Transcompiling Firewalls

Chiara Bodei[1], Pierpaolo Degano[1], Riccardo Focardi[2], Letterio Galletta[1,3](✉), and Mauro Tempesta[2]

[1] Università di Pisa, Pisa, Italy
galletta@di.unipi.it
[2] Università Ca' Foscari, Venice, Italy
[3] IMT School for Advanced Studies, Lucca, Italy

Abstract. Porting a policy from a firewall system to another is a difficult and error prone task. Indeed, network administrators have to know in detail the policy meaning, as well as the internals of the firewall systems and of their languages. Equally difficult is policy maintenance and refactoring, e.g., removing useless or redundant rules. In this paper, we present a transcompiling pipeline that automatically tackles both problems: it can be used to port a policy into an equivalent one, when the target firewall language is different from the source one; when the two languages coincide, transcompiling supports policy maintenance and refactoring. Our transcompiler and its correctness are based on a formal intermediate firewall language that we endow with a formal semantics.

1 Introduction

Firewalls are one of the standard mechanisms for protecting computer networks. Configuring and maintaining them is very difficult also for expert system administrators since firewall policy languages are varied and usually rather complex, they account for low-level system and network details and support non trivial control flow constructs. Additional difficulties come from the way in which packets are processed by the network stack of the operating system and further issues are due to Network Address Translation (NAT), the mechanism for translating addresses and performing port redirection while packets traverse the firewall.

A configuration is typically composed of a large number of rules and it is often hard to figure out the overall firewall behavior. Also, firewall rules interact with each other, e.g., some shadow others making them redundant or preventing them to be triggered. Often administrators resort to policy *refactoring* to solve these issues and to obtain minimal and clean configurations. Software Defined Network (SDN) paradigm has recently been proposed for programming the network as a whole at a high level, making network and firewall configuration simpler and less

Work partially supported by CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks (www.filierasicura.it) funded by CISCO Systems Inc. and Leonardo SpA.

© The Author(s) 2018
L. Bauer and R. Küsters (Eds.): POST 2018, LNCS 10804, pp. 303–324, 2018.
https://doi.org/10.1007/978-3-319-89722-6_13

error prone. However, network administrators have still to face the *porting* of firewall configurations from a variety of legacy devices into this new paradigm.

Both policy refactoring and porting are demanding operations because they require system administrators to have a deep knowledge about the policy meaning, as well as the internals of the firewall systems and of their languages. To automatically solve these problems we propose here a transcompiling pipeline composed of the following stages:

1. decompile the policy in the source language into an intermediate language;
2. extract the meaning of the policy as a set of non overlapping declarative rules describing the accepted packets and their translations in logical terms;
3. compile the declarative rules into the target language.

Another key contribution of this paper is to formalize this pipeline and to prove that it preserves the meaning of the original policy (Theorems 1, 2 and 3). The core of our proposal is the intermediate language IFCL (Sect. 4), which offers all the typical features of firewall languages such as NAT, jumps, invocations to rulesets and stateful packet filtering. This language unveils the bipartite structure common to real firewall languages: the rulesets determining the destiny of packets and the control flow in which the rules are applied. The relevant aspects of IFCL are its independence from specific firewall systems and their languages, and its formal semantics (Sect. 5). Remarkably, stage 1 provides real languages, which usually have no formal semantics, with the one inherited by the decompilation to IFCL. In this way the meaning of a policy is formally defined, so allowing algorithmic manipulations that yield the rules of stage 2 (Sect. 6). These rules represent minimal configurations in a declarative way, covering all accepted packets and their transformations, with neither overlapping nor shadowing rules. These two stages are implemented in a tool appearing in a companion paper [1] and surveyed below, in the section on related work. The translation algorithm of stage 3 (Sect. 7) distributes the rules determined in the previous stage on the relevant points of the firewall where it decides the destiny of packets.

To show our transcompilation at work, we consider `iptables` [2] and `pf` [3] (Sect. 2), since they have very different packet processing schemes making policy porting hard. In particular, we apply the stages of our pipeline to port a policy from `iptables` to `pf` (Sect. 3). For brevity, we do not include an example of refactoring, which occurs when the source and the target languages coincide.

Related Work. Formal methods have been used to model firewalls and access control, e.g., [4–6]. Below we restrict our attention to language-based approaches.

Transcompilation is a well-established technique to address the problem of code refactoring, automatic parallelization and porting legacy code to a new programming language. Recently, this technique has been largely used in the field of web programming to implement high level languages into JavaScript, see e.g., [7,8]. We tackle transcompilation in the area of firewall languages to support porting and refactoring of policies.

To the best of our knowledge, the literature has no approaches to mechanically porting firewall policies, while it has some to refactoring. The proposal in [9] is similar to ours, in that it "cleans" rulesets, then analyzes them by an automatic tool. It uses a formal semantics of `iptables` (without NAT) and a semantics-preserving ruleset simplification. The tool FIREMAN [10] detects inconsistencies and inefficiencies of firewall policies (without NAT). The Margrave policy analyzer [11] analyzes IOS firewalls, and is extensible to other languages. However the analysis focuses on finding specific problems in policies rather then synthesizing a high-level policy specification. Another tool for discovering anomalies is Fang [12,13], which also synthesizes an abstract policy. Our approach differs from the above proposals mainly because *at the same time* it (i) is language-independent; (ii) defines a formal semantics of firewall behavior; (iii) gives a declarative, concise and neat representation of such a behavior; (iv) supports NAT; (v) generates policies in a target language.

Among the papers that formalize the semantics of firewall languages, we mention [14,15] that specify abstract filtering policies to be then compiled into the actual firewall systems. More generally, NetKat [16] proposes linguistic constructs for programming a network as a whole within the SDN paradigm. All these approaches propose their own high level language with a formal semantics, and then compile it to a specific target language (cf. our stage 3). Instead, IFCL intermediates between real source and target languages. It thus takes from real languages actions both for filtering/rewriting packets (notably NAT and MARK) and for controlling the inspection flow, widely used in practice.

Our companion paper [1] describes the design of an automated tool and its application to real cases. The tool implements the first two stages of our pipeline and supports system administrators in the verification of some properties of a given firewall policy. In particular, the user can ask queries to check implication, equivalence and difference of policies, and reachability among hosts. The tool uses the same syntax of Sect. 4 but only sketches how to obtain the declarative representation of a given policy, while here we fully formalize the process and prove it correct (Sect. 6.2). In detail, the present paper partially overlaps with [1] on Sect. 4, where the language is presented, and on Sect. 6.2, where the logical characterization is introduced. Besides the technical details and theorems, which support the semantics and the correctness of the whole approach missing in [1], here we also address the issue of compiling the declarative firewall representation to a target language, enabling transcompilation (cf. Sects. 3 and 7).

2 Background

Usually, system administrators classify networks into security domains. Through firewalls they monitor the traffic and enforce a predetermined set of access control policies (*packet filtering*), possibly performing some network address translation.

Firewalls are implemented either as proprietary, special devices, or as software tools running on general purpose operating systems. Independently of their actual implementations, they are usually characterized by a set of rules that determine which packets reach the different subnetworks and hosts, and how they are modified or translated. We briefly review `iptables` [2] and `pf` [3] that are two of the most used firewall tools in Linux and Unix.

iptables. It is the default in Linux distributions, and operates on top of Netfilter, the standard framework for packets processing of the Linux kernel [2]. This tool is based on the notions of *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 [17]: `PreRouting`, when the packet reaches the host; `Forward`, when the packet is routed through the host; `PostRouting`, right before the packet leaves the host; `Input`, when the packet is routed to the host; `Output`, when the packet is generated by the host. Moreover, users can define additional chains, besides the built-in ones.

