Language-Independent Synthesis of Firewall Policies

Chiara Bodei, Pierpaolo Degano, Letterio Galletta
Dipartimento di Informatica, Università di Pisa, Italy
{*chiara,degano,galletta*}@di.unipi.it

Riccardo Focardi, Mauro Tempesta, Lorenzo Veronese
DAIS, Università Ca' Foscari Venezia, Italy
{*focardi,tempesta*}@unive.it, 852058@stud.unive.it

Abstract—Configuring and maintaining a firewall configuration is notoriously hard. Policies are written in low-level, platform-specific languages where firewall rules are inspected and enforced along non trivial control flow paths. Further difficulties arise from Network Address Translation (NAT), since filters must be implemented with addresses translations in mind. In this work, we study the problem of *decompiling* a real firewall configuration into an abstract specification. This abstract version throws the low-level details away by exposing the meaning of the configuration, i.e., the allowed connections with possible address translations. The generated specification makes it easier for system administrators to check if: (i) the intended security policy is actually implemented; (ii) two configurations are equivalent; (iii) updates have the desired effect on the firewall behavior. The peculiarity of our approach is that is independent of the specific target firewall system and language. This independence is obtained through a generic intermediate language that provides the typical features of real configuration languages and that separates the specification of the rulesets, determining the destiny of packets, from the specification of the platform-dependent steps needed to elaborate packets. We present a tool that decompiles real firewall configurations from different systems into this intermediate language and uses the Z3 solver to synthesize the abstract specification that succinctly represents the firewall behavior and the NAT. Tests on real configurations show that the tool is effective: it synthesizes complex policies in a matter of minutes and, and it answers to specific queries in just a few seconds. The tool can also point out policy differences before and after configuration updates in a simple, tabular form.

1. Introduction

Firewalls are one of the standard mechanisms for protecting computer networks but, as any other security mechanism, they become useless when incorrectly configured. Configuring a firewall may be a very difficult task also for expert system administrators. In fact, a configuration is typically composed of a large number of rules and it is often hard to figure out what they imply in terms of firewall behavior. In addition, configurations need to be modified according to the updates of the desired security policies. Since rules interact with each other, incautious modifications may unexpectedly impact on the overall behavior of the firewall, with possible severe consequences on the functionality and the security of the network. When a network is protected by more than one firewall the situation complicates further, since the configurations of the various firewalls need to be kept coherent: enabling or disabling a connection typically requires to modify the configuration of all the firewalls that are potentially traversed by the considered connection.

Firewall policy languages are varied and usually rather complex, accounting for low-level system and network details and supporting non trivial control flow constructs, such as jumps and gotos. The way firewall configurations are enforced typically depends on how packets are processed by the network stack of the operating system. Further difficulties for network administrators come from Network Address Translation (NAT), a pervasive component of IPv4 networking that operates while packets traverse the firewall. In IPv4, NAT is indispensable for performing port redirection and translating addresses, e.g., when a single public address is used for a whole private network.

Over the past few years, there has been a growing interest in high level languages for *programming* the network as a whole. The Software Defined Network (SDN) paradigm decouples network control and forwarding functions, by abstracting the underlying infrastructure from applications and network services [1]. A unified, high level paradigm to configure networks and firewalls is appealing and might, in principle, make firewall configuration simpler and less error-prone. However, SDN requires a suitable infrastructure and, even if it seems to be spreading fast, it will take time before "old" technology is dismissed in favor of it. In the years to come, we still have to face a variety of firewall configuration languages, including the ones running on legacy devices.

In this work, we study the problem of *decompiling* a real firewall configuration into an abstract specification that represents the set of the allowed connections. This abstract version of the firewall policy throws away the low-level details, e.g., the control flow, so exposing the *meaning* of the configuration and making it easier for system administrators to check whether or not the intended security policy is correctly implemented. Moreover, by comparing two abstract specifications an administrator can detect the differences between actual configurations and can check that updates have the desired effect on the firewall behavior. Decompilation also opens the way to cross-platform re-compilation into a

different firewall system. This is particularly useful when migrating to a different infrastructure or to a new network configuration paradigm such as SDN.

In the literature there are many tools and techniques for the analysis of specific firewall systems. Our approach is peculiar because of its independence of the specific target firewall system and language. The core of our approach is a generic intermediate language that incorporates all typical features of firewall languages such as NAT, jumps, invocations to rulesets and stateful packet filtering. Interestingly, the intermediate language uncovers the *bipartite structure* common to real firewall languages: the first component consists of the rulesets determining the destiny of packets, the second one specifies the steps needed to elaborate packets and the order in which rulesets are applied. While the format of the rules and the actions are largely shared by the available firewall languages, apart from minor syntactic differences, the second component is peculiar to each operating system and each firewall tool and, intuitively, summarizes the specific low-level behavior of a particular system.

We have developed a tool named FireWall Synthesizer (FWS) that translates real configurations from different firewall systems into the intermediate language and automatically synthesizes an abstract firewall policy, in tabular form. The intermediate language has been developed so to make it relatively easy to compile real firewall configuration languages into it. As a proof of concept, we have developed compilers for the most used firewall tools in Linux and Unix [2], [3], [4] and, partially, for Cisco IOS routers. Interestingly, once a configuration has been translated into the intermediate language, the synthesis can be performed independently of the initial firewall language and system. Therefore new firewall systems are easily supported by just providing a compiler into the intermediate language.

The core of the synthesis is the automatic translation of the firewall configuration from the intermediate language into a first order logic predicate that determines which are the packets accepted by the configuration in hand, with all the possible translations. FWS uses the Z3 solver [5] to derive the actual set of packets accepted and translated by the firewall. The resulting abstract specification is more readable than the standard rulesets, because each row in the generated tables declaratively represents a set of accepted packets with their possible translations. Moreover, rows are *independent* from each other, while in real firewall configurations the meaning of a rule depends on the others and on the firewall control flow (cf. Section 3.4). Moreover, FWS is provided with a query language that allows administrators to perform specific queries about, e.g., which subnets, hosts and ports are reachable from other hosts and subnets. Due to Z3, FWS can also efficiently compute policy equivalence, implication and difference. This makes it possible for an administrator to easily observe the effect of adding, deleting or modifying a rule in the actual firewall configuration.

Contributions. Our contributions can be summarized as follows:

(i) we present FWS, a language-independent tool that, given a real configuration, synthesizes an abstract firewall specification. FWS can efficiently compute policy equivalence, implication and difference and can answer to specific queries about host reachability. The tool is available for download at [6];

(ii) we introduce a new language for firewalls that decomposes a configuration into rule sets and a *control diagram* describing the packet processing flow through the system. The language is the core component of FWS that makes the tool language-independent, but it is also of independent interest as a generic firewall configuration language;

(iii) we provide a new characterization of a firewall configuration with NAT in terms of a first order logic predicate that determines which are the packets accepted by the configuration in hand, with all the possible translations. The formal characterization is, by itself, insightful and provides a formal setting for reasoning on firewall configurations;

(iv) we present results of experiments performed with FWS on real firewall configurations. The tool synthesizes complex policies in a matter of minutes and, and it answers to specific queries just in a few seconds. Policy implication and equivalence are also checked very efficiently.

Related work. The literature has many proposals for simplifying and analyzing firewall configurations. Some are based on formal methods, others consist of *ad hoc* configuration and analysis tools. Many works take a top-down approach, proposing ways to specify abstract filtering policies that can be possibly compiled into the actual firewall systems, e.g., [7], [8], [9], [10], [11], [12], [13]. There also exist several tools for facilitating firewall management and detecting misconfigurations, still following a top-down approach, such as [14], [15], [16], [17], [18], [19], [20], [21], [22].

In this paper we take a dual, bottom-up approach: we synthesize a specification from the actual firewall configuration. Below, we revise papers that take a bottom-up approach and adopt formal methods. To the best of our knowledge, ours is the only one providing at the same time: (i) a language for analysing multiple firewall systems; (ii) an effective technique for synthesizing abstract policies; (iii) a support for NAT; (iv) a formal characterization of firewall behaviour.

Some researchers focused on analyzing `iptables`: Jeffrey et al. introduce in [23] a formal model of firewall policy, based on `iptables`, and investigate the properties of reachability and cyclicity of firewall policy configurations. The proposal by Diekmann et al. [24] has some similarities with ours. In particular, the authors provide a "cleaned" ruleset that an automatic tool can easily analyze, using a formal semantics of `iptables` and a semantics-preserving ruleset simplification (e.g., chain unfolding) with a treatment of unknown match conditions, due to a ternary logic. They give a semantics to a subset of `iptables` that includes access control flow actions, but not packet modification such

as NAT. Our approach supports NAT and is based on a generic language that can target languages different from `iptables`. ITVal [25] is a tool that parses `iptables` rules and can be queried to discover host reachability. The tool is specific for `iptables` and does not aim at synthesizing an abstract firewall specification.

