns true if ACL $a$ is not jumped to from some other ACL
$\text{is\_go}(act)$	Returns whether the action $act$ is $\text{GO}(a)$ for some arbitrary $a$
$\text{reaches\_end}(\mathcal{M}, p, a)$	$\text{reaches}(\mathcal{M}, p, a, \text{acl\_length}(\mathcal{M}, a))$
$\text{reaches\_return}(\mathcal{M}, p, a)$	$\text{reaches}(\mathcal{M}, p, a, r) \wedge \text{rule\_action}(\mathcal{M}, a, r) == \text{RETURN}$
$\text{reaches\_exit}(\mathcal{M}, p, a)$	$\text{reaches\_end}(\mathcal{M}, p, a) \vee \text{reaches\_return}(\mathcal{M}, p, a)$
$\text{matches\_rule}(\mathcal{M}, p, a, r)$	$\text{matches\_criteria}(\mathcal{M}, p, a, r) \wedge \text{reaches}(\mathcal{M}, p, a, r)$
$\text{matches\_example}(p, e)$	Packet $p$ matches the criteria of an example $e$
$\text{protocol}(p)$	The protocol of packet $p$
$\text{acl\_length}(\mathcal{M}, a)$	Returns the number of rules in ACL $a$
$\text{max\_packets}$	Returns the maximum number of packets to be considered
$\text{terminates\_with}(\mathcal{M}, p)$	Returns if the FirewallMap $\mathcal{M}$ would ACCEPT or DROP packet $p$
$\text{rule\_action}(\mathcal{M}, a, r)$	Returns the action of rule $r$ in ACL $a$ in FirewallMap $\mathcal{M}$
$\text{insert\_rule}(\mathcal{M}, R, a, r)$	Returns FirewallMap $\mathcal{M}$ , but with rule $R$ inserted in ACL $a$ as rule $r$
$\text{equivalent}(\mathcal{M}, n)$	Returns the set of IDs in FirewallMap $\mathcal{M}$ for the ACL named $n$
$\text{go\_acl}(act)$	For $act = \text{GO}(a)$ returns $a$ , otherwise -1

Table II: Formulas to model a firewall, and packets that firewall is processing.

$a_1 \neq a_2 \wedge \text{reaches}(\mathcal{M}, p, a_1, 0) \wedge \text{starting\_acl}(\mathcal{M}, a_1) \wedge \text{starting\_acl}(\mathcal{M}, a_2) \implies \neg \text{reaches}(\mathcal{M}, p, a_2, 0)$
$\text{reaches}(\mathcal{M}, p, a, r) \wedge \neg \text{matches\_criteria}(\mathcal{M}, p, a, r) \implies \text{reaches}(\mathcal{M}, p, a, r + 1)$
$\text{reaches}(\mathcal{M}, p, a, r + 1) \implies \text{reaches}(\mathcal{M}, p, a, r)$
$\text{matches\_rule}(\mathcal{M}, p, a, r) \wedge \text{is\_go}(\text{rule\_action}(\mathcal{M}, a, r)) \equiv \text{reaches}(\mathcal{M}, p, \text{go\_acl}(\text{rule\_action}(\mathcal{M}, a, r)), 0)$
$\text{matches\_rule}(\mathcal{M}, p, a, r) \wedge \text{is\_go}(\text{rule\_action}(\mathcal{M}, a, r)) \implies \text{reaches\_exit}(\mathcal{M}, p, \text{go\_acl}(\text{rule\_action}(\mathcal{M}, a, r))) = \text{reaches}(\mathcal{M}, p, a, r + 1)$
$\text{reaches}(\mathcal{M}, p, a, r) \wedge \neg \text{is\_go}(\text{rule\_action}(\mathcal{M}, a, r)) \wedge \text{rule\_action}(\mathcal{M}, a, r) \neq \text{RETURN} \wedge \neg \text{terminating}(\mathcal{M}, a, r) \implies \text{reaches}(\mathcal{M}, p, a, r + 1)$
$\text{reaches\_return}(\mathcal{M}, p, a) \implies \neg \text{reaches}(\mathcal{M}, p, a, r + 1)$
$\text{matches\_rule}(\mathcal{M}, p, a, r) \wedge \text{terminating}(\text{rule\_action}(\mathcal{M}, p, a, r)) \implies \neg \text{reaches}(\mathcal{M}, p, a, r + 1)$
$\text{matches\_rule}(\mathcal{M}, p, a, r) \wedge \text{terminating}(\text{rule\_action}(\mathcal{M}, p, a, r)) \implies \text{terminates\_with}(\mathcal{M}, p) = \text{rule\_action}(\mathcal{M}, p, a, r)$
$\text{reaches\_end}(\mathcal{M}, p, a, r) \wedge \text{starting\_acl}(\mathcal{M}, a) \implies \text{terminates\_with}(\mathcal{M}, p) = \text{policy}(\mathcal{M}, a)$

Table III: Logical formulas related to limits, all variables are implicitly universally quantified with additional constraints that rule  $r$  in ACL  $a$  has a limit with main ID  $i$  and secondary ID  $j$ , and  $0 \leq p < \text{max\_packets}$ . We use  $\text{j\_max}(i)$  to denote the maximum secondary ID for the limit with main ID  $i$ .