Each rule specifies a condition and a target. If the packet matches the condition then it is processed according to the specified target. The most common targets are: ACCEPT and DROP, to accept and discard packets; DNAT/SNAT, to perform destination/source NAT; MARK to mark a packet with a numeric identifier which can be used in the conditions of other rules, even placed in different chains; RETURN, to stop examining the current chain and resume the processing of a previous chain. When the target is a user-defined chain, two "jumping" modes are available: *call* and *goto*. They differ 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.

pf. This is the standard firewall of OpenBSD [3] and is included in macOS since version 10.7. Similarly to `iptables`, each rule consists of a predicate which is used to select packets and an action that specifies how to process the packets satisfying the predicate. The most frequently used actions are `pass` and `block` to accept and reject packets, `rdr` and `nat` to perform destination and source NAT. Packet marking is supported also by `pf`: if a rule containing the `tag` keyword is applied, the packet is marked with the specified identifier and then processed according to the rule's action.

Differently from other firewalls, the action taken on a packet is determined by the *last matched rule*, unless otherwise specified. `pf` has a single ruleset that is inspected both when the packet enters and exits the host. When a packet enters the host, DNAT rules are examined first and filtering is performed after the address translation. Similarly when a packet leaves the host: first its source address is translated by the relevant SNAT rules, and then the resulting packet is possibly filtered. Notice also that packets belonging to established connections are accepted by default, thus bypassing the filters.

3 Porting a Policy: An Example

Consider the simple, yet realistic network of Fig. 1, where the IP addresses 10.0.0.0/8 identify the private LAN; 54.230.203.0/24 identify servers and production machines in the demilitarized zone DMZ that also hosts the HTTPS server

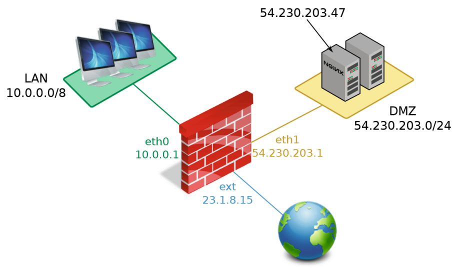

Fig. 1. A network.

Table 1. Declarative representation of the configuration in Fig. 2.

Src IP	Src Port	SNAT IP	SNAT Port	DNAT IP	DNAT Port	Dest IP	Dest Port	Prot	State
*	*	-	-	-	-	{54.230.203.47}	443	tcp	NEW
10.0.0.0/8	*	-	-	-	-	54.230.203.0/24	*	*	NEW
10.0.0.0/8	*	23.1.8.15	-	-	-	* \ { 10.0.0.0/8 54.230.203.0/24 127.0.0.0/8 }	80 443	tcp	NEW
*	*	*	*	*	*	*	*	*	ESTABLISHED

with address 54.230.203.47. The firewall has three interfaces: `eth0` connected to the LAN with IP 10.0.0.1, `eth1` connected to the DMZ with IP 54.230.203.1 and `ext` connected to the Internet with public IP 23.1.8.15.

The `iptables` configuration in Fig. 2 enforces the following policy on the traffic: (i) hosts from the Internet can connect to the HTTPS server; (ii) LAN hosts can freely connect to any host in the DMZ; (iii) LAN hosts can connect to the Internet over HTTP and HTTPS (with source NAT). Now, suppose the system administrator has to migrate the firewall configuration of Fig. 2 from `iptables` to `pf`. Performing this porting by hand is complex and error prone because the administrator has to write the `pf` configuration from scratch and test that it is equivalent to the original one. Furthermore, this requires a deep understanding of the policy meaning, as well as of both `iptables` and `pf` and of their configuration languages. We apply below the stages of our pipeline to solve this problem, guaranteeing by construction that the firewall semantics is preserved. The next sections detail the following intuitive description.

First we extract the meaning of the `iptables` configuration represented by a table, in our case Table 1 (stages 1 and 2). For instance, its second row says that the packets of a new connection with source address in the range 10.0.0.0/8 (i.e., from the LAN) can reach the hosts in the range 54.230.203.0/24 (the DMZ), with no NAT, regardless of the protocol and the port. The last row says that packets of an already established connection are always allowed. Note that each row in the table declaratively describes a set of packets accepted by the firewall, and their network translation. Actually, Table 1 is a clean, *refactored* policy automatically generated by the tool of [1]. Indeed, each row is disjoint from the others, so they need not to be ordered and none of the typical firewall anomalies arises, like

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

#(iii) Apply SNAT on connections from the LAN towards the Internet
-A POSTROUTING -s 10.0.0.0/8 -o ext -j MASQUERADE

COMMIT

*filter
# DROP policy in filtering chains
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]

# Allow established packets
-A FORWARD -m state --state ESTABLISHED -j ACCEPT
#(ii) LAN hosts can connect to DMZ
-A FORWARD -s 10.0.0.0/8 -d 54.230.203.0/24 -j ACCEPT
#(iii) LAN hosts can connect to the Internet over HTTP/HTTPS
-A FORWARD -s 10.0.0.0/8 -o ext -p tcp --dport 80 -j ACCEPT
-A FORWARD -s 10.0.0.0/8 -o ext -p tcp --dport 443 -j ACCEPT
#(i) Any host can connect to the HTTPS server in the DMZ
-A FORWARD -d 54.230.203.47 -p tcp --dport 443 -j ACCEPT

COMMIT
```

Fig. 2. Firewall configuration in `iptables`.

```
nat proto tcp from 10.0.0.0/8 to {!10.0.0.0/8, !54.230.203.0/24, !127.0.0.0/8}
  port {80, 443} tag T1 -> 23.1.8.15

