Mignis: A semantic based tool for firewall configuration

P. Adão

SQIG, Instituto de Telecomunicações
Instituto Superior Técnico, Universidade de Lisboa
Email: pedro.adao@ist.utl.pt

C. Bozzato, G. Dei Rossi, R. Focardi and F.L. Luccio

DAIS, Università Ca' Foscari Venezia, Italy
Email: {cbozzato,deirossi,focardi,luccio}@dsi.unive.it

Abstract—**The management and specification of access control rules that enforce a given policy is a non-trivial, complex, and time consuming task. In this paper we aim at simplifying this task both at specification and verification levels. For that, we propose a formal model of Netfilter, a firewall system integrated in the Linux kernel. We define an abstraction of the concepts of chains, rules, and packets existent in Netfilter configurations, and give a semantics that mimics packet filtering and address translation. We then introduce a simple but powerful language that permits to specify firewall configurations that are unaffected by the relative ordering of rules, and that does not depend on the underlying Netfilter chains. We give a semantics for this language and show that it can be translated into our Netfilter abstraction. We then present Mignis, a publicly available tool that translates abstract firewall specifications into real Netfilter configurations. Mignis is currently used to configure the whole firewall of the DAIS Department of Ca' Foscari University.**

I. INTRODUCTION

Protecting networks from external and internal attacks is a crucial task. System administrators rely on the usage of firewalls that examine the network traffic and enforce policies based on specified rules. However, implementing correct policies is a non-trivial task: if a policy is too weak the system may be attacked by exploiting its weaknesses, while if it is too restrictive legitimate traffic may be filtered out.

Manually proving that implementations comply with a firewall policy is a too much time-consuming practice given that firewall rules are usually written in low-level, platform-specific languages, thus automatic tools for testing them have been developed [1], [2]. These tools however do not prevent users from introducing new flaws when modifying such policies. Some flaws may derive from the wrong order of firewall rules (consistency problems), and some others from the lack of matching rules for every packet that crosses the firewall (completeness problems). Another approach proposed by Liu et al. [3], is based on a firewall design process that passes through different verification stages, but this is also time and resource consuming. Policy visualization tools have also been developed [4], [5], [6], [7], but they are not sufficiently helpful in dynamically changing networks where new services are added over time, as these typically impose very articulated firewalls composed of hundreds or even thousands of interacting rules. It is in fact very difficult to keep the number of rules small also because of redundancies (compactness problem).

In our opinion, there is an increasing need for formal and general tools to reason about the security of firewalls. Existing tools are however still far from the intended goal and we propose in this paper one further step in that direction.

Our contribution: Netfilter is a firewall system integrated in the Linux kernel [8]. A firewall in Netfilter is implemented as a series of chains, tables and rules that are executed in a precise given order. In this paper we propose a model of Netfilter in which we abstract the concept of chains, rules and packets, and introduce the notion of state that records the information about exchanged packets. We give a semantics for this abstraction, close to the real one, that specifies how packets are dealt by the firewall in a specific state.

The novel features of our model allow us to introduce a new simple declarative language that specifies firewall policies by abstracting both the order in which rules are applied, and the different chains that Netfilter provides. The main advantage of this language is that transitions are defined in a single-step fashion, contrary to the multi-step semantics associated with the evaluation of the different tables of Netfilter.

We then show how this language can be translated into our Netfilter abstraction, and we provide sufficient conditions under which a specification given in this language and its translation into Netfilter abstraction have the same effect on packets, both in terms of filtering and network address translation.

It is important to stress that, in our high level setting, any order of rules is acceptable and irrelevant for the semantics, whereas in Netfilter the order in which rules are written is fundamental and in general not interchangeable. Indeed, a well-known difficulty that reduces significantly the usability of Netfilter is that adding/deleting/modifying rules is context-dependent and might potentially break the whole firewall policy. This makes it painful for system administrators to modify complex Netfilter configurations. Our firewall language, instead, makes it very easy to modify a configuration as the relative order of rules never affects the behavior of the generated Netfilter rules. This language, in spite of its simplicity, is expressive and powerful enough to specify the most commonly used network security policies.

In order to demonstrate the feasibility and illustrate the simplicity and advantages of this approach we also present MIGNIS, a novel publicly available tool that translates, according to the aforementioned results, abstract firewall specifications into real Netfilter configurations. We then show an example of how MIGNIS can be used in a realistic, large scale,

and non-trivial setting: MIGNIS is currently used to configure the firewall of the DAIS Department of the Ca' Foscari University of Venice. Using the overlap-detecting capabilities of MIGNIS and its simple syntax we were able to tackle the compactness problem by capturing many redundancies in the initial Netfilter configurations, and we could thus drastically reduce the number of configuration lines. Moreover, we have run some experiments by querying the MIGNIS specification and we were able to extract information such as the rules that affect packets from a certain host or whether a certain rule is already included or not in the specification.

Related work: Formal approaches to firewall policy configurations have been studied in the literature. Gouda et al. propose in [9] a method, called Structured Firewall Design, which allows the design of a firewall with non-conflicting and compact rules. The technique is very interesting but it limited to *stateless firewalls* in which a decision is taken only depending on the received packet, while our solution instead is more general given that it applies also to *stateful firewalls*, in which the fate of some packet also depends on the history of previously received packets.

In [10] Jeffrey et al. introduce a formal model of firewall policy, based on Netfilter, and investigate two properties of firewall policy configurations, namely reachability and cyclicity. They also provide an NP-completeness proof of the problem of analyzing a firewall configuration with respect to those properties.

As we have previously pointed out, an accurate semantics of a high level language allows the security administrator to avoid firewall misconfigurations. The work that is closer to our proposal is the one of Cuppens et al. [11], that provides an access control language based on the XML syntax that is supported by the access control model Or-BAC. It also presents an example of the mapping of the abstract policy into some Netfilter firewall rules. In spite of the proposed language being more generic than ours, it lacks some important elements of Netfilter configurations such as the specification of the network address translation (NAT). Moreover, and contrary to our solution, no formal proof of the correctness of the translation process between the abstract language and the low level Netfilter language is presented.

Recently, some *network programming languages*, that create an abstraction of network programs over some hardware-oriented APIs, have been proposed. An example is the language proposed in the Frenetic project [12], which works at two distinct levels and is able to capture dynamic policies. The language has evolved during the years but still it is not clear which primitives are essential, and how new added constructs should interact with the existing ones. Moreover, it does not address interesting questions such as network reachability or cyclicity. These features and others, such as reachability, traffic isolation, and compiler correctness, have been captured by NetKAT a new network programming language equipped with a sound and complete equational theory [13]. However, this language is not specifically targeted at firewalls, and thus does not consider some issues such as NAT, and does not provide a translation mechanism to real-world packet filters.

For what concerns firewall configuration tools, there are many, but, as far as we know, for none of them a semantics of their specification language has been provided. Some of those languages may be compiled to several low level languages, and admit any order of rules, and thus are more general than the specification language that we propose. However, due to their generality, their complexity is close to the one of the low level specification, thus making their adoption harder. Moreover, they usually also have a lack of expressiveness in other fields: Firmato [14] only permits the use of allow rules, FLIP [15] does not support the overlap between rule selectors, and NeTSPoC [16] does not allow to specify custom filters over the generated rules. Other languages are simpler but not expressive enough, e.g., HLFL [17] does not support port ranges in its syntax. A widely adopted user-friendly tool is Uncomplicated Firewall (UFW) [18], the default firewall tool for Ubuntu. However, in this tool the order of the rules is relevant, in contrast to our solution. Shorewall [19] is a quite mature and flexible tool that similarly to MIGNIS tries to express firewall policies at a higher level of abstraction avoiding complex control-flow. The idea is that each *zone* has a policy and rules are exceptions to these policies. However, Shorewall is not as declarative as MIGNIS since rules are still evaluated one after the other. Moreover, configurations are spread into different files which makes it harder than in MIGNIS to have a general view of what the policy is. Pyroman [20] is a tool developed to configure the firewall of the Center for Digital Technology and Management in Munich. However, given that its language is very close to the Netfilter configuration language, it is not easy to manage. Graphical tools, such as kmyfirewall [21], firestarter [22] or the more complete Firewall builder [23], suffer from the same shortcoming of language-based tools in either not being expressive enough for the sake of simplicity or exposing low-level characteristics of Netfilter, such as the evaluation order of chains.

There are proposals for representing access control policies as XML documents [24], [25]. These approaches are interesting but too complex given that they are meant to be applicable in more general security policies and applications.

Systems for the management of large scale infrastructures such as Chef [26], LCFG [27] and Puppet [28], usually offer facilities to automatically deploy host-based firewalls on machines, providing configuration templates or allowing for the specification of basic rules. In both cases, there is no formal semantics for the specification nor any attempt to transform or verify the provided rules.

Bottom-up approaches that extract the model of the access control policy from the configuration files of the firewalls and allow one to reason about their application have been proposed [29], [30], [31], [32]. We instead take a top-down approach that allows one to write simple, easy to modify firewall rules that are translated into a concrete Netfilter firewall configuration.