$\forall p. p \geq 1 \implies \text{arrival\_time}(p) \geq \text{arrival\_time}(p - 1)$
$\Delta t(p) = \begin{cases} \text{arrival\_time}(p) - \text{arrival\_time}(p - 1) & \text{if } 1 \leq p < \text{max\_packets} \\ 0 & \text{otherwise} \end{cases}$
$\text{counter\_pre}(\mathcal{M}, i, j, p) = \begin{cases} \text{counter\_post}(\mathcal{M}, i, j - 1, p) & \text{if } j \geq 1 \\ \min(\text{counter\_post}(\mathcal{M}, i, \text{j\_max}(i), p - 1) + ra * \Delta t(p), b) & \text{if } p \geq 1 \text{ and } j = 0 \\ b & \text{otherwise} \end{cases}$
$\text{counter\_post}(\mathcal{M}, i, j, p) = \begin{cases} \text{counter\_pre}(\mathcal{M}, i, j, p) - \text{sub} & \text{if } \text{counter\_pre}(i, j, p) \geq \text{sub} \wedge \text{matches\_rule}(\mathcal{M}, p, a, r) \\ \text{counter\_pre}(\mathcal{M}, i, j, p) & \text{otherwise} \end{cases}$

$$\forall p, a. a \in \text{equivalent}(\mathcal{M}, n) \wedge \text{reaches}(\mathcal{M}, p, a, 0) \\ \wedge \text{matches\_example}(p, e) \implies \text{terminates\_with}(\mathcal{M}, p) = act$$

Formula  $\mathcal{F}$  states that every packet arriving to ACL  $n$  and satisfying criteria  $c$  terminates with action  $a$ . Note that when negated, the formula is only existentially quantified.

To verify a list of examples with times,  $e_k = (n_k, c_k \implies act_k, t_k)$ , for  $0 \leq k \leq N$  we apply a similar procedure. After setting up all packets with appropriate times, the verification condition states that at least one packet does not terminate as desired (expressed already in the negated form):

$$\forall k \exists a. 0 \leq k \leq N \wedge a \in \text{equivalent}(\mathcal{M}, n_k) \\ \wedge \text{reaches}(\mathcal{M}, p_k, a_k, 0) \wedge \text{matches\_example}(p_k, e_k) \\ \wedge \left( \bigvee_{0 \leq j \leq N} \text{terminates\_with}(\mathcal{M}, p_j) \neq act_j \right)$$

### C. Adding Rules

Here we outline how to create rules from the provided examples. We describe in more detail how to create rate limiting rules in Sec. IV-F. We first focus on stateless rules. We use the repair algorithm, Algorithm 2 in Sec. IV-G, to assign each rule a position. After positions are assigned, it is straightforward to add the rules to the firewall. We simply copy the old rules from the original firewall, convert the new rules from our internal language to the iptables language, and insert them at the appropriate positions. For example, the tuple (INPUT, protocol = 6, source\_ip = 1.2.3.4 => ACCEPT,  $\emptyset$ ) would become iptables -A INPUT -p 6 -s 1.2.3.4 -j ACCEPT.

## D. Consistency Checking

The purpose of consistency checking is both to let the administrator know whether the provided examples contradict each other, and to detect when to invoke the algorithm for addressing limits. Consider the two examples below:

```
repair(INPUT, protocol = 17 => ACCEPT),
repair(INPUT, source_ip = 1.1.0.0/16 => DROP)
```

If a packet with `protocol = 17` and a source IP address of `1.1.1.1` enters the INPUT ACL, it is not clear whether such a packet should be accepted or dropped. We consider these examples *rule inconsistent*.

Formally, we say two examples,  $(n_1, c_1 \Rightarrow act_1, t_1)$  and  $(n_2, c_2 \Rightarrow act_2, t_2)$  are rule inconsistent if  $n_1 = n_2$ ,  $c_1 \wedge c_2$  is satisfiable by a single packet, and  $act_1 \neq act_2$ . We find the contradictory examples by using an SMT solver and we inform the administrator about ambiguities. Note that this definition makes no reference to time, and handling of rule inconsistent examples with different times will be covered in Sec IV-F.

## E. Formal Guarantees for Repaired Firewalls

FireMason offers two guarantees on the behavior of repaired firewalls. The first guarantee is the packets or sequences of packets described by the examples are correctly routed in the repaired firewall. The second guarantee is that the changes have a minimal effect; that is, that the routing of every packet not described by the examples is the same as it was in the original firewall. Together, these guarantees allow an administrator to be confident that the repairs had the intended effect, and only the intended effect.

Here we give formulas which can be used by an SMT solver to check if the formal guarantees hold.

