Formal Methods in System Design (2020) 56:127–153
https://doi.org/10.1007/s10703-020-00346-0

Automated repair by example for firewalls

William T. Hallahan[1] · Ennan Zhai[1,2] · Ruzica Piskac[1]

Published online: 30 September 2020

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.

Keywords Synthesis · Firewall repair · Programming by example · Network configuration

1 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

This research was supported in part by the NSF under Grants CCF-1302327, CCF-1553168 and CCF-1715387.

✉ Ruzica Piskac
ruzica.piskac@yale.edu

William T. Hallahan
william.hallahan@yale.edu

Ennan Zhai
ennan.zhai@alibaba-inc.com

1 Computer Science Department, Yale University, New Haven, CT 06511, USA

2 Present Address: Alibaba Group, Seattle, USA

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 [1,2], verification [3], and root-cause troubleshooting [4–6]. For instance, systems like Margrave [4] and Fang [1] 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 a framework, called FireMason, that is the first to not only detects errors in firewall behaviors, but also automatically repair 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 [7], 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.

Furthermore, FireMason is also a stand-alone verification tool. The user specifies a property of interest, and FireMason will either prove that the given property holds, or if it does not hold, it produces counterexamples. As an illustration, if the user wants to verify that all packets with the IP address 1.2.3.4 should be dropped, FireMason either confirms that as correct, or it outputs an example of a packet with an IP address of 1.2.3.4 that would be accepted by the firewall.

By having a description of a firewall as a set of first-order logic formulas we reduce verification to the formula entailment problem, which we decide again using an SMT solver. Additionally, this description is useful as a formal specification of the correct behavior of a firewall implementation. The only existing specification for iptables is a man page [8], which, as a textual description, is inherently imprecise.

Due to this imprecision, developing the set of first-order logic formulas in this work required two steps. First, we careful read the man page specification. When the man page was unclear, we turned to testing on actual implementations, to decide how to resolve the ambiguously. By specifying the behavior as first-order logic formulas, we provide future tool implementors with a precise description of iptables behavior.

Previous work has modeled firewalls using less expressive logics. For example, Zhang et al. [6] use SAT and QBF formulas, while Margrave [4], uses first-order relational logic (specifically, through the use of KodKod [2]). 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. In comparison to the conference version of this work [9], we have expanded the input language, and taken advantage of the new input options with further case studies.

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, we make the following contributions:

- We describe a formalism to model firewalls and their behavior. This formalism allows us to use SMT solvers to prove formal guarantees, which is useful both for verification and repair.
- We explain the first method to automatically repair firewalls based on easily specified examples. Administrators can conveniently specify their desired behaviors, and automate the repair process.
- We describe using SMT solvers to efficiently reason about limit rules, which are not considered by any existing tool.
- We built a workable system that scales well with real-world examples and larger-scale datasets.

This work is an extension of [9], and includes an extended grammar for the input example language, and an expanded evaluation.

2 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 [10,11]. 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 only a small 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 (ACL) firewalls, such as iptables [8], Juniper [12], and Cisco firewalls [13], are

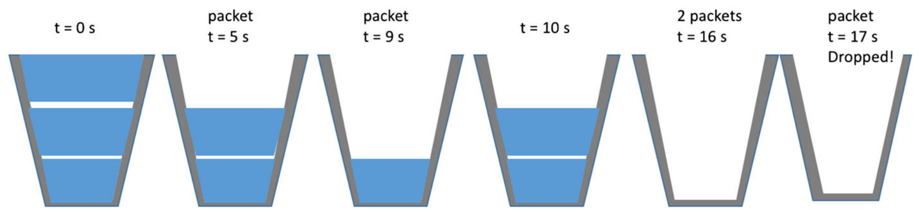


Fig. 1 An visualization of a limit, as packets try to match a rule

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) [14]. 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 with criteria 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
```

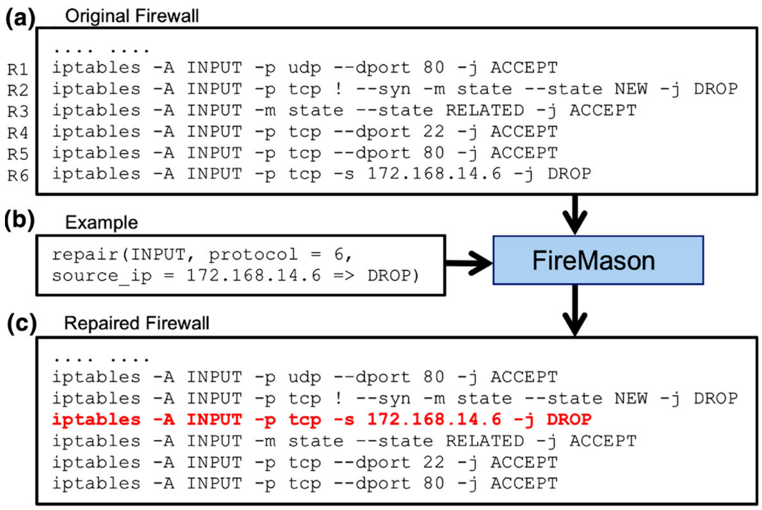
has criteria denoting packets with a protocol of 16 and a source IP address of `123.23.12.1`, and an action specifying those packets should be dropped.

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 checks other rules in the ACL. Non-terminating actions (such as printing to a log file) allow a packet to continue traversing the ACL rules and match 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 cannot 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 [15].

Rate limiting rules Rate limiting rules are used when an administrator wants to restrict the amount of packets matching a certain rule, for example the amount of packets arriving from some IP address. We call a firewall with such rules a *rate limiting firewall*. In many firewalls, including iptables [15], Juniper [12], 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 a match is possible only if a packet satisfies rule's criteria and $l > 0$. A rate limiting behavior is specified through 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.

Rate limiting firewalls use the token bucket algorithm [16] 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:



```
.... ....
R1 iptables -A INPUT -p udp --dport 80 -j ACCEPT
R2 iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
R3 iptables -A INPUT -m state --state RELATED -j ACCEPT
R4 iptables -A INPUT -p tcp --dport 22 -j ACCEPT
R5 iptables -A INPUT -p tcp --dport 80 -j ACCEPT
R6 iptables -A INPUT -p tcp -s 172.168.14.6 -j DROP
```

```
repair(INPUT, protocol = 6,
source_ip = 172.168.14.6 => DROP)
```

```
.... ....
iptables -A INPUT -p udp --dport 80 -j ACCEPT
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
iptables -A INPUT -p tcp -s 172.168.14.6 -j DROP
iptables -A INPUT -m state --state RELATED -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

Fig. 2 An example of a firewall repair problem

Example. Suppose that we set a limit on incoming packets, with $ra = 6$ packet/min and $b = 3$ packets. The firewall is initialized with $l = b = 3$. If we do not exceed the limit, we will accept incoming packets. If we do exceed it, we will drop them. As shown in Fig. 1 suppose that at times 0, 5, 9, and 17 s, we receive 1 packet, and at time 16 s we receive 2 packets.

At the end of the fifth second, $l = 3 - 1 = 2$ since 1 packet arrives. Similarly, at the end of the ninth second, $l = 2 - 1 = 1$ since 1 packet arrives. At the beginning of the tenth second, l is incremented again to 2. At the sixteenth second, we receive two packets. Both will be accepted, but it drains the limit completely. Therefore, since $l = 0$ when the fourth packet arrives, that packet does not match the limit, and is dropped.

3 Motivating examples

Stateless firewall repairing Example 1 An example given in Fig. 2 demonstrates the basic functionality of FireMason. The example is inspired by a StackExchange post [17]. An administrator is maintaining firewall rules written in iptables [8], 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 Fig. 2. Such an action is very common in enterprise-scale firewall management, because administrators prefer appending a new rule to the existing rules [18].

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

```
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 Fig. 2b.

FireMason returns a repaired firewall, Fig. 2c, 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), this method 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, the rule is positioned so that any packet matching the user provided example is guaranteed to be dropped. Rule R1 specifies a protocol other than TCP, and so never matches such a packet. A packet matching the example could match rule R2, but rule R2 drops any matching packet anyway.

This whole example also showed that placing a rule at a wrong place can change the behavior of a firewall. FireMason provides formal guarantees that for every packet, which is not covered by the user provided examples, the original firewall and in the repaired firewall will invoke the same action.

Stateless firewall repairing Example 2 Inspired by posts on ServerFault [19,20], consider an administrator who wants to ensure that the local host, and only the local host, can access the web server at 1.2.3.4. Any traffic not from the local host, but trying to access the ip address 1.2.3.4, should be dropped. To solve this with FireMason, one approach would be to write two examples:

```
repair(INPUT, destination_ip - 1.2.3.4,source_ip = 127.0.0.1
 => ACCEPT)
repair(INPUT, destination_ip - 1.2.3.4, not source_ip =
127.0.0.1 => DROP)
```

Unfortunately, this is redundant and hard to read: it is easy to miss the not on the second line. To make this sort of task easy, we introduce two keywords: onlyif and unless. We can demonstrate the desired behavior in a single example, using the onlyif keyword, as follows:

```
repair(INPUT, destination_ip - 1.2.3.4 => ACCEPT
    onlyif source_ip = 127.0.0.1)
```

Equivalently, the administrator could write the example with the unless keyword:

```
repair(INPUT, destination_ip - 1.2.3.4 => DROP
    unless source_ip = 127.0.0.1)
```

In either case, FireMason will create and add two new rules:

```
iptables -A INPUT -d 1.2.3.4/32 -s 127.0.0.1/32 -j ACCEPT
iptables -A INPUT -d 1.2.3.4/32 ! -s 127.0.0.1/32  -j DROP
```

which ensure the firewall has the desired behavior.

Rate limiting rule repairing 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 s (a task inspired by a forum post on StackExchange [21].) To do this, the administrator may provide a sequence of example packets and relative times, in seconds:

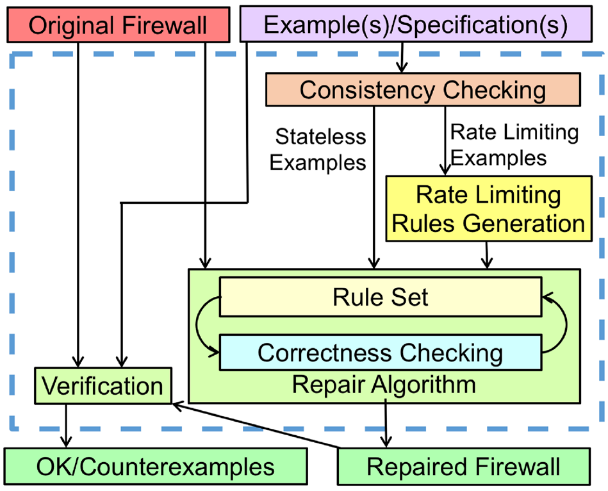


Fig. 3 The workflow overview of FireMason