Other proposals in the literature are more general and target, in principle, various firewall systems. Below, we discuss the main differences with respect to our work. A model-driven approach is proposed in [26] to derive network access-control policies from real firewall configuration. A proof of concept is given only for `iptables`. Moreover, compared to our proposal this paper does not address NAT. In [27] the authors propose an algorithmic solution to detect and correct specific anomalies on stateful firewalls. However, the proposed approach does not aim at synthesizing an abstract specification, as we do. FIREMAN [28] is a tool that detects inconsistencies and inefficiencies of firewall policies. It does not support NAT though. In [29] the Margrave policy analyzer is applied to the analysis of IOS firewalls. The approach is rather general and extensible to other languages, however the analysis focuses on finding specific problems in policies rather then synthesizing a high-level policy specification. A framework for the static analysis of networks is proposed in [30]. It provides sophisticated insights about network configurations but does not specifically analyze real firewall configurations and, as for the previous papers, there is no synthesis of high-level specifications. Fang [31] is another tool for querying real policies in order to discover anomalies. Authors state that it synthesizes an abstract policy that resembles the one we propose here, but the tool is unavailable and the paper does not describe the tool internals, making any comparison with our approach impossible.

Jayaraman et al. [32] propose an approach for validating network connectivity policies, implemented by the tool SECGURU. They extract logical specifications from real Cisco IOS routers and solve them in Z3. In our paper we have extended their approach under two main aspects: (i) we treat NAT, i.e., we deal with transformations happening to packets while they are filtered. This is non trivial and will be modeled through logical predicates on pairs of packets: the original and the translated one; (ii) we perform our analysis on a generic language that can be used to represent various real configuration languages, by taking into account the platform-depended packet processing flow. As a consequence, we provide a common logical characterization that fits any real languages modeled in our settings. More details will be given in Section 4, in particular on our original algorithms for generating abstract specifications.

Structure of the paper. In Section 2, we briefly survey `iptables` and `ipfw`, two widespread firewall systems that are supported by FWS. For lack of space we will not describe the other supported firewall systems in the paper. Section 3 illustrates how FWS works on a small yet realistic case study. In particular we focus on how network administrators can exploit FWS to check firewall configurations

for host reachability, policy implication, equivalence and difference. In Section 4, we present the internals of FWS and how it exploits the Z3 solver to automatically synthesize an abstract specification and to perform policy analysis. In Section 5, we discuss the scalability of our approach by illustrating tests on various real firewall policies. In Section 6 we present our intermediate language and we provide a logical characterization of all accepted packets with possible translations. We conclude in Section 7.

2. Background

System administrators use firewalls to monitor the traffic and to enforce a predetermined set of access control policies among the various hosts and subnetworks (*packet filtering*). System administrators can also use a firewall to connect a network with private IPs to other (public IP) networks or to the Internet and to perform connection redirections through Network Address Translation (NAT).

Firewalls can be implemented either as specialized hardware or as software tools running on general purpose operating systems. Independently from their actual implementations, the underlying idea is the same: they are characterized by a set of rules that determine which packets reach the protected network and how they are modified.

For lack of space, in the rest of the paper we will only consider the `iptables` and `ipfw` firewall systems that are two of the most used firewall tools in Linux and Unix. Also, their very different packet processing schemes make it hard understanding if an `iptables` policy filters as an `ipfw` one with no mechanical tool (see Section 3).

`iptables`. It is one of the most used tools for packet filtering as it is the default one in Linux distributions. It operates on top of Netfilter, the standard framework for packets processing implemented in the Linux kernel [33].

The basic notions of `iptables` are *tables* and *chains*. Intuitively, a table is a collection of ordered lists of policy rules called chains. The most commonly used tables are: `filter` for packet filtering; `nat` for network address translation; `mangle` for packet alteration. There are five built-in chains that are inspected at specific moments of the packet life cycle [34]: `PreRouting`, when the packet reaches the host; `Forward`, when the packet is routed through the host; `PostRouting`, when the packet is about to leave the host; `Input`, when the packet is routed to the host; `Output`, when the packet is generated by the host. Tables do not necessarily contain all the predefined chains and further user-defined chains can be added.

Each rule specifies a condition and a target. If the packet matches the condition then it is processed according to the specified target, which can be a built-in target or a user-defined chain. The most commonly used targets are: `ACCEPT`, to accept the packet; `DROP`, to discard the packet; `RETURN`, to stop examining the current chain and resume the processing of a previous chain; `DNAT`, to perform destination NAT, i.e., a translation of the destination address; `SNAT`, to perform source NAT, i.e., a translation of the

source address. When the target is a user-defined chain, two "jumping" modes are available: *call* and *goto*. The difference between the two arises when a `RETURN` is executed or the end of the chain is reached: the evaluation resumes from the rule following the last matched call. Built-in chains have a user-configurable default policy (`ACCEPT` or `DROP`): if the evaluation reaches the end of a built-in chain without matches, its default policy is applied.

ipfw. It is the standard firewall for FreeBSD [3]. In filtering a packet, rules are inspected sequentially until the first match occurs and the corresponding action is taken, similarly to `iptables`. The packet is dropped if there is no matching rule. The sequential order of inspection can be altered by rules containing `skipto`, `call` and `return`; in particular, `skipto` behaves as a *goto* in `iptables`, but it is only possible to jump to a rule that follows the current one. Another difference is that the targets of these actions are rules instead of rulesets. In `ipfw` there is a single ruleset that is inspected twice, when the packet enters the firewall and when it exits. It is possible to specify when a certain rule should be applied using the keywords `in` and `out`. Packets that belong to established connections, e.g., in connection-oriented protocols, can be accepted by using a `check-state` rule.

3. The FWS Tool at Work

This section illustrates how the FWS tool can be used to analyze and manage real firewall configurations. We consider a small yet realistic scenario and we show how FWS supports system administrators in reasoning on and managing a firewall, spotting mistakes and refactoring the configuration so to fix them. In particular, we show how the following behavioral properties can be checked:

- *Reachability*, i.e., whether or not a certain address is reachable from another one, possibly through NAT;
- *Policy implication and equivalence*, i.e., if the packets accepted by one configuration are at least/exactly the same accepted by another configuration;
- *Policy difference*, i.e., what packets are accepted by one configuration and denied by another configuration. This feature is particularly useful when maintaining a policy to check how updates affect the firewall behavior.
- *Related rules*, i.e., which configuration rules affect the processing of the packets identified by a user-provided query.

In Section 3.1 we describe the case study. We then provide two firewall configurations in `iptables` (Section 3.2) and `ipfw` (Section 3.3), and we apply FWS to decompile the actual configurations in a tabular, human-readable format and check whether they meet the requirements stated in Section 3.1. Finally, in Section 3.4 we show how FWS can help administrators in maintaining a firewall configuration.

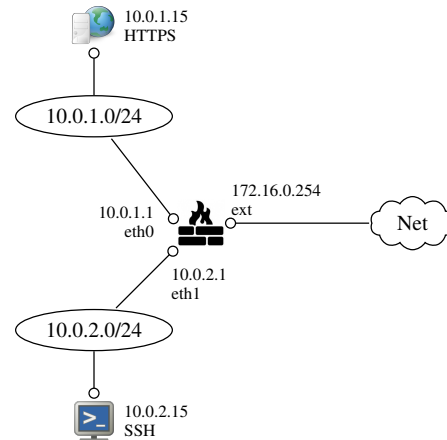FWS, together with all the examples of this section, is available for download at [6].

Figure 1: A case study of a firewall with three interfaces.

3.1. Case Study

As running example, we consider the typical network setup of a small company shown in Figure 1. The internal network consists of two subnetworks:

- network 10.0.1.0/24 contains servers and production machines, including a HTTPS server (10.0.1.15) that runs the company website on port 443;
- network 10.0.2.0/24 contains the machines of the employees, including the computer of the system administrator (10.0.2.15) where a SSH service is running on port 22.

The firewall has three network interfaces: `eth0` connected to 10.0.1.0/24 with IP 10.0.1.1, `eth1` connected to 10.0.2.0/24 with IP 10.0.2.1 and `ext` connected to the Internet with public IP 172.16.0.254.

Requirements. We want to enforce the following requirements on the internal and external traffic:

1) internal networks can freely communicate;
2) connections to the public IP on ports 443 and 22 are translated (DNAT) to 10.0.1.15 and 10.0.2.15, respectively. This condition permits external hosts to access the website by connecting to the public IP address 172.16.0.254 at port 443, that is redirected to the corresponding internal host; similarly for accessing the SSH server;
3) connections from the internal hosts to the Internet are allowed only towards HTTP and HTTPS web servers, i.e., with destination ports 80 and 443, respectively;
4) source addresses of connections from the internal hosts to the Internet are translated (SNAT) to the external IP address of the firewall. This allows hosts with private IPs to access the Internet;
5) the firewall can connect to any other host.