For given examples of the form  $e_k = (n_k, crit_k \Rightarrow act_k, \emptyset)$ , for  $0 \leq k < N$ , the first guarantee can be written with Formula (1),

$$\begin{aligned} \forall k, a. 0 \leq k < N \wedge a \in \text{equivalent}(\mathcal{M}, n_k) \wedge \\ \text{matches\_example}(k, e_k) \wedge \text{reaches}(\mathcal{M}', k, a, 0) \\ \implies \text{terminates\_with}(\mathcal{M}', k) = act_k \end{aligned} \quad (1)$$

Now suppose we have examples with relative times,  $e_k = (n_k, crit_k \Rightarrow act_k, t_k)$ . Without loss of generality, assume that for  $k_1 < k_2$ , we have  $t_{k_1} < t_{k_2}$ . In this case we ensure that packets arriving at the appropriate times, with the appropriate criteria, are correctly routed, given that no other packets matching the examples criteria are processed before their arrival. Formally, we write:

$$\begin{aligned} \forall k, a. 0 \leq k < N \wedge a \in \text{equivalent}(\mathcal{M}, n_k) \\ \bigwedge_{0 \leq m \leq k} \left( \text{arrival\_time}(m) = t_m \wedge \text{matches\_example}(m, e_m) \right) \\ \wedge \text{reaches}(\mathcal{M}, m, a, 0) \bigwedge_{m' > k} \text{nonexample}(\mathcal{M}, m', k) \\ \implies \text{terminates\_with}(\mathcal{M}, k) = act_k \end{aligned} \quad (2)$$

where we use the predicate `nonexample` to determine if the packet  $p$  either does not correspond to or arrives after the last relevant example.

$$\begin{aligned} \text{nonexample}(\mathcal{M}, p, e) = \\ \forall k, a. 0 \leq k < e \wedge a \in \text{equivalent}(\mathcal{M}, n_k) \implies \\ t_e < \text{arrival\_time}(p) \vee \neg \text{reaches}(\mathcal{M}, p, a, 0) \\ \vee \left( \bigwedge_{0 \leq m \leq k} \neg \text{matches\_example}(p, e_m) \right) \end{aligned}$$

The second guarantee, that the changes we make are minimal, is stated as Formula (3):

$$\begin{aligned} \forall p. \text{terminates\_with}(\mathcal{M}, p) = \text{terminates\_with}(\mathcal{M}', p) \quad (3) \\ \vee \left( \exists k, a. 0 \leq k < N \wedge a \in \text{equivalent}(\mathcal{M}, n_k) \right. \\ \left. \wedge \text{matches\_example}(p, e_k) \wedge \text{reaches}(\mathcal{M}, k, a, 0) \right) \end{aligned}$$

## F. Rate Limiting Rules Generation

---

### Algorithm 1: Limit Generating Algorithm

---

**input** :  $E$ , the list of examples, all with relative times, optional parameters *minRulesAndLimits* and *minTotalSub* (both default to  $\emptyset$ )

**output** :  $r$  a list of rules

- 1  $E' \leftarrow []$ ;
- 2 **foreach**  $(n, r, t) \in E$  **do**
- 3      $r_2 \leftarrow r$ , with a limit template, consisting of symbolic values for *ra*, *b*, *sub*, and *useLimit*, and a Boolean *enableRule* added to the criteria
- 4      $E'.\text{append}((n, r_2, t))$ ;
- 5 **sortByNameByTime**( $E'$ );
- 6 **if** *minRulesAndLimits*  $\neq \emptyset$  and *minTotalSub*  $\neq \emptyset$  **then**
- 7     Assert *rulesAndLimits*  $<$  *minRulesAndLimits*  
        $\vee (\text{rulesAndLimits} = \text{minRulesAndLimits} \wedge \text{totalSub} < \text{minTotalSub})$
- 8 Convert  $E'$  to SMT formulas, create formulas defining score and totalSub, run SMT Solver;
- 9  $\text{sat} \leftarrow \text{getSat}$ ;
- 10 **if**  $\text{sat} = \text{UNSAT}$  **then**
- 11      $r \leftarrow \text{getRulesFromModel}(\text{model})$ ;
- 12     return  $r$ ;
- 13 **else**
- 14      $\text{model} \leftarrow \text{getModel}$ ;
- 15      $(\text{rulesAndLimits}, \text{totalSub}) \leftarrow \text{getScore}(\text{model})$ ;
- 16     call this recursively, to lexicographically minimize  $(\text{rulesAndLimits}, \text{totalSub})$ ;

---

After the consistency checking, some examples may have to be resolved via rate limiting. Specifically, this is required for rules that are rule inconsistent, but have relative times. Algorithm 1 generates rate limiting rules satisfying these examples. Our algorithm takes a list of rule inconsistent examples,  $E$ , each with a time. It returns an ordered list of

satisfying rules, which are later inserted into the firewall using Algorithm 2.

Recall that we may express an example as consisting of an ACL name, a rule, and a time. We create  $E'$  from  $E$ , by adding two criteria to each examples rule. The first is a limit template, which uses variables in place of actual integers for  $ra$ ,  $b$ , and  $sub$ . It also has a Boolean variable  $useLimit$ , which enables and disables the limit. The second criterion is a Boolean,  $enableRule$ . Packets can match the rule if and only if  $enableRule$  is true. We will use this template with an SMT solver to search for the solution that requires the fewest limits and rules.

We sort  $E'$  into distinct groups according to which ACL the rules are meant to be added to, and then sort each group by ascending time, at line 5. We extract the rules from  $E'$  into lists (ACLs) to form a templated FirewallMap  $\mathcal{M}$ . This allows us to convert to an SMT formula, using exactly the same formulas and logic as in Sec. IV-A.

For each original example,  $e_p = (n_p, c_p \Rightarrow act_p, t_p)$ , we pick  $a \in \text{equivalent}(\mathcal{M}, n_p)$  and assert that the packet with ID  $p$  matches the requirements of that example:

$$\begin{aligned} & \text{arrival\_time}(p) \wedge \text{matches\_example}(p, e_p) \\ & \wedge \text{reaches}(\mathcal{M}, p, a, 0) \wedge \text{terminates\_with}(\mathcal{M}, p) = act_p \end{aligned} \quad (4)$$

For all the pairs  $0 \leq r, q < \text{length}(E'), r \neq q$ , we check if  $c_r \wedge \neg c_q$  is satisfiable by a single packet. For each pair which is, we assert:

$$\neg \text{matches\_example}(r, e_q) \quad (5)$$

The SMT solver can then find values for each  $ra$ ,  $b$ ,  $sub$ ,  $u$ , and  $enableRule$  that guide the packets as required by the examples. Formula (4) ensures that the found solution satisfies the requirements of the examples sequence. Formula (5) ensures that the SMT solver does not make assumptions about packets criteria that the user likely does not intend. For example, if the administrator provided the examples:

```
repair(
  INPUT, protocol = 17, time = 0 => ACCEPT;
  INPUT, protocol = 17, time = 5 => DROP;
  INPUT, source_ip = 1.1.0.0/16, time = 10 => ACCEPT;
  INPUT, source_ip = 1.1.0.0/16, time = 15 => DROP)
```

Formula (5) would prevent the SMT solver finding a solution that required any of the packets satisfying `protocol = 17 AND source_ip = 1.1.0.0/16`.

Such a model is always possible to find. One valid solution is to set all the  $enableRule$  to true, all the bursts to 1, and all the rates and subs such that the limit recharging even once takes longer than the total time between the first and last packet arriving. Then, each packet will be sorted according to the rule that came from its modified example.

To make our solution capable of handling more general cases, we assign a lexicographic score to our formula. The first value is calculated by adding the number of limits and the number of non-ignored rules, which we call  $rulesAndLimits$ . The second value is the sum of the limit's  $sub$  values, which we call  $totalSub$ . We aim to make this score as small as possible. This can be done by repeatedly asserting there exists a formula

---

### Algorithm 2: Rule Adding Repair Algorithm

---

**input** :  $E$ , the list of examples;  $\mathcal{M}$ , a FirewallMap  
**output** : a FirewallMap with a rule for each  $e \in E$  added

- 1 **foreach**  $(n, newR, t) \in E$  **do**
- 2      $a' \leftarrow$  ACL id of an arbitrary representation of the ACL  $n$  in  $\mathcal{M}$ ;
- 3      $res \leftarrow SAT$  ;
- 4      $maxR \leftarrow \text{acl\_length}(a') - 1$ ;
- 5     **while**  $res = SAT$  **do**
- 6         Pick  $r' \leq maxR$ , using a similarity measure to  $newR$ ;
- 7          $\mathcal{M}' \leftarrow \text{insertRule}(\mathcal{M}, newR, a', r')$  ;
- 8          $res \leftarrow \text{SMTCheckCorrectness}(\mathcal{M}, \mathcal{M}', e)$ ;
- 9         **if**  $res = SAT$  **then**
- 10              $maxR \leftarrow r' - 1$ ;
- 11      $\mathcal{M} \leftarrow \mathcal{M}'$

---

with a better score. If  $(minRulesAndLimits, minTotalSub)$  is the current best score, we assert:

$$\begin{aligned} & rulesAndLimits < minRulesAndLimits \vee (rulesAndLimits = \\ & minRulesAndLimits \wedge totalSub < minTotalSub) \end{aligned}$$

When the SMT solver returns UNSAT, we can guarantee we found the solution which minimizes the number of rules plus the number of limits used.

There are two small potential problems with this approach, and luckily, both have straightforward solutions. First, recall from Sec. IV-A that the model involves the value of  $ra * \Delta t(p)$ , but to stay in the theory of LIA, we must avoid multiplying two variables. In that section, there was an assumption that the value of  $ra$  was known, whereas here it clearly is not. Fortunately, while we do not know the value of  $ra$ , we can precompute, and fix as a constant, the time difference between neighboring packets,  $\Delta t(p)$ .

Second, some firewalls languages constrain the value of  $sub$  to a fixed list of possible values  $s_1, \dots, s_v$ . This can be handled through one additional assertion per  $sub$  value,  $\bigvee_{u=1}^v sub = s_u$ . This occasionally leads to cases where there is no valid way to generate the limits, but such cases can be detected when the first call to the SMT solver is UNSAT.

### G. Repair Algorithms

Given the formulas representing the target firewall and examples, we need to run a repair algorithm to generate a correct firewall based on the examples. We will first consider rule insertion for non-rule inconsistent examples. Then, we will explain how this same algorithm can be used to insert the rate limiting rules found by Algorithm 1. Suppose we have  $N$  non-rule inconsistent examples,  $e_1 = (n_1, r_1 = (c_1 \Rightarrow act_1), t_1), \dots, e_N = (n_N, r_N, t_N)$ . Given a firewall represented by a FirewallMap  $\mathcal{M}$ , our goal is to find a new FirewallMap  $\mathcal{M}'$  which ensures all the examples are satisfied, but that guarantees all non-described packets maintain the same behavior. We also want  $\mathcal{M}'$  to be well organized, meaning that



“similar rules” all appear together. Our procedure (omitted due to space restrictions) to decide the similarity assigns a score based on the number and kinds of criteria used in the rules, but could be replaced by any desired scoring algorithm.

Consider the  $k^{\text{th}}$  example,  $1 \leq k \leq N$ . We express the desired condition with respect to example  $e_k$  by instantiating  $k$  in Formulas 1 and 3. We then show that Algorithm 2 outputs a firewall which satisfies this condition. For each example  $e_i = (n_i, r_i, t_i)$ , we take some  $a' \in \text{equivalent}(\mathcal{M}, n_i)$  and find the ID  $r'$  of the existing rule most similar to  $r_i$  in ACL  $a'$ . Next we set  $\mathcal{M}' = \mathcal{M}$ , and run `insert_rule`( $\mathcal{M}', r_i, a', r'$ ) to insert  $r_i$  in all ACLs equivalent to  $a'$  at position  $r'$  in  $\mathcal{M}'$ .

We convert both  $\mathcal{M}$  and  $\mathcal{M}'$  to SMT formulas, and use an SMT solver to check that Formulas 1 and 3 are valid. To do this, we must eliminate the two universal quantifiers that remain after instantiating  $k$ . There are only a finite number of values that  $a$  may attain - namely, it can only be the values in `equivalent_to_name`( $\mathcal{M}, a'$ ). Using this observation, we can easily eliminate the universal quantifier using finite instantiation. Once the formula is only universally quantified by  $p$ , we negate it, and try to show that its negation is unsatisfiable.

If the SMT solver does find the formula to be unsatisfiable, we know that the original formula was valid, i.e. the firewall satisfies the considered example. However, if the formula is satisfiable, we search for a different place to insert the rule, that comes before rule  $r'$  in ACL  $a'$ . We do not consider any rule after this rule, as any route along which  $\mathcal{M}$  and  $\mathcal{M}'$  could incorrectly diverge would also exist if the new rule was inserted after  $a'$ . Also note that the condition is guaranteed to hold if the new rule is inserted as rule 0 in ACL  $a'$ ; and although this placement is often not ideal for the structure of the firewall, it does guarantee termination.

When rules are from consistent examples, we can insert them in any order. By definition, two consistent examples cannot describe any of the same packets, so it does not matter which corresponding rule comes before the other in the firewall. However, the rules found by Algorithm 1 are rule inconsistent. In this case, insertion of the rules must be done in reverse order of the corresponding example’s times. This ensures that the inconsistent rules have the same relative order in  $E'$  (from Sec. IV-F) as in  $\mathcal{M}'$ , and thus we can expect the same behavior from the examples in both  $E'$  and  $\mathcal{M}'$ .

#### H. Redundant Rule Detection

The final step in repairing the firewall is removing *redundant* rules – that is, rules which cannot be matched by any packet. Thanks to the SMT model, this is straightforward.

As before, the firewall is converted to an SMT formula. Then, for each ACL name and rule ID,  $n$  and  $r$ , respectively, check that there exists a packet that matches the rule, or some equivalent rule by asserting

$$\exists a'. a' \in \text{equivalent}(\mathcal{M}, n) \wedge \text{matches\_rule}(\mathcal{M}, p, a', r)$$

If this call returns SAT, then clearly there exists some packet that matches the rule, and the rule is therefore not redundant. If it returns UNSAT, then there was no packet that matched the rule, and it is therefore redundant. In this case, it can be

commented out. This does involve a large number of calls to the SMT solver, but these calls tend to be fast.

## V. IMPLEMENTATION AND EVALUATION

FireMason is developed in Haskell and fully implements the design described in Sec. IV. The default firewall language that we support is the iptables language [4], but the framework can be easily extended to other firewall languages, such as Juniper [18] and Cisco firewalls [13]. The syntax of these languages varies, but the semantics are largely the same. Therefore, only the translation step (essentially a parser) needs to be rewritten for a particular language, which means that FireMason can easily be adapted to repair firewalls written in other languages. As an SMT solver we used Microsoft’s Z3 [24]. The source code for our implementation is available at <https://github.com/BillHallahan/FireMason>.

The evaluation was conducted with an Intel Xeon Quad Core HT 3.7 GHz.

**Scalability evaluation.** We evaluated the scalability of FireMason with regard to real-world network sizes by using three examples as specification, and varying the number of rules in the target firewall between 100 and 500. These firewalls were randomly generated. As shown in Figure 3, FireMason scales well to large-scale firewalls.

One might expect the rate limiting rules insertion to be slower than the non rate limiting rules insertion, due to the additional runtime of Algorithm 1. However, Algorithm 1’s runtime depends only on the number of examples, and not on the number of rules in the original firewall, its runtime is constant across the rate limiting tests. In the rate limiting case our three examples result in only two rules to insert, whereas in the non rate limiting case, we insert three rules. Thus, the additional runtime is due to Algorithm 2.

We also evaluated the performance of FireMason for different numbers of provided examples, as shown in Table V. In the stateless case this scales linearly. In the rate limiting case, the time required increases rather sharply as the number of examples generating a single limit increases. However, this is not a major concern, as we have found that a small number of examples is typically sufficient to find an appropriate limit.

**Case study: Repairing real-world firewalls.** To demonstrate that FireMason can repair real-world firewalls, we found firewall repair problems on Server Fault [7] and Stack Overflow [8]. We recreated each scenario, and generated corrected firewalls using FireMason.

Table IV presents five such problems. We list the examples which an administrator may provide to clarify how the firewall should be repaired and present the resulting repairs to the firewall. We also include the running time, the number of calls to the SMT solver, and the number of rules in the original iptables script.

We manually checked the correctness of each result and compared them to the repairs suggested on the forums. We found that the output returned by FireMason not only fixed the problems, but also avoided any side effects. Furthermore, we manually confirmed the “minimality” of the repairs, in terms of the impact on the firewalls overall behavior. In some cases,



Table IV: Case study: Sampled firewall repair problems and our solutions.

<b>Case Study 1 [1]</b>	An administrator appended a rule <code>iptables -A INPUT -s 73.143.129.38 -j DROP</code> , but can still receive packets from 73.143.129.38.
<b>Input example</b>	<code>repair(INPUT, source_ip = 73.143.129.38 =&gt; DROP)</code>
<b>Results</b>	1. Remove the appended rule, and insert a new rule <code>iptables -A INPUT -s 73.143.129.38/32 -j DROP</code> in front of an original rule <code>iptables -A INPUT -i lo -j ACCEPT</code> .
<b>Original Rule Count</b>	11
<b>Repair Time</b>	.109 s
<b>SMT Solver calls</b>	26
<b>Case Study 2 [2]</b>	An administrator wants to allow SSH access from the IP address 71.82.93.101, but does not know how.
<b>Input examples</b>	1. <code>repair(INPUT, protocol = 22, source_ip = 71.82.93.101 =&gt; ACCEPT)</code> , 2. <code>repair(INPUT, protocol = 22, not_source_ip = 71.82.93.101 =&gt; DROP)</code>
<b>Results</b>	Insert new rules <code>iptables -I INPUT 0 -p 22 -s 71.82.93.101/32 -j ACCEPT</code> and <code>iptables -I INPUT 0 -p 22 ! -s 71.82.93.101/32 -j DROP</code> in front of an original rule <code>iptables -I INPUT -p icmp --icmp-type time-exceeded -j ACCEPT</code> .
<b>Original Rule Count</b>	11
<b>Repair Time</b>	.088 s
<b>SMT Solver calls</b>	23
<b>Case Study 3 [3]</b>	An administrator is trying to limit the number of inbound SSH packets, but it just seems to lock her out.
<b>Input examples</b>	1. <code>repair(INPUT, protocol = 22, time = 0 =&gt; ACCEPT)</code> , 2. <code>repair(INPUT, protocol = 22, time = 20 =&gt; ACCEPT)</code> , 3. <code>repair(INPUT, protocol = 22, time = 30 =&gt; ACCEPT)</code> , 4. ... (In total, this repair uses 8 examples, we cannot list all the examples due to limited space)
<b>Results</b>	Insert new rules <code>iptables -A INPUT -m limit --limit 2/minute --limit-burst 4 -p 22 -j ACCEPT</code> and <code>iptables -A INPUT -p 22 -j DROP</code> at the beginning of the original firewall.
<b>Original Rule Count</b>	9
<b>Repair Time</b>	21.10 s
<b>SMT Solver calls</b>	44
<b>Case Study 4 [6]</b>	A server is attacked by TCP SYN flooding, so the administrator wants a limit on SYN packets per second.
<b>Input examples</b>	1. <code>repair(INPUT : source_ip = 192.132.209.0/24, SYN, time = 10 =&gt; ACCEPT)</code> , 2. <code>repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 11 =&gt; ACCEPT)</code> , 3. <code>repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 12 =&gt; ACCEPT)</code> , 4. <code>repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 13 =&gt; DROP)</code> , 5. <code>repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 19 =&gt; DROP)</code> , 6. <code>repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 21 =&gt; ACCEPT)</code>
<b>Results</b>	Append two new rules, <code>iptables -I INPUT 0 -s 192.132.209.0/24 -p 6 --tcp-flags SYN -j DROP</code> and <code>iptables -I INPUT 0 -m limit --limit 6/minute --limit-burst 3 -s 192.132.209.0/24 -p 6 --tcp-flags SYN SYN -j ACCEPT</code> , to the original firewall.
<b>Original Rule Count</b>	11
<b>Repair Time</b>	6.046 s
<b>SMT Solver calls</b>	42
<b>Case Study 5 [5]</b>	An administrator has the IP address 192.168.1.99, and wants to SSH to the IP address 192.168.1.15. She appended a rule <code>iptables -A INPUT -p tcp -i eth0 --dport 22 -m state --state NEW,ESTABLISHED -j ACCEPT</code> but still cannot SSH 192.168.1.15.
<b>Input example</b>	1. <code>repair(OUTPUT, protocol = , destination_ip = 192.168.1.15 =&gt; ACCEPT)</code> , 2. <code>repair(INPUT, source_ip = 192.168.1.15 =&gt; ACCEPT)</code>
<b>Results</b>	Insert two new rules <code>iptables -A INPUT -s 192.168.1.15/32 -j ACCEPT</code> and <code>iptables -A OUTPUT -d 192.168.1.15/32</code> in front of the fourth and fifth rules in the original firewall, respectively.
<b>Original Rule Count</b>	4
<b>Repair Time</b>	.054 s
<b>SMT Solver calls</b>	14

Table V: Scalability for number of examples (when inserting into a firewall with 100 rules).

Number of examples	Stateless Time (s)	Rate Limiting Time
3	3.567	2.177
6	4.545	2.004
9	5.804	36.37

FireMason outputs a different solution from the posted solution. After manual comparison, we found that both solutions work correctly, but FireMason’s output required adding fewer new rules.

Interestingly, the case studies involving rate limits took significantly longer than those only involving stateless examples. This is not at odds with the results of the scalability evaluation. As shown in table V, for a small number of examples, rate limit rule generation is generally faster, whereas for a larger number of examples, stateless rule generation is faster.

## VI. RELATED WORK

This section presents existing efforts on firewall analysis, verification and generation, and discusses why these efforts are

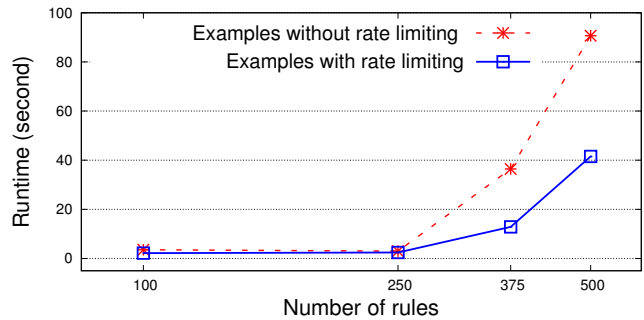


Figure 3: Scalability for number of rules.

not helpful to our target.

**Firewall synthesis.** Zhang *et al.* [34] proposed a symbolic firewall synthesis approach such that the synthesized firewall has the same behavior as a given firewall, but with the smallest possible number of rules. As this approach focuses on automatically simplifying redundant rules, rather than repairing an observed error, it is not applicable to our goal.

As software defined networks (SDN) have become increas-

ingly popular, automatic programming approaches for SDN have been proposed [27], [33]. Yuan *et al.* [33] proposed an automatic SDN policy generation approach, named NetEgg, based on a scenario-based programming technique. NetEgg can only generate a new policy, it can not account for the effect of a new policy on existing policies in the network. Furthermore, NetEgg can not synthesize rate limiting rules.

**Firewall analysis and verification.** Mayer *et al.* [23] developed the first systematic firewall analysis engine, Fang, to analyze diverse properties of firewalls. Fang and its sequel Lumeta [31] allow checking the correctness of firewall configurations by sending their analysis engines queries. Other efforts [10], [16] propose packet-filter based schemes to detect conflicting or violated rules. Frantzen *et al.* [17] and Kamara *et al.* [19] proposed different data-flow based approaches to analyze vulnerability risks in firewalls. Wool [32] conducted a case study on understanding and classifying the configuration errors of firewalls.

The Margrave firewall verification tool [26] encodes firewall rules and queries into first-order logic. It uses KodKod [30] to search for finite state models. Compared with another firewall verification tool, NoD [22], Margrave cannot produce all differences between policies in a compact way, and does not scale for large firewall rule sets.

**Firewall testing.** El-Atawy *et al.* [15] proposed targeting test packets for better fault coverage. Al-Shaer *et al.* [11] developed a system-wide framework to generate targeted packets and obtain good coverage during firewall testing. Brucker *et al.* [12] proposed a formal firewall conformance testing approach, which uses Isabelle/HOL to generate test-cases from constraint satisfaction problems.

## VII. CONCLUSION

In this paper, we have presented FireMason, an automatic tool for formally verifying and repairing firewalls. To this end, we use a first-order intermediary language to model firewalls, which allows us use of an SMT solver to obtain formal guarantees on the correctness of verification and repair. We showed that FireMason not only generates correctly repairs real-world firewall scripts, but also is able to scale to large-scale firewalls. We hope that this work will inspire more reasoning about firewalls in the formal methods community.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. We also thank Avi Silberschatz and Arvind Swaminathan for their valuable feedback on the earlier version of this work. This research was supported in part by CCF-1553168.

## REFERENCES

- [1] Cannot Block IP Address, <http://stackoverflow.com/questions/16142446/why-cant-i-block-an-ip-address-with-iptables-on-debian-6>
- [2] Good iptables starting rules for a webserver, <http://serverfault.com/questions/118669/good-iptables-starting-rules-for-a-webserver>
- [3] How can I rate limit SSH connections with iptables, <http://serverfault.com/questions/298954/how-can-i-rate-limit-ssh-connections%20-with-iptables>
- [4] iptables, <http://ipset.netfilter.org/iptables.man.html>

- [5] iptables issue cannot SSH remote machines, <http://askubuntu.com/questions/476626/iptables-issue-cant-ssh-remote-machines>
- [6] Is there a rule for iptables to limit the amount of SYN packets, <http://askubuntu.com/questions/240360/is-there-a-rule-for-iptables-to-limit-%20the-amount-of-syn-packets-a-24-range-of-i>
- [7] Server Fault, <http://serverfault.com/>
- [8] Stack Overflow, <http://stackoverflow.com/>
- [9] iptables Linux User's Manual (June 2015)
- [10] Adishesu, H., Suri, S., Parulkar, G.M.: Detecting and resolving packet filter conflicts. In: 19th IEEE International Conference on Computer Communications (INFOCOM) (Mar 2000)
- [11] Al-Shaer, E., El-Atawy, A., Samak, T.: Automated pseudo-live testing of firewall configuration enforcement. *IEEE Journal on Selected Areas in Communications* 27(3), 302–314 (2009)
- [12] Brucker, A.D., Brügger, L., Wolff, B.: Hol-TestGen/fw - An environment for specification-based firewall conformance testing. In: 10th Theoretical Aspects of Computing (ICTAC) (Sep 2013)
- [13] Cisco: Policing and Shaping Overview, [http://www.cisco.com/c/en/us/td/docs/ios/12\\_2/qos/configuration/guide/fqos\\_c/qcfcpolsh.html](http://www.cisco.com/c/en/us/td/docs/ios/12_2/qos/configuration/guide/fqos_c/qcfcpolsh.html)
- [14] Cypher, A., Halbert, D.: Watch what I Do: Programming by Demonstration. MIT Press (1993)
- [15] El-Atawy, A., Ibrahim, K., Hamed, H.H.: Policy segmentation for intelligent firewall testing. In: IEEE Workshop on Secure Network Protocols (NPsec) (Nov 2005)
- [16] Eppstein, D., Muthukrishnan, S.: Internet packet filter management and rectangle geometry. In: 12th Symposium on Discrete Algorithms (SODA) (Jan 2001)
- [17] Frantzen, M., Kerschbaum, F., Schultz, E.E., Fahmy, S.: A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals. *Computers & Security* 20(3), 263–270 (2001)
- [18] Juniper: Traffic Policier Overview, [http://www.juniper.net/documentation/en\\_US/junos12.3/topics/concept/policer-overview.html](http://www.juniper.net/documentation/en_US/junos12.3/topics/concept/policer-overview.html)
- [19] Kamara, S., Fahmy, S., Schultz, E.E., Kerschbaum, F., Frantzen, M.: Analysis of vulnerabilities in Internet firewalls. *Computers & Security* 22(3), 214–232 (2003)
- [20] Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-Miner: A tool for finding copy-paste and related bugs in operating system code. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
- [21] Lieberman, H.: Your Wish Is My Command: Programming by Example. Morgan Kaufmann (2001)
- [22] Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: 12th USENIX Symposium on Networked System Design and Implementation (NSDI) (May 2015)
- [23] Mayer, A.J., Wool, A., Ziskind, E.: Fang: A firewall analysis engine. In: IEEE Symposium on Security and Privacy (IEEE S&P) (May 2000)
- [24] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: 14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Mar 2008)
- [25] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.* 1(2), 245–257 (Oct 1979), <http://doi.acm.org/10.1145/357073.357079>
- [26] Nelson, T., Barratt, C., Dougherty, D.J., Fidler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: 24th Large Installation System Administration Conference (LISA) (Nov 2010)
- [27] Plotkin, G.D., Bjørner, N., Lopes, N.P., Rybalchenko, A., Varghese, G.: Scaling network verification using symmetry and surgery. In: 43rd ACM Symposium on Principles of Programming Languages (POPL) (Jan 2016)
- [28] Qian, J., Hinrichs, S., Nahrstedt, K.: ACLA: A Framework for Access Control List (ACL) Analysis and Optimization, pp. 197–211. Springer Science+Business Media New York (2001)
- [29] Tanenbaum, A.: Computer Networks. Prentice Hall, 4th edn. (2002)
- [30] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Mar 2007)
- [31] Wool, A.: Architecing the Lumeta firewall analyzer. In: USENIX Security Symposium (Aug 2001)
- [32] Wool, A.: A quantitative study of firewall configuration errors. *IEEE Computer* 37(6), 62–67 (2004)
- [33] Yuan, Y., Lin, D., Alur, R., Loo, B.T.: Scenario-based programming for SDN policies. In: ACM CoNEXT (CoNEXT) (Dec 2015)
- [34] Zhang, S., Mahmoud, A., Malik, S., Narain, S.: Verification and synthesis of firewalls using SAT and QBF. In: 20th IEEE International Conference on Network Protocols (ICNP) (Oct 2012)