block all
pass proto tcp from any to 54.230.203.47 port 443
pass from 10.0.0.0/8 to 54.230.203.0/24
pass tagged T1
```

Fig. 3. The policy in Fig. 2 ported in `pf`.

shadowing, rule overlapping, etc. According to stage 3, we compile the refactored policy in `pf`, in two steps. First, the rows are translated in a sequence of IFCL rules that are then compiled in `pf`. The result is in Fig. 3 and was computed with a proof-of-concept extension of [1] based on the theory presented in Sect. 7.

4 The Intermediate Firewall Configuration Language

We now present our intermediate firewall configuration language (IFCL). It is parametric w.r.t. the notion of state and the steps performed to elaborate packets. For generality, we do not detail the format of network packets. In the following we only use $sa(p)$ and $da(p)$ to denote the source and destination addresses of a given packet p; additionally, $tag(p)$ returns the tag m associated with p. An address a consists of an IP address $ip(a)$ and possibly a port $port(a)$. An *address range* n is a pair consisting of a set of IP addresses and a set of ports, denoted $IP(n){:}port(n)$. An 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., for ICMP packets we only check if the IP address is in the range.

Firewalls modify packets, e.g., through network address translations. We write $p[da \mapsto a]$ and $p[sa \mapsto a]$ to denote a packet identical to p, except for the destination address da and source address sa, which is equal to a, respectively. Similarly, $p[tag \mapsto m]$ denotes the packet with a modified tag m.

Here we consider *stateful* firewalls that keep track of the state s of network connections and 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. In this way, filtering and translation decisions are not only based on administrator-defined rules, but also on the information built by previous packets belonging to the same connection. We omit a precise definition of a state, but we assume that it tracks at least the source and destination ranges, NAT operations and the state of the connection, i.e., established or not. When receiving a packet p one may check whether it matches the state s or not. We left unspecified the match between a packet and the state because it depends on the actual shape of the state. When the match succeeds, we write $p \vdash_s \alpha$, where α describes the actions to be carried on p; otherwise we write $p \nvdash_s$.

A firewall rule is made of two parts: a predicate ϕ expressing criteria over packets, and an action t, called *target*, defining the "destiny" of matching packets. Here we consider a core set of actions included in most of the real firewalls. These actions not only determine whether or not a packet passes across the firewall, but also control the flow in which the rules are applied. They are the following:

ACCEPT	a packet passes
DROP	a packet is discarded
CALL(R)	invoke the ruleset R (see below)
GOTO(R)	jump to the ruleset R
RETURN	exit from the current ruleset
NAT(n_d, n_s)	network translation
MARK(m)	marking with tag m
CHECK-STATE(X)	examine the state

The targets CALL(_) and RETURN implement a procedure-like behavior; GOTO(_) is similar to unconditional jumps. In the NAT action n_d and n_s are address ranges used to translate the destination and source address of a packet, respectively; in the following we use the symbol $\star$ to denote an identity translation, e.g., $n : \star$ means that the address is translated according to n, whereas the port is kept unchanged. The MARK action marks a packet with a tag m. The argument $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ 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 $\leftrightarrow$ both. Formally:

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

A packet p matches a rule r with target t whenever ϕ holds.

Definition 2 (Rule match). *Given a rule $r = (\phi, t)$ we say that p matches r with target t, denoted $p \models_r t$, iff $\phi(p)$. We write $p \not\models_r$ when p does not match r.*

We can now define how a packet is processed given a possibly empty list of rules (denoted with ϵ), hereafter called *ruleset*. Similarly to real implementations of firewalls, we inspect the rules in the list, one after the other, until we find a matching one, which establishes the destiny (or target) of the packet. For sanity, we assume that no GOTO(R) and CALL(R) occur in the ruleset R, so avoiding self-loops. We also assume that rulesets may have a default target denoted by $t_d \in \{\text{ACCEPT}, \text{DROP}\}$, which accepts or drops according to the will of the system administrator.

Definition 3 (Ruleset match). *Given a ruleset $R = [r_1, \dots, r_n]$, we say that p matches the i-th rule with target t, denoted $p \models_R (t, i)$, iff*

$$i \leq n \,.\, r_i = (\phi, t) \wedge p \models_{r_i} t \wedge \forall j < i \,.\, p \not\models_{r_j}.$$

We also write $p \not\models_R$ if p matches no rules in R, formally if $\forall r \in R \,.\, p \not\models_r$. Afterwords, we will omit the index i when immaterial, and we simply write $p \models_R t$.

In our model we do not explicitly specify the steps performed by the kernel of the operating system to process a single packet passing through the host. We represent this algorithm through a *control diagram*, i.e., a graph where nodes represent different processing steps and the arcs determine the sequence of steps. The arcs are labeled with a predicate describing the requirements a packet has to meet in order to pass to the next processing phase. Therefore, they are not finite state auomata. We assume that control diagrams are deterministic, i.e., that every pair of arcs leaving the same node has mutually exclusive predicates. For generality, we let these predicates abstract, since they depend on the specific firewall.

Definition 4 (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;*
- *$A \subseteq Q \times \Psi \times Q$ is the set of arcs, such that whenever $(q, \psi, q'), (q, \psi', q'') \in A$ and $q' \neq q''$ then $\neg(\psi \wedge \psi')$;*
- *q_i,$q_f \in Q$ are special nodes denoting the start and the end of elaboration.*

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

Definition 5 (Transition function). *Let (Q, A, q_i, q_f) be a control diagram and let p be a packet. The transition function $\delta \colon Q \times Packet \mapsto Q$ is defined as*

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

We can now define a firewall in IFCL.

Definition 6 (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* correspondence *mapping from the nodes of $\mathcal{C}$ to the actual rulesets.*

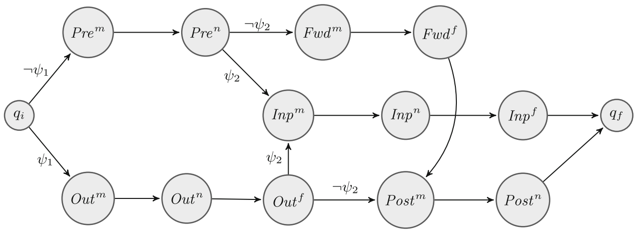

Fig. 4. The control diagram of `iptables`

4.1 Decompiling Two Real Languages into IFCL

Here we encode the two *de facto* standard Unix firewalls `iptables` and `pf` as triples $(\mathcal{C}, \rho, c)$ of our framework (stage 1). An immediate fallout is a formal semantics for both `iptables` and `pf` defined in terms of that of IFCL (see Sect. 5).

Modelling `iptables`. Let $\mathcal{L}$ be the set of local addresses of a host; and let ψ_1 and ψ_2 predicates over packets defined as follows:

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

Figure 4 shows the control diagram $\mathcal{C}$ of `iptables`, where unlabeled arcs carry the label "*true*." It also implicitly defines the transition function according to Definition 5. In `iptables` there are twelve built-in chains, each of which correspond to a single ruleset. So we can define the set $\rho_p \subseteq \rho$ of primitive rulesets as the one made of R^{man}_{INP}, R^{nat}_{INP}, R^{fil}_{INP}, R^{man}_{OUT}, R^{nat}_{OUT}, R^{fil}_{OUT}, R^{man}_{PRE}, R^{nat}_{PRE}, R^{man}_{FOR}, R^{fil}_{FOR}, R^{man}_{POST} and R^{nat}_{POST}, where the superscript represents the chain name and the subscript the table name. Note that the set $\rho \setminus \rho_p$ contains the user-defined chains.

The mapping function $c \colon Q \mapsto \rho$ is defined as follows:

$$\begin{array}{ccc}
c(q_i) = R & c(q_f) = R & c(Pre^m) = R^{man}_{\mathrm{PRE}} \\
c(Pre^n) = R^{nat}_{\mathrm{PRE}} & c(Inp^m) = R^{man}_{\mathrm{INP}} & c(Fwd^f) = R^{fil}_{\mathrm{FOR}} \\