Below, we encode the requirements above as queries that are checked by FWS. To save space, the specific encodings are detailed along their presentation.

```
### NAT rules ###
*nat
# Default policy ACCEPT in nat chains
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

# Requirement 2: Redirect incoming SSH and HTTPS connections to hosts 10.0.2.15 and 10.0.1.15 (DNAT)
-A PREROUTING -p tcp -d 172.16.0.254 --dport  22 -j DNAT --to 10.0.2.15
-A PREROUTING -p tcp -d 172.16.0.254 --dport 443 -j DNAT --to 10.0.1.15
# Requirement 4: Connections towards the Internet exit with source address 172.16.0.254 (SNAT)
-A POSTROUTING -s 10.0.0.0/16 ! -d 10.0.0.0/16 -j SNAT --to 172.16.0.254

COMMIT

### Filtering rules ###
*filter
# Default ACCEPT in output (Requirement 5), DROP in the other chains
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

# Allow established packets
-A FORWARD -m state --state ESTABLISHED -j ACCEPT
-A INPUT   -m state --state ESTABLISHED -j ACCEPT
# Requirement 1: Allow arbitrary traffic between internal networks
-A FORWARD -s 10.0.0.0/16 -d 10.0.0.0/16 -j ACCEPT
# Requirement 3: Allow HTTP/HTTPS outgoing traffic
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 80  -j ACCEPT
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 443 -j ACCEPT
# Requirement 2: Allow SSH/HTTPS incoming traffic to the corresponding hosts
-A FORWARD -p tcp -d 10.0.2.15 --dport  22 -j ACCEPT
-A FORWARD -p tcp -d 10.0.1.15 --dport 443 -j ACCEPT

COMMIT
```

Figure 2: Example policy of Section 3.1 in `iptables`.

3.2. Compliant Configuration in **iptables**

We provide a configuration in `iptables` for the example of Section 3.1. Then, we use FWS to decompile and analyze the configuration and check if it complies with the requirements 1-5 of Section 3.1.

Configuring the firewall with iptables. Figure 2 shows the policy for our example in the `iptables-save` format, i.e., the standard format used to store `iptables` rules in a configuration file.

The first sequence of commands delimited by `*nat` and `COMMIT` keywords sets the default policies of all `nat` chains to `ACCEPT`, inserts into the `nat` `PREROUTING` chain the rules for redirecting the incoming connections to the internal servers (requirement 2) and adds to the `nat` `POSTROUTING` chain the rule for SNAT (requirement 4).

The subsequent block from lines `*filter` to `COMMIT` specifies a default `DROP` policy for the `INPUT` and `FORWARD` chains and a default `ACCEPT` policy for the `OUTPUT` chain, letting the firewall communicate with any host (requirement 5). The first two filtering rules allow the packets belonging to connections flagged as established to go through and towards the firewall, i.e., whenever a new connection is allowed any further packet belonging to the same connection will also be allowed. This is not explicitly required by the policy but is necessary to ensure functionality of connection-oriented protocols. Then, we

have `ACCEPT` rules corresponding to the requirements 1, 3 and 2, respectively. Notice that requirement 2 has also rules in the `nat` table above.

Decompiling and analyzing the configuration. We now use FWS to check that the configuration of Figure 2 meets the requirements 1-5 of Section 3.1. First, we ask the tool the following query:

```
( (srcIp == 10.0.1.0/24 && dstIp == 10.0.2.0/24) ||
  (srcIp == 10.0.2.0/24 && dstIp == 10.0.1.0/24) )
&& state == NEW
```

where `srcIp`, `dstIp` represent the fields for source and destination address of the IP packet entering the firewall interfaces, and `state` tells if a connection is new or established. The query checks whether hosts with `srcIp` 10.0.1.0/24 can start new connections towards those with `dstIp` 10.0.2.0/24, or vice versa, as stated by requirement 1. The operator `==` constrains a variable to be equal to a value or inside a certain interval; the operators `&&` and `||` stand for logical conjunction and disjunction.

The output we obtain from the tool is in Figure 3a, where `*` denotes any value. The table contains all of the allowed connections matching the query, confirming that requirement 1 is satisfied.

We now check that external hosts can access the web and the SSH servers only by connecting to the firewall IP address 172.16.0.254 at port 443 and 22 respectively

Source IP	Source Port	Destination IP	Destination Port	Protocol	State
10.0.2.0/24	*	10.0.1.0/24	*	*	NEW
10.0.1.0/24	*	10.0.2.0/24	*	*	NEW

(a) Requirement 1

Source IP	Source Port	DNAT IP	DNAT Port	Destination IP	Destination Port	Protocol	State
*	*	10.0.1.15	–	172.16.0.254	443	tcp	NEW
*	*	10.0.2.15	–	172.16.0.254	22	tcp	NEW

(b) Requirement 2

Source IP	Source Port	SNAT IP	SNAT Port	Destination IP	Destination Port	Protocol	State
10.0.0.0/16	*	172.16.0.254	–	* \ {10.0.0.0/16}	80 443	tcp	NEW

(c) Requirements 3 and 4

Source IP	Source Port	Destination IP	Destination Port	Protocol	State
172.16.0.254	*	*	*	*	NEW

(d) Requirement 5

Figure 3: Results of FWS when checking the `iptables` configuration of Figure 2

(requirement 2). To do that, we ask which packets can reach the hosts with addresses 10.0.1.15 and 10.0.2.15:

```
(dstIp' == 10.0.1.15  || dstIp' == 10.0.2.15) &&
state == NEW
```

The variable `dstIp'` represents the destination address of the packet possibly translated by a NAT: in the queries the variables with a prime "'", e.g., `dstIp'` above, denote constraints applied to packets exiting a firewall interface; variables without primes instead constrain packets entering the firewall. The result in Figure 3b confirms that requirement 2 is satisfied: indeed, the servers 10.0.1.15 and 10.0.2.15 are reachable from any host connecting to the firewall on ports 443 and 22 only.

The next query checks the requirements 3 and 4 together:

```
srcIp == 10.0.0.0/16 && not(dstIp' == 10.0.0.0/16)
&& state == NEW
```

Intuitively, the query asks for the new connections that are allowed from an internal source to an external destination. The answer in Figure 3c shows that both the requirements are met. Indeed, the notation * \ {10.0.0.0/16} represents all destination addresses except those in the subnet 10.0.0.0/16 (in the standard CIDR notation). Finally, by checking requirement 5 with the query

```
srcIp == 172.16.0.254 && state == NEW
```

we obtain the output of Figure 3d showing that the firewall can reach any host.

We can thus conclude that the configuration in Figure 2 is correct with respect to the requirements.

3.3. Non-Compliant Configuration in `ipfw`

Figure 4 implements the example policy in `ipfw`. On purpose, we introduce subtle but realistic differences with respect to the one in `iptables` and we show how FWS spots them in a clear and concise way.

Configuring the firewall with `ipfw`. The first command declares NAT rules, named `nat 1`, that will be activated by the following rules. Notice that next commands have numbers (after the `add` keyword) that can be used for jumps, as we will see below. We refer to those numbers in the description. Command `00001` accepts all the packets that belong to already established connections (command `check-state`). As for `iptables` this is important to ensure functionality of connection-oriented protocols. Command `00010` enables traffic between internal networks (requirement 1). Command `00100` applies `nat 1` to the packets received via the interface `ext`, implementing the destination NAT of requirement 2. The actual connections to hosts 10.0.1.15 and 10.0.2.15, respectively on ports 443 and 22, are enabled by the next commands `00200`, `00201`, `00300` and `00301`. Notice that packets coming from those hosts are handled by jumping (command `skipto 1000`) to the last but one line, which applies `nat 1`, translating source address to 172.0.16.254 (SNAT). Then, packets are accepted by command `01001`. Next line (command `00500`) implements the requirements 3 and 4 similarly to previous rules, i.e., by jumping to `01000` which enforces the SNAT on outgoing connections. Command `keep-state` is the counter-part of `check-state`: the connection is saved in the firewall state so that packets belonging to the same connection will be allowed through the firewall by rule `00001`. Rule `00501` allows the firewall host to communicate to any host. Finally, command `00999` rejects any packet that does not match any previous rule, implementing a default deny policy.

```
# NAT setup. The first line defines the SNAT for packets leaving the firewall through the interface
# ext (Requirement 4), the other two lines specify to perform DNAT on packets arriving to the ports
# 22 and 443 of the firewall (Requirement 2)
ipfw -q nat 1 config if ext unreg_only reset \
                       redirect_port tcp 10.0.1.15:443 443 \
                       redirect_port tcp 10.0.2.15:22  22


# Allow established packets
ipfw -q add 00001 check-state
# Requirement 1: Allow arbitrary traffic between internal networks
ipfw -q add 00010 allow all from 10.0.0.0/16 to 10.0.0.0/16
# Requirement 2: Apply DNAT on packets arriving to the external interface of the firewall
ipfw -q add 00100 nat 1 ip from any to 172.16.0.254 in recv ext
# Requirement 2: Allow SSH/HTTPS incoming traffic to the corresponding hosts and responses from these services
ipfw -q add 00200 allow tcp from any to 10.0.1.15 443
ipfw -q add 00201 skipto 1000 tcp from 10.0.1.15 443 to any
ipfw -q add 00300 allow tcp from any to 10.0.2.15 22
ipfw -q add 00301 skipto 1000 tcp from 10.0.2.15 22 to any
# Requirements 3 and 4: Allow HTTP/HTTPS outgoing traffic
ipfw -q add 00500 skipto 1000 tcp from 10.0.0.0/16 to any 80,443 setup keep-state
# Requirement 5: Allow arbitrary outgoing traffic by the firewall
ipfw -q add 00501 allow ip from me to any setup keep-state
# Drop all the other packets
ipfw -q add 00999 deny all from any to any
# Apply SNAT to outgoing connnections
ipfw -q add 01000 nat 1 ip from any to not 10.0.0.0/16 out
ipfw -q add 01001 allow ip from any to any
```

Figure 4: Policy in `ipfw`.