Finally, there are other interesting low level open-source

firewalls such as ipfilter [33], Packet Filter (PF) [34], and pfSense [35]. They were developed for Unix like systems, OpenBSD and FreeBSD operating system respectively, and provide stateful firewalls and network address translation (NAT). They have low-level iptables-like rules, thus are not very easy to manage. In our opinion a mapping from the MIGNIS to all these firewalls is possible given their stateful nature and their handling of NATs. This translation is an on-going work.

This paper is organized as follows: in Section II we provide background on Netfilter, and in Section III we give a formal model of Netfilter chains, rules and packets. In Section IV we introduce a simple firewall language that abstracts Netfilter chains and the order in which rules are applied, and in Section V we provide a translation of this language into Netfilter model. In Section VI we illustrate MIGNIS, a publicly available tool that translates firewall specifications into real Netfilter configurations, and we briefly illustrate and discuss the DAIS Department firewall configuration. We conclude in Section VII.

II. BACKGROUND ON NETFILTER

Netfilter is a standard framework for packet filtering in Linux kernels [8]. It provides `iptables`, a tool that enables configuring Netfilter for packet filtering, network address translation (NAT) and packet mangling. The extreme flexibility of `iptables` makes it very powerful but also non-trivial to use. In fact, Netfilter configurations are inspected following a flow that allows for jumping into subsets of rules, going back to the 'callee', similarly to what happens in function calls, exiting when a decision on the packet is taken, and applying a default policy when no matching rule is found. In this section we present the basic notions behind packet filtering and NAT with `iptables`. Readers familiar with it can safely skip this section.

Chains and tables: Netfilter is based on *tables*, each containing lists of *rules* called *chains*. We focus on the three most commonly used tables: `mangle` for packet alteration, `nat` for NATs, and `filter` for packet filtering. There are five predefined chains that are inspected in specific moments of a packet life cycle:

(i) `PreRouting`, as soon as the packet reaches the host;
(ii) `Forward`, when the packet is routed through the host;
(iii) `PostRouting`, when the packet is about to leave the host;
(iv) `Input`, when packets are routed to the host;
(v) `Ouput`, when packets are generated by the host.

We have four possible execution flows: *(i)* $\longrightarrow$ *(ii)* $\longrightarrow$ *(iii)*, for packets passing through the host; *(i)* $\longrightarrow$ *(iv)*, for packets routed to the host; *(v)* $\longrightarrow$ *(iii)* for packets generated by the host and *(v)* $\longrightarrow$ *(iii)* $\longrightarrow$ *(i)* $\longrightarrow$ *(iv)* for packets that are both generated by the host and routed to the host.

Tables do not necessarily contain all of the above predefined chains: table `mangle` contains all of them; `nat` contains only `PreRouting`, `Input` (since kernel version 2.6.34), `Ouput`,
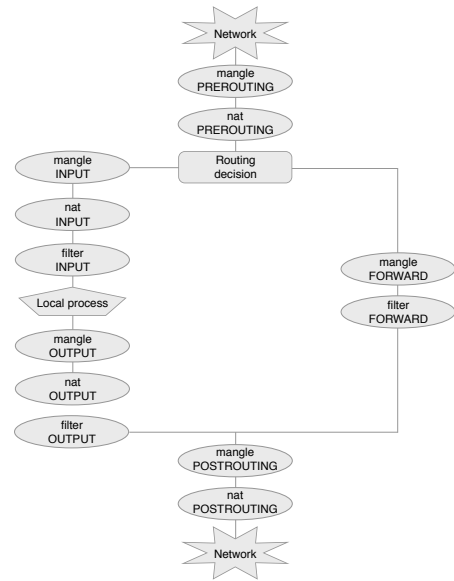

Fig. 1. Chain and table traversal.

and `PostRouting`, whereas `filter` contains only `Input`, `Forward` and `Ouput`. Tables are inspected in the following order: `mangle`, `nat`, `filter`. Figure 1 summarizes the resulting execution flow. After `mangle-PreRouting` and `nat-PreRouting` there is a branching and a packet is either processed through the `mangle-Input` and `filter-Input` chains if it is addressed to the local host, or through the `mangle-Forward` and `filter-Forward` chains if it is addressed to another host. Packets forwarded to other hosts are then inspected by `mangle-PostRouting` and `nat-PostRouting`. The same happens to packets originated by the local host which are first processed through `mangle-Ouput`, `nat-Ouput` and `filter-Ouput`.

Rules: Chains are lists of rules that are inspected one after the other. Rules specify criteria to match a packet and a target. If the packet does not match the criteria, the next rule in the chain is examined; if it does match, then the packet is processed as specified in the target. Here we only consider a subset of targets: `ACCEPT`, for accepting the packet, `DROP`, for dropping it, `DNAT`, for destination NAT, and `SNAT` for source NAT. Chains may also have a default policy that is triggered if none of the rules in the chain matches the packet.

Example 1 (Default drop policy)**.** *By default Netfilter does not filter packets. We can list chains of a specific table using options `-t table` and `-L`. Here we inspect table `filter`:*

```
# iptables -t filter -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

All chains are empty as rules would appear under the fields `target`, `prot`, etc. Moreover, the default policy is `ACCEPT`.

3

To change the default policy we can use option `-P` *as follows (we omit option* `-t filter` *as* `filter` *is the default table):*

```
# iptables -P INPUT DROP
# iptables -L
Chain INPUT (policy DROP)
...
```

The default policy for the `Input` *chain in table* `filter` *has been changed to* DROP. *When we ping* `localhost` *we find that all packets are lost since the* `Input` *chain is inspected when packets are routed to the local host:*

```
#  ping -v localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
^C
--- localhost ping statistics ---
31 packets transmitted,0 received,100% packet loss ...
```

Example 2 (Simple accept rule). *By setting the default policy for the* `Input` *chain in table* `filter` *to* DROP *we forbid any external connection to the host. To enable ssh connections it is enough to add (`-A chain` option) an* ACCEPT *rule in the same chain and table specifying 22 as the destination port:*

```
# iptables -A INPUT -p tcp --dport 22 -j ACCEPT
# iptables -L
Chain INPUT (policy DROP)
target    prot   opt   source       destination
ACCEPT    tcp    --    anywhere     anywhere     tcp dpt:ssh
...
```

We can see that we now have a rule that accepts any tcp packet with destination port 22, i.e., ssh and in fact it is now possible to connect to the host via ssh. It is worth mentioning that adding now another similar rule but with target DROP *to the end of this chain would still allow the ssh connections as rules are inspected in the order in which they appear in the chain.*

Example 3 (NATs). *Address translation is useful in various situations. For example, when accessing the Internet from an internal LAN, the source private address is translated into a public IP and all packets received back are translated and forwarded to the originating address (source NAT). Another example is a router that redirects external requests to a certain port of a web server located in the internal LAN (destination NAT). Destination NATs are defined in the* `nat` *table and in the* `PreRouting` *chain as follows:*

```
iptables -t nat -A PREROUTING -p tcp -i eth0 -d 192.168.0.1
       --dport 80 -j DNAT --to-destination 192.168.0.100:80
```

In this example, all connections from interface `eth0` *directed to* 192.168.0.1 *on port 80 are translated into* 192.168.0.100 *on the same port. So any web connection to the first host will be redirected to the second. Interestingly, answers from host 100 will be translated as coming from host 1 so that they will transparently reach the browser that initiated the connection. In fact, NAT rules are inspected only when initiating new connections. The translations are then stored and suitably applied to all packets belonging to the same connection. We illustrate this below.*

Conntrack: Netfilter uses the `conntrack` module to keep track of the established connections. The module stores the source and destination address of the packet that initiated the connection plus the source and destination address of the expected answer. This allows for correctly dealing with NATs.

Example 4 (Tracking destination NATs). *For the destination NAT of Example 3,* `conntrack` *records the following information:*

