From Firewalls to Functions and Back

Lorenzo Ceragioli[1], Letterio Galletta[2], and Mauro Tempesta[3,4]

[1] Università di Pisa, Pisa, Italy
lorenzo.ceragioli@phd.unipi.it
[2] IMT School for Advanced Studies, Lucca, Italy
letterio.galletta@imtlucca.it
[3] Università Ca' Foscari, Venezia, Italy
[4] TU Wien, Vienna, Austria
tempesta@unive.it

Abstract. Designing and maintaining firewall configurations is hard also for expert system administrators. Indeed, policies are made of a large number of rules and are written in low-level configuration languages that are specific to the firewall system in use. To simplify the work of system administrators, some authors of the present paper proposed in previous work a transcompilation pipeline and a tool that **(i)** *extracts* the meaning of a real configuration by representing it into a tabular form; **(ii)** *refactors* a configuration by removing redundant rules; **(iii)** *ports* the policy from a firewall system to another. Here, we extend this pipeline by proposing a new characterization that models rulesets and firewalls as functions from packets to transformations. Transformations specify which packets are accepted by the firewall and how they are translated. Using this functional characterization we propose two new algorithms that simplify the treatment of the pipeline.

CEUR-WS.org/Vol-2315/paper04.pdf

1 Introduction

Firewalls are the fundamental mechanisms for the protection of computer networks. The effectiveness of a firewall system crucially depends on the correctness of its configuration, since even a small flaw may severely impact the security or the functionality of the entire network.

Policies are typically written in low-level configuration languages that are specific to the firewall system in use and support non-trivial control flow constructs, such as calls and gotos. A configuration usually consists of a large number of rules interacting with each other. Indeed, some rules may shadow others or prevent them to be triggered depending on the order in which they appear in the configuration. This context-dependency makes it hard to understand the effect of a single rule on the overall firewall behaviour. Moreover, when writing a policy, network administrators must also take into account how packets are processed by the network stack of the operating system running on the firewall. The scenario becomes even worse when Network Address Translation (NAT) is considered, since packets can be modified while they traverse the firewall by translating IP addresses and performing port redirection.

To simplify the work of system administrators, some of the authors proposed a transcompilation pipeline [4] and a tool [5] to **(i)** *decompile* real configurations into abstract specifications representing the set of the permitted connections; **(ii)** perform *policy refactoring* by removing redundant rules thus obtaining minimal and clean configurations; **(iii)** automatically *port* a configuration written for a system into the language of another system. The proposed transcompiling pipeline is made of the following stages:

1. decompile a policy from the source language to an intermediate language;
2. extract the meaning of the policy as a table describing how the accepted packets are translated;
3. compile the semantic table into the target language.

Core of the stage 1 is the intermediate language IFCL equipped with a formal semantics and with all the typical features of firewall languages. IFCL enables the algorithmic manipulation of stage 2, where a SAT-based procedure is used to derive a minimal declarative configuration in a tabular form that shows all accepted packets and their NAT, without overlapping or shadowed rules.

In this paper, we extend the above pipeline by proposing new algorithms for stage 2 and 3. The first one does not rely on a SAT solver, but on a denotational semantics that directly represents a configuration as a function from packets to *packet transformations*. These transformations specify whether packets are accepted or not and how they are rewritten by NAT. Furthermore, the new algorithm for stage 3 works with the new functional representation and preserves the translation applied by the original configuration. Differently from [4], the algorithm does not rely on tagging packets, thus simplifying the treatment of firewall systems with limited support of this tagging feature, e.g., `pf`. Exploiting the properties of IFCL, we characterize the target systems on which the new algorithm is granted to work for all configurations, like `iptables`. We remark that representing rulesets and firewalls as functions allows us to simplify the treatment of the stages because we resort to a more abstract and handy domain.

The rest of the paper is organized as follows. Section 2 presents our proposal via a small yet realistic example and also compares it to the previous approach of [4]. Our new algorithms are described in Sections 3 and 4. In Section 5 we present related works and in Section 6 we conclude and discuss some future work.

2 Overview of the Pipeline

We consider as an example the network in Figure 1, where the firewall is connected to the LAN 192.168.0.0/24 with IP 192.168.0.1, and to the Internet using the IP 151.15.185.183. Notice that hosts in the LAN have private addresses that cannot be routed on the Internet, hence NAT must be used to rewrite the source address of outgoing packets and the destination address of incoming packets. Notice also that such hosts cannot communicate directly with each other, messages have to pass through the firewall. We wish to configure a firewall that satisfies the following requirements. **(R1)** When a packet from the Internet with destination

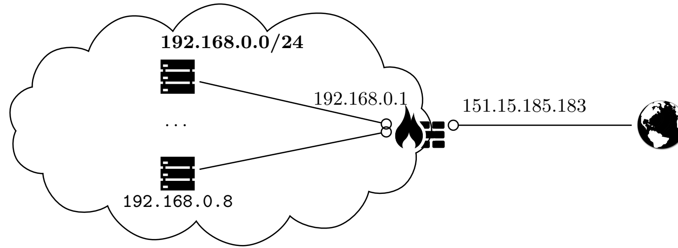

Fig. 1: A simple case of study of a firewall between a local network and Internet.