Decompiling and analyzing the configuration. We now use FWS to check if the configuration of Figure 4 meets the requirements 1-5 of Section 3.1. We perform exactly the same queries we did for `iptables` in Section 3.2. In fact, one of the advantages of our approach is that the analysis is fully independent of the particular firewall system and of the language analyzed.

Queries for the requirements 1 and 5 give exactly the same results we got for `iptables` (cf. figures 5a, 5d and 3a, 3d). For requirement 2, instead, we get an interesting difference. For the `ipfw` configuration we obtain that hosts 10.0.1.15 and 10.0.2.15 cannot be reached by the internal network and by the firewall host via DNAT (cf. Figure 5b). This is because in the `ipfw` configuration, rule 00100 is defined for interface `ext`, i.e., for packets received from the Internet. In fact, requirement 2 could be interpreted in this stricter way by a system administrator, as hosts 10.0.1.15 and 10.0.2.15 are anyway reachable from internal hosts even without DNAT. FWS is able to spot this subtle difference in the two configurations. To make the `ipfw` configuration behave as the `iptables` one (for requirement 2), it is enough to remove `recv ext` from rule 00100.

In checking the requirements 3 and 4, FWS reports that the hosts 10.0.1.15 and 10.0.2.15 can start new connections from source ports 443 and 22, respectively, to any other host. This is due to rules 00201 and 00301 that enable the two hosts to answer connections done through the DNAT and constitutes an alternative way to make connection-oriented protocols work without exploiting the `check-state` command. In principle, this should be considered non-compliant with requirement 3 as new connections from 443 and 22 from the two hosts will access any port and not just 80 and 443, as requested. Again, FWS spots this difference in the policy. This error can be rectified by removing rules 00201

and 00301 from the policy and adding the `keep-state` keyword to the rules 00200 and 00300.

Interestingly, FWS can compute the equivalence of configurations written for different firewall systems/languages. In this particular case, FWS outputs that the fixed `ipfw` configuration and the `iptables` one are equivalent, relatively to the five requirements.

3.4. Maintaining Firewall Configurations

In this section, we show how FWS can be used to perform maintenance of the `iptables` policy presented in Section 3.2.

The company has added a new machine to the subnet 10.0.1.0/24, which has been assigned the IP address 10.0.1.22. Differently from the other hosts of the network, we want to allow Internet access (with SNAT) to this machine only over HTTPS. The other requirements on the traffic should be preserved. For this purpose, we can add the following rule to the FORWARD chain, which drops connections to port 80 from host 10.0.1.22:

```
-A FORWARD -s 10.0.1.22 -p tcp --dport 80 -j DROP
```

However, we must be careful about the position where to place this rule in order to fulfill the desired requirement and avoid to unintentionally block legal traffic.

If we place the new rule at the end of the FORWARD chain, the *policy equivalence* analysis implemented in FWS reports that the new policy is equivalent to the previous version. We can use the *related rules* analysis to understand which rules are relevant for processing HTTP packets. We find out that the output of the analysis includes only the following filtering rule from the FORWARD chain:

```
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 80
                             -j ACCEPT
```

The above rule accepts all the HTTP traffic from the internal networks and is evaluated before the new `DROP` rule. Hence, our new rule should be placed before this one.

If we add the new rule before those of the other requirements, e.g., after the rules that allow packets of incoming connections, FWS reports that the policy is not equivalent to the previous one. We can check the impact of our changes by running the *policy difference* analysis projected over the HTTP traffic:

```
protocol == tcp && dstPort == 80
```

The output of the analysis is shown in Figure 6a. The first column is + or − for lines that appear in the synthesis or disappear after the updates, respectively. We can see that host 10.0.1.22 is now unable to connect to the Internet, as desired (second table of Figure 6a). However, our update also prevents communications over HTTP with other machines on the internal networks, thus violating requirement 1 (first table of Figure 6a).

The correct place where to add the new rule is between the rule for requirement 1 and those for requirement 3. In this way we allow HTTP traffic from 10.0.1.22 only to the internal networks. If we repeat the analysis, we see that now the only difference is just in the HTTP traffic towards the Internet, as desired (cf. Figure 6b).

4. FWS Architecture and Algorithms

FWS [6] consists of one analysis module and many front-end compilers from real configuration languages into the intermediate language presented in detail in Section 6.

The analysis module is implemented in Haskell. It takes configurations specified in the intermediate language and performs various analyses, as already illustrated in Section 3. Each front-end compiler takes a real firewall configuration for a specific firewall system, and compiles it into the intermediate language. The complexity of this compilation is usually small, as we will discuss in Section 6.3. At the moment, FWS supports the three main Unix firewall systems (`iptables`, `ipfw` and `pf`) and the subset of the Cisco IOS configuration language concerning standard, extended and named ACLs. The front-end compilers are implemented in Python in order to facilitate the extension to new firewall languages and systems, even from potential external contributors.

Encoding in Z3. The analysis module takes as input a firewall configuration expressed in the intermediate language and builds a set of logical predicates $\mathcal{P} = \{\mathcal{P}_1(p, \tilde{p}), \ldots, \mathcal{P}_n(p, \tilde{p})\}$ defined over pairs of packets that characterize the firewall behavior. In particular, $\mathcal{P}_i(p, \tilde{p})$ evaluates to true if the input packet p is accepted as $\tilde{p}$ by the firewall. The union of the pairs that satisfy one of the predicates in $\mathcal{P}$ represents the overall firewall behavior. We work with pairs since the input packet can be modified as it traverses the firewall due to the presence of NAT: of course, when no NAT occurs, p and $\tilde{p}$ coincide. The technical details of the logic encoding will be given in Section 6.4.

Algorithm 1 All-BVSAT*

Input: Formula φ over bit-vectors with free variables $\vec{x}$
Output: Set of multi-cubes $\mathcal{M}$ that are models of φ
1: $B \leftarrow \varphi$
2: $\mathcal{M} \leftarrow \emptyset$
3: **while** B is satisfiable **do**
4:     $\vec{v} \leftarrow$ a satisfiable assignment to B
5:     **for each** multi-cube $\vec{M} \in \mathcal{M}$ **do**
6:         Extend $\vec{M}$ with $\vec{v}$ if possible
7:         $B \leftarrow B \wedge (\vec{x} \notin \vec{M})$
8:     **if** $B \wedge \vec{x} = \vec{v}$ is still satisfiable **then**
9:         $\vec{C} \leftarrow \{v_1\} \times \ldots \times \{v_n\}$
10:         **for each** i in $1..n$ **do**
11:             Expand interval C_i
12:         $\mathcal{M} \leftarrow \mathcal{M} \cup \{\vec{C}\}$
13:         $B \leftarrow B \wedge (\vec{x} \notin \vec{M})$
14: **return** $\mathcal{M}$

We adopt the algorithms presented in [32] to synthesize, through the Z3 solver [5], an abstract firewall specification, starting from the predicates in $\mathcal{P}$. We further exploit Z3 to perform interesting analyses on firewall policies, such as checking for policy equivalence and implication or synthesizing policy differences.

We model packets as tuples of Z3 bit-vector variables of appropriate size (`srcIP`, `srcPort`, `dstIP`, `dstPort`, `protocol`, `state`) that represent source and destination IPs and ports, the protocol and the packet state. Rule constraints are expressed as logical formulas on those packet variables. For example, the constraint

$$\texttt{dstIp} \equiv 10.0.2.15 \wedge \texttt{dstPort} == 22$$

selects packets with destination 10.0.2.15 and port 22. We write $\texttt{dstIp} \equiv 10.0.2.15$ as a shortcut for equating `dstIp` with the numerical representation of the IP address 10.0.2.15. Intervals are encoded with two $\leq$ constraints.

Enumerating accepted packets. In order to succinctly enumerate all the packets accepted by the firewall, FWS adopts the *multi-cubes* representation proposed in [32]. A multi-cube maps each variable v to a union of disjoint intervals I_v to which the value of v belongs. For instance, the solutions of the formula

$$(\texttt{dstIp} \equiv 10.0.2.15 \ \vee \ \texttt{dstIp} \equiv 10.0.1.0/24) \wedge$$
$$(\texttt{dstPort} == 22 \vee \texttt{dstPort} == 443)$$

can be expressed with the following multi-cube:

$$\texttt{dstIp} = \{10.0.2.15\} \cup [10.0.1.0, \ 10.0.1.255],$$
$$\texttt{dstPort} = \{22\} \cup \{443\}$$

For each formula $\mathcal{P}_i$ defined over bit-vector variables, we compute the satisfying multi-cubes using Algorithm 1 [32].