```
# conntrack -L
...
tcp   6 431994 ESTABLISHED
src= 10.0.0.1      dst=192.168.0.1   sport=49303    dport=80
src=192.168.0.100 dst=10.0.0.1       sport=80       dport=49303
```

We can see that the packet originated by 10.0.0.1 *and addressed to* 192.168.0.1 *on port 80 expects an answer from* 192.168.0.100 *addressed to* 10.0.0.1. *This information is enough to apply the suitable translation to all subsequent packets of this connection: for packets originated by* 10.0.0.1 *Netfilter translates (again) the destination into* 192.168.0.100; *for packets coming from* 192.168.0.100 *and addressed to* 10.0.0.1 *Netfilter translates the source address to* 192.168.0.1, *behaving this time as a source NAT. This latter translation will be done automatically in* `PostRouting` *as an effect of the established destination NAT.*

III. A FORMAL MODEL OF NETFILTER

We now give a formal model for a significant subset of Netfilter covering all of the predefined chains: `PreRouting`, `Forward`, `PostRouting`, `Input` and `Ouput`.

A. Modelling conntrack

As explained in Section II, the `conntrack` module keeps track of the active connections relating the source/destination IP of a packet with the source/destination of the expected response. Our abstraction of a state s is very close to the real implementation of `conntrack`. A state s is a set of tuples $(src, dst, id, src', dst', id')$ where src, dst are the source and destination addresses of the initial packet, src', dst' are the source and destination of the expected answer, and id, id' model protocol-specific information that is used to track established connections (e.g., the unique identifier of ICMP echo requests and replies). By recording this information in the state, the `conntrack` module is able to perform all the `PreRouting` and `PostRouting` translations. For example, if host a starts a new `tcp` connection with host b through a source NAT with address c, then the state is enriched with the tuple $(a, b, \_, b, c, \_)$ as the expected answer should be a packet from b to c. When the answer arrives, we have enough information to transparently forward it to a.

We do not abstract packets since, as we will see later, our model allows for any constraint that is expressible in `iptables` syntax. Given a concrete packet p, we write $sa(p)$, $da(p)$, $id(p)$ to denote respectively the source and destination addresses and identifier (if applicable) of p. When receiving p one may check whether or not it belongs to an already established connection by searching for tuples with matching src, dst, id or src', dst', id'. Moreover, if p belongs to an established connection, one may also determine the source and destination of its expected answer.

Definition 1 (Established connections). *A packet p belongs to an established connection in s, denoted $p \vdash_s src, dst$, if one of the following holds:*

$$i) \quad (sa(p), da(p), id(p), src, dst, id) \in s$$
$$ii) \quad (src, dst, id, sa(p), da(p), id(p)) \in s$$

where src, dst are the source and destination addresses of the expected answer to p.

In the following we will write $p \vdash_s$ to denote $p \vdash_s src, dst$ for some src and dst, and we will write $p \nvdash_s$ to denote that $p \vdash_s$ does not hold.

B. Modelling chains, tables and rules

A firewall is implemented as a series of chains, tables and rules that are executed in a given order. Let us first consider rules that do not perform address translation. For the sake of simplicity we only consider targets ACCEPT and DROP.

Rules are defined in terms of address ranges n. An address range n is a pair consisting of a set of IP addresses and a set of ports, denoted $IP(n):port(n)$. Given an address $addr$ we write $addr \in n$ to denote $port(addr) \in port(n)$, if $port(addr)$ is defined, and $IP(addr) \in IP(n)$. Notice that if $addr$ does not specify a port (for example in ICMP packets) we only check if the IP address is in the range. We will use the wildcard $*$ to denote any possible address or port.

Definition 2 (Basic rule). *A basic rule r is a tuple (n_1, n_2, ϕ, t) where n_1 and n_2 are respectively the ranges of source and destination addresses, ϕ is a formula over a packet and a state, and $t \in \{\text{ACCEPT}, \text{DROP}\}$ is the target of the rule.*

More specifically, a formula ϕ is any option expressible in iptables that is checkable over a packet and possibly a state, excluding options that:

- alter the packets, except for address translation that we support explicitly in special *translation rules* defined below; other packet alterations are not supported at the moment;
- alter the connection state which, to the best of our knowledge, is not possible in core iptables modules;
- alter the chains, as we do not want chains to change *on-the-fly*. For example option -F would flush a chain removing all the existing rules and altering the semantics of the firewall.

Example 5. *The iptables rule of Example 2:*

```
# iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

is modelled as ($*$:$*$, $*$:22, -p tcp, ACCEPT). We notice that --dport 22 is the only constraint on addresses and is included in the second component $*$:22. In this case, formula ϕ is -p tcp.

We extend this to consider rules that perform address translation. We still formalize rules as quadruples but in this case, instead of a target, we specify the address range on which we perform the translation. This uniform notation for rules will simplify our semantics.

Definition 3 (Translation rule). *A translation rule r is a tuple (n_1, n_2, ϕ, t) where n_1 and n_2 are respectively the ranges of source and destination addresses, ϕ is a formula over a packet and a state, and t is the range to which the addresses are translated. We abusively call this range of addresses t the target of the rule.*

Example 6. *The iptables rule of Example 3:*

```
iptables -t nat -A PREROUTING -p tcp -i eth0 -d 192.168.0.1
        --dport 80 -j DNAT --to-destination 192.168.0.100:80
```

is modelled as ($*$:$*$, 192.168.0.1:80, -p tcp, 192.168.0.100:80) assuming that eth0 is connected to the Internet, i.e., all source addresses are possible.

A packet p matches a rule r in a state s whenever its source and destination addresses are in the specified ranges and ϕ holds.

Definition 4 (Rule match). *Given a rule $r_i = (n_1, n_2, \phi, t)$ we say that p matches r_i in state s, denoted $p, s \models_r r_i$, if $sa(p) \in n_1$, $da(p) \in n_2$ and $\phi(p, s)$.*

We can now define how a packet is processed given a list of rules. Similarly to real implementations of Netfilter we inspect rules one after the other until we find a matching one, which establishes the destiny (target) of the packet.

Definition 5 (Rule list match). *Given a rule list $R = [r_1, \dots, r_n]$, we say that p matches R in state s with target t, denoted $p, s \models_R t$, if*

$$\exists i \le n \,.\, r_i = (n_1, n_2, \phi, t) \wedge p, s \models_r r_i \wedge \forall j < i \,.\, p, s \nvDash_r r_j.$$

We also write $p, s \nvDash_R$ if p does not match any of the rules in R, formally $\forall r_i \in R \,.\, p, s \nvDash_r r_i$.

We can now define a Netfilter firewall as a set of lists of rules, each corresponding to a chain in a certain table. For the nat table we have lists of translation rules that we denote by T. Albeit standard descriptions of Netfilter do not include Input chain in nat table this is part of Netfilter since kernel version 2.6.34, and we have decided to include it in our model.

Definition 6 (Netfilter Firewall). *A Netfilter firewall $\mathcal{F}$ is composed of twelve lists of rules L_{PRE}^{man}, T_{PRE}^{nat}, L_{INP}^{man}, T_{INP}^{nat}, L_{INP}^{fil}, L_{OUT}^{man}, T_{OUT}^{nat}, L_{OUT}^{fil}, L_{FOR}^{man}, L_{FOR}^{fil}, L_{POST}^{man}, T_{POST}^{nat}, where the subscript represents the chain name and the superscript the table name. L lists are composed of basic rules while T lists are composed of translation rules.*

C. Netfilter firewall semantics

We can now define how a packet p is dealt by a Netfilter firewall $\mathcal{F}$ in a state s. The semantics is presented in Table I. To distinguish this semantics from the abstract and intermediate ones introduced in the following sections, we index rule names and transitions with ll (meaning low-level). Semantics is given in terms of three different transition relations.

The first relation is $(s, p) \downarrow_{ll}^{\delta} \tilde{p}$, meaning that p is accepted as $\tilde{p}$ by chain δ in state s. This is where destination NAT happens and δ can be either PRE or OUT. Rule DEST_{ll} is applied to

$$\frac{p \vdash_s src, dst}{(s,p) \downarrow_{ll}^{\delta} p[da \mapsto src]} \; [\text{DEst}_{ll}] \qquad \frac{p \nvdash_s \quad p, s \nvDash_{T_{\delta}^{nat}}}{(s,p) \downarrow_{ll}^{\delta} p} \; [\text{DNew}_{ll}]$$

$$\frac{p \nvdash_s \quad p, s \vDash_{T_{\delta}^{nat}} t \quad dst \in t}{(s,p) \downarrow_{ll}^{\delta} p[da \mapsto dst]} \; [\text{DNAT}_{ll}]$$

$$\frac{p \vdash_s src, dst}{(s,p,\tilde{p}) \downarrow_{ll}^{\sigma} \tilde{p}[sa \mapsto dst]} \; [\text{SEst}_{ll}] \qquad \frac{p \nvdash_s \quad \tilde{p}, s \nvDash_{T_{\sigma}^{nat}}}{(s,p,\tilde{p}) \downarrow_{ll}^{\sigma} \tilde{p}} \; [\text{SNew}_{ll}]$$

$$\frac{p \nvdash_s \quad \tilde{p}, s \vDash_{T_{\sigma}^{nat}} t \quad src \in t}{(s,p,\tilde{p}) \downarrow_{ll}^{\sigma} \tilde{p}[sa \mapsto src]} \; [\text{SNAT}_{ll}]$$

$$\frac{\begin{array}{c} sa(p) \notin \mathcal{L} \quad da(\tilde{p}) \notin \mathcal{L} \quad p, s \vDash_{L_{\text{PRE}}^{man}} \text{ACCEPT} \quad (s,p) \downarrow_{ll}^{\text{PRE}} \tilde{p} \\ \tilde{p}, s \vDash_{L_{\text{FOR}}^{man}} \text{ACCEPT} \quad \tilde{p}, s \vDash_{L_{\text{FOR}}^{fil}} \text{ACCEPT} \\ \tilde{p}, s \vDash_{L_{\text{POST}}^{man}} \text{ACCEPT} \quad (s,p,\tilde{p}) \downarrow_{ll}^{\text{POST}} p' \end{array}}{s \xrightarrow{p,p'}_{ll} s \uplus (p,p')} \; [\text{Forward}_{ll}]$$