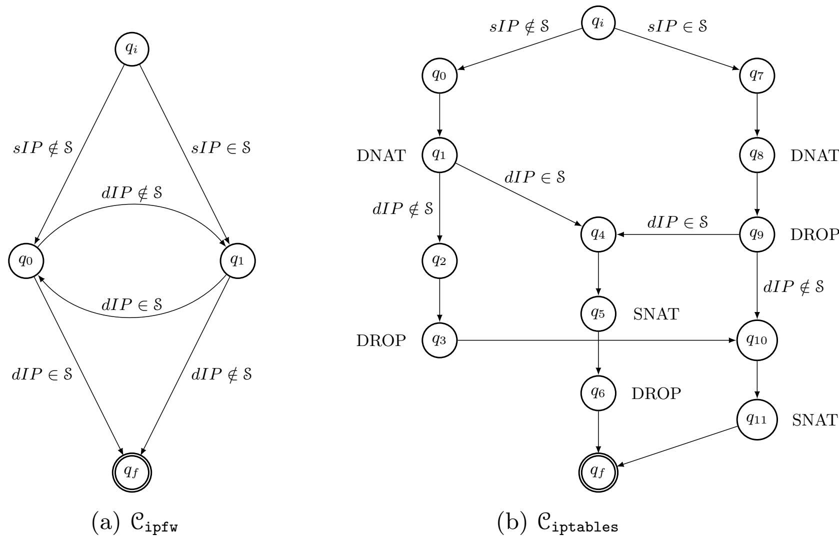

Fig. 2: Control diagrams of `ipfw` and `iptables` (q_i and q_f being the initial and final nodes).

port 22 reaches the external IP address of the firewall, it is redirected to the SSH server at 192.168.0.8. **(R2)** Every LAN host is allowed to access HTTP servers on the Internet. NAT is applied to rewrite the source address of outgoing packets with the public address of the firewall, so that responses from web servers can be routed back. **(R3)** Connections to the address 8.8.8.8 are not allowed. **(R4)** All local hosts can connect to the SSH server in the LAN. **(R5)** The administrator can connect to SSH on the firewall from host 192.168.0.8.

The `ipfw` firewall configuration in Fig. 3 implements the above requirements. Lines `1` and `2` respectively configure the NAT module according to requirements (R1) and (R2), while rules in lines `6` and `7` apply NAT. Note that line `8` imposes an additional condition: the firewall communicates with hosts in the Internet only over port 80. Line `5` is for requirement (R3), and line `9` for requirement (R4), besides (R2). Line `10` implements requirement (R5), while line `11` blocks any other traffic.

```
1  ipfw -q nat 1 config redirect_port tcp 192.168.0.8:22 22
2  ipfw -q nat 2 config ip 151.15.185.183
3
4  ipfw -q add 0010 check-state
5  ipfw -q add 0020 deny tcp from any to 8.8.8.8
6  ipfw -q add 0030 nat 1 tcp from not 192.168.0.0/24 to 151.15.185.183 22
7  ipfw -q add 0040 nat 2 tcp from 192.168.0.0/24 to not 192.168.0.0/24 80
8  ipfw -q add 0050 allow tcp from 151.15.185.183 to not 192.168.0.0/24 80
9  ipfw -q add 0060 allow tcp from any to 192.168.0.8 22
10 ipfw -q add 0070 allow tcp from 192.168.0.8 to 192.168.0.1 22
11 ipfw -q add 0080 deny all from any to any
```

Fig. 3: An `ipfw` configuration.

Before showing the result of the decompilation (stage 1 of our pipeline), we briefly introduce the intermediate language IFCL. A firewall configuration in IFCL consists of a set of rules, with a format common to most of the real firewalls, and a control diagram that describes how packets are processed by the firewall system under consideration. Specifically, a control diagram is a graph $\mathcal{C}$ where every node is associated to a ruleset that is applied to packets reaching that node. Arcs are labeled with predicates that encode the routing decisions performed by the firewall. Intuitively, a packet p is accepted by the firewall if there exists a path from the initial to the final node of $\mathcal{C}$ such that p is accepted (and possibly transformed) by the rulesets associated to the nodes of the path.