Intuitively, each iteration of the while loop selects an assignment of variables $\vec{v}$ that is not covered by any of the existing multi-cubes. First the algorithm tries to extend the

Source IP	Source Port	Destination IP	Destination Port	Protocol	State
10.0.2.0/24	*	10.0.1.0/24	*	*	NEW
10.0.1.0/24	*	10.0.2.0/24	*	*	NEW

(a) Requirement 1

Source IP	Source Port	DNAT IP	DNAT Port	Destination IP	Destination Port	Protocol	State

* \ {
  10.0.1.0-10.0.2.255
  127.0.0.0/8
}
	*	10.0.2.15	–	172.16.0.254	22	tcp	NEW

* \ {
  10.0.1.0-10.0.2.255
  127.0.0.0/8
}
	*	10.0.1.15	–	172.16.0.254	443	tcp	NEW

(b) Requirement 2

Source IP	Source Port	SNAT IP	SNAT Port	Destination IP	Destination Port	Protocol	State
10.0.2.15	22	172.16.0.254	–	* \ {10.0.0.0/16}	*	tcp	NEW
10.0.1.15	443	172.16.0.254	–	* \ {10.0.0.0/16}	*	tcp	NEW
10.0.0.0/16	*	172.16.0.254	–	* \ {10.0.0.0/16}	80 443	tcp	NEW

(c) Requirements 3 and 4

Source IP	Source Port	Destination IP	Destination Port	Protocol	State
172.16.0.254	*	*	*	*	NEW

(d) Requirement 5

Figure 5: Results of FWS when checking the `ipfw` configuration of Figure 4

+/-	Source IP	Source Port	Destination IP	Destination Port	Protocol	State

+	10.0.0.0/16 \ {
	  10.0.1.22
	}
		*	10.0.0.0/16	80	tcp	NEW

–	10.0.0.0/16	*	10.0.0.0/16	80	tcp	NEW

+/-	Source IP	Source Port	SNAT IP	SNAT Port	Destination IP	Destination Port	Protocol	State

+	10.0.0.0/16 \ {
	  10.0.1.22
	}
		*	172.16.0.254	–	* \ {
					  10.0.0.0/16
					}
						80	tcp	NEW

–	10.0.0.0/16	*	172.16.0.254	–	* \ {
					  10.0.0.0/16
					}
						80	tcp	NEW

(a) Policy differences after the wrong update

+/-	Source IP	Source Port	SNAT IP	SNAT Port	Destination IP	Destination Port	Protocol	State

+	10.0.0.0/16 \ {
	  10.0.1.22
	}
		*	172.16.0.254	–	* \ {
					  10.0.0.0/16
					}
						80	tcp	NEW

–	10.0.0.0/16	*	172.16.0.254	–	* \ {
					  10.0.0.0/16
					}
						80	tcp	NEW

(b) Policy differences after the correct update

Figure 6: Maintenance of the `iptables` configuration

existing multi-cubes with the values in $\vec{v}$; next, if the formula is still satisfiable, a new multi-cube is created.

During the extension/creation of multi-cubes, the algorithm performs an expansion step that extends as much as possible the intervals both downwards and upwards. This step uses a variant of the binary search algorithm to find the bounds of the maximal interval that satisfies the given formula. The complexity of this step is linear in the size in bits of the variable under consideration. We refer the interested reader to [32] for additional details about the algorithm.

Dealing with NAT. In [32], since NAT is not supported, the logical predicate representing firewall behavior can be defined on a single packet and Algorithm 1 is applied directly. With NAT, synthesis gets more complicated: NAT can happen in different moments of packet processing that, in turn, introduce many variables in the formulas, representing intermediate address values for the packet. Some variables, however, are not touched by NAT and this needs to be represented in the predicates, as discussed in the following.

A natural way is to impose equality constraints on variables that are not touched by NAT. Despite intuitive, this approach has a severe drawback: equality constraints do not work well with Algorithm 1. For instance, consider the formula $1 \leq v_1 \leq 5 \wedge v_1 = v_2$: the algorithm uses the SMT solver to find a solution of the formula (e.g., $v_1 = v_2 = \{3\}$) and tries to expand, one after the other, the intervals associated to v_1 and v_2. However, the expansion fails because increasing the interval of v_1 would violate the equality constraint with v_2. The result of the algorithm are 5 different multi-cubes, i.e., $v_1 = v_2 = \{i \in [1..5]\}$.

To solve this problem we introduce new variables only for the packet features that are modified by NAT rules and implicitly model equality constraints by sharing the same variable in the input and in the output packet. For instance, if a NAT rule modifies the destination address of the input packet p, the output packet $\tilde{p}$ is represented with the same variables as p with the exception of the destination address that uses the fresh variable `dstIp'`.

Since the introduction of these fresh variables is only required for the packets that are subject to NAT, we consider separate predicates covering the different cases: DNAT, SNAT and filtering. In DNAT and SNAT all variables will be the same except for the destination and source address, respectively. In filtering, all variables will coincide, as the input and the output packets are the same.

In principle, this separation could lead to an explosion of the number of predicates. However, when studying the existing firewall systems, we found that the maximum number of packets that we need to consider is three: the input packet, the packet after applying destination NAT and the packet after applying source NAT. In fact, in real systems NAT is applied at most twice during packet processing. For this reason, the proposed approach works very well in practice.

Supported Analyses. Besides synthesizing high-level specifications, once we have a firewall expressed as logical constraints in Z3, the FWS tool can perform various interesting fully automated analyses:

Analysis	Multi-cubes	Time (m:s.cs)
$N_1 \to N_2$	35	0:53.73
$N_1 \to N_3$	28	0:37.77
$N_1 \to Out$	25	1:20.65
$N_2 \to N_1$	45	0:45.32
$N_2 \to N_3$	39	0:34.27
$N_2 \to Out$	31	0:57.40
$N_3 \to N_1$	47	2:19.16
$N_3 \to N_2$	17	0:05.68
$N_3 \to Out$	8	0:09.45
$Out \to N_1$	52	6:02.08
$Out \to N_2$	10	0:11.41
$Out \to N_3$	8	0:08.12
Complete policy	138	17:09.31

Table 1: Tests performed on our department policy

- *Reachability*: to check whether or not a certain address is reachable from another one, possibly through NAT, it suffices to impose the desired constraints on the packet variables using the query language of FWS and check satisfiability;
- *Implication*: we can see if policy P_1 is implied by policy P_2 by asking Z3 if there exists no pair of packets that is accepted by P_2 and rejected by P_1;
- *Equivalence*: we can verify if two policies are equivalent by checking mutual implication;
- *Differences*: given two policies, possibly projected, we can synthesize them and show the differences in the extracted multi-cubes;
- *Related rules*: to identify the rules affecting the processing of the packets selected by user-provided query, we remove, one at a time, a rule from the policy and check whether the new policy is not equivalent to the original one.

5. Experiments with Real Configurations

We have used our tool on several policies to assess how our approach scales to real-world scenarios. We have performed our tests on a desktop PC (running Ubuntu 16.04.2) equipped with an Intel i7-3770 CPU and 16 GB of RAM.

Stanford University Backbone Network. It is a medium-sized network that contains 16 operational zone Cisco routers [35]. From the configuration files of these routers we have extracted 252 ACL policies containing 1916 filtering rules in total. Our tool separately synthesized all the policies in 2 minutes and 17.46 seconds; the largest ACL, made of 111 rules, has been analyzed in 16.36 seconds and the corresponding specification consists of 12 multi-cubes. The encoding of the ACL policies in our framework has required a simple, mechanized syntactic translation from the Cisco routers configuration syntax into the intermediate language.

Description	Rules	Multi-cubes	Time (m:s.cs)
Policy from Github	15	11	00:00.765
Ticket from OpenWRT	65	11	00:01.519
Kerberos server	8	14	00:01.635
Policy from a blog	28	25	00:02.572
Eduroam laptop	21	15	00:01.018
Memphis testbed	34	15	00:01.233
Kornwall	52	23	00:02.362
Shorewall	77	48	00:28.154
Home router	76	36	00:05.879
Medium-sized company	90	20	00:25.289
`veroneau.net`	263	7	05:55.690

Table 2: Tests performed on real-world policies

Analysis	$<$1m	1-3m	3-5m	5-10m	10-20m
Subnet $\rightarrow$ Subnet	0	405	37	20	0
Subnet $\rightarrow$ Internet	14	5	1	1	0
Internet $\rightarrow$ Subnet	5	13	1	0	2

Table 3: Tests performed on the *Chair for Network Architectures and Services* firewall policy

Our department policy. The network of our department is logically partitioned in the main network N_1, the labs network N_2 and a mixed network N_3. The firewall acts as a router between these networks and is connected to the Internet via other routers. The policy is written in `iptables`, consists of 530 rules (including both SNAT and DNAT) and contains 5 user-defined chains. In Table 1 we report the execution times and the sizes of the obtained specifications when running our tool on the policy projected on specific source and destination networks, as well as the time required to synthesize the entire firewall policy. The analysis on specific source and destination networks takes less that one minute most of the times and six minutes in the worst case.