$$\frac{\begin{array}{c} sa(p) \notin \mathcal{L} \quad da(\tilde{p}) \in \mathcal{L} \quad p, s \vDash_{L_{\text{PRE}}^{man}} \text{ACCEPT} \quad (s,p) \downarrow_{ll}^{\text{PRE}} \tilde{p} \\ \tilde{p}, s \vDash_{L_{\text{INP}}^{man}} \text{ACCEPT} \quad (s,p,\tilde{p}) \downarrow_{ll}^{\text{INP}} p' \quad p', s \vDash_{L_{\text{INP}}^{fil}} \text{ACCEPT} \end{array}}{s \xrightarrow{p,p'}_{ll} s \uplus (p,p')} \; [\text{Input}_{ll}]$$

$$\frac{\begin{array}{c} sa(p) \in \mathcal{L} \quad da(\tilde{p}) \notin \mathcal{L} \quad p, s \vDash_{L_{\text{OUT}}^{man}} \text{ACCEPT} \quad (s,p) \downarrow_{ll}^{\text{OUT}} \tilde{p} \\ \tilde{p}, s \vDash_{L_{\text{OUT}}^{fil}} \text{ACCEPT} \quad \tilde{p}, s \vDash_{L_{\text{POST}}^{man}} \text{ACCEPT} \quad (s,p,\tilde{p}) \downarrow_{ll}^{\text{POST}} p' \end{array}}{s \xrightarrow{p,p'}_{ll} s \uplus (p,p')} \; [\text{Output}_{ll}]$$

$$\frac{\begin{array}{c} sa(p) \in \mathcal{L} \quad da(\tilde{p}) \in \mathcal{L} \quad p, s \vDash_{L_{\text{OUT}}^{man}} \text{ACCEPT} \quad (s,p) \downarrow_{ll}^{\text{OUT}} \tilde{p} \\ \tilde{p}, s \vDash_{L_{\text{OUT}}^{fil}} \text{ACCEPT} \quad \tilde{p}, s \vDash_{L_{\text{POST}}^{man}} \text{ACCEPT} \quad (s,p,\tilde{p}) \downarrow_{ll}^{\text{POST}} p' \\ p', s \vDash_{L_{\text{PRE}}^{man}} \text{ACCEPT} \quad p', s \vDash_{L_{\text{INP}}^{man}} \text{ACCEPT} \quad p', s \vDash_{L_{\text{INP}}^{fil}} \text{ACCEPT} \end{array}}{s \xrightarrow{p,p'}_{ll} s \uplus (p,p')} \; [\text{Local}_{ll}]$$

TABLE I
NETFILTER FIREWALL SEMANTICS.

packets that are already defined in the state. These do not go through the T_{δ}^{nat} table as their destination address is already recorded in the state; we translate the destination address of these packets into the source address of their expected answer, since this is how they should be delivered. Rule DNew_{ll} states that new packets for which there is no rule in T_{δ}^{nat} are not translated; rule DNAT_{ll} on the contrary states that for new packets for which there is a translation rule in T_{δ}^{nat}, the destination address is translated to some dst belonging to the range of possible destinations t.

Relation $(s,p,\tilde{p}) \downarrow_{ll}^{\sigma} p'$, means that p, previously translated into $\tilde{p}$ in chains PRE or OUT, is now accepted as p' by chain σ in state s. This models source NAT and in this case σ is either POST or INP. This relation is defined by rules SEst_{ll}, SNew_{ll} and SNAT_{ll} that are dual of the previous ones.

Finally, relation $s \xrightarrow{p,p'}_{ll} s'$ represents the state transition from s to s' where packet p is accepted and translated into p'. This relation is defined through four rules corresponding to the four possible execution flows discussed in Section II: Forward_{ll} is for packets that pass through the host; Input_{ll} is for packets routed to the host; Output_{ll} for packets generated by the host; and Local_{ll} for the ones both generated by and routed to the host.

We denote by $\mathcal{L}$ the set of addresses corresponding to the host, typically 127.0.0.1 and all the IPs of the various interfaces with no port restriction. Given a packet p and its translation $\tilde{p}$ after destination NAT, we select one of the four rules depending on whether or not p and $\tilde{p}$ belong to $\mathcal{L}$. For example, rule Forward_{ll} is applied whenever $sa(p) \notin \mathcal{L}$ and $da(\tilde{p}) \notin \mathcal{L}$, while rule Output_{ll} is applied whenever $sa(p) \in \mathcal{L}$ and $da(\tilde{p}) \notin \mathcal{L}$.

Our semantics assumes a default DROP policy, i.e., a packet is dropped if not explicitly accepted by some basic rule in each of the traversed chains. Address translation is instead optional: if there is no matching translation rule the packet is delivered unchanged. Rule Forward_{ll} states that a packet p is accepted as p' if it is:

(1) accepted by mangle-PreRouting list L_{PRE}^{man};
(2) possibly transformed into $\tilde{p}$ by nat-PreRouting list T_{PRE}^{nat}, or by the fact that it belongs to an established connection subject to NAT (relation $(s,p) \downarrow_{ll}^{\text{PRE}} \tilde{p}$);
(3) accepted as $\tilde{p}$ by mangle-Forward, filter-Forward and mangle-PostRouting lists L_{FOR}^{man}, L_{FOR}^{fil} and L_{POST}^{man}; and
(4) possibly translated from $\tilde{p}$ into p' by list T_{POST}^{nat}, or when on an established connection (relation $(s,p,\tilde{p}) \downarrow_{ll}^{\text{POST}} p'$).

State update is achieved through the partial function $s \uplus (p,p')$ that yields:

- s if $p \vdash_s$
- $s \cup \{(sa(p), da(p), id(p), sa(p_{ans}), da(p_{ans}), id(p_{ans}))\}$ if $p \nvdash_s$ and $p_{ans} \nvdash_s$ where p_{ans} is the expected answer to p, i.e., $p_{ans} = p'[sa \mapsto da(p'), da \mapsto sa(p')]$

Intuitively, when p belongs to an established connection the state is not modified and $s' = s$. When instead p does not belong to any established connection we extend the state with a new tuple in which the source and destination addresses of p' are swapped, as we want to store the source and destination addresses of the expected answer p_{ans} to p. The check $p_{ans} \nvdash_s$ avoids that the new tuple overlaps with existing connections. This models the fact that in real scenarios it is not possible to connect from/to a busy port or to reuse an already taken connection identifier.

The other three rules model the corresponding three execution paths similarly to Forward_{ll}. Notice that local communication only performs address translation in chains Ouput and PostRouting: when the packet comes in, the chains PreRouting and Input of table nat are not inspected. Rule Local_{ll} correctly mimics this behaviour.

To illustrate the application of rule Forward_{ll} and state update let us consider the following example.

Example 7. *Consider the destination NAT rule from Example 6 in chain* PreRouting *of table* nat *and let*

$$T_{\text{PRE}}^{nat} = [(*{:}*, 192.168.0.1{:}80, \text{-}p\ tcp, 192.168.0.100{:}80)]$$

We assume that all other translation lists T are empty, and that all basic lists L are $[(*, *, \top, \text{ACCEPT})]$, that is, they accept

every packet. Moreover, let $\mathcal{L} = \{127.0.0.1{:}*, 192.168.0.1{:}*\}$ be the set of addresses of our firewall.

Consider now an empty state s and a packet p coming from $sa(p) = 1.2.3.4{:}5678$ on the Internet and addressed to $da(p) = 192.168.0.1{:}80$. Let us for readability write such packet as $p = (1.2.3.4{:}5678 \rightsquigarrow 192.168.0.1{:}80)$.

Since $sa(p) \notin \mathcal{L}$ we can only apply either $Forward_{ll}$ or $Input_{ll}$. Since state s is empty, thus $p \nvdash_s$, and given the rule above, we have $p, s \models_{T_{\text{PRE}}^{nat}} 192.168.0.100{:}80$ which by application of $DNAT_{ll}$, gives $(s, p) \downarrow_{ll}^{\text{PRE}} \tilde{p}$ for $\tilde{p} = p[da \mapsto 192.168.0.100{:}80] = (1.2.3.4{:}5678 \rightsquigarrow 192.168.0.100{:}80)$. Intuitively, $\tilde{p}$ is packet p after destination NAT.

Since $da(\tilde{p}) \notin \mathcal{L}$ we apply rule $Forward_{ll}$ and this packet will be forwarded to the web server $192.168.0.100$. Also, since T_{POST}^{nat} is empty, rule $SNew_{ll}$ gives $(s, p, \tilde{p}) \downarrow_{ll}^{\text{POST}} p'$ with $p' = \tilde{p}$.