A firewall rule consists of a predicate ϕ over packets and an action t, called *target*, defining how packets matching ϕ are processed. We consider the targets:

ACCEPT	the packet is accepted
DROP	the packet is discarded
NAT(n_d, n_s)	apply NAT
MARK(m)	marking with tag m

In the NAT action, n_d and n_s specify how to translate the destination and source addresses/ports of a packet and we use $\star$ to denote the identity translation. For instance, $n_d = n : \star$ means that the destination address of a packet is translated according to n, while the port is kept unchanged. The MARK action marks a packet with a tag m. Predicates of the rules in a ruleset may select packets based on tags assigned by preceding rules of the firewall. Notice that, since the tag is not part of the network packet, it is lost when the packet leaves the firewall. Here we consider a subset of targets used in [4] and we restrict the NAT target to addresses only not ranges as in [4]. We do not lose generality because the initial part of stage 2 normalises the configuration by removing the targets concerning the flow control and because performing NAT towards a range of addresses is rarely used in practice. Additionally, we assume all connections to be new as done in [5].

Fig. 2a shows $\mathcal{C}_{\texttt{ipfw}}$, the control diagram of `ipfw`. Guards label the arcs to check whether the source/destination address of packets (in symbols *sIP*/*dIP*) belong to $\mathcal{S}$ (the addresses assigned to the firewall). Stage 1 produces the firewall consisting of $\mathcal{C}_{\texttt{ipfw}}$ and the assignment to both nodes q_0 and q_1 of the ruleset

$(\texttt{dIP} = 8.8.8.8,\ \text{DROP});$

Received packets				Accepted packets			
sIP	**sPort**	**dIP**	**dPort**	**sIP**	**sPort**	**dIP**	**dPort**
192.168.0.8	*	192.168.0.1	22	-	-	-	-
*	*	192.168.0.8	22	-	-	-	-

151.15.185.183	*	* \ {
		8.8.8.8
		192.168.0.0/24
		}
			80	-	-	-	-

192.168.0.0/24	*	* \ {
		8.8.8.8
		192.168.0.0/24
		}
			80	151.15.185.183	-	-	-

* \ {
192.168.0.0/24
}
	*	151.15.185.183	22	-	-	192.168.0.8	-

(a) The semantics of the configuration in Fig. 3.

Received packets				Accepted packets			
sIP	**sPort**	**dIP**	**dPort**	**sIP**	**sPort**	**dIP**	**dPort**
192.168.0.1 151.15.185.183 127.0.0.1	*	192.168.0.8	22	-	-	-	-

151.15.185.183	*	* \ {
		8.8.8.8
		192.168.0.0/24
		}
			80	-	-	-	-

192.168.0.1	*	* \ {
		8.8.8.8
		192.168.0.0/24
		}
			80	-	-	-	-

(b) Function applied by node q_9 of $\mathcal{C}_{\texttt{iptables}}$.

Table 1: Tabular representation of the semantics and of the function in node q_9.

$$(\texttt{sIP} \notin 192.168.0.0/24 \wedge \texttt{dIP} = 151.15.185.183 \wedge \texttt{dPort} = 22,\ \textsc{nat}(192.168.0.8 : \star, \star : \star));$$
$$(\texttt{sIP} \in 192.168.0.0/24 \wedge \texttt{dIP} \notin 192.168.0.0/24 \wedge \texttt{dPort} = 80,\ \textsc{nat}(\star : \star, 151.15.185.183 : \star));$$
$$(\texttt{sIP} = 151.15.185.183 \wedge \texttt{dIP} \notin 192.168.0.0/24 \wedge \texttt{dPort} = 80,\ \textsc{accept});$$
$$(\texttt{dIP} = 192.168.0.8 \wedge \texttt{dPort} = 22,\ \textsc{accept});$$
$$(\texttt{sIP} = 192.168.0.8 \wedge \texttt{dIP} = 192.168.0.1 \wedge \texttt{dPort} = 22,\ \textsc{accept});$$
$$(true,\ \textsc{drop})$$

The second stage of our pipeline defines a function that specifies whether a packet is accepted or not and eventually how it is transformed. In other words, this function is the denotational semantics of the configuration in hand. Table 1a displays the functional meaning of the example in Fig. 3. To obtain it we first compute the functions associated to the ruleset of each node and then we compose them along the paths of the control diagram. In this specific example the composition is trivial since the rulesets associated with q_0 and q_1 are the same.

The last stage of our pipeline compiles the semantics into the target language, here `iptables`. This stage consists of three steps. In the first, we consider the control diagram $\mathcal{C}_{\texttt{iptables}}$ in Fig. 2b, where, by construction, the operations over

packets are only allowed on specific nodes, as suggested by the labels in the figure. For transcompiling, we are thus left to associate with the nodes of $\mathcal{C}_{\texttt{iptables}}$ a suitable function, computed from the semantics in Table 1a. Essentially, we project it along the paths of $\mathcal{C}_{\texttt{iptables}}$, taking care of the actions that `iptables` prescribes in the traversed nodes.