Other real-world policies. The authors of [24] have collected a set of anonymized `iptables` configurations from several institutions and from the Internet. Table 2 reports the time needed to perform a complete synthesis for a selected subset of these policies, together with their size and the number of multi-cubes of the synthesized specification.

The repository also contains the firewall configuration of the lab the authors of [24] are affiliated to. The firewall has 22 network interfaces and its policy consists of 4841 `iptables` rules. We have slightly modified the policy to remove checks on MAC addresses since they are currently not supported by FWS. In Table 3 we provide a summary of the time required to produce a synthesis for each possible pair of input/output interfaces and to communicate with the Internet. Most of the analyses terminate in less than 3 minutes and just a couple of cases involving particularly complex subnets take more than 10 minutes to be completed.

Queries. We have performed some tests to evaluate the expressiveness of the output produced by FWS. For instance, in the *Home router* example, we can check which hosts in the private LAN are reachable via the public IP address of the router by running the query

```
dstIp == 117.195.222.105 && state == NEW
```

FWS succinctly reports that external hosts can access the internal server 192.168.1.130 on ports 22, 80, 443 and 1194 via DNAT. For hosts in the private LAN 192.168.1.0/24, both SNAT and DNAT are applied to connections towards the public IP address to avoid the problem of asymmetric routing (this technique is called NAT reflection). For lack of space, we do not discuss the remaining examples that are available online [6].

6. The Intermediate Language

We now present the intermediate language used for decompiling firewall configurations. The language uncovers the bipartite structure common to real firewall languages. The first component consists of a set of rules that are applied to packets, in order to determine their destiny. The format of the rules and the actions they prescribe are largely shared by the available firewall languages, apart from minor syntactic differences. The second component specifies which rulesets are applied and which steps are performed to elaborate packets. This is peculiar to each operating system and each firewall tool and, intuitively, summarizes the specific low-level behavior of a particular system.

6.1. Firewall Rulesets

We now define the format of the rules and when a packet matches a rule. It is convenient to introduce some notation.

Given a packet p, we write $sa(p)$ and $da(p)$ to denote the source and destination addresses of p, respectively. An address a consists of an IP address and possibly a port, which are denoted as $ip(a)$ and $port(a)$. An *address range* n is a pair consisting of a set of IP addresses and a set of ports, written $ip(n):port(n)$. We say that the address a is in the range n (written $a \in n$) if $ip(a) \in ip(n)$, and $port(a) \in port(n)$, when $port(a)$ is defined, e.g., we only check the IP address for ICMP packets. We write $p[da \mapsto a]$ to denote a packet identical to p, except for the destination address da that is translated into a; similarly, with $p[sa \mapsto a]$ we represent a packet equal to p, except for the source address sa that is equal to a. This is useful to model NAT.

We consider *stateful* firewalls that keep track of the state s of network connections and possibly use this information to process a packet. Any existing network connection can be described by several protocol-specific properties, e.g., source and destination addresses or ports, and by the translations to apply. This is fundamental with NAT since the translation applied to the packet that started a new connection needs to be transparently applied to any further packet on the same connection. Thus, filtering and translation decisions may depend on the previous packets belonging to the same connection.

A *firewall rule* is composed of two parts: a predicate ϕ expressing criteria over packets, and an action t, called *target*, defining the "destiny" of matching packets. Let Actions be the set of actions a firewall may perform over a packet.

Here we consider a core set of actions included in most of the available firewall languages:

ACCEPT	packet is accepted
DROP	packet is discarded
CALL(R)	invoke the ruleset R
RETURN	exit from the current ruleset
GOTO(R)	jump to the ruleset R
NAT(n_d, n_s)	network address translation
CHECK-STATE(X)	examine the state

Intuitively, ACCEPT and DROP accept and drop a packet, respectively; the targets CALL(_) and RETURN implement a behavior close to procedure call; GOTO(_) is an unconditional jump; NAT performs address translation and n_d and n_s are address ranges used to specify the translated destination and source address of a packet, respectively; in the following we use the symbol $\star$ to denote an identity translation, e.g., $ip(n) : \star$ means that the IP addresses are translated according to $ip(n)$, whereas ports are unmodified; CHECK-STATE examines the state and, for matching packets, it reapplies established translations. The argument $X \in \{\rightarrow, \leftarrow, \leftrightarrow\}$ of the CHECK-STATE action denotes the fields of the packets that are rewritten according to the information from the state. More precisely, $\rightarrow$ rewrites the destination address, $\leftarrow$ the source one, and $\leftrightarrow$ rewrites both.

Formally, a rule is defined as follows.

Definition 1 (Firewall rule). *A firewall rule r is a pair (ϕ, t) where ϕ is a logical formula over a packet, and $t \in$ Actions is the target of the rule.*

A packet p matches a rule (ϕ, t) whenever $\phi(p)$ evaluates to true. Rules are organized in (possibly empty) lists called *rulesets*. Similarly to real firewall implementations, we inspect the rules, one after the other, until we find a matching one, which establishes the destiny of the packet. Rulesets have a default target denoted by $t_d \in \{\text{ACCEPT}, \text{DROP}\}$, which accepts or drops the packet when no other rule matches.

6.2. Control Diagram

The other peculiar component of our intermediate language specifies the steps performed by the kernel of the operating system to process a single packet passing through the firewall. A packet is subject to different rulesets and we represent the control flow of rule inspection through a labeled directed graph called *control diagram*.

The nodes of a control diagram represent different processing steps and arcs determine their sequence. The arcs are labeled with a predicate describing the requirements a packet has to meet in order to pass to the next processing step. We assume control diagrams to be deterministic, i.e., that every pair of arcs leaving the same node has mutually exclusive predicates.

Definition 2 (Control diagram). *Let Ψ be a set of predicates over packets. A* control diagram *$\mathcal{C}$ is a tuple (Q, A, q_i, q_f) where*
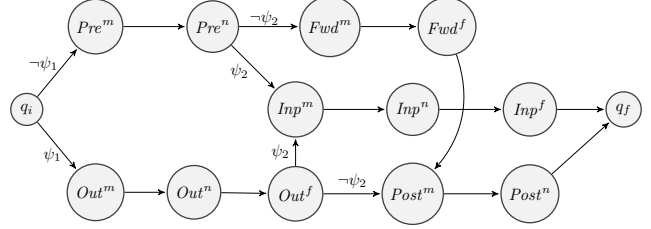
- *Q is the set of nodes;*

Figure 7: The control diagram of iptables

- *$A \subseteq Q \times \Psi \times Q$ is the set of arcs, such that whenever $((q, \psi, q'), (q, \psi', q'') \in A \wedge q' \neq q'')$ then $\neg(\psi \wedge \psi')$;*
- *$q_i, q_f \in Q$ are distinguished nodes denoting the starting and final point of elaboration.*

The firewall manipulates, possibly translates and filters a given packet by traversing a control diagram accordingly to the following transition function.

Definition 3 (Transition function). *Let $\mathcal{C} = (Q, A, q_i, q_f)$ be a control diagram and let p be a packet. The transition function $\delta \colon Q \times Packets \mapsto Q$ is defined as follows*

$$\delta(q, p) = q' \quad iff \quad \exists (q, \psi, q') \in A. \ \psi(p) \ holds.$$

We remark that, even though a control diagram looks like a finite state automata, it is *not* because its arcs are labeled by mutually exclusive predicates, rather than letters.

6.3. Firewall Definition and Examples

We define a firewall as the combination of the two compontents: rulsets (cf. Section 6.1) and control diagram (cf. Section 6.2).

Definition 4 (Firewall). *A firewall $\mathcal{F}$ is a triple $(\mathcal{C}, \rho, c)$, where $\mathcal{C}$ is a control diagram; ρ is a set of rulesets; and $c \colon Q \mapsto \rho$ is the mapping from the nodes of $\mathcal{C}$ to the actual rulesets.*

To illustrate, we instantiate the above definition to iptables and ipfw.

Modelling iptables. We define the firewall $(\mathcal{C}, \rho, c)$ for iptables. Let $\mathcal{L}$ be the set of local addresses of a host and let ψ_1 and ψ_2 be the following predicates over packets:

$$\psi_1(p) = sa(p) \in \mathcal{L} \qquad \psi_2(p) = da(p) \in \mathcal{L}.$$

Figure 7 shows the control diagram $\mathcal{C}$ of iptables, where the nodes q_i and q_f mark start and end of packet processing and untagged arcs carry the label "$true$". The transition function for iptables is defined accordingly to Definition 3, starting from the control diagram $\mathcal{C}$, e.g., we have

$$\delta(Pre^n, p) = \begin{cases} Inp^m & \text{if } \psi_2(p) \\ Fwd^m & \text{if } \neg\psi_2(p) \end{cases} \quad \delta(Fwd^f, p) = Post^m$$

Each of the twelve built-in chains of iptables correspond to a single ruleset and is represented by a different node in the diagram. The node name is an abbreviation of
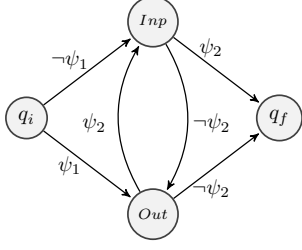
Figure 8: The control diagram of `ipfw`

the chain name while the superscript is an abbrevation of the table name. For example Pre^n is the PREROUTING chain of the NAT (n) table.