Finally, $L_{\text{PRE}}^{man}, L_{\text{FOR}}^{man}, L_{\text{POST}}^{man}, L_{\text{FOR}}^{fil}$ accept every packet and so we have $s \xrightarrow{p, p'}_{ll} s'$ with

$$s' = \{ (1.2.3.4{:}5678, 192.168.0.1{:}80, \_, \\ 192.168.0.100{:}80, 1.2.3.4{:}5678, \_) \}$$

Consider now reply $p_r = (192.168.0.100{:}80 \rightsquigarrow 1.2.3.4{:}5678)$ from the web server. Notice that these addresses match the second part of the tuple in s' meaning that p_r is in an established connection and so $p_r \vdash_{s'} 1.2.3.4{:}5678, 192.168.0.1{:}80$. By rules $DEst_{ll}$ we obtain $(s, p_r) \downarrow_{ll}^{\text{PRE}} p_r[da \mapsto 1.2.3.4{:}5678] = p_r$, and by rule $SEst_{ll}$ $(s, p_r, p_r) \downarrow_{ll}^{\text{POST}} p_r'$ with $p_r' = p_r[sa \mapsto 192.168.0.1{:}80] = (192.168.0.1{:}80 \rightsquigarrow 1.2.3.4{:}5678)$ which gives $s' \xrightarrow{p_r, p_r'}_{ll} s'$.

Intuitively, the established connection in s' applies a source NAT to the reply packets so that they go out with the source address that was initially contacted by $1.2.3.4$, i.e., $192.168.0.1{:}80$. The state s' is unchanged because this answer belongs to an established connection.

D. Maintaining Netfilter configurations

We have shown that the semantics of Netfilter is based on inspecting rules of various chains and tables in the order they appear. This makes understanding and modifying a configuration a non-trivial task. For example, to discover what a certain host `h` can do, one cannot inspect the rules involving `h` in isolation as the order and the context in which they appear is relevant. At the same time, if we need to modify the policy for host `h` it is neither immediate to understand which rule to add nor where to add it as, even in this case, the semantics depends on the context. Adding a new rule has implications on the subsequent ones and might change the existing policy in a subtle way.

Example 8 (Understanding a configuration). *Consider the following Netfilter configuration where* `U1`, `U2` *and* `U3` *are ranges of IP addresses. We let L_{FOR}^{fil} be the following list of rules:*

1) *(*`U1`*:*, *:*, -p tcp,* `ACCEPT`*)*
2) *(*:*, *:139, -p tcp,* `DROP`*)*
3) *(*`U2`*:*, *:*, -p tcp,* `ACCEPT`*)*
4) *(*:*, *:21, -p tcp,* `DROP`*)*
5) *(*`U3`*:*, *:*, -p tcp,* `ACCEPT`*)*

Intuitively, tcp packets from IPs in `U1` *can go everywhere (Rule 1); tcp packets directed to port 139 are dropped (Rule 2); tcp packets from IPs in* `U2` *are accepted (Rule 3); tcp packets directed to port 21 are dropped (Rule 4); and tcp packets from IPs in* `U3` *are accepted (Rule 5).*

One relevant question for system administrators is to know which packets are allowed to pass through a firewall and which ones are discarded. To obtain this information, and since the order of the rules matters, if one wants to know, for example, what happens to packets coming from `U3` *it would be necessary to consider the context and analyze the previous (overlapping) rules. In fact, tcp packets from IPs in* `U3` *that are neither in* `U1` *nor* `U2` *are accepted only if they are not directed to ports 21 and 139. Packets from IPs in* `U3` *that are also in* `U2` *but not in* `U1` *are accepted only if they are not directed to 139, and so on. This complicates even further when rules belong to different chains and tables as the entire flow and possible translations have also to be considered.*

Example 9 (Maintaining a configuration). *Suppose now we want to perform the following changes to the configuration of Example 8: (a) allow all communications from IPs in* `U4`*, (b) allow all communications from IPs in* `U5` *except those addressed to port 21.*

Once we understand the configuration, (a) is relatively easy to add since it is enough to place an `ACCEPT` *rule before any* `DROP` *rule, i.e., before Rule 2. As for (b), achieving it is not as immediate since packets with destination 139 are dropped before the ones with destination 21. So if we place a new* `ACCEPT` *rule after Rule 4 we are also dropping packets with destination 139. A solution for this is to add a specific* `DROP` *rule for packets coming from* `U5` *with destination 21, as follows (new rules are 2, 3 and 4):*

1) *(*`U1`*:*, *:*, -p tcp,* `ACCEPT`*)*
2) *(*`U4`*:*, *:*, -p tcp,* `ACCEPT`*)*
3) *(*`U5`*:*, *:21, -p tcp,* `DROP`*)*
4) *(*`U5`*:*, *:*, -p tcp,* `ACCEPT`*)*
5) *(*:*, *:139, -p tcp,* `DROP`*)*
6) *(*`U2`*:*, *:*, -p tcp,* `ACCEPT`*)*
7) *(*:*, *:21, -p tcp,* `DROP`*)*
8) *(*`U3`*:*, *:*, -p tcp,* `ACCEPT`*)*

Notice that the insertion of rules 2, 3, and 4 affect the policy for IPs in `U2` *and* `U3` *if they also belong to* `U4` *or* `U5`*. The above configuration, even if extremely small, is far from being easy to read and maintain.*

The language proposed in the next section greatly simplifies the understanding and maintenance of firewall policies. Due to its declarative style, it makes configurations trivial to understand and allows one to add new requisites and to enforce them regardless of the place where the corresponding rules are placed.

IV. FIREWALL SPECIFICATION LANGUAGE

In order to simplify the specification of firewall policies, we propose a simple *firewall specification language* that abstracts the many different chains existing in Netfilter and the order in which rules are applied. We later provide a translation

of this simple language into a Netfilter firewall and show that, under reasonable assumptions, this translation preserves the semantics, that is, firewall configurations written in this language can be translated into equivalent configurations for (our model of) Netfilter.

We emphasize that the declarative nature of our firewall specification language in which the order of rules does not matter, makes the specification of a firewall very close to its semantics: a packet goes through (possibly translated) only if it matches a positive rule and is not explicitly denied. This allows administrators to write and inspect rules independently of the order in which they are specified. Moreover, the declarative approach makes it easy to detect possible conflicts and redundancies, as we will discuss in Section VI, and to query for a subset of the specification involving specific hosts, as we exemplify below.

A. Firewall language syntax

The firewall language has the following syntax:

$$
\begin{aligned}
r \quad ::= \quad & \mathbf{n}_1 \ / \ \mathbf{n}_2 \mid \phi && (\text{DROP}) \\
\mid \ & \mathbf{n}_1 > \mathbf{n}_2 \mid \phi && (\text{ACCEPT}) \\
\mid \ & \mathbf{n}_1 > [\mathbf{n}_2] \ \mathbf{n}_\mathbf{t} \mid \phi && (\text{DNAT}) \\
\mid \ & \mathbf{n}_1 \ [\mathbf{n}_\mathbf{t}] > \mathbf{n}_2 \mid \phi && (\text{SNAT})
\end{aligned}
$$

Intuitively, the DROP rule $\mathbf{n}_1 \ / \ \mathbf{n}_2 \mid \phi$ forbids packets p that satisfy $\phi(p, s)$ to flow from $\mathbf{n}_1$ to $\mathbf{n}_2$, even through NATs, and has priority over any other rule. It applies to any packet p either in a new or in an established connection with $\mathbf{n}_1$ and $\mathbf{n}_2$ as communication endpoints, that is, after destination NAT and before source NAT are applied; the ACCEPT rule $\mathbf{n}_1 > \mathbf{n}_2 \mid \phi$ enables new connections from $\mathbf{n}_1$ to $\mathbf{n}_2$, as long as ϕ holds. We write $\mathbf{n}_1 > [\mathbf{n}_2] \ \mathbf{n}_\mathbf{t} \mid \phi$ to indicate that $\mathbf{n}_\mathbf{t}$ is behind a destination NAT (DNAT) and $\mathbf{n}_1$ may start new connections with it by sending a packet satisfying ϕ to an address in $\mathbf{n}_2$; we use $\mathbf{n}_1 \ [\mathbf{n}_\mathbf{t}] > \mathbf{n}_2 \mid \phi$ to indicate that $\mathbf{n}_1$ may start new source NAT (SNAT) connections with $\mathbf{n}_2$ sending packets satisfying ϕ and with source address $\mathbf{n}_\mathbf{t}$. Established connections are always allowed in both directions, unless they are explicitly dropped by DROP. A set of these firewall rules is called a *configuration*.

The firewall language allows administrators to specify firewalls in a purely declarative style. There is no control flow: new packets that match at least one of ACCEPT, DNAT, SNAT and established packets always go through unless they are explicitly dropped by DROP. Notice that, in most of the cases, DROP is not necessary since the same effect could be obtained by restricting the scope of the other rules in order to accept less packets. However, DROP is very useful in practice as it makes it possible, for example, to forbid all packets from/to specific hosts or ports independently of the many different ACCEPT, DNAT or SNAT rules these hosts and ports might match. This is the reason why we give higher priority to the DROP rule.