c(Inp^n) = R^{nat}_{\mathrm{INP}} & c(Inp^f) = R^{fil}_{\mathrm{INP}} & c(Out^m) = R^{man}_{\mathrm{OUT}} \\
c(Out^n) = R^{nat}_{\mathrm{OUT}} & c(Out^f) = R^{fil}_{\mathrm{OUT}} & c(Fwd^m) = R^{man}_{\mathrm{FOR}} \\
c(Fwd^f) = R^{fil}_{\mathrm{FOR}} & c(Post^m) = R^{man}_{\mathrm{POST}} & c(Post^n) = R^{nat}_{\mathrm{POST}}
\end{array}$$

where R is an empty ruleset with ACCEPT as default policy.

Finally, note that the action CALL(_) implements the built in target JUMP(_).

Modelling `pf`. Differently from `iptables`, `pf` has a single ruleset and the rule applied to a packet is the last one matched, apart from the case of the so-called `quick` rules: as soon as one of these rules matches the packet, its action is applied and the remaining part of the ruleset is skipped.

Figure 5 shows the control diagram $\mathcal{C}_{pf}$ for `pf` that also defines the transition function. The nodes Inp^n and Inp^f represent the procedure executed when an

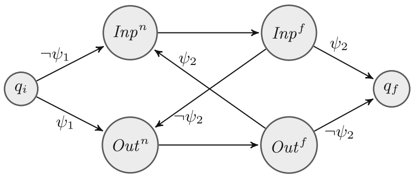

Fig. 5. The control diagram of `pf`

IP packet reaches the host from the net. Dually, Out^n and Out^f are for when the packet leaves the host. The predicates ψ_1 and ψ_2 are those defined for `iptables`. Given the `pf` ruleset $R_{\texttt{pf}}$ we include the following rulesets in ρ_{pf}:

– R_{dnat} contains the rule $(state == 1, \text{CHECK-STATE}(\rightarrow))$ as the first one, followed by all the rules `rdr` of $R_{\texttt{pf}}$;
– R_{snat} contains the rule $(state == 1, \text{CHECK-STATE}(\leftarrow))$ as the first one, followed by all the rules `nat` of $R_{\texttt{pf}}$;
– R_{finp} contains the rule $(state == 1, \text{ACCEPT})$ followed by all the `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `out`, and finally the rule $(true, \text{GOTO}(R_{finpr}))$;
– R_{finpr} contains all the no `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `out`, in reverse order;
– R_{fout} contains the rule $(state == 1, \text{ACCEPT})$ followed by all the `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `in`, and $(true, \text{GOTO}(R_{foutr}))$ as last rule;
– R_{foutr} includes all the no `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `in` in reverse order.

Given the ruleset R with the only rule for ACCEPT as default policy, the mapping function c_{pf} is defined as follows:

$$\begin{array}{lll} c_{pf}(q_i) = R & c_{pf}(Inp^n) = R_{dnat} & c_{pf}(Out^n) = R_{snat} \\ c_{pf}(q_f) = R & c_{pf}(Inp^f) = R_{finp} & c_{pf}(Out^f) = R_{fout} \end{array}$$

5 Formal Semantics

Now, we formally define the semantics of a firewall through two transition systems operating in a master-slave fashion. The master has a labeled transition relation of the form $s \xrightarrow{p,p'} s'$. The intuition is that the state s of a firewall changes to s' when a new packet p reaches the host and becomes p'.

The configurations of the slave transition system are triples (q, s, p) where: (i) $q \in Q$ is a control diagram node; (ii) s is the state of the firewall; (iv) p is the packet. A transition $(q, s, p) \rightarrow (q', s, p')$ describes how a firewall in a state s deals with a packet p and possibly transforms it in p', according to the control diagram $\mathcal{C}$. Recall that the state records established connections and other kinds of information that are updated after the transition.

In the slave transition relation, we use the following predicate, which describes an algorithm that runs a ruleset R on a packet p in the state s

$$p, s \models_R^S (t, p')$$

This predicate searches for a rule in R matching the packet p through $p \models_R (t, i)$. If it finds a match with target t, t is applied to p to obtain a new packet p'.

Recall that actions CALL(R), RETURN and GOTO(R) are similar to procedure calls, returns and jumps in imperative programming languages. To correctly deal with them, our predicate $p, s \models_R^S (t, p')$ uses a stack S to implement a behavior similar to the one of procedure calls. We will denote with ϵ the empty stack and with $\cdot$ the concatenation of elements on the stack. This stack is also used to detect and prevent loops in ruleset invocation, as it is the case in real firewalls.

In the stack S we overline a ruleset R to indicate that it was pushed by a GOTO(_) action and it has to be skipped when returning. Indeed, we use the following *pop** function in the semantics of the RETURN action:

$$pop^*(\epsilon) = \epsilon \qquad pop^*(R \cdot S) = (R, S) \qquad pop^*(\overline{R} \cdot S) = pop^*(S)$$

In case there is a non-overlined ruleset on the top of S, it behaves as a standard pop operation; otherwise it extracts the first non-overlined ruleset. When S is empty, we assume that *pop** returns ϵ to signal the error.

Furthermore, in the definition of $p, s \models_R^S (t, p')$ we use the notation R_k to indicate the ruleset $[r_k, ..., r_n]$ $(k \in [1, n])$ resulting from dropping the first $k-1$ rules from the given ruleset $R = [r_1, ..., r_n]$.

We also assume the function *establ* that, taken an action α from the state, a packet p and the fields $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ to rewrite, returns a possibly changed packet p', e.g., in case of an established connection. Also this function depends on the specific firewall we are modeling, and so it is left unspecified.

Finally, we assume as given a function $nat(p, s, d_n, s_n)$ that returns the packet p translated under the corresponding NAT operation in the state s. The argument d_n is used to modify the destination range of p, i.e., destination NAT (DNAT), while s_n is used to modify the source range, i.e., source NAT (DNAT). Recall that a range of the form $\star : \star$ is interpreted as the identity translation, whereas one of the form $a : \star$ modifies only the address. Also this function is left abstract.

Table 2 shows the rules defining $p, s \models_R^S (t, p')$. The first inference rule deals with the case when the packet p matches a rule that says ACCEPT or DROP; in this case the ruleset execution stops returning the found action and leaving p unmodified. When a packet p matches a rule with action CHECK-STATE, we query the state s: if p belongs to an established connection, we return ACCEPT and a p' obtained rewriting p. If p belongs to no existent connection the packet is matched against the remaining rules in the ruleset. When a packet p matches a NAT rule, we return ACCEPT and the packet resulting by the invocation of the function *nat*. There are two cases if a packet p matches a GOTO(_). If the ruleset R' is not already in the stack, we push the current ruleset R onto the stack overlined to record that this ruleset dictated a GOTO(_). Otherwise, if R' is in the stack, we detect a loop and discard p. The case when a packet p matches a rule with action CALL(_) is similar,

Table 2. The predicate $p, s \models_R^S (t, p')$.

$$(1)\ \frac{p \models_R (t, i) \quad t \in \{\texttt{ACCEPT}, \texttt{DROP}\}}{p, s \models_R^S (t, p)} \qquad (2)\ \frac{p \models_R (\texttt{CHECK-STATE}(X), i) \quad p \vdash_s \alpha \quad p' = establ(\alpha, X, p)}{p, s \models_R^S (\texttt{ACCEPT}, p')}$$

$$(3)\ \frac{p \models_R (\texttt{CHECK-STATE}(X), i) \quad p \not\vdash_s \quad p, s \models_{R_{i+1}}^S (t, p')}{p, s \models_R^S (t, p')} \qquad (4)\ \frac{p \models_R (\texttt{NAT}(d_n, s_n), i)}{p, s \models_R^S (\texttt{ACCEPT}, nat(p, s, d_n, s_n))}$$

$$(5)\ \frac{p \models_R (\texttt{GOTO(R')}, i) \quad R' \notin S \quad p, s \models_{R'}^{\overline{R} \cdot S} (t, p')}{p, s \models_R^S (t, p')} \qquad (6)\ \frac{p \models_R (\texttt{GOTO(R')}, i) \quad R' \in S}{p, s \models_R^S (\texttt{DROP}, p)}$$

$$(7)\ \frac{p \models_R (\texttt{CALL(R')}, i) \quad R' \notin S \quad p, s \models_{R'}^{R_{i+1} \cdot S} (t, p')}{p, s \models_R^S (t, p')} \qquad (8)\ \frac{p \models_R (\texttt{CALL(R')}, i) \quad R' \in S}{p, s \models_R^S (\texttt{DROP}, p)}$$

$$(9)\ \frac{p \models_R (\texttt{RETURN}, i) \quad pop^*(S) = (R', S') \quad p, s \models_{R'}^{S'} (t, p')}{p, s \models_R^S (t, p')} \qquad (10)\ \frac{p \models_R (\texttt{RETURN}, i) \quad pop^*(S) = \epsilon}{p, s \models_R^S (t_d, p)}$$

$$(11)\ \frac{p \not\models_R \quad S \neq \epsilon \quad pop^*(S) = (R', S') \quad p, s \models_{R'}^{S'} (t, p')}{p, s \models_R^S (t, p')} \qquad (12)\ \frac{p \not\models_R \quad (S = \epsilon \ \vee \ pop^*(S) = \epsilon)}{p, s \models_R^S (t_d, p)}$$