We also assume that the first rule of NAT rulesets Pre^n, $Post^n$, Inp^n and Out^n is CHECK-STATE. In fact, `iptables` applies NAT chains only to new connections, i.e., packets belonging to established connections are accepted by default. The twelve built-in chains are included in the set ρ of rulesets (cf. Definition 4) but notice that ρ can also contain additional user-defined chains. In the initial and final states q_i and q_f we assume to have an empty ruleset with ACCEPT as default policy.

The translation of the actual `iptables` configuration into the rulesets of the intermediate language mainly amounts to a simple syntactic translation, performed by the corresponding front-end complier of FWS. Notice that invocation of user defined chains is implemented by the action CALL(_).

Modelling `ipfw`. The control diagram $\mathcal{C}$ of `ipfw`, displayed in Figure 8, is simpler than the one of `iptables` (cf. Figure 7). The node *Inp* represents the procedure executed when an IP packet reaches the host from the net. Dually, *Out* is for when the packet leaves the host. The predicates ψ_1, ψ_2 are defined as for `iptables` and check whether the packet has been generated by the host or is addressed to the host itself, respectively. The transition function δ easily follows from $\mathcal{C}$, according to Definition 3.

We present the construction of the rulesets associated to the node *Inp*. Let $R = [r_{id_1}, \ldots, r_{id_k}]$ be the unique ruleset of `ipfw`, where the id_i's are the numeric identifiers associated to the rules and r_{id_k} is the rule encoding the default policy set by the user. The idea is to generate k different rulesets R_i^I, one for each rule in R. If the rule r_{id_i} contains the keyword `out`, i.e., the rule is not considered when the packet enters the firewall, we let $R_i^I = [(true, \text{GOTO}(R_{i+1}^I))]$. Otherwise, we define $R_i^I = [trs(r_{id_i}), (true, \text{GOTO}(R_{i+1}^I))]$, where the translation trs is defined by cases below:

$$trs(r) = \begin{cases} (\phi, \text{GOTO}(R_n^I)) & \text{if } r \text{ is skipto } id_n \ \phi \\ (\phi, \text{CALL}(R_n^I)) & \text{if } r \text{ is call } id_n \ \phi \\ (\phi, \text{t}) & \text{if } r \text{ is t } \phi \end{cases}$$

The construction of the rulesets R_i^O for the node *Out* is similar, but in this case the rules containing the keyword `in` should be ignored. The mapping function c returns R_1^I for

Inp, R_1^O for *Out*, and empty ruleset with ACCEPT as default policy for q_i and q_f. These rulesets form the component ρ.

6.4. Formalization

We now formalize the behavior of firewall expressed in the intermediate language by providing a logical predicate on packet pairs $(p, \tilde{p})$ that holds true whenever a packet p is accepted as $\tilde{p}$ by the firewall. This formalization is the one used by FWS (cf. Section 4). We first discuss how to remove control flow actions from the firewall specification. Then, we translate a firewall specification with no control flow actions into a logical predicate.

Firewall unfolding. By using rules with control flow actions GOTO(_), CALL(_) and RETURN we can deal with firewalls with an involved control flow. Now we show how to avoid these actions without reducing the expressiveness of our language. To this aim, we introduce an unfolding procedure that, given a ruleset R, produces an equivalent ruleset $[\![R]\!]$ that contains no control flow actions.

We process in order the rules contained in R. When a return rule (ϕ, RETURN) is encountered, we delete it and add the conjunct $\neg\phi$ to the predicates of the remaining rules of the ruleset. In the case of a call rule $(\phi, \text{CALL}(R'))$ we recursively unfold R', add the conjunct ϕ to all the rules in $[\![R']\!]$ and substitute the call rule with $[\![R']\!]$. The case of a goto rule $(\phi, \text{GOTO}(R'))$ is similar to the previous one, with the difference that the conjunct $\neg\phi$ is added to the predicates of the remaining rules in the ruleset. Essentially, $(\phi, \text{GOTO}(R'))$ is equivalent to a rule $(\phi, \text{CALL}(R'))$ followed by the rule (ϕ, RETURN). Rules having a target that is not related to control flow are left unchanged.

The procedure described so far does not terminate in case of rulesets that are mutually calling each other. In such cases, existing firewall systems discard the packet involved in the loop. Similarly, when we detect a loop during the unfolding, the target of the call/goto rule that is causing the loop is replaced with a DROP.

We can use the procedure defined above to transform an entire firewall. Intuitively, given a firewall it suffices to iterate over the nodes of the control diagram and unfold the associated rulesets.

Logical characterization of firewalls. We construct a logical predicate that characterizes all the packets accepted by a ruleset, together with the relevant translations. Hereafter we will only consider unfolded firewalls, i.e., firewalls with no control flow actions like GOTO(_), CALL(_) and RETURN.

To deal with NAT, we define an auxiliary function tr that takes a packet p and computes the set of packets resulting from all possible NAT translations that the firewall may apply to p. Intuitively, it modifies the destination and source addresses according to ranges d_n, s_n. Similarly to CHECK-STATE(X), the parameter $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ specifies

$$P_\epsilon(p, \tilde{p}) = dp(R) \land p = \tilde{p}$$

$P_{r;R}(p, \tilde{p}) = (\phi(p) \land p = \tilde{p}) \lor (\neg\phi(p) \land P_R(p, \tilde{p}))$	if $r = (\phi, \text{ACCEPT})$
$P_{r;R}(p, \tilde{p}) = \neg\phi(p) \land P_R(p, \tilde{p})$	if $r = (\phi, \text{DROP})$
$P_{r;R}(p, \tilde{p}) = (\phi(p) \land \tilde{p} \in tr(p, d_n, s_n, \leftrightarrow)) \lor (\neg\phi(p) \land P_R(p, \tilde{p}))$	if $r = (\phi, \text{NAT}(d_n, s_n))$
$P_{r;R}(p, \tilde{p}) = (\phi(p) \land \tilde{p} \in tr(p, *:*, *:*, X)) \lor (\neg\phi(p) \land P_R(p, \tilde{p}))$	if $r = (\phi, \text{CHECK-STATE}(X))$

Table 4: Translation of rulesets into logical predicates.

if the translation applies to source, destination or both addresses, respectively.

$$tr(p, d_n, s_n, \leftrightarrow) \triangleq \{p[da \mapsto a_d, sa \mapsto a_s] \mid a_d \in d_n, a_s \in s_n\}$$
$$tr(p, d_n, s_n, \rightarrow) \triangleq \{p[da \mapsto a_d] \mid a_d \in d_n\}$$
$$tr(p, d_n, s_n, \leftarrow) \triangleq \{p[sa \mapsto a_s] \mid a_s \in s_n\}$$

Furthermore, we model the default policy of a ruleset R with the predicate dp, where $dp(R)$ is true when the policy is ACCEPT, false otherwise.

Given an unfolded ruleset R, we define a predicate P_R on pairs of packets: when $P_R(p, \tilde{p})$ holds, then the packet p can be accepted as $\tilde{p}$ by R. Its definition is in Table 4, that induces on the rules in R. Intuitively, the empty ruleset applies the default policy $dp(R)$ and does not transform the packet, which is encoded by the constraint $p = \tilde{p}$; rule (ϕ, ACCEPT) is the conjunction of two cases: $\phi(p)$ and $\neg\phi(p)$. In the first case, the packet is accepted as it is, while in the other case $(p, \tilde{p})$ is accepted only if the continuation R (represented as $P_R(p, \tilde{p})$) accepts it. Rule (ϕ, DROP) accepts pairs accepted by the continuation R only if $\phi(p)$ does not hold. The NAT rule $(\phi, \text{NAT}(d_n, s_n))$ is treated similarly to (ϕ, ACCEPT): the only difference is that $\tilde{p}$ is obtained by applying the NAT translation to p, written $tr(p, d_n, s_n, \leftrightarrow)$. Finally, $(\phi, \text{CHECK-STATE}(X))$ is like a NAT that applies all possible translations of kind X (written as $tr(p, *:*, *:*, X)$). Intuitively, we abstract away from the actual state of established connections and we over-approximate it by considering any possible translations. At run-time, only the connections corresponding to the actual state will be possible.

We define the predicate associated to the firewall as follows.