For example, take node q_9: only packets with a source IP address in $\mathcal{S}$ can traverse it. Moreover, packets subjected to DNAT will reach q_9 with the translated destination address, since DNAT is applied in node q_8. Differently, SNAT will occur later on the path (in node q_5 or q_{11}). We build the Table 1b associated with node q_9 by transforming each row of Table 1a by applying the specified DNATand then keeping only packets with source address in $\mathcal{S}$. The second step discards the first row of Table 1a since $192.168.0.8 \notin \mathcal{S}$. Instead, filtering out the non-local addresses in the second row yields the first row of Table 1b. The third row is left unchanged. We select the only local address from the fourth row and we defer the SNAT to the rulesets associated to nodes q_5 and q_{11}. Applying the specified DNAT to the fifth row turns 151.15.185.183 into 192.168.0.8, which is already represented by the first row of Table 1b.

The third step is straightforward: each row of the tabular representation of the obtained function originates an IFCL rule. In this example, the result for nodes q_5 and q_{11} is

$$(\texttt{sIP} \in 192.168.0.0/24 \land \texttt{dIP} \notin 192.168.0.0/24 \land \texttt{dPort} = 80,\ \textsc{nat}(\star:\star, 151.15.185.183:\star))$$

while for q_1 and q_8 we have

$$(\texttt{sIP} \notin 192.168.0.0/24 \land \texttt{dIP} = 151.15.185.183 \land \texttt{dPort} = 22,\ \textsc{nat}(192.168.0.8:\star, \star:\star))$$

and for nodes q_3, q_6 and q_9 respectively

$$(\texttt{dIP} = 192.168.0.8 \land \texttt{dPort} = 22,\ \textsc{accept});$$
$$(\texttt{sIP} \in 192.168.0.0/24 \land \neg(\texttt{dIP} \in 192.168.0.0/24 \lor \texttt{dIP} \in 8.8.8.8) \land \texttt{dPort} = 80,\ \textsc{accept});$$
$$(\mathit{true},\ \textsc{drop})$$

$$(\texttt{sIP} = 192.168.0.8 \land \texttt{dIP} = 192.168.0.1 \land \texttt{dPort} = 22,\ \textsc{accept});$$
$$(\texttt{sIP} = 151.15.185.183 \land \texttt{dIP} \notin 192.168.0.0/24 \land \texttt{dPort} = 80,\ \textsc{accept});$$
$$(\mathit{true},\ \textsc{drop})$$

$$(\texttt{sIP} = 151.15.185.183 \land \neg(\texttt{dIP} \in 192.168.0.0/24 \lor \texttt{dIP} \in 8.8.8.8) \land \texttt{dPort} = 80,\ \textsc{accept});$$
$$(\texttt{sIP} = 192.168.0.1 \land \neg(\texttt{dIP} \in 192.168.0.0/24 \lor \texttt{dIP} \in 8.8.8.8) \land \texttt{dPort} = 80,\ \textsc{accept});$$
$$(\texttt{dIP} = 192.168.0.8 \land \texttt{dPort} = 22,\ \textsc{accept});$$
$$(\mathit{true},\ \textsc{drop})$$

All remaining nodes have an empty ruleset. Eventually, these rules are compiled into the `iptables` configuration shown in Fig. 4. Note that some target languages may constrain the guard of rules to have a specific form, more restrictive than the one used by IFCL. Hence, before generating the target configuration, we transform every IFCL rule into an equivalent form, possibly implemented with multiple rules. For example, in an `iptables` rule you cannot impose an IP

```
 1 *nat
 2 :PREROUTING ACCEPT [0:0]
 3 :INPUT ACCEPT [0:0]
 4 :OUTPUT ACCEPT [0:0]
 5 :POSTROUTING ACCEPT [0:0]
 6
 7 -A PREROUTING -p tcp ! -s 192.168.0.0/24 -d 151.15.185.183 --dport 22
 8                                              -j DNAT --to 192.168.0.8
 9 -A OUTPUT -p tcp ! -s 192.168.0.0/24 -d 151.15.185.183 --dport 22
10                                              -j DNAT --to 192.168.0.8
11 -A INPUT -p tcp -s 192.168.0.0/24 ! -d 192.168.0.0/24 --dport 80
12                                              -j SNAT --to 151.15.185.183
13 -A POSTROUTING -p tcp -s 192.168.0.0/24 ! -d 192.168.0.0/24 --dport 80
14                                              -j SNAT --to 151.15.185.183
15 COMMIT
16
17 *filter
18 :INPUT DROP [0:0]
19 :FORWARD DROP [0:0]
20 :OUTPUT DROP [0:0]
21
22 -A OUTPUT -p tcp -s 151.15.185.183 --match iprange
23                 --dst-range 0.0.0.0-8.8.8.7 --dport 80 -j ACCEPT
24 -A OUTPUT -p tcp -s 151.15.185.183 --match iprange
25                 --dst-range 8.8.8.9-192.167.255.255 --dport 80 -j ACCEPT
26 -A OUTPUT -p tcp -s 151.15.185.183 --match iprange
27                 --dst-range 192.168.1.0-255.255.255.255 --dport 80 -j ACCEPT
28 -A OUTPUT -p tcp -s 192.168.0.1 --match iprange
29                 --dst-range 0.0.0.0-8.8.8.7 --dport 80 -j ACCEPT
30 -A OUTPUT -p tcp -s 192.168.0.1 --match iprange
31                 --dst-range 8.8.8.9-192.167.255.255 --dport 80 -j ACCEPT
32 -A OUTPUT -p tcp -s 192.168.0.1 --match iprange
33                 --dst-range 192.168.1.0-255.255.255.255 --dport 80 -j ACCEPT
34 -A OUTPUT -p tcp -d 192.168.0.8 --dport 22 -p tcp -j ACCEPT
35
36 -A FORWARD -p tcp -d 192.168.0.8 --dport 22 -p tcp -j ACCEPT
37 -A FORWARD -p tcp -s 192.168.0.0/24 --match iprange
38                 --dst-range 0.0.0.0-8.8.8.7 --dport 80 -j ACCEPT
39 -A FORWARD -p tcp -s 192.168.0.0/24 --match iprange
40                 --dst-range 8.8.8.9-192.167.255.255 --dport 80 -j ACCEPT
41 -A FORWARD -p tcp -s 192.168.0.0/24 --match iprange
42                 --dst-range 192.168.1.0-255.255.255.255 --dport 80 -j ACCEPT
43
44 -A INPUT -p tcp -s 192.168.0.8 -d 192.168.0.1 --dport 22 -j ACCEPT
45 -A INPUT -p tcp -s 151.15.185.183 ! -d 192.168.0.0/24 --dport 80 -j ACCEPT
46
47 COMMIT
```