```
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
```

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

4 System design

Figure 3 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 Sect. 4.1) produces two sets of EUF+LIA formulas [7], which means we can use an SMT solver to reason about firewalls.

The verification process (described in Sect. 4.2) 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 (Sect. 4.4). 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. (Sect. 4.6). FireMason next runs the repair algorithm (Sect. 4.7). Finally, FireMason adds the rules to the firewall (Sect. 4.3), checks if there are redundant rules in the newly generated firewall (Sect. 4.8), and outputs a correct firewall.

4.1 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:

$$\begin{aligned}
\mathit{comm} &:= \mathsf{verify}(\{(\mathit{acl}, \mathit{rule})\}^{+}) \mid \mathsf{repair}(\{(\mathit{acl}, \mathit{rule})\}^{+})\\
\mathit{rule} &:= \mathit{precon}^{+} \Rightarrow \mathit{action} \mid \mathit{precon}^{+} \Rightarrow \mathit{action}\ \mathit{cond}\\
\mathit{precon} &:= \mathsf{protocol} = \textsc{Int} \mid \mathsf{source\_ip} = \text{IP\_}\textsc{Address}\\
&\quad \mid \mathsf{destination\_port} = \textsc{Int} \mid \ \dots \ \mid \mathsf{not}\ \mathit{precon}\\
\mathit{action} &:= \textsc{Accept} \mid \textsc{Drop} \mid \ \dots\\
\mathit{cond} &:= \mathsf{onlyif}\ \mathit{precon} \mid \mathsf{unless}\ \mathit{precon}\\
\mathit{acl} &:= \textsc{String} \qquad\qquad \backslash\backslash \text{ ACL Name}
\end{aligned}$$

We represent every example by a tuple (n, r, t), where n 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 $\mathsf{repair}(\mathsf{protocol} = 16, \mathsf{time} = 5 \Rightarrow \textsc{Accept})$ is translated to $(\text{INPUT}, \mathsf{protocol} = 16 \Rightarrow \textsc{Accept}, 5)$.

Adding a *cond* to a rule allows for stronger statements about the desired behavior. On forums, we noticed users would often ask for rules that implement a certain behavior only if (or, conversely, unless) some condition is met. To help users write down these types of conditions, we introduce two keywords: onlyif and unless. To define these precisely, we use a function on the actions:

$$\text{flipAction}(a) = \begin{cases} \textsc{Accept} & a = \textsc{Drop}\\ \textsc{Drop} & a = \textsc{Accept} \end{cases}$$

Then, we translate the example $p \Rightarrow a$ onlyif $q_1 \dots q_n$ into the rules:

$$\begin{gathered}
p \wedge q_1 \wedge \cdots \wedge q_n \Rightarrow a\\
p \wedge \neg q_1 \Rightarrow \text{flipAction}(a)\\
\dots\\
p \wedge \neg q_n \Rightarrow \text{flipAction}(a)
\end{gathered}$$

and, similarly, translate $p_1 \Rightarrow a$ unless $q_1 \wedge \cdots \wedge q_n$ into the rules:

$$\begin{gathered}
p_1 \wedge \neg q_1 \wedge \cdots \wedge \neg q_n \Rightarrow \text{flipAction}(a)\\
p_1 \wedge q_1 \Rightarrow a\\
\dots\\
p_1 \wedge q_n \Rightarrow a
\end{gathered}$$

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

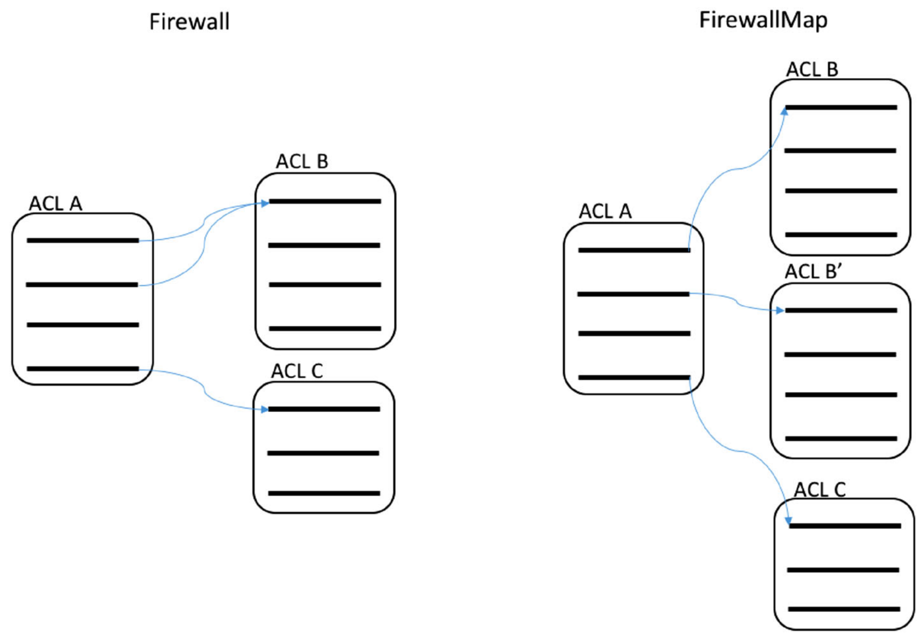


Fig. 4 In a FirewallMap, ACL's are duplicated for each point that they can be jumped to from

$\mathcal{A}_1$. Let us assume that the ACL $\mathcal{A}_1$ has some limit rules. If a packet has to go through both the jumps, then when it reaches the limit in $\mathcal{A}_1$ the second time, the limit in $\mathcal{A}_1$ 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, as shown in Fig. 4. 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 1 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:

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

Table 1 Partial list of predicates and functions used to model firewalls

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

The predicates are designed to make it easy to write formulas with important properties. For example, reaches is used to described which rules a packet is evaluated against, while matches_criteria indicates whether a packet *would* satisfy the criteria of a rule. Building on these, matches_rule is true if and only if a packet both reaches a rule and satisfies the rule.