Definition 5. *Let* $\mathcal{F} = (\mathcal{C}, \rho, c)$ *be a firewall and* $\mathcal{C} = (Q, A, q_i, q_f)$ *its corresponding control diagram.*

The predicate associated to the firewall is defined as

$$\mathcal{P}_\mathcal{F}(p, \tilde{p}) \triangleq \mathcal{P}_{q_i}^{\emptyset}(p, \tilde{p}) \quad \text{where}$$

$$\mathcal{P}_{q_f}^I(p, \tilde{p}) \triangleq p = \tilde{p}$$

$$\mathcal{P}_q^I(p, \tilde{p}) \triangleq \exists p'. P_{c(q)}(p, p') \land \left(\bigvee_{\substack{(q, \psi, q') \in A \\ q' \notin I}} \psi(p') \land \mathcal{P}_{q'}^{I \cup \{q\}}(p', \tilde{p}) \right)$$

for all $q \in Q$ *such that* $q \neq q_f$. *Note that* $P_{c(q)}$ *is the predicate constructed from the ruleset associated to the node* q *of the control diagram.*

Intuitively, in the final state we just accept the pair $(p, \tilde{p})$. In all the other states, we look for an intermediate packet p', such that the pair (p, p') is accepted by the ruleset $c(q)$ of state q, packet p' satisfies one of the $\psi's$ in the branches of the control diagram and the pair $(p', \tilde{p})$ is accepted in the reached state q'. Thus, $(p, \tilde{p})$ will be accepted if and only if there is a path starting from p in the control diagram that obtains $\tilde{p}$ through intermediate transformations. The set I is used to avoid infinite loops in the generation of the predicate. Notice that we can ignore paths with loops, although some control diagrams may contain them, because firewalls have mechanisms to detect and discard a packet when its elaboration loops.

7. Conclusions

We have presented FWS [6], a tool for the decompilation of real firewall configurations. It parses real configuration files and produces an abstract tabular specification summarizing all accepted packets with possible address translation. FWS also features various policy analyses that can be used by system administrators to confirm the effect of a configuration update in terms of packet filtering and NAT.

The peculiarity of our approach is the adoption of an intermediate language for firewalls that separates the specification of filtering/NAT rules from the platform-dependent processing of the packet, expressed in terms of a control diagram. The intermediate language provides a generic way to specify and configure a firewall and is, by itself, an original contribution. We have shown that real firewall systems can be encoded in the intermediate language by providing a specific control diagram. Then, a real configuration can be compiled into the intermediate language by translating the firewall rules into the intermediate language syntax.

We have provided the language with a characterization in terms of a logical predicate on pairs of packets $(p, \tilde{p})$, which holds true whenever packet p is accepted by the firewall as $\tilde{p}$. Working on pairs of packets accounts for NAT. We have reused existing algorithms [32] to extract a succinct representation of all the accepted pairs using the Z3 solver. The results on real configurations show that the tool is effective and can typically answer to queries in just a few seconds. Complex queries or configurations, however, can require minutes because of the complexity of the predicate and the repeated invocations of the Z3 solver.

Future work. We want to extend our work in several directions: encode more languages, e.g., from specialized

firewall devices; add a graphical user interface to improve the usability of the tool; perform further experiments and get feedback from network administrators; model MAC addresses and other features of TCP/IP Level 2. We also plan to explore firewall configuration for new network configuration paradigms such as SDNs.

We will explore new techniques to synthesize the abstract specification in order to improve the tool performance. In particular, we want to take into account the initial intervals in the configurations while computing the abstract specification. The adopted algorithm in Z3 forgets this information by encoding the initial intervals as constraints on address variables in the logical predicate but, in fact, the bounds of the computed multi-cubes directly depends on the bounds of the intervals specified in the firewall rules. A promising alternative could be considering an approach based on Header Space Analysis [30].

We intend to investigate the problem of *compiling* an abstract specification into concrete firewall languages along the lines of [36]. This would enable cross-compilation from one firewall system to another, complementing what we presented in this paper. In particular, we intend to develop a compiler that is parametric with respect to the control diagram of the target language.

Finally, it would be very interesting to extend our approach to deal with networks with more than one firewall. The idea would be to combine the synthesized specifications based on network topology and routing.

Acknowledgment

References

[1] Open Networking Foundation, "Software-Defined Networking," https://www.opennetworking.org/sdn-resources/sdn-definition.

[2] "Netfilter," https://www.netfilter.org/.

[3] "The IPFW Firewall," https://www.freebsd.org/doc/handbook/firewalls-ipfw.html.

[4] "Packet Filter (PF)," https://www.openbsd.org/faq/pf/.

[5] Microsoft Research, "The Z3 Theorem Prover," https://github.com/Z3Prover/z3.

[6] FireWall Synthesizer (FWS): Tool and Examples, https://github.com/secgroup/fws.

[7] P. Adão, C. Bozzato, G. Dei Rossi, R. Focardi, and F. L. Luccio, "Mignis: A Semantic Based Tool for Firewall Configuration," in *proc. of the 27th IEEE CSF*, 2014, pp. 351–365.

[8] P. Adão, R. Focardi, J. D. Guttman, and F. L. Luccio, "Localizing firewall security policies," in *proc. of the 29th IEEE CSF, Lisbon, Portugal, June 27 - July 1*, 2016, pp. 194–209.

[9] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège, "A Formal Approach to Specify and Deploy a Network Security Policy," in *proc. of 2nd IFIP FAST*, 2004, pp. 203–218.

[10] M. G. Gouda and A. X. Liu, "Structured Firewall Design," *Computer Networks*, vol. 51, no. 4, pp. 1106–1120, 2007.

[11] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: semantic foundations for networks," in *proc. of 41st ACM POPL*, 2014, pp. 113–126.

[12] S. N. Foley and U. Neville, "A Firewall Algebra for OpenStack," in *Proceedings of the 3rd IEEE Conference on Communications and Network Security, CNS 2015*, 2015, pp. 541–549.

[13] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool, "Firmato: A novel Firewall Management Toolkit," *ACM Transactions on Computer Systems*, vol. 22, no. 4, pp. 381–420, 2004.

[14] B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher, "Specifications of a High-Level Conflict-Free Firewall Policy Language for Multi-Domain Networks," in *12th ACM SACMAT*, 2007, pp. 185–194.

[15] "High Level Firewall Language," https://www.cusae.com/hlfl/, 2003.

[16] "NeTSPoC: A Network Security Policy Compiler," https://hknutzen.github.io/Netspoc/, 2011.

[17] "Uncomplicated Firewall," https://help.ubuntu.com/community/UFW.

[18] "Pyroman," http://pyroman.alioth.debian.org/, 2011.

[19] "Shorewall," http://www.shorewall.net/, 2014.

[20] "KMyFirewall," https://sourceforge.net/projects/kmyfirewall/, 2008.

[21] "Firestarter," http://www.fs-security.com/, 2007.

[22] "Firewall Builder," http://www.fwbuilder.org/, 2012.

[23] A. Jeffrey and T. Samak, "Model checking firewall policy configurations," in *Proceedings of the 10th IEEE International Symposium on Policies for Distributed Systems and Networks, POLICY 2009*, 2009, pp. 60–67.

[24] C. Diekmann, J. Michaelis, M. P. L. Haslbeck, and G. Carle, "Verified iptables Firewall Analysis," in *the 15th IFIP Networking Conference, Vienna, Austria, May 17-19, 2016*, 2016, pp. 252–260.

[25] R. M. Marmorstein, "Formal Analysis of Firewall Policies," Ph.D. dissertation, College of William and Mary, May 2008.

[26] S. M. Perez, J. Cabot, J. García-Alfaro, F. Cuppens, and N. Cuppens-Boulahia, "A Model-Driven Approach for the Extraction of Network Access-Control Policies," in *Proceedings of the Workshop on Model-Driven Security Workshop, MDsec 2012*, 2012.

[27] F. Cuppens, N. Cuppens-Boulahia, J. García-Alfaro, T. Moataz, and X. Rimasson, "Handling Stateful Firewall Anomalies," in *Proceedings of the 27th IFIP Information Security and Privacy Conference, SEC 2012*, 2012, pp. 174–186.

[28] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra, "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis," in *27th IEEE S&P*, 2006, pp. 199–213.

[29] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The Margrave Tool for Firewall Analysis," in *Proceedings of the 24th Large Installation System Administration Conference, LISA 2010*, 2010.

[30] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*, 2012, pp. 113–126.

[31] A. J. Mayer, A. Wool, and E. Ziskind, "Fang: A Firewall Analysis Engine," in *proc. of the 21st IEEE S&P 2000*, 2000, pp. 177–187.

[32] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman, "Automated Analysis and Debugging of Network Connectivity Policies," Microsoft, Tech. Rep., 2014.

[33] R. Russell, "Linux 2.4 Packet Filtering HOWTO," http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html.

[34] The Netfilter Project, "Traversing of Tables and Chains," http://www.iptables.info/en/structure-of-iptables.html.

[35] "Stanford University Backbone Network Configuration Ruleset," https://bitbucket.org/peymank/hassel-public/.

[36] C. Bodei, P. Degano, R. Focardi, L. Galletta, and M. Tempesta, "Transcompiling firewalls," in *Proc. 7th International Conference on Principles of Security and Trust*, ser. LNCS, 2018.