Fig. 4: The `ipfw` configuration in Fig. 3 transcompiled to `iptables`.

address to be anything but 8.8.8.8 or 192.168.0.0/24[5]. You can achieve the same goal by considering the ranges obtained by taking the complement of the union of those sets, as in lines 22-27 of Fig. 4.

The synthesis is similar to the one proposed in [4], where logical predicates $\mathcal{P}(p, \tilde{p})$ were assigned to the rulesets. These predicates are true if and only if the packet p is accepted as $\tilde{p}$ by the corresponding ruleset. The predicate corresponding to a firewall was, thus, obtained by the composition of predicates

[5] You can write something like `! -d 8.8.8.8, 192.168.0.0/24`, but it will be interpreted as $dIP \notin 192.168.0.0/24 \vee dIP \neq 8.8.8.8$ i.e. in a really useless requirement.

for rulesets in the paths of the control diagram. The table was then produced by computing the model of the firewall predicate using a variant of the bisection method in which every check resulted in a call to the SAT solver. Since we have chosen an handy representation here, the resulting table can be computed directly, hence gaining flexibility and avoiding the bottleneck of multiple invocation to the solver. On the contrary, the last stage of the pipeline is very different from the one presented in [4]. Indeed, there one specific ruleset was produced for each specific task: filtering the network, performing `SNAT` and `DNAT`. The actions to be performed on each packet were known thanks to the tags previously associated by the firewall. Furthermore, the assignment of rulesets to the nodes of the target control diagram was defined in an ad hoc way for the supported languages (`iptables` and `pf`). That approach is not immediately extensible to other systems, especially those with minimal control diagram like `ipfw`, where some nodes may accomplish all the tasks. Finally, the last translation into the target language was not treated in [4]. We believe that the new translation algorithm simplifies this last step because we do not need to deal with tagging whose support is very heterogeneous on the various systems.

3 From a Configuration to a Function: Synthesis

We present the algorithm for the second stage of our pipeline. Given an IFCL configuration we compute a synthetic representation of its behaviour as a function on the ongoing traffic: we first formally define the domains on which the resulting function operates, and we then introduce a convenient representation from which we derive the algorithm.

Domains Let $\mathbb{P}$ be the set of all network packets. We define a semantic function of type $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ that given a packet $p \in \mathbb{P}$ either discards it (returning $\bot$) or accepts it, possibly with some changes due to NAT (returning a transformation $t \in \mathcal{T}(\mathbb{P})$, defined field by field). Below we denote by $t(p)$ the packet obtained by applying the transformation t to the packet p. Furthermore, given two transformations t and t', we denote by $t \ltimes t'$ (t updates t') the transformation resulting by first applying t' and then t, formally $(t \ltimes t')(p) = t(t'(p))$ — as we will see later on, $\ltimes$ is not standard function composition since it deals only with the outputs, which are accumulated.