B. Firewall language semantics

We extend the notation of Section III to the firewall specification language. Let C be a configuration. We write $p, s \models_C \text{DROP}, \text{ACCEPT}, \text{DNAT}(\mathbf{n}_\mathbf{t}), \text{SNAT}(\mathbf{n}_\mathbf{t})$ to respectively

$$
\frac{p, s \models_C \text{ACCEPT} \quad p, s \not\models_C \text{DROP}}{(s, p) \downarrow^{hl} p} \ [\text{ACCEPT}_{hl}]
$$

$$
\frac{p, s \models_C \text{DNAT}(\mathbf{n}_\mathbf{t}) \quad dst \in \mathbf{n}_\mathbf{t} \quad p[da \mapsto dst], s \not\models_C \text{DROP}, \text{SNAT}}{(s, p) \downarrow^{hl} p[da \mapsto dst]} \ [\text{DNAT}_{hl}]
$$

$$
\frac{p, s \models_C \text{SNAT}(\mathbf{n}_\mathbf{t}) \quad src \in \mathbf{n}_\mathbf{t} \quad p, s \models_C \text{DROP}, \text{DNAT}}{(s, p) \downarrow^{hl} p[sa \mapsto src]} \ [\text{SNAT}_{hl}]
$$

$$
\frac{\begin{array}{c} p, s \models_C \text{DNAT}(\mathbf{n}_\mathbf{t}) \quad dst \in \mathbf{n}_\mathbf{t} \quad \tilde{p} = p[da \mapsto dst] \\ \tilde{p}, s \not\models_C \text{DROP} \quad \tilde{p}, s \models_C \text{SNAT}(\mathbf{n}'_\mathbf{t}) \quad src \in \mathbf{n}'_\mathbf{t} \end{array}}{(s, p) \downarrow^{hl} \tilde{p}[sa \mapsto src]} \ [\text{DSNAT}_{hl}]
$$

$$
\frac{p \not\vdash_s \quad (s, p) \downarrow^{hl} p'}{s \xrightarrow{p, p'}_{hl} s \uplus (p, p')} \ [\text{NEW}_{hl}]
$$

$$
\frac{\begin{array}{c} p \vdash_s src, dst \quad p[da \mapsto src], s \not\models_C \text{DROP} \\ p' = p[da \mapsto src, sa \mapsto dst] \end{array}}{s \xrightarrow{p, p'}_{hl} s} \ [\text{EST}_{hl}]
$$

TABLE II
FIREWALL LANGUAGE SEMANTICS.

denote that there exists in C rule $\mathbf{n}_1 \ / \ \mathbf{n}_2 \mid \phi$ or rule $\mathbf{n}_1 > \mathbf{n}_2 \mid \phi$ or rule $\mathbf{n}_1 > [\mathbf{n}_2] \ \mathbf{n}_\mathbf{t} \mid \phi$ or rule $\mathbf{n}_1 \ [\mathbf{n}_\mathbf{t}] > \mathbf{n}_2 \mid \phi$, such that $sa(p) \in \mathbf{n}_1, da(p) \in \mathbf{n}_2$, and $\phi(p, s)$. When no such matching rules exist we respectively write $p, s \not\models_C \text{DROP}, \text{ACCEPT}, \text{DNAT}, \text{SNAT}$.

Firewall language semantics is presented in Table II. Semantics is given in terms of two relations. Relation $(s, p) \downarrow^{hl} p'$ states that p is accepted as p' in state s by the firewall configuration. Differently from the low-level semantics of Section III, this relation is derived in one step from single or pairs (in the case of DSNAT) of firewall language rules. In particular, rule ACCEPT_{hl} applies when p matches an ACCEPT firewall rule of the form $\mathbf{n}_1 > \mathbf{n}_2 \mid \phi$ and it is not dropped. The packet is accepted with no translation. Rule DNAT_{hl} applies to $\mathbf{n}_1 > [\mathbf{n}_2] \ \mathbf{n}_\mathbf{t} \mid \phi$ and checks that the translated packet is not dropped neither has an applicable SNAT rule (otherwise rule DSNAT_{hl} should be used). In this case the packet is accepted as $p[da \mapsto dst]$ where $dst \in \mathbf{n}_\mathbf{t}$. Rule SNAT_{hl} applies to $\mathbf{n}_1 \ [\mathbf{n}_\mathbf{t}] > \mathbf{n}_2 \mid \phi$ and it is the dual of DNAT_{hl}. Notice that in this case we check that the original p (before source NAT) is not dropped: as we explained before, DROP forbids packets between the communication endpoints and so we check it after destination NAT and before source NAT. Rule DSNAT_{hl} simply combines the two rules above and is applied when there are matching DNAT and SNAT rules. This is useful when we want to have both source and destination NAT on the same packet.

Relation $s \xrightarrow{p, p'}_{hl} s'$ denotes the state transition and is given by two rules. Rule NEW_{hl} applies to any packet p that establishes a new connection and is accepted as p' by one of the previous rules. The state is updated so to include the new connection, using the same $\uplus$ operator of Table I. Rule EST_{hl} simply inspects the state for established connections and performs the corresponding translation as long as there is no

8

explicit rule dropping this packet (again, after destination NAT is applied).

Example 10. *Consider the destination NAT rule from Examples 6 and 7. In the firewall language this is specified as*

$$C = \{*{:}* > [192.168.0.1{:}80]\ 192.168.0.100{:}80 \mid \textit{-p tcp}\}$$

Differently from Example 7, we do not have to specify anything else: the firewall will only allow packets through this destination NAT and all the established answers to them.

Consider again an empty state s and the same packet $p = (1.2.3.4{:}5678 \rightsquigarrow 192.168.0.1{:}80)$ as in Example 7. We have $p, s \models_C \textsc{Dnat}(192.168.0.100{:}80)$ and clearly $p', s \not\models_C \textsc{Drop}, \textsc{Snat}$ for $p' = p[da \mapsto 192.168.0.100{:}80] = (1.2.3.4{:}5678 \rightsquigarrow 192.168.0.100{:}80)$ since there are no \textsc{Drop} nor \textsc{Snat} rules in C. By $\textsc{Dnat}_{hl}$ we obtain $(s, p) \downarrow^{hl} p'$ and thus $s \xrightarrow{p,p'}_{hl} s'$ with s' as in Example 7.

A possible reply $p_r = (192.168.0.100{:}80 \rightsquigarrow 1.2.3.4{:}5678)$ from the web server is now in an established connection, i.e. formally, $p_r \vdash_{s'} 1.2.3.4{:}5678, 192.168.0.1{:}80$. By rule $\textsc{Est}_{hl}$ we directly have $s' \xrightarrow{p_r,p'_r}_{hl} s'$ with $p'_r = p_r[sa \mapsto 192.168.0.1{:}80, da \mapsto 1.2.3.4{:}5678] = (192.168.0.1{:}80 \rightsquigarrow 1.2.3.4{:}5678)$.

Notice that for this specific packets the configuration C behaves like the one of Example 7. However these two firewalls are not equivalent: here the destination NAT is the only way to send a packet to the server and any other packet is rejected, apart from established ones. In Example 7 we configured all chains to accept any packet, making the firewall completely open.

To implement a Netfilter configuration that behaves like this single $\textsc{Dnat}$ rule in the firewall language we would need to add 3 more rules in the Netfilter chains as discussed in the next section.

C. Maintaining firewall configurations

We can now discuss how Examples 8 and 9 can be addressed in a simpler way in our new formalism. We use symbol \ to exclude addresses from a set. So, for example, we write $*{:}*\backslash(\textrm{p})$ to denote all addresses except the ones with port p.

Example 11. *The Netfilter configuration of Example 8 can be written in the firewall specification language as*

U1:*	>	*:*	| *-p tcp*
U2:*	>	*:*\(139)	| *-p tcp*
U3:*	>	*:*\(139, 21)	| *-p tcp*

This configuration has exactly the same semantics as the one in Example 8. The alternation of `DROP` *and* `ACCEPT` *rules in Example 8, that avoids filtering the ports in Rules 3 and 5, is in our case explicitly declared in the port restriction rules. In this way, our rules exactly state what is allowed and if one wants to know what happens to packets coming from an IP in* U3 *that is not in* U1 *nor* U2*, one only needs to extract from the configuration the rules in which such an IP appears, i.e.,* U3:* > *:*\(139, 21) | *-p tcp. If instead the IP is in* U3 *and in* U2 *but not in* U1 *we additionally get*

U2:* > *:*\(139) | *-p tcp. Contrary to the Netfilter model, we do not need to consider the context nor the complete flow of the packet through the firewall as in this case we have a single step semantics which is immediate to understand.*

Example 12. *As for maintenance, and since our language is order-independent, adding the requisites (a) and (b) of Example 9 results in adding those rules anywhere in the configuration:*

U1:*	>	*:*	| *-p tcp*
U2:*	>	*:*\(139)	| *-p tcp*
U3:*	>	*:*\(139, 21)	| *-p tcp*
U4:*	>	*:*	| *-p tcp*
U5:*	>	*:*\(21)	| *-p tcp*

which we can concisely write as:

(U1, U4):*	>	*:*	| *-p tcp*
U2:*	>	*:*\(139)	| *-p tcp*
U3:*	>	*:*\(139, 21)	| *-p tcp*
U5:*	>	*:*\(21)	| *-p tcp*

V. FROM THE FIREWALL LANGUAGE TO NETFILTER

In this section we show that the firewall specification language presented in Section IV can be translated into our Netfilter model of Section III, and that under reasonable conditions the accepted packets are the same. Since there are significant differences between the two approaches, e.g., Netfilter assumes ordering among the rules while the firewall language does not, we propose an intermediate step where the language is close to the one presented in Section III but where the ordering is still irrelevant. We then map the firewall language to this intermediate language, and finally this intermediate language to the Netfilter model.