$$(13)\ \frac{p \models_R (\texttt{MARK}(m), i) \qquad p[tag \mapsto m], s \models_{R_{i+1}}^S (t, p')}{p, s \models_R^S (t, p')}$$

except that the ruleset pushed on the stack is not overlined. When a packet p matches a rule with action RETURN, we pop the stack and match p against the top of the stack. Finally, when no rule matches, an implicit return occurs: we continue from the top of the stack, if non empty. The MARK rule simply changes the tag of the matching packet to the value m. If none of the above applies, we return the default action t_d of the current ruleset.

We can now define the slave transition relation as follows.

$$\frac{c(q) = R \quad p, s \models_R^\epsilon (\texttt{ACCEPT}, p') \quad \delta(q, p') = q'}{(q, s, p) \rightarrow (q', s, p')}$$

The rule describes how we process the packet p when the firewall is in the elaboration step represented by the node q with a state s. We match p against the ruleset R associated with q and if p is accepted as p', we continue considering the next step of the firewall execution represented by the node q'.

Finally, we define the master transition relation that transforms states and packets as follows (as usual, below $\rightarrow^+$ stands for the transitive closure of $\rightarrow$):

$$\frac{(q_i, s, p) \rightarrow^+ (q_f, s, p')}{s \xrightarrow{p,p'} s \uplus (p, p')}$$

This rule says that when the firewall is in the state s and receives a packet p, it elaborates p starting from the initial node q_i of its control diagram. If this elaboration succeeds, i.e., it reaches the node q_f accepts p as p', we update the state s by storing information about p, its translation p' and the connection they belong to, through the function $\uplus$, left unspecified for the sake of generality.

Example 1. Suppose to have the user-defined chains below

Chain C_B	Chain u_1	Chain u_2
$(\phi_1, \texttt{DROP})$	$(\phi_{11}, \texttt{ACCEPT})$	$(\phi_{21}, \texttt{ACCEPT})$
$(\phi_2, \texttt{CALL}(u_1))$	$(\phi_{12}, \texttt{CALL}(u_2))$	$(\phi_{22}, \texttt{RETURN})$
$(\phi_3, \texttt{ACCEPT})$	$(\phi_{13}, \texttt{DROP})$	$(\phi_{23}, \texttt{DROP})$

and that the condition $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ holds for a packet p. Then, the semantic rules (a), (b) and (c) are applied in order:

$$(a)\ \frac{p \models_{C_B} (\texttt{CALL(u}_1\texttt{)}, i) \quad u_1 \notin S \quad p, s \models_{u_1}^{C_{B_3}\cdot\epsilon} (\texttt{ACCEPT}, p)}{p, s \models_{C_B}^{\epsilon} (\texttt{ACCEPT}, p)}$$

$$(b)\ \frac{p \models_{u_1} (\texttt{ACCEPT}, 1)}{p, s \models_{u_1}^{C_{B_3}\cdot\epsilon} (\texttt{ACCEPT}, p)} \qquad (c)\ \frac{c(q) = C_B \quad p, s \models_{C_B}^{\epsilon} (\texttt{ACCEPT}, p) \quad \delta(q, p) = q'}{(q, s, p) \to (q', s, p)}$$

6 From Operational to Declarative Descriptions

We now extract the meaning of a firewall written in our intermediate language by transforming it in a declarative, logical presentation that preserves the semantics (stage 2). This transformation is done in three steps: (i) generate an unfolded firewall with a single ruleset for each node of the control diagram; (ii)transform the unfolded firewall in a first-order formula; (iii)determine a model for the obtained formula, through a SAT solver (the procedure for this step is described in [1] and is omitted here). The correctness of stage 2 follows from Theorem 1, which guarantees that the unfolded firewall is semantically equivalent to the original one, and from Theorem 2, which ensures that the derived formula characterizes exactly the accepted packets and their translations.

6.1 Unfolding Chains

Our intermediate language can deal with involved control flows, by using the targets $\texttt{GOTO(\_)}$, $\texttt{CALL(\_)}$ and $\texttt{RETURN}$ (see Example 1). The following unfolding operation $[\![\_]\!]$ rewrites a ruleset into an equivalent one with no control flow rules.

Hereafter, let $r; R$ be a non empty ruleset consisting of a rule r followed by a possibly empty ruleset R; and let $R_1 @ R_2$ be the concatenation of R_1 and R_2.

The unfolding of a ruleset R is defined as follows:

$$\begin{aligned}
[\![R]\!] &= [\![R]\!]_{\{R\}}^{true} \\
[\![\epsilon]\!]_I^f &= \epsilon \\
[\![(\phi, t); R]\!]_I^f &= (f \wedge \phi, t); [\![R]\!]_I^f \quad \text{if } t \notin \{\texttt{GOTO(R')}, \texttt{CALL(R')}, \texttt{RETURN}\} \\
[\![(\phi, \texttt{RETURN}); R]\!]_I^f &= [\![R]\!]_I^{f \wedge \neg\phi} \\
[\![(\phi, \texttt{CALL(R')}); R]\!]_I^f &= \begin{cases} [\![R']\!]_{I \cup \{R'\}}^{f \wedge \phi} @ [\![R]\!]_I^f & \text{if } R' \notin I \\ (f \wedge \phi, \texttt{DROP}); [\![R]\!]_I^f & \text{otherwise} \end{cases}
\end{aligned}$$

$$[\![(\phi, \texttt{GOTO(R')}); R]\!]_I^f = \begin{cases} [\![R']\!]_{I \cup \{R'\}}^{f \wedge \phi} @ [\![R]\!]_I^{f \wedge \neg \phi} & \text{if } R' \notin I \\ (f \wedge \phi, \texttt{DROP}); [\![R]\!]_I^{f \wedge \neg \phi} & \text{otherwise} \end{cases}$$

The auxiliary procedure $[\![R]\!]_I^f$ recursively inspects the ruleset R. The formula f accumulates conjuncts of the predicate ϕ; the set I records the rulesets traversed by the procedure and helps detecting loops. If a rule does not affect control flow, we just substitute the conjunction $f \wedge \phi$ for ϕ, and continue to analyze the rest of the ruleset with the recursive call $[\![R]\!]_I^f$.

In the case of a return rule $(\phi, \texttt{RETURN})$ we generate no new rule, and we continue to recursively analyze the rest of the ruleset, by updating f with the negation of ϕ. For the rule $(\phi, \texttt{CALL(R')})$ we have two cases: if the callee ruleset R' is not in I, we replace the rule with the unfolding of R' with $f \wedge \phi$ as predicate, and append $\{R'\}$ to the traversed rulesets. If R' is already in I, i.e., we have a loop, we replace the rule with a $\texttt{DROP}$, with $f \wedge \phi$ as predicate. In both cases, we continue unfolding the rest of the ruleset. We deal with the rule $(\phi, \texttt{GOTO(R')})$ as the previous one, except that the rest of the ruleset has $f \wedge \neg\phi$ as predicate.

Example 2. Back to Example 1, unfolding the chain C_B gives the following rules:

$$\begin{aligned} [\![C_B]\!] = {} & (\phi_1, \texttt{DROP}); \\ & (\phi_2 \wedge \phi_{11}, \texttt{ACCEPT}); \\ & (\phi_2 \wedge \phi_{12} \wedge \phi_{21}, \texttt{ACCEPT}); \\ & (\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}, \texttt{DROP}); \\ & (\phi_2 \wedge \phi_{13}, \texttt{DROP}); \\ & (\phi_3, \texttt{ACCEPT}); \\ & \epsilon \end{aligned}$$

We just illustrate the first three steps:

$$\begin{aligned} [\![C_B]\!] = & [\![(\phi_1, \texttt{DROP}); C_{B2}]\!]_{\{C_B\}}^{true} = (\phi_1, \texttt{DROP}); [\![(\phi_2, \texttt{CALL}(u_1)); C_{B3}]\!]_{\{C_B\}}^{true} \\ = & [\![u_1]\!]_{\{C_B\} \cup \{u_1\}}^{true \wedge \phi_2} @ [\![C_{B3}]\!]_{\{C_B\}}^{true} \end{aligned}$$

Note that our transformation does not change the set of accepted packets, e.g., all packets satisfying $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ are still accepted by the unfolded ruleset.

An unfolded firewall is obtained by repeatedly rewriting the rulesets associated with the nodes of its control diagram, using the procedure above. Formally,

Definition 7 (Unfolded firewall). *Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$, its unfolded version $[\![\mathcal{F}]\!]$ is $(\mathcal{C}, \rho', c')$ where $\forall q \in \mathcal{C}.\, c'(q) = [\![c(q)]\!]$ and $\rho' = \{[\![c(q)]\!] \mid q \in \mathcal{C}\}$.*

We now prove that a firewall $\mathcal{F}$ and its unfolded version $[\![\mathcal{F}]\!]$ are semantically equivalent, i.e., they perform the same action over a given packet p in a state s, and reach the same state s'. Formally, the following theorem holds:

Table 3. Translation of rulesets into logical predicates.

$P_\epsilon(p,\tilde{p}) = dp(R) \wedge p = \tilde{p}$	
$P_{r;R}(p,\tilde{p}) = (\phi(p) \wedge p = \tilde{p}) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p}))$	if $r = (\phi, \texttt{ACCEPT})$
$P_{r;R}(p,\tilde{p}) = \neg\phi(p) \wedge P_R(p,\tilde{p})$	if $r = (\phi, \texttt{DROP})$