The succinct representation we use is based on multi-cubes, which are a generalization of geometric cubes [7]. In a n-dimensional space, a cube is the cartesian product of n segments, whereas a multi-cube is the product of n unions of segments. Since firewall rules put requirements on packets field by field, the set of packets matching them is naturally expressible as a multi-cube. For example, the set of packets satisfying the predicate

$$\texttt{sIP} = 192.168.0.8 \wedge \texttt{sPort} = 22 \wedge (\texttt{dIP} \in 192.168.0.0/24 \vee \texttt{dIP} = 127.0.0.1) \wedge \texttt{dPort} \in \{22, 80\}$$

can be expressed using the following multi-cube:

$$\{192.168.0.8\} \times \{22\} \times (192.168.0.0/24 \cup \{127.0.0.1\}) \times (\{22\} \cup \{80\})$$

Algorithm 1

1: **function** RULESET_SYNTHESIS(P: set of packets, R: ruleset, t: transformation)
2: **if** $R = [\,]$ **then return** $\{(P, t)\}$
3: $(\phi, target) \cdot R' \leftarrow R$
4: $(P_s, \mathbf{P}_n) \leftarrow$ SPLIT(P, t, ϕ)
5: $\lambda' \leftarrow \bigcup_{P'_n \in \mathbf{P}_n}$(RULESET_SYNTHESIS($P'_n$, R', t))
6: **if** $target =$ ACCEPT **then return** $\{(P_s, t)\} \cup \lambda'$
7: **else if** $target =$ DROP **then return** $\{(P_s, \bot)\} \cup \lambda'$
8: **else if** $target =$ NAT(d_n, s_n) **then return** $\{(P_s, tr_{nat}(d_n, s_n) \ltimes t)\} \cup \lambda'$
9: **else if** $target =$ MARK(m) **then return** RULESET_SYNTHESIS(P_s, R', $t_{tag(m)} \ltimes t) \cup \lambda'$

We represent a semantic function as a set A of pairs (P, t), where $P \subseteq \mathbb{P}$ is a multi-cube and t is either the transformation associated with every packet in P or it is $\bot$. Note that the projection of A on its first component is a partition of $\mathbb{P}$. The tables of the above example have rows corresponding to a pair (P, t): the first four columns are the input (the multi-cube) and the last four are the output (the transformation). For brevity, the rows with output $\bot$ are omitted.

Algorithms The stage 2 first computes a semantic function for every ruleset of the control diagram through RULESET_SYNTHESIS in Algorithm 1; then, it composes the results in a single table using COMPOSE in Algorithm 2.

Consider first the recursive algorithm RULESET_SYNTHESIS. Its parameters are respectively: a multi-cube P (initially $\mathbb{P}$) representing the set of packets we are interested in; the ruleset R we are analyzing, scanned rule by rule; and the transformation t (initially the identity) already computed for P. Taken a rule (line 3) the set P is split in the two sets P_s and $\mathbf{P}_n$ (line 4). The first is the multi-cube of the packets that verify the rule when transformed by t, thus, the algorithm returns it unchanged. The (complementary) set $\mathbf{P}_n$ is instead a disjoint union of non-empty multi-cubes, on which the function is recursively called on (line 5). Lines 6 and 7 are trivial. In line 8 the transformation dictated by the NAT (represented by tr_{nat}) updates the accumulated one. Note that, as stated before, $\ltimes$ is not the standard function composition. Besides updating the transformation, line 9 calls recursively the function, because MARK does not terminate packet processing. The base case of recursion, when R is empty (line 2), yields the set of packets as it is and the computed transformation over them.

Now we exploit COMPOSE in Algorithm 2 to combine the results returned by the previous algorithm. The underlining idea is to visit the control diagram backward from the final node to the initial one, thus composing the semantic function of a node q with the result of COMPOSE applied to each successor q'. Consequently, the table assigned to q' is the combination of the functions of all the following nodes from q' to q_f. More precisely, line 3 computes the packets that are dropped by node q, i.e., the pairs $(P, \bot)$. Then, for each successor q', reachable through an arc labeled with ψ, line 6 propagates the remaining packets

Algorithm 2

1: **function** COMPOSE(q: node)
2: $\lambda_R \leftarrow$ function assigned to node q
3: $\lambda_q \leftarrow \{(P,t) \in \lambda_R \mid t = \bot\}$
4: **for** node q' reachable from q with guard ψ on the arc **do**
5: $\lambda_{q'} \leftarrow$ function assigned to node q'
6: $\lambda_{(q,q')} \leftarrow \{(P',t) \mid (P,t) \in \lambda_R \wedge P' = \psi(t(P))\}$
7: $\lambda_q \leftarrow \lambda_q \cup \{(P'', t' \ltimes t) \mid (P,t) \in \lambda_{(q,q')} \wedge (P',t') \in \lambda_{q'} \wedge P'' = t^{-1}(P' \cap t(P))\}$
8: **return** λ_q

that verify ψ after the application of t[6]. Finally, line 7 composes each propagated pair (P,t), with each pair (P',t') of q', if there is a nonempty subset P'' of P included in P' after the application of t[7]. The result of the composition is the pair (P'',t''), where t'' is the transformation t updated with t'.