All technical proofs can be found in [36].

Definition 7 (Intermediate Firewall). *An intermediate firewall $\mathcal{F}_\mathcal{I}$ is composed of five sets of rules $S_{D_1}, S_{DNAT}, S_D, S_A, S_{SNAT}$ respectively to drop packets before destination NAT, to perform destination NAT translations, to drop and accept connections, and finally to perform source NAT translations. Rules in S_{D_1}, S_D and S_A are basic rules while rules in S_{DNAT} and S_{SNAT} are translation rules (cf. Definitions 2 and 3). Moreover, rules in S_{D_1} and S_D have target* DROP *while rules in S_A have target* ACCEPT.

The following definition is similar to Definition 5 but works on sets instead of lists.

Definition 8 (Rule set match). *Given a set of rules $S = \{s_1, \ldots, s_n\}$, we say that p matches S in state s with target t, denoted $p, s \models_S^{il} t$, if*

$$\exists s_i \in S . s_i = (n_1, n_2, \phi, t) \wedge p, s \models_r s_i.$$

We also write $p, s \not\models_S^{il}$ if p does not match any of the rules in S, formally $\forall s_i \in S . p, s \not\models_r s_i$.

9

$$\frac{p \vdash_s src, dst}{(s,p) \downarrow_{il}^{\text{DNAT}} p[da \mapsto src]} \ [\text{DEst}_{il}] \qquad \frac{p \nvdash_s \quad p,s \not\models_{S_{\text{DNAT}}}^{il}}{(s,p) \downarrow_{il}^{\text{DNAT}} p} \ [\text{DNew}_{il}]$$

$$\frac{p \nvdash_s \quad p,s \models_{S_{\text{DNAT}}}^{il} t \quad dst \in t}{(s,p) \downarrow_{il}^{\text{DNAT}} p[da \mapsto dst]} \ [\text{DNAT}_{il}]$$

$$\frac{p \vdash_s src, dst}{(s,p,\tilde{p}) \downarrow_{il}^{\text{SNAT}} \tilde{p}[sa \mapsto dst]} \ [\text{SEst}_{il}] \qquad \frac{p \nvdash_s \quad \tilde{p},s \not\models_{S_{\text{SNAT}}}^{il}}{(s,p,\tilde{p}) \downarrow_{il}^{\text{SNAT}} \tilde{p}} \ [\text{SNew}_{il}]$$

$$\frac{p \nvdash_s \quad \tilde{p},s \models_{S_{\text{SNAT}}}^{il} t \quad src \in \texttt{t}}{(s,p,\tilde{p}) \downarrow_{il}^{\text{SNAT}} \tilde{p}[sa \mapsto src]} \ [\text{SNAT}_{il}]$$

$$\frac{\begin{array}{c} sa(p) \notin \mathcal{L} \vee da(\tilde{p}) \notin \mathcal{L} \\ p \vdash_s \vee p, s \not\models_{S_{D_1}}^{il} \quad (s,p) \downarrow_{il}^{\text{DNAT}} \tilde{p} \\ \tilde{p},s \not\models_{S_D}^{il} \quad p \vdash_s \vee \tilde{p},s \models_{S_A}^{il} \quad (s,p,\tilde{p}) \downarrow_{il}^{\text{SNAT}} p' \end{array}}{s \xrightarrow{p,p'}_{il} s \uplus (p,p')} \ [\text{Ext}_{il}]$$

$$\frac{\begin{array}{c} sa(p) \in \mathcal{L} \wedge da(\tilde{p}) \in \mathcal{L} \\ p \vdash_s \vee p, s \not\models_{S_{D_1}}^{il} \quad (s,p) \downarrow_{il}^{\text{DNAT}} \tilde{p} \\ \tilde{p},s \not\models_{S_D}^{il} \quad p \vdash_s \vee \tilde{p},s \models_{S_A}^{il} \quad (s,p,\tilde{p}) \downarrow_{il}^{\text{SNAT}} p' \\ p \vdash_s \vee p', s \not\models_{S_{D_1}}^{il} \quad p',s \not\models_{S_D}^{il} \quad p \vdash_s \vee p',s \models_{S_A}^{il} \end{array}}{s \xrightarrow{p,p'}_{il} s \uplus (p,p')} \ [\text{Local}_{il}]$$

A. Intermediate firewall semantics

Intermediate level firewall semantics is presented in Table III and similarly to the semantics presented in Table I for Netfilter is also given in terms of three different transition relations (indexed by il).

The first relation is $(s,p) \downarrow_{il}^{\text{DNAT}} \tilde{p}$, meaning that p is accepted in state s as $\tilde{p}$ after performing a destination NAT translation. It is similar to relation $(s,p) \downarrow_{ll}^{\delta} \tilde{p}$ of Table I using the set of translation rules S_{DNAT} instead of the list T_δ^{nat}. The second relation is $(s,p,\tilde{p}) \downarrow_{il}^{\text{SNAT}} p'$ and it means that p, previously translated into $\tilde{p}$ by some destination NAT, is now accepted in state s as p' after performing a source NAT translation. It is similar to relation $(s,p,\tilde{p}) \downarrow_{ll}^{\sigma} p'$ of Table I using the set of translation rules S_{SNAT} instead of the list T_σ^{nat}. Finally, relation $s \xrightarrow{p,p'}_{il} s'$ represents the state transition from s to s' where packet p is accepted and translated into p', and is defined through rules Ext_{il} and Local_{il}.

Rule Ext_{il} reflects the flow of non-local packets. Rules in S_{D_1} are all of type DROP and will be responsible for dropping packets that try to circumvent destination NATs. A packet to be accepted has then either to belong to an already established connection or to match none of these rules; if p passes this test it will be (possibly) translated into $\tilde{p}$ by some destination NAT. The resulting packet $\tilde{p}$ should not match any DROP rule in S_D, and is required to either match an ACCEPT rule in S_A, as the default policy is DROP, or the original packet to be in an established connection. If $\tilde{p}$ passes all these tests the source NAT translation produces the final p'.

Rule Local_{il} applies to local packets and is equal to rule

$$\frac{\mathbf{n_1 > n_2 \mid \phi}}{(\mathbf{n_1},\mathbf{n_2},\phi,\text{ACCEPT}) \in S_A} \qquad \frac{\mathbf{n_1 / n_2 \mid \phi}}{(\mathbf{n_1},\mathbf{n_2},\phi,\text{DROP}) \in S_D}$$

$$\frac{\mathbf{n_1 > [n_2]\,n_t \mid \phi}}{\begin{array}{c} (\mathbf{n_1},\mathbf{n_t},\phi,\text{DROP}) \in S_{D_1} \quad (\mathbf{n_1},\mathbf{n_2},\phi,\mathbf{n_t}) \in S_{\text{DNAT}} \\ (\mathbf{n_1},\mathbf{n_t},\phi^\star,\text{ACCEPT}) \in S_A \end{array}}$$

$$\frac{\mathbf{n_1\,[n_t] > n_2 \mid \phi}}{(\mathbf{n_1},\mathbf{n_2},\phi,\text{ACCEPT}) \in S_A \quad (\mathbf{n_1},\mathbf{n_2},\phi,\mathbf{n_t}) \in S_{\text{SNAT}}}$$

Ext_{il} except that the packet p' is then sent back to the local machine and has to pass the initial verifications. Notice that in this case we do not apply again destination nor source NATs as this is the expected behavior in Netfilter.

We stress three ideas from this intermediate semantics. The first is that packets belonging to established connections are always accepted unless explicitly dropped by a rule in S_D; the second is that we have all DROP rules (S_{D_1}, S_D) checked before the ACCEPT rules (S_A) and so acceptance implies the non-match of any of the DROP rules; and finally that we allow ourselves to consult the state of the original packet p even after performing a destination or a source NAT translation. This might look counter-intuitive but it is in fact what happens in real Netfilter semantics. This will become clearer once we define later in this Section the translation from this intermediate level to the Netfilter model.

B. From the firewall language into the intermediate level

We will now show that our abstract firewall language can be encoded into this intermediate level firewall. Translation is given by the least sets $S_{D_1}, S_D, S_A, S_{\text{DNAT}}$ and S_{SNAT} satisfying the rules of Table IV.