$P_{r;R}(p,\tilde{p}) = (\phi(p) \wedge \tilde{p} \in tr(p, d_n, s_n, \leftrightarrow)) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p}))$	if $r = (\phi, \texttt{NAT}(d_n, s_n))$
$P_{r;R}(p,\tilde{p}) = (\phi(p) \wedge \tilde{p} \in tr(p, *{:}*, *{:}*, X) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p}))$	if $r = (\phi, \texttt{CHECK-STATE}(X))$
$P_{r;R}(p,\tilde{p}) = (\phi(p) \wedge P_R(p[tag \mapsto m],\tilde{p})) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p}))$	if $r = (\phi, \texttt{MARK}(m))$

Theorem 1 (Correctness of unfolding). *Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall and $[\![\mathcal{F}]\!]$ its unfolding. Let $s \xrightarrow{p,p'}_X s'$ be a step of the master transition system performed by the firewall $X \in \{\mathcal{F}, [\![\mathcal{F}]\!]\}$. Then, it holds*

$$s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{[\![\mathcal{F}]\!]} s'.$$

6.2 Logical Characterization of Firewalls

We construct a logical predicate that characterizes all the packets accepted by an unfolded ruleset, together with the relevant translations.

To deal with NAT, we define an auxiliary function tr that computes the set of packets resulting from all possible translations of a given packet p. The parameter $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ specifies if the translation applies to source, destination or both addresses, respectively, similarly to CHECK-STATE(X).

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

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

Given an unfolded ruleset R, we build the predicate $P_R(p, \tilde{p})$ that holds when the packet p is accepted as $\tilde{p}$ by R. Its definition is in Table 3 that induces on the rules in R. Intuitively, the empty ruleset applies the default policy $dp(R)$ and does not transform the packet, encoded by the constraint $p = \tilde{p}$. The rule $(\phi, \texttt{ACCEPT})$ considers two cases: when $\phi(p)$ holds and the packet is accepted as it is; when instead $\neg\phi(p)$ holds, p is accepted as $\tilde{p}$ only if the continuation R accepts it. The rule $(\phi, \texttt{DROP})$ accepts p only if the continuation does and $\phi(p)$ does not hold. The rule $(\phi, \texttt{NAT}(d_n, s_n))$ is like an $(\phi, \texttt{ACCEPT})$: the difference is when $\phi(p)$ holds, and it gives $\tilde{p}$ by applying to p the NAT translations $tr(p, d_n, s_n, \leftrightarrow)$. Finally, $(\phi, \texttt{CHECK-STATE}(X))$ is like a NAT that applies all possible translations of kind X (written as $tr(p, *{:}*, *{:}*, X)$). The idea is that, since we abstract away from the actual established connections, we over-approximate the state by considering any possible translations. At run-time, only the connections corresponding to the actual state will be possible. The rule $(\phi, \texttt{MARK}(m))$ is like a NAT, but when $\phi(p)$ holds it requires that the continuation accepts p tagged by m as $\tilde{p}$.

Example 3. The predicate of the unfolded ruleset in Example 2 when $dp(C_B) = F$ is

$$\begin{aligned}
P_{[\![C_B]\!]}(p,\tilde{p}) = \neg\phi_1 \wedge (&\\
&(\phi_2 \wedge \phi_{11} \wedge p = \tilde{p}) \vee (\neg(\phi_2 \wedge \phi_{11}) \wedge (\\
&(\phi_2 \wedge \phi_{12} \wedge \phi_{21} \wedge p = \tilde{p}) \vee (\neg(\phi_2 \wedge \phi_{12} \wedge \phi_{21}) \wedge (\\
&\neg(\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}) \wedge (\\
&\neg(\phi_2 \wedge \phi_{13}) \wedge (\\
&(\phi_3 \wedge p = \tilde{p}) \vee (\neg\phi_3 \wedge (\\
&F \wedge p = \tilde{p})))))))))
\end{aligned}$$

Note that if $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ holds then the formula trivially holds and therefore the formula accepts the packet as the semantics does.

As a further example, consider the case in which $\phi_2, \phi_{12}, \phi_{22}, \phi_{23}, \phi_3$ hold for a packet p, while all the other ϕ's does not. Then, p is accepted as it is: the rule $(\phi_{23}, \texttt{DROP})$ is not evaluated since ϕ_{22} holds and the RETURN is performed (cf. Example 1). Indeed, the predicate $P_{[\![C_B]\!]}(p,p)$ evaluates to:

$$T \wedge (F \vee (T \wedge (F \vee (T \wedge (T \wedge (T \wedge (T \vee (F \wedge F)))))))) = T$$

Instead, if ϕ_{13} holds too, the packet is rejected as expected:

$$T \wedge (F \vee (T \wedge (F \vee (T \wedge (T \wedge (F \wedge (T \vee (F \wedge F)))))))) = F$$

The predicate in Table 3 is semantically correct, because if a packet p is accepted by a ruleset R as p', then $P_R(p,p')$ holds, and vice versa. Formally,

Lemma 1. *Given a ruleset R we have that*

1. $\forall p, s.\ p, s \models^{\epsilon}_{R} (\textit{ACCEPT}, p') \implies P_R(p,p')$*; and*
2. $\forall p, p'.\ P_R(p,p') \implies \exists s. p, s \models^{\epsilon}_{R} (\textit{ACCEPT}, p')$

We eventually define the predicate associated with a whole firewall as follows.

Definition 8. *Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall with control diagram $\mathcal{C} = (Q, A, q_i, q_f)$. The predicate associated with $\mathcal{F}$ is defined as*

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

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

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

Intuitively, in the final node q_f we accept p as it is. In all the other nodes, p is accepted as $\tilde{p}$ if and only if there is a path starting from p in the control diagram that obtains $\tilde{p}$ through intermediate transformations. More precisely, we look for an intermediate packet p', provided that (*i*) p is accepted as p' by the ruleset $c(q)$ of node q; (*ii*) p' satisfies one of the predicates ψ labeling the branches of the control diagram; and (*iii*) p' is accepted as $\tilde{p}$ in the reached node q'. Note that we ignore paths with loops, because firewalls have mechanisms to detect and discard a packet when its elaboration loops. To this aim, our predicate uses the set I for recording the nodes already traversed.

We conclude this section by establishing the correspondence between the logical formulation and the operational semantics of a firewall. Formally, $\mathcal{F}$ accepts the packet p as $\tilde{p}$ if the predicate $\mathcal{P}_{\mathcal{F}}(p, \tilde{p})$ is satisfied, and vice versa:

Theorem 2 (Correctness of the logical characterization). *Given a firewall* $\mathcal{F} = (\mathcal{C}, \rho, c)$ *and its corresponding predicate* $\mathcal{P}_{\mathcal{F}}$ *we have that*

1. $s \xrightarrow{p,p'} s \uplus (p, p') \implies \mathcal{P}_{\mathcal{F}}(p, p')$
2. $\forall p, p'.\ \mathcal{P}_{\mathcal{F}}(p, p') \implies \exists s. s \xrightarrow{p,p'} s \uplus (p, p')$

Recall that the logical characterization abstracts away the notion of state, and thus $\mathcal{P}_{\mathcal{F}}(p, p')$ holds if and only if there exists a state s in which p is accepted as p'. In particular, if the predicate holds for a packet p that belongs to an established connection, p will be accepted only if the relevant state is reached at runtime. This is the usual interpretation of firewall rules for established connections.

7 Policy Generation

The declarative specification extracted from a firewall policy (cf. Table 1) can be mapped to a firewall $\mathcal{F}_S$ whose control diagram has just one node. The ruleset R_S associated with this node only contains ACCEPT and NAT rules, each corresponding to a line of the declarative specification. In Sect. 3 we showed that each line is disjoint from the others. Hence, the ordering of rules in R_S is irrelevant.

Here we compile $\mathcal{F}_S$ into an equivalent firewall $\mathcal{F}_C$. First, we introduce an algorithm that computes the basic rulesets of $\mathcal{F}_C$. Then, we map these rulesets to the nodes of the control diagram of a real system. Finally, we prove the correctness of the compilation.

For simplicity, we produce a firewall that automatically accepts all the packets that belong to established connections with the appropriate translations. We claim this is not a limitation, since it is the default behavior of some real firewall systems (e.g., `pf`) and it is quite odd to drop packets, once the initial connection has been established. Moreover, this is consistent with the over-approximation on the firewall state done in Sect. 6.2.

Algorithm 1. Generation of the rulesets R_{dnat}, R_{fil}, R_{snat}, R_{mark} from R_S

```
1: $R_{dnat} = R_{fil} = R_{snat} = R_{mark} = \epsilon$
2: for $r$ in $R_S$ do
3:     if $r = (\phi, \texttt{ACCEPT})$ then
4:         add $r$ to $R_{fil}$
5:     else if $r = (\phi, \texttt{NAT}(d_n, s_n))$ then
6:         generate fresh tag $m$
7:         add $(\phi \wedge tag(p) = \bullet, \texttt{MARK}(m))$ to $R_{mark}$
8:         add $(tag(p) = m, \texttt{NAT}(d_n, \star))$ to $R_{dnat}$
9:         add $(tag(p) = m, \texttt{NAT}(\star, s_n))$ to $R_{snat}$
10:    end if
11: end for
12: add $(tag(p) \neq \bullet, \texttt{ACCEPT})$ and $(true, \texttt{DROP})$ to $R_{fil}$
13: prepend $R_{mark}$ to $R_{dnat}$, $R_{fil}$ and $R_{snat}$
```

7.1 Compiling a Firewall Specification

Our algorithm takes as input the ruleset R_S derived from a synthesized specification and yields the rulesets R_{fil}, R_{dnat}, R_{snat} (with default ACCEPT policy) containing filtering, DNAT and SNAT rules. This separation reflects that all the real systems we have analyzed impose constraints on where NAT rules can be placed, e.g., in `iptables`, DNAT is allowed only in rulesets R^{nat}_{PRE} and R^{nat}_{OUT}, while SNAT only in R^{nat}_{INP} and R^{nat}_{POST}.

Intuitively, Algorithm 1 produces rules that assign different tags to packets that must be processed by different NAT rules (lines 6 and 7). Each NAT rule is split in a DNAT (line 8) and an SNAT (line 9), where the predicate ϕ becomes a check on the tag of the packet. Filtering rules are left unchanged (line 4). Packets subject to NAT are accepted in R_{fil} while the others are dropped (line 12). We prepend R_{mark} to all rulesets making sure that packets are always marked, independently of which ruleset will be processed first (line 13). We use $\bullet$ to denote the empty tag used when a packet has never been tagged.

Recall that the @ operator combines rulesets in sequence. Note that R_{fil} drops by default and shadows any ruleset appended to it. In practice, the only interesting rulesets are $\mathcal{R} = \{R_\epsilon, R_{fil}, R_{dnat}, R_{snat}, R_{dnat} @ R_{fil}, R_{snat} @ R_{fil}\}$ where R_ϵ is the empty ruleset with default ACCEPT policy. Since here we do not discuss `ipfw` [18] and other firewalls with a minimal control diagram, we neither use $R_{dnat} @ R_{fil}$ nor $R_{snat} @ R_{fil}$.

We now introduce the notion of *compiled firewall.*

Definition 9 (Compiled firewall). *A firewall $\mathcal{F}_C = (\mathcal{C}, \rho, c)$ with control diagram $\mathcal{C} = (Q, A, q_i, q_f)$ is a* compiled firewall *if*

- $c(q_i) = c(q_f) = R_\epsilon$
- $c(q) \in \mathcal{R}$ *for every* $q \in Q \setminus \{q_i, q_f\}$
- *every path π from q_i to q_f in the control diagram $\mathcal{C}$ traverses a node q such that $c(q) \in \{R_{fil}, R_{dnat} @ R_{fil}, R_{snat} @ R_{fil}\}$*

Intuitively, the above definition requires that only rulesets in $\mathcal{R}$ are associated with the nodes in the control diagram and that all paths pass at least one through a node with the filtering ruleset.

Example 4. Now we map the rulesets to the nodes of the control diagrams of the real systems presented in Sect. 4.1. For `iptables` we have:

$$\begin{array}{llll} c(Pre^n) = R_{dnat} & c(Out^n) = R_{dnat} & c(Inp^n) = R_{snat} & c(Post^n) = R_{snat} \\ c(Fwd^f) = R_{fil} & c(Inp^f) = R_{fil} & c(Out^f) = R_{fil} & \end{array}$$

while the remaining nodes get the empty ruleset R_ϵ. For `pf` we have:

$$c(Inp^n) = R_{dnat} \qquad c(Out^n) = R_{snat} \qquad c(Inp^f) = R_{fil} \qquad c(Out^f) = R_{fil}$$

7.2 Correctness of the Compiled Firewall

We start by showing that a compiled firewall $\mathcal{F}_C$ accepts the same packets as $\mathcal{F}_S$, possibly with a different translation.

Lemma 2. *Let $\mathcal{F}_C$ be a* compiled firewall*. Given a packet p, we have that*

$$\exists p'. \mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \exists p''. \mathcal{P}_{\mathcal{F}_C}(p, p'').$$

Let be $\mathcal{T} = \{id, dnat, snat, nat\}$ the set of translations possibly applied to a packet while it traverses a firewall. The first, id, represents the identity, $dnat$ and $snat$ are for DNAT and SNAT, while nat represents both DNAT and SNAT. Also, let $(\mathcal{T}, <)$ be the partial order such that $id < dnat$, $id < snat$, $dnat < nat$ and $snat < nat$. Finally, given a packet p and a firewall $\mathcal{F}$, let $\pi_{\mathcal{F}}(p)$ be the path in the control diagram of $\mathcal{F}$ along which p is processed. Note that there exists a unique path for each packet because the control diagram is deterministic.

The following function computes the *translation capability* of a path π, i.e., which translations can be performed on packets processed along π.

Definition 10 (Translation capability). *Let $\pi = \langle q_1, \ldots, q_n \rangle$ be a path on the control diagram of a compiled firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$. The* translation capability *of π is*

$$tc(\pi) = \text{lub}\left(\bigcup_{q_i \in \pi} \gamma(c(q_i))\right)$$

where lub *is the least upper bound of a set $T \subseteq \mathcal{T}$ w.r.t. $<$ and γ is defined as*

$$\begin{aligned} \gamma(R) &= \{id\} \text{ for } R \in \{R_\epsilon, R_{fil}\} \\ \gamma(R_t) &= \{t\} \text{ for } t \in \{dnat, snat\} \\ \gamma(R_1 \mathbin{@} R_2) &= \gamma(R_1) \cup \gamma(R_2) \end{aligned}$$

We write $p \approx p'$ to denote that $p' = p[tag \mapsto m]$ for some marking m. In addition, let t_β be a function that, given a packet p and its translation p', computes a packet p'' where only the translation $\beta \in \mathcal{T}$ is applied to p, defined as:

$$t_{id}(p, p') = p \qquad t_{dnat}(p, p') = p[da \mapsto da(p')]$$
$$t_{nat}(p, p') = p' \qquad t_{snat}(p, p') = p[sa \mapsto sa(p')]$$

The following theorem describes the relationship between a compiled firewall $\mathcal{F}_C$ and the firewall $\mathcal{F}_S$. Intuitively, $\mathcal{F}_S$ accepts a packet p as p' if and only if $\mathcal{F}_C$ accepts a packet p as p'' where p' and p'' only differ on marking and NAT. More specifically, p'' is derived from p by applying all the translations available on the path $\pi_{\mathcal{F}_C}(p)$ in the control diagram of $\mathcal{F}_C$, along which p is processed.

Theorem 3. *Let p, p' be two packets such that p is accepted by both $\mathcal{F}_S$ and $\mathcal{F}_C$. Moreover, let $p'' \approx t_\beta(p, p')$ where $\beta = tc(\pi_{\mathcal{F}_C}(p))$. We have that*

$$\mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \mathcal{P}_{\mathcal{F}_C}(p, p'').$$

Example 5. Consider again Example 4. Any path π in `iptables` has $tc(\pi) = nat$, which implies $p' \approx p''$, i.e., $\mathcal{F}_C$ behaves exactly as $\mathcal{F}_S$. Interestingly, paths $\pi_1 = \langle q_i, Inp^n, Inp^f, q_o \rangle$ and $\pi_2 = \langle q_i, Out^n, Out^f, q_o \rangle$ in `pf` have $tc(\pi)$ equal to *dnat* and *snat*, respectively. In fact, `pf` cannot perform *snat* and *dnat* on packets directed to and generated from the host, respectively.

8 Conclusions

We have proposed a transcompling pipeline for firewall languages, made of three stages. Its core is IFCL, an intermediate language equipped here with a formal semantics. It has the typical actions of real configuration languages, and it keeps them apart from the way the firewall applies them, represented by a control diagram. In stage 1, a real firewall policy language can be encoded in IFCL by simply instantiating the state and the control diagram. As a by-product, we give a formal semantics to the source language, which usually has none. In stage 2, we have built a logical predicate that describes the flow of packets accepted by the firewall together with their possible translations. From that, we have synthesized a declarative firewall specification, in the form of a table that succinctly represents the firewall behavior. This table is the basis for supporting policy analysis, like policy implication and comparison, as described in our companion paper [1]. The declarative specification is the input of stage 3, which compiles it to a real target language. To illustrate, we have applied these stages on two among the most used firewall systems in Linux and Unix: `iptables` and `pf`. We have selected these two systems because they exhibit very different packet processing schemes, making the porting of configurations very challenging. All the stages above have been proved to preserve the semantics of the original policy, so guaranteeing that our transcompilation is correct. As a matter of fact, we have proposed a way to mechanically implement policy refactoring, when the source and the target languages coincide. This is because the declarative specification has no anomalies, e.g., rule overlapping or shadowing, so helping the system administrator also in policy maintenance. At the same time, we have put forward a manner to mechanically port policies from one firewall system to another, when their languages differ. We point out that, even though [1] intuitively presents and implements the first two stages of our transcompiling pipeline, the overlap with this paper is only on Sects. 4 and 6.2. Indeed, the theory, the semantics, the compilation of stage 3 and the proofs of the correctness of the whole transcompilation are original material.

As a future work, we intend to further experiment on our proposal by encoding more languages, e.g., from specialized firewall devices, like commercial Cisco IOS, or within the SDN paradigm. We plan to include a (more refined) policy generator of stage 3 in the existing tool [1] that implements the stages 1 and 2, and can deal with configurations made of hundreds of rules. Also testing and improving the performance of our transcompiler, as well as providing it with a friendly interface would make it more appealing to network administrators. For example, readability can be improved by automatically grouping rules and by adding comments that explain the meaning of refactored configurations. 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.

References

1. Bodei, C., Degano, P., Focardi, R., Galletta, L., Tempesta, M., Veronese, L.: Language-independent synthesis of firewall policies. In: Proceedings of the 3rd IEEE European Symposium on Security and Privacy (2018)
2. The Netfilter Project. https://www.netfilter.org/
3. Packet Filter (PF). https://www.openbsd.org/faq/pf/
4. Cuppens, F., Cuppens-Boulahia, N., Sans, T., Miège, A.: A formal approach to specify and deploy a network security policy. In: Dimitrakos, T., Martinelli, F. (eds.) Formal Aspects in Security and Trust. IFIP, vol. 173, pp. 203–218. Springer, Boston, MA (2005). https://doi.org/10.1007/0-387-24098-5_15
5. Gouda, M., Liu, A.: Structured firewall design. Comput. Netw. **51**(4), 1106–1120 (2007)
6. Foley, S.N., Neville, U.: A firewall algebra for openstack. In: 2015 IEEE Conference on Communications and Network Security, CNS 2015, pp. 541–549 (2015)
7. Babel: The compiler for writing next generation JavaScript. https://babeljs.io
8. Runtime converter. http://www.runtimeconverter.com
9. Diekmann, C., Michaelis, J., Haslbeck, M.P.L., Carle, G.: Verified iptables firewall analysis. In: Proceedings of the 15th IFIP Networking Conference, Vienna, Austria, 17–19 May 2016, pp. 252–260 (2016)
10. Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C., Mohapatra, P.: FIREMAN: a toolkit for firewall modeling and analysis. In: IEEE Symposium on Security and Privacy (S&P 2006), May 2006, Berkeley, California, USA, pp. 199–213 (2006)
11. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010, San Jose, CA, USA, 7–12 November 2010 (2010)

12. Mayer, A.J., Wool, A., Ziskind, E.: Fang: a firewall analysis engine. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, 14–17 May 2000, pp. 177–187 (2000)
13. Mayer, A.J., Wool, A., Ziskind, E.: Offline firewall analysis. Int. J. Inf. Secur. **5**(3), 125–144 (2006)
14. Adão, P., Bozzato, C., Rossi, G.D., Focardi, R., Luccio, F.L.: Mignis: a semantic based tool for firewall configuration. In: IEEE 27th Computer Security Foundations Symposium, CSF 2014, pp. 351–365 (2014)
15. Bartal, Y., Mayer, A.J., Nissim, K., Wool, A.: Firmato: a novel firewall management toolkit. ACM Trans. Comput. Syst. **22**(4), 381–420 (2004)
16. Anderson, C.J., Foster, N., Guha, A., Jeannin, J.B., Kozen, D., Schlesinger, C., Walker, D.: NetKAT: semantic foundations for networks. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014). ACM (2014)
17. The Netfilter Project: Traversing of tables and chains. http://www.iptables.info/en/structure-of-iptables.html
18. The IPFW Firewall. https://www.freebsd.org/doc/handbook/firewalls-ipfw.html