The resulting table associated with q_i of the control diagram is the synthesis representing the semantic function of the firewall configuration.

4 From a Function to a Configuration: Generation

Here we describe the last stage of our pipeline which compiles the output of the previous stage into the target language. In particular, we focus on an algorithm that computes the functions for the rulesets and assigns them to the nodes of the target control diagram. Note that when a firewall system is not capable of expressing the input semantic function, the algorithm detects it. Moreover, it is granted to compile (if possible) whenever the paths in the target control diagram do not have duplicated NAT labels (e.g., `DNAT` repeated more than once).

Intuitively, our algorithm projects the table of the system to be compiled onto the paths of the control diagram of the target system. Since the modeled systems constrain the nodes of the control diagram to perform only certain actions, the projection must take into account the permitted actions of the traversed nodes. We represent these constraints by labeling each node with a subset of $\{\texttt{SNAT}, \texttt{DNAT}, \texttt{DROP}\}$, as we did in Fig. 2b. Only assignments that meet the constraints are granted to be expressible in the target language.

The tables assigned to nodes are generated incrementally, starting with an empty set and then gradually adding pairs (P,t) implementing the expected behaviour. First, we deal with rows specifying accepted packets, by assigning them to the corresponding path in the target control diagram. For example, the first line of Table 1a is assigned to the path $\pi = q_i, q_0, q_1, q_4, q_5, q_6, q_f$ in $\mathcal{C}_{\texttt{iptables}}$. Given such a path, we project a pair (P,t) along its nodes, by first computing the transformations and then the corresponding multi-cubes. As regards the transformation t, we decompose it and we generate a new transformation for each

[6] In the Algorithm 2 we abuse the notation, denoting $\{p \in P \mid \phi(p)\}$ with $\phi(P)$.

[7] Note that t is not an invertible function, hence with $t^{-1}(P' \cap t(P))$ we denote the preimage of $P' \cap t(P)$ w.r.t. t, inside the domain P.

node following the labels. For example, a row with target $\texttt{NAT}(d_n, s_n)$ associated with the path π in $\mathcal{C}_{\texttt{iptables}}$, results in $\texttt{NAT}(d_n, \star : \star)$ in q_1, $\texttt{NAT}(\star : \star, s_n)$ in q_5 and in id in all the other nodes of π. As for the multi-cube P, we assign to each node the result of the application of the predecessor transformation to its corresponding multi-cube. The first node of the path is assigned to P. In the example above we assign the pair (P, id) to q_i and q_0, (P, t') to q_1, where t' is the transformation implementing $\texttt{NAT}(d_n, \star : \star)$, and $(t'(P), id)$ to q_4. After repeating this procedure for all the rows of the table, the accepted packets are managed correctly, however the sets of pairs assigned to each node q can be incomplete: some packets may be included in no multi-cubes of q. We call them *free packets* of q. In other words, we still need to define the behaviour of the node for some cases. Next we need to complete the just assigned sets in order to manage the packets discarded by the input semantic function. Given that we do not modify the pairs already added, it is not possible to compromise the previous result. Moreover, since we are left to deal with dropped packets only, every packet that is not already included in a multi-cube should be dropped.

We start from the nodes in the control diagram labeled with DROP, where the choice is trivial: we assign $\perp$ to every free packet. Then, we use a recursive procedure to configure all other nodes to transform their free packets in such a way that they reach some node $q_\perp$ labeled with DROP as free packets of $q_\perp$. Let $\mathbf{P}_\perp$ be the set of multi-cubes containing the packets dropped by the function associated to q (either directly or by passing them to another node that discards them), and assume that q' is a predecessor of node q reachable through an arch labeled with ψ. The idea is to assign to the free packets of q' a transformation that maps them inside $\psi(\mathbf{P}_\perp)$. We filter $\mathbf{P}_\perp$ using ψ because we want to send the packets along the arc pointing to q. For each multi-cube of free packets of q' we check if it is possible to map some of them inside $\psi(\mathbf{P}_\perp)$, updating the table accordingly. In each node, the transformations that we can exploit depend on the labels. If a node is labeled with both SNAT and DNAT, then we can map all the free packets to a single value inside $\psi(\mathbf{P}_\perp)$. If we have no labels assigned, we can only apply id, and then, for each multi-cube of free packets, the subset of the dropped ones is computed as the intersection with $\psi(\mathbf{P}_\perp)$. The case where only DNAT (resp. SNAT) is assigned to a node is a composition of the previous ones, where for the source addresses (resp. destination addresses) we take the intersection with each multi-cube in $\psi(\mathbf{P}_\perp)$, and we map the other addresses to some value inside the same multi-cube. After updating the table of q' the back-propagation continues recursively.