All translations in Table IV are intuitive except the DNAT one, where the rules added to S_A and S_{D_1} might need some explanation. For the first, one can easily see that if a packet p matches a DNAT rule $\mathbf{n_1 > [n_2]\,n_t \mid \phi}$, the formula ϕ that is matched by p before the translation should not be the same as the $\phi^\star$ that is matched by $\tilde{p}$ after the translation. As an example consider $\phi = (da(p) = IP(n_2))$. This formula is true before the translation, as p is addressed to $\mathbf{n_2}$, but is no longer true after the translation, where $\tilde{p}$ is now addressed to $\mathbf{n_t}$. Since we do not want to introduce any structure in ϕ, the most that we can require for this to be well-defined is $\phi^\star$ to be such that for any packet p, state s, $\mathbf{n_1 > [n_2]\,n_t \mid \phi} \in C$ and $dst \in \mathbf{n_t}$, we have $\phi(p,s)$ iff $\phi^\star(p[da \mapsto dst], s)$. This way we ensure that ϕ is somehow preserved by the destination NAT translation. A trivial way to enforce this condition is to require that the ϕ used in DNAT rules cannot refer $da(p)$ and define $\phi^\star = \phi$.

For the rule in S_{D_1} the *rationale* is that the ACCEPT rule added to S_A would make it possible to send packets directly from $\mathbf{n_1}$ to $\mathbf{n_t}$, that is more than what we want when defining the DNAT rule $\mathbf{n_1 > [n_2]\,n_t \mid \phi}$. To prevent this, we have to drop those packets before destination NAT is applied. However, and as a side effect, these rules in S_{D_1} will also

disable legitimate communications of rules $n_1 > n_t \mid \phi$ and $n_1 > [n_t] \, n_2' \mid \phi$ and $n_1 \, [n_2'] > n_t \mid \phi$ which poses a problem when we think about completeness.

While the soundness of our translation is always guaranteed, *i.e.*, if $s \xrightarrow{p,p'}_{il} s'$ then $s \xrightarrow{p,p'}_{hl} s'$, its completeness needs some extra conditions. In particular, one needs to exclude the same behaviors that are excluded by S_{D_1}.

We say that a configuration is *NAT-safe* if for every packet p, state s, and $n_1 > [n_2] \, n_t \mid \phi$ in C, if $p, s \models_C$ ACCEPT or $p, s \models_C$ DNAT(n_t') or $p, s \models_C$ SNAT(n_t') then $sa(p) \notin n_1 \vee da(p) \notin n_t \vee \neg\phi(p,s)$. Intuitively we are saying that if there is a DNAT with end-points n_1 and n_t, then there can be no other (non-drop) rule that allows direct access from n_1 to n_t.

The above condition can be syntactically enforced (although more restrictively) requiring $n_1 \cap n_1' = \emptyset$ or $n_t \cap n_2' = \emptyset$ for every rule $r_i \in C$, where $r_i = n_1' > n_2' \mid \phi'$, or $r_i = n_1' > [n_2'] \, n_t' \mid \phi'$, or $r_i = n_1' \, [n_t'] > n_2' \mid \phi'$. Notice that in the particular case that $r_i = n_1 > [n_2] \, n_t \mid \phi$ the definition above enforces n_2 and n_t to be disjoint.

Lemma 1. *Let C be an abstract firewall configuration and $\mathcal{F}_\mathcal{I}$ the intermediate firewall obtained by the translation presented in Table IV. Then for any state s and packet p we have that:*

(i) $p, s \models^{il}_{S_{DNAT}} n_t$ iff $p, s \models_C$ DNAT(n_t)

(ii) $p, s \models^{il}_{S_{SNAT}} n_t$ iff $p, s \models_C$ SNAT(n_t)

(iii) $p, s \not\models^{il}_{S_D}$ iff $p, s \not\models_C$ DROP.

In order to achieve completeness, we need to require an extra condition. We can see that if rules $n_1 > n_2 \mid \phi$ and $n_1 \, [n_t] > n_2 \mid \phi$ are both present in a configuration C, then any packet from a source in n_1 to a destination in n_2, satisfying ϕ, can follow directly to n_2 or through the SNAT. Similarly, if $n_1 > n_2 \mid \phi$ and $n_1 > [n_2] \, n_t \mid \phi$ are both present in configuration C, one cannot know if the packet will be forwarded to n_2, or to an address in n_t. The first case potentially allows one to expose private IPs to the outside, whereas the second introduces non-determinism in the specification. To disallow these we require that for every packet p, and state s $p, s \models_C$ ACCEPT iff $p, s \not\models_C$ SNAT and $p, s \not\models_C$ DNAT. We call such configurations *NAT-consistent*.

This condition enforces that an ACCEPT rule is matched iff there is no matching NAT rule, and can be syntactically enforced (although more restrictively) by checking that if $r = n_1 > n_2 \mid \phi$ and $r' = n_1' \, [n_t'] > n_2' \mid \phi'$ or $r' = n_1' > [n_2'] \, n_t' \mid \phi'$ are in C then $n_1 \cap n_1' = \emptyset$ or $n_2 \cap n_2' = \emptyset$.

Finally, we say that a SNAT rule r is *local* if $r = n_1 \, [n_t] > n_2 \mid \phi$ and $n_1 \cap \mathcal{L} \neq \emptyset$ and $n_2 \cap \mathcal{L} \neq \emptyset$.

Definition 9 (Well-formedness). *Let C be an abstract firewall configuration. If C is NAT-safe, NAT-consistent, and has no local SNAT rules, we say that C is well-formed.*

Theorem 2. *Let C be an abstract firewall configuration and $\mathcal{F}_\mathcal{I}$ the intermediate firewall obtained by the translation presented in Table IV. Then for any states s, s' and packets p, p' we have that:*

(i) if $s \xrightarrow{p,p'}_{il} s'$ then $s \xrightarrow{p,p'}_{hl} s'$;

(ii) if C is well-formed and $s \xrightarrow{p,p'}_{hl} s'$ then $s \xrightarrow{p,p'}_{il} s'$.

C. Relating intermediate and Netfilter firewalls

We now relate intermediate firewalls $\mathcal{F}_\mathcal{I}$ introduced earlier in this section with the Netfilter firewalls $\mathcal{F}$ of Section III. Given a packet p' we will denote by $orig(p')$ the packet p' before performing any NAT translation, that is, (i) if $(s,p) \downarrow^\delta_{ll} p'$ then $orig(p') = p$, (ii) if $(s,p) \downarrow^\delta_{ll} \tilde{p}$ and $(s,p,\tilde{p}) \downarrow^\sigma_{ll} p'$ then $orig(p') = p$; and (iii) if no NAT translation is performed then $orig(p') = p'$.

Definition 10. *Let $\mathcal{F}_\mathcal{I} = \langle S_{D_1}, S_{DNAT}, S_D, S_A, S_{SNAT} \rangle$ be an intermediate firewall. The translation into a Netfilter firewall $\mathcal{F} = \langle L^{man}_{\text{PRE}}, T^{nat}_{\text{PRE}}, L^{man}_{\text{INP}}, T^{nat}_{\text{INP}}, L^{fil}_{\text{INP}}, L^{man}_{\text{OUT}}, T^{nat}_{\text{OUT}}, L^{fil}_{\text{OUT}}, L^{man}_{\text{FOR}}, L^{fil}_{\text{FOR}}, L^{man}_{\text{POST}}, T^{nat}_{\text{POST}} \rangle$ is defined as follows:*

(i) Let $S^{est}_{D_1} = \{(n_1, n_2, \phi \wedge orig(p) \nvdash_s, t) \mid (n_1, n_2, \phi, t) \in S_{D_1}\}$. Let list L^{man}_{PRE} be any possible ordering of the set $S^{est}_{D_1}$; add rule $(*, *, \top, ACCEPT)$ to the end of this list;

(ii) Let list T^{nat}_{PRE} be any possible ordering of the set S_{DNAT};

(iii) Let list L^{man}_{INP} be the concatenation of any possible ordering of the set S_D with any possible ordering of the set $S_A \cup \{(*, *, orig(p) \vdash_s, ACCEPT)\}$; add rule $(*, *, \top, DROP)$ to the end of this list;

(iv) Let list T^{nat}_{INP} be any possible ordering of the set S_{SNAT};

(v) Let L^{man}_{OUT} be defined as (i);

(vi) Let T^{nat}_{OUT} be defined as (ii);

(vii) Let L^{fil}_{OUT} be defined as (iii);

(viii) Let L^{fil}_{FOR} be defined as (iii);

(ix) Let T^{nat}_{POST} be defined as (iv);

(x) Let list $L^{fil}_{\text{INP}} = L^{man}_{\text{FOR}} = L^{man}_{\text{POST}} = \{(*, *, \top, ACCEPT)\}$.

Several remarks should be made concerning the translation above. First, notice that the formula $orig(p) \nvdash_s$ used in the rules of L^{man}_{PRE}, item *(i)*, is not in iptables language. However, this is only for convenience of the exposition as this formula can be written as the option -m state --state NEW. Moreover, the module state always inspects the state of the packet before any address translation and so it will always return the same result regardless of being applied to the original p, to the $\tilde{p}$ obtained after destination NAT translation, or to the p' obtained after source NAT translation. Similarly, the formula $orig(p) \vdash_s$ used in L^{man}_{INP}, item *(iii)*, is also not written in iptables language. This is again for convenience of the exposition as, for the same reason as above, this formula can be written as the option -m state --state ESTABLISHED.

Notice also that since all rules in S_{D_1} and S_D have target DROP the order in