Table 2 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 $0 \leq p < \mathsf{max\_packets}$ and $\mathsf{valid\_rule}(\mathcal{M}, a, r)$. 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 1) can be finitely instantiated. Thus, no universal quantifiers are needed, and we encode the firewalls in the decidable EUF+LIA logic [7].

Largely, the axioms in Table 2 describe reachability, and how reaches interacts with the other predicates. As an example, consider:

$$\mathsf{reaches}(\mathcal{M}, p, a, r) \wedge \neg\mathsf{matches\_criteria}(\mathcal{M}, p, a, r) \implies \mathsf{reaches}(\mathcal{M}, p, a, r + 1$$

and

$$\mathsf{reaches}(\mathcal{M}, p, a, r + 1) \implies \mathsf{reaches}(\mathcal{M}, p, a, r)$$

The first axioms captures the property that, if a packet has reached a rule, and does not match (satisfy) the criteria of that rule, the packet will reach the next rule. The second axioms states

Table 2 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 3 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 $0 \leq p <$ max_packets

$$\forall p . p \geq 1 \implies \text{arrival\_time}(p) \geq \text{arrival\_time}(p-1)$$

$$\Delta t(p) = \begin{cases} \text{arrival\_time}(p) - \text{arrival\_time}(p-1) & \text{if } 1 \leq p < \text{max\_packets} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{counter\_pre}(\mathcal{M}, i, j, p) = \begin{cases} \text{counter\_post}(\mathcal{M}, i, j-1, p) & \text{if } j \geq 1 \\ \text{min}(\text{counter\_post}(\mathcal{M}, i, & \text{if } p \geq 1 \text{ and } j = 0 \\ \quad \text{j\_max}(i), p-1) + ra * \Delta t(p), b) & \\ b & \text{otherwise} \end{cases}$$

$$\text{counter\_post}(\mathcal{M}, i, j, p) = \begin{cases} \text{counter\_pre}(\mathcal{M}, i, j, p) - sub & \text{if counter\_pre}(i, j, p) \geq sub \wedge \\ & \quad \text{matches\_rule}(\mathcal{M}, p, a, r) \\ \text{counter\_pre}(\mathcal{M}, i, j, p) & \text{otherwise} \end{cases}$$

We use j_max(i) to denote the maximum secondary ID for the limit with main ID i

that in order for a packet to reach a rule, a packet must have also reached the rule that directly precedes it.

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 $.51\overline{6}$ 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 3 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 Sect. 4.6 we explain how $\Delta t(p)$ is known in advance from the examples, so we can obtain ra from the SMT solver.

4.2 Firewall verification

Since firewalls are not annotated with standard specifications, systems for verifying firewalls, such as Margrave [4], verify firewalls against user provided queries. When performing the verification process, FireMason also checks if the given examples violate the firewall rules. In particular, it is helpful to be able to check that packets with certain attributes will be accepted or dropped by a firewall. For example, an administrator might want to verify that any packet received from a certain IP address will be dropped by the firewall.

We first explain the verification process for examples without time (limit) constraints. Given an example, $e = (n, c \Rightarrow act, \emptyset)$ (as described in Sect. 4.1), and a firewall $\mathcal{M}$, we verify e against $\mathcal{M}$ by showing that the following formula $\mathcal{F}$ is valid:

$$\begin{aligned}&\forall p, a.\ a \in \mathsf{equivalent}(\mathcal{M}, n) \wedge \mathsf{reaches}(\mathcal{M}, p, a, 0)\\ &\quad \wedge \mathsf{matches\_example}(p, e) \Rightarrow \mathsf{terminates\_with}(\mathcal{M}, p) = act\end{aligned}$$

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, $e_k = (n_k, c_k \Rightarrow act_k, t_k)$, 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):

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

4.3 Adding rules

Here we outline how to create firewall rules from the provided examples. We first focus on stateless rules. Generating rate limiting rules is described in Sect. 4.6. The repair algorithm, Algorithm 4.7 from Sect. 4.7, assigns each rule a position where it should be placed. After positions are assigned, translating user provided examples to to the iptables language is rather straightforward. For example, the tuple

(`INPUT`, `protocol = 6, source_ip = 1.2.3.4 => ACCEPT`, $\emptyset$)

translates to a rule

```
iptables -A INPUT -p 6 -s 1.2.3.4 -j ACCEPT.
```

4.4 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 4.6.

4.5 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 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),

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

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:

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

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

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

The second guarantee, that the changes we make do not affect more packets than intended, is stated as Formula (3):

$$\forall p.\mathsf{terminates\_with}(\mathcal{M}, p) = \mathsf{terminates\_with}(\mathcal{M}', p) \lor \Big(\exists k, a. 0 \le k < N \land a \in \mathsf{equivalent}(\mathcal{M}, n_k) \land \mathsf{matches\_example}(p, e_k) \land \mathsf{reaches}(\mathcal{M}, k, a, 0)\Big) \tag{3}$$

4.6 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      $E'$.append($(n, r_2, t)$);
5  sortByNameByTime($E'$);
6  if $minRulesAndLimits \neq \emptyset$ and $minTotalSub \neq \emptyset$ then
7      Assert $rulesAndLimits < minRulesAndLimits$
       $\lor(rulesAndLimits = minRulesAndLimits \land totalSub < minTotalSub)$
8  Convert $E'$ to SMT formulas, create formulas defining score and totalSub, run SMT Solver;
9  $sat \leftarrow$ getSat;
10 if $sat = UNSAT$ then
11     $r \leftarrow$ getRulesFromModel(model);
12     return $r$;
13 else
14     model $\leftarrow$ getModel;
15     $(rulesAndLimits, totalSub) \leftarrow$ getScore(model);
16     call this recursively, to lexicographically minimize $(rulesAndLimits, totalSub)$;
```

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 4.6 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 4.7.

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 Sect. 4.1.

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

$$\mathsf{arrival\_time}(p) \wedge \mathsf{matches\_example}(p, e_p) \qquad (4)$$
$$\wedge\, \mathsf{reaches}(\mathcal{M}, p, a, 0) \wedge \mathsf{terminates\_with}(\mathcal{M}, p) = act_p$$

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

$$\neg \mathsf{matches\_example}(r, e_q) \qquad (5)$$

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

$$\mathit{rulesAndLimits} < \mathit{minRulesAndLimits} \vee (\mathit{rulesAndLimits} =$$
$$\mathit{minRulesAndLimits} \wedge \mathit{totalSub} < \mathit{minTotalSub})$$

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 Sect. 4.1 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 $s_1, \ldots s_v$. This can be handled through one additional assertion per *sub* value, $\vee_{u=1}^{v} sub = s_u$. 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.

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      $maxR \leftarrow \mathsf{acl\_length}(a') - 1$;
5      while $res = SAT$ do
6          Pick $r' \leq maxR$ , using a similarity measure to $newR$;
7          $\mathcal{M}$' $\leftarrow$ insertRule($\mathcal{M}$, $newR$, $a'$, $r'$) ;
8          $res \leftarrow$ SMTCheckCorrectness($\mathcal{M}$, $\mathcal{M}$', $e$);
9          if $res = SAT$ then
10             $maxR \leftarrow r' - 1$;
11     $\mathcal{M} \leftarrow \mathcal{M}$'
```

4.7 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 4.6. Suppose we have N non-rule inconsistent examples, $e_1 = (n_1, r_1 = (c_1 \Rightarrow act_1), t_1), ..., e_N = (n_N, r_N, t_N)$. Given a firewall represented by a FirewallMap $\mathcal{M}$, our goal is to 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. The choice of how similar two rules are is a heuristic. Our procedure to decide the similarity assigns a score based on the number and kinds of criteria used in the rules, as well as the action taken by the rule. However, this procedure could be replaced by any desired scoring algorithm.

Consider the k^{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 4.7 outputs a firewall which satisfies this condition. For each example $e_i = (n_i, r_i, t_i)$, we take some $a' \in \mathsf{equivalent}(\mathcal{M}, n_i)$ 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 $\mathsf{insert\_rule}(\mathcal{M}', r_i, a', r')$ 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 $\mathsf{equivalent\_to\_name}(\mathcal{M}, a')$. 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 4.6 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 Sect. 4.6) as in $\mathcal{M}'$, and thus we can expect the same behavior from the examples in both E' and $\mathcal{M}'$.

4.8 Redundant rule detection

The final step in repairing the firewall is removing *redundant* rules—that is, rules which cannot be matched by any packet. An existing approach to redundant rule detection [6] can be adapted to and implemented in our SMT model. We briefly summarize this approach here.

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

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

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.

5 Implementation and evaluation

FireMason is developed in Haskell and fully implements the design described in Sect. 4. The default firewall language that we support is the iptables language [8], but the framework can be easily extended to other firewall languages, such as Juniper [12] 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 [22]. 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 first 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 Fig. 5, 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 4.6. However, Algorithm 4.6'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 4.7.

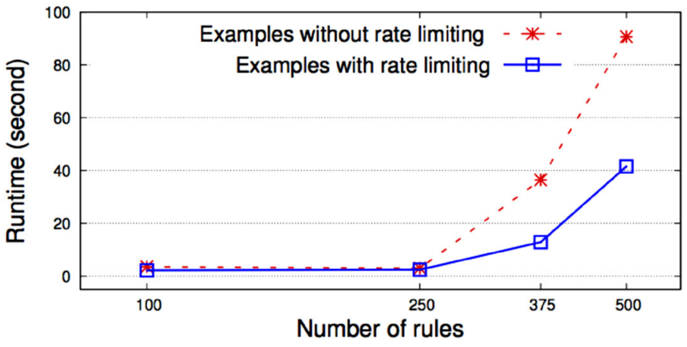


Fig. 5 Scalability for number of rules

We also evaluated the performance of FireMason for different numbers of provided examples, as shown in Table 6. 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 We next demonstrate that FireMason can repair real-world firewalls. To do that, we found firewall repair problems on Server Fault [23] and Stack Overflow [24]. We recreated each scenario, and generated corrected firewalls using FireMason.

Tables 4 and 5 present ten 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, 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, two of 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 6, 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.

6 Related work

This section presents existing efforts on firewall analysis, verification and generation, and discusses why these efforts are not helpful to our target.

Firewall repair and synthesis Chen et al. [31,32] describes an approach to repair stateless firewalls. The paper develops techniques to localize specific forms of faulty rules, as opposed to our approach of building a general model. Unlike our approach, rate-limiting rules are not considered.

Table 4 Case study: sampled stateless firewall repair problems and our solutions

Case study 1 [17]	An administrator appended a rule `iptables -A INPUT -s 73.143.129.38 -j DROP` but can still receive packets from `73.143.129.38`.
Input example	1. `repair(INPUT, source_ip = 73.143.129.38 => DROP)`
Results	Remove the appended rule, and insert a new rule `iptables -A INPUT -s 73.143.129.38/32 -j DROP` in front of an original rule `iptables -A INPUT -i lo -j ACCEPT`.
Original rule count	11
Repair time	.109 s
SMT solver calls	26
Case study 2 [25]	An administrator wants to allow SSH access from the IP address `71.82.93.101`, but does not know how.
Input examples	1. `repair(INPUT, protocol = 22, source_ip = 71.82.93.101 => ACCEPT)` 2. `repair(INPUT, protocol = 22, not source_ip = 71.82.93.101 => DROP)`
Results	Insert new rules `iptables -I INPUT 0 -p 22 -s 71.82.93.101/32 -j ACCEPT` and `iptables -I INPUT 0 -p 22 ! -s 71.82.93.101/32 -j DROP` in front of an original rule `iptables -I INPUT -p icmp -icmp-type time-exceeded -j ACCEPT`.
Original rule count	11
Repair time	.088 s
SMT solver calls	23
Case study 3 [26]	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 `iptables -A INPUT -p tcp -i eth0 -dport 22 -m state -state NEW,ESTABLISHED -j ACCEPT` but still cannot SSH 192.168.1.15.
Input examples	1. `repair(OUTPUT, destination_ip = 192.168.1.15 => ACCEPT)` 2. `repair(INPUT, source_ip = 192.168.1.15 => ACCEPT)`

Table 4 continued

Results	Insert two new rules `iptables -A INPUT -s 192.168.1.15/32 -j ACCEPT` and `iptables -A OUTPUT -d 192.168.1.15/32` in front of the fourth and fifth rules in the original firewall, respectively.
Original rule count	4
Repair time	.054 s
SMT solver calls	14
Case s 4 [19]	An administrator wants to allow only the localhost to have access to a given port, but is having trouble figuring out the right iptables commands.
Input example	1. `repair(INPUT, destination_port = 44344 => ACCEPT` `onlyif destination_ip = 127.0.0.1)`
Results	Inserted four new rules `iptables -A INPUT -p 17 -dport 44344 -d 127.0.0.1/32 -j ACCEPT`, `iptables -A INPUT -p 6 -dport 44344 -d 127.0.0.1/32 -j ACCEPT`, `iptables -A INPUT -p 17 -dport 44344 ! -d 127.0.0.1/32 -j DROP`, and `iptables -A INPUT -p 6 -dport 44344 ! -d 127.0.0.1/32 -j DROP` in the firewall.
Original rule count	6
Repair time	.204 s
SMT solver calls	14
Case study 5 [20]	An administrator wants to prevent all other users from using HTTP or HTTPS connections.
Input example	1. `repair(INPUT, protocol = 6, destination_port = 80 => DROP` `unless source_ip = 10.1.1.2)` 2. `repair(INPUT, protocol = 6, destination_port = 443 => DROP`

Table 4 continued

	`unless source_ip = 10.1.1.2)`
Results	Inserted four new rules `iptables -A INPUT -p 6 -dport 443 ! -s 10.1.1.2/32 -j DROP`, `iptables -A INPUT -p 6 -dport 443 -s 10.1.1.2/32 -j ACCEPT`, `iptables -A INPUT -p 6 -dport 80 ! -s 10.1.1.2/32 -j DROP`, and `iptables -A INPUT -p 6 -dport 80 -s 10.1.1.2/32 -j ACCEPT` in the firewall.
Original rule count	6
Repair time	.246 s
SMT solver calls	13
Case study 6 [27]	An administrator wants to accept connections on a range of ports, but does not know how to do so.
Input example	1. `repair(INPUT, protocol = 17, 1000 <= destination_port <= 2000 => ACCEPT)`
Results	`iptables -A INPUT -p 17 -dport 1000:2000 -j ACCEPT`
Original rule count	6
Repair time	.057 s
SMT solver calls	5
Case study 7 [28]	An administrator wants to block a range of ip addresses, rather than a specific ip address.
Input example	1. `repair(INPUT, source_ip = 116.10.191.* => DROP))`
Results	`iptables -A INPUT -s 116.10.191.0/24 -j DROP`
Original rule count	6
Repair time	0.106 s
SMT solver calls	7

Table 5 Case study: sampled rate limiting firewall repair problems and our solutions

Case study 8 [29]	An administrator is trying to limit the number of inbound SSH packets, but it just seems to lock her out.
Input examples	1. `repair(INPUT, protocol = 22, time = 0 => ACCEPT)`
	2. `repair(INPUT, protocol = 22, time = 20 => ACCEPT)`
	3. `repair(INPUT, protocol = 22, time = 30 => ACCEPT)`
	4. ... (In total, this repair uses 8 examples.)
Results	Insert new rules `iptables -A INPUT -m limit -limit 2/minute -limit-burst 4 -p 22 -j ACCEPT` and `iptables -A INPUT -p 22 -j DROP` at the beginning of the original firewall.
Original rule count	9
Repair time	21.10 s
SMT solver calls	44
Case study 9 [21]	A server is attacked by TCP SYN flooding, so the administrator wants a limit on SYN packets per second.
Input examples	1. `repair(INPUT : source_ip = 192.132.209.0/24, SYN, time = 10 => ACCEPT)`
	2. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 11 => ACCEPT)`
	3. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 12 => ACCEPT)`
	4. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 13 => DROP)`
	5. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 19 => DROP)`
	6. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 21 => ACCEPT)`
Results	Append two new rules, `iptables -I INPUT 0 -s 192.132.209.0/24 -p 6 -tcp-flags SYN -j DROP` and `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`, to the original firewall.

Table 5 continued

Original rule count	11
Repair time	6.046 s
SMT solver calls	42
Case study 10 [30]	An administrator wants to rate limit the number of new TCP connections to there server.
Input examples	1. `repair(INPUT, protocol = 6, destination_port = 22, SYN, time = 0 => ACCEPT)`
	2. `repair(INPUT, protocol = 6, destination_port = 22, SYN, time = 0 => ACCEPT)`
	3. `repair(INPUT, protocol = 6, destination_port = 22, SYN, time = 0 => ACCEPT)`
	4. `repair(INPUT, protocol = 6, destination_port = 22, SYN, time = 1 => DROP)`
Results	Append two new rules, `iptables -A INPUT -m limit -limit 54/minute -limit-burst 3 -p 6 -dport 22 -tcp-flags SYN SYN -j ACCEPT` and `iptables -A INPUT -p 6 -dport 22 -tcp-flags SYN SYN -j DROP`, to the original firewall.
Original rule count	6
Repair time	0.509 s
SMT solver calls	10

Table 6 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

Zhang *et al.* [6] 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 increasingly popular, automatic programming approaches for SDN have been proposed [33,34]. Yuan *et al.* [34] proposed an automatic SDN policy generation approach, named NetEgg, based on a scenario-based programming technique. NetEgg can only generate a new policy, it cannot account for the effect of a new policy on existing policies in the network. Furthermore, NetEgg cannot synthesize rate limiting rules.

Firewall analysis and verification Mayer *et al.* [1] developed the first systematic firewall analysis engine, Fang, to analyze diverse properties of firewalls. Fang and its sequel Lumeta [5] allow checking the correctness of firewall configurations by sending their analysis engines queries. Other efforts [35,36] propose packet-filter based schemes to detect conflicting or violated rules. Frantzen et al. [37] and Kamara et al. [38] proposed different data-flow based approaches to analyze vulnerability risks in firewalls. Yuan et al. [39] used BDDs to detect policy violations and misconfigurations in firewalls. Wool [40] conducted a case study on understanding and classifying the configuration errors of firewalls.

The Margrave firewall verification tool [4] encodes firewall rules and queries into first-order logic. It uses KodKod [2] to search for finite state models. Compared with another firewall verification tool, NoD [3], 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.* [41] proposed targeting test packets for better fault coverage. Al-Shaer *et al.* [42] developed a system-wide framework to generate targeted packets and obtain good coverage during firewall testing. Brucker *et al.* [43] proposed a formal firewall conformance testing approach, which uses Isabelle/HOL to generate test-cases from constraint satisfaction problems.

7 Conclusion

In this paper, we have presented FireMason, a tool for verification and repair of 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. Our empirical evaluation suggests that FireMason could be both practical and effective in assisting administrators with firewall management. Our goal is to inspire further work on reasoning about firewalls in the formal methods community.

Acknowledgements We thank all anonymous reviewers for their insightful comments and suggestions. We also thank the FMCAD'17 organizers, Daryl Stewart and Georg Weissenbacher, for their help in preparing this paper. We thank Avi Silberschatz and Arvind Swaminathan for their valuable feedback on the earlier version of this work.

References

1. Mayer AJ, Wool A, Ziskind E (2000) Fang: a firewall analysis engine. In: IEEE symposium on security and privacy (IEEE S&P)
2. Torlak E, Jackson D (2007) Kodkod: a relational model finder. In: 13th international conference on tools and algorithms for the construction and analysis of systems (TACAS)
3. Lopes NP, Bjørner N, Godefroid P, Jayaraman K, Varghese G (2015) Checking beliefs in dynamic networks. In: 12th USENIX symposium on networked system design and implementation (NSDI)
4. Nelson T, Barratt C, Dougherty DJ, Fisler K, Krishnamurthi S (2010) The Margrave tool for firewall analysis. In: 24th large installation system administration conference (LISA)
5. Wool A (2001) Architecting the Lumeta firewall analyzer. In: USENIX security symposium
6. Zhang S, Mahmoud A, Malik S, Narain S (2012) Verification and synthesis of firewalls using SAT and QBF. In: 20th IEEE international conference on network protocols (ICNP)
7. Nelson G, Oppen DC (1979) Simplification by cooperating decision procedures. ACM Trans Program Lang Syst 1(2):245–257. https://doi.org/10.1145/357073.357079
8. iptables. http://ipset.netfilter.org/iptables.man.html
9. Hallahan WT, Zhai E, Piskac R (2017) Automated repair by example for firewalls. In: 2017 formal methods in computer aided design (FMCAD). IEEE, pp 220–229
10. Cypher A, Halbert D (1993) Watch what I do: programming by demonstration. MIT Press, Cambridge
11. Lieberman H (2001) Your wish is my command: programming by example. Morgan Kaufmann, Burlington
12. Juniper: traffic policier overview. http://www.juniper.net/documentation/en_US/junos12.3/topics/concept/policer-overview.html
13. Cisco: policing and shaping overview. http://www.cisco.com/c/en/us/td/docs/ios/12_2/qos/configuration/guide/fqos_c/qcfpolsh.html
14. Qian J, Hinrichsa S, Nahrstedt K (2001) ACLA: a framework for access control list (ACL) analysis and optimization. Springer, New York, pp 197–211
15. iptables Linux User's Manual (2015)
16. Tanenbaum A (2002) Computer networks, 4th edn. Prentice Hall, Upper Saddle River
17. Cannot block IP address. http://stackoverflow.com/questions/16142446/why-cant-i-block-an-ip-address-with-iptables-on-debian-6
18. Li Z, Lu S, Myagmar S, Zhou Y (2004) 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)
19. IPTables only allow localhost access. https://serverfault.com/questions/247176/iptables-only-allow-localhost-access
20. IP tables allow everything but restrict HTTP for only one IP. https://goo.gl/VTRBui
21. 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
22. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: 14th tools and algorithms for the construction and analysis of systems (TACAS)
23. Server fault. http://serverfault.com/
24. Stack overflow. http://stackoverflow.com/
25. Good iptables starting rules for a webserver. http://serverfault.com/questions/118669/good-iptables-starting-rules-for-a-webserver
26. iptables issue cannot SSH remote machines. http://askubuntu.com/questions/476626/iptables-issue-cant-ssh-remote-machines
27. What is the correct way to open a range of ports in iptables. https://goo.gl/tNXsLp
28. Block range of IP addresses. https://serverfault.com/questions/592061/block-range-of-ip-addresses
29. How can I rate limit SSH connections with iptables. http://serverfault.com/questions/298954/how-can-i-rate-limit-ssh-connections%20-with-iptables
30. Limiting the number of global connections per second. https://serverfault.com/questions/378124/limiting-the-number-of-global-connections-per-second

31. Chen F, Liu AX, Hwang J, Xie T (2012) First step towards automatic correction of firewall policy faults. ACM TAAS 7(2):27
32. Chen F, Liu A.X, Hwang J, Xie T (2010) First step towards automatic correction of firewall policy faults. In: Proceedings of the 24th international conference on large installation system administration. USENIX Association, pp 1–8
33. Plotkin GD, Bjørner N, Lopes NP, Rybalchenko A, Varghese G (2016) Scaling network verification using symmetry and surgery. In: 43rd ACM symposium on principles of programming languages (POPL)
34. Yuan Y, Lin D, Alur R, Loo BT (2015) Scenario-based programming for SDN policies. In: ACM CoNEXT (CoNEXT)
35. Eppstein D, Muthukrishnan S (2001) Internet packet filter management and rectangle geometry. In: 12th symposium on discrete algorithms (SODA)
36. Adiseshu H, Suri S, Parulkar GM (2000) Detecting and resolving packet filter conflicts. In: 19th IEEE international conference on computer communications (INFOCOM)
37. Frantzen M, Kerschbaum F, Schultz EE, Fahmy S (2001) A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals. Comput Secur 20(3):263–270
38. Kamara S, Fahmy S, Schultz EE, Kerschbaum F, Frantzen M (2003) Analysis of vulnerabilities in Internet firewalls. Comput Secur 22(3):214–232
39. Yuan L, Chen H, Mai J, Chuah CN, Su Z, Mohapatra P (2006) Fireman: a toolkit for firewall modeling and analysis. In: 2006 IEEE symposium on security and privacy. IEEE, p 15
40. Wool A (2004) A quantitative study of firewall configuration errors. IEEE Comput 37(6):62–67
41. El-Atawy A, Ibrahim K, Hamed HH (2005) Policy segmentation for intelligent firewall testing. In: IEEE workshop on secure network protocols (NPSec)
42. Al-Shaer E, El-Atawy A, Samak T (2009) Automated pseudo-live testing of firewall configuration enforcement. IEEE J Sel Areas Commun 27(3):302–314
43. Brucker AD, Brügger L, Wolff B (2013) Hol-TestGen/fw—an environment for specification-based firewall conformance testing. In: 10th theoretical aspects of computing (ICTAC)