5 Related Work

In this work we extend the pipeline defined in [4], of which the implementation of the first two phases is described in [5]. To the best of our knowledge, [4] and this work are the only approaches in literature to mechanically port firewall policies, while there are other tools for some firewall administration tasks. There are proposals that deal with specific problems without deriving an abstract rep-

resentation of the firewall policies, like [6] for refactoring, FIREMAN [11] and Margrave policy analyzer [10] for error correction. Other approaches, like ours, are based on abstract policies: they can be divided into those that start from a real configuration and derive an abstract one to perform analyses, like Fang [8,9], that can discover anomalies; and those that starting from an abstract policy generate a real configuration, like [1,3]. All these approaches propose their own high level language with a formal semantics, and define a compilation from the real configuration language to the abstract one (cf. our stage 1 and 2) or vice-versa (cf. our stage 3). Instead our approach defines the compilation in both directions, dealing with 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. NetKat [2] embraces a different approach, proposing linguistic constructs for programming a network as a whole within the SDN paradigm. This approach is orthogonal to ours: we consider real firewall configuration languages as low-level machine languages, and provide the needed tools for supporting legacy systems.

Our approach and that of [4,5] differ from the other proposals mainly because *at the same time* it **(i)** is language-independent; **(ii)** defines a formal semantics of firewall behavior; **(iii)** gives a concise and neat representation of such a behavior; **(iv)** supports NAT, MARK and targets for changing the control flow; **(v)** both input and output are real configuration languages.

6 Conclusion

We presented two new algorithms for stages 2 and 3 of the transcompilation pipeline of [4]. The new algorithm for stage 2 directly represents a configuration as a function (in a tabular form) from packets to transformations, specifying which packets are accepted and how they are re-written by NAT. The second algorithm starts from the functional representation returned by stage 2 and compiles it into another system, preserving all the NAT of the original configuration and not relying on any tag mechanism. Finally, we characterized the properties of the target systems for which the algorithm is granted to work for all input configurations. In particular, there are cases like `ipfw` where the algorithm is not granted to work, in the sense that it can fail even when a solution would be possible. This is due to the fact that the control diagram does not impose enough constraints on which nodes can apply `SNAT` and `DNAT`. A limit of the proposed approach is that multi-cubes can be split but not merged by the algorithms, causing the synthesis to produce tables with many rows, and, consequently, the compilation to produce long configurations.

As future work we plan to implement the new algorithms inside the tool proposed in [5] and to carry out an experimental evaluation to test whether the new functional representation improves the performance of the tool when dealing with real-world configurations. We also plan to implement and evaluate a procedure for multi-cube merging. Furthermore, we aim at improving the readability of the generated policies by automatically grouping rules and by adding

comments that explain their meaning. Finally, it would be very interesting to extend our approach to deal with networks with more than one firewall.

References

1. Adão, P., Bozzato, C., Dei Rossi, G., Focardi, R., Luccio, F.L.: Mignis: A Semantic Based Tool for Firewall Configuration. In: proc. of the 27th IEEE CSF. pp. 351–365 (2014)
2. Anderson, C.J., Foster, N., Guha, A., Jeannin, J., Kozen, D., Schlesinger, C., Walker, D.: Netkat: semantic foundations for networks. In: proc. of 41st ACM POPL. pp. 113–126 (2014)
3. Bartal, Y., Mayer, A.J., Nissim, K., Wool, A.: Firmato: A novel Firewall Management Toolkit. ACM Transactions on Computer Systems **22**(4), 381–420 (2004)
4. Bodei, C., Degano, P., Focardi, R., Galletta, L., Tempesta, M.: Transcompiling firewalls. In: Proc. POST 2018. LNCS (2018)
5. Bodei, C., Degano, P., Focardi, R., Galletta, L., Tempesta, M., Veronese, L.: Language-independent synthesis of firewall policies. In: Proc. 3rd IEEE European Symposium on Security and Privacy (2018)
6. Diekmann, C., Michaelis, J., Haslbeck, M.P.L., Carle, G.: Verified iptables Firewall Analysis. In: the 15th IFIP Networking Conference. pp. 252–260 (2016)
7. Jayaraman, K., Bjørner, N., Outhred, G., Kaufman, C.: Automated Analysis and Debugging of Network Connectivity Policies. Tech. rep., Microsoft (2014)
8. Mayer, A.J., Wool, A., Ziskind, E.: Fang: A Firewall Analysis Engine. In: proc. of the 21st IEEE S&P 2000. pp. 177–187 (2000)
9. Mayer, A.J., Wool, A., Ziskind, E.: Offline firewall analysis. Int. J. Inf. Sec. **5**(3), 125–144 (2006)
10. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave Tool for Firewall Analysis. In: Proc. of LISA 2010 (2010)
11. Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C., Mohapatra, P.: FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In: 27th IEEE S&P. pp. 199–213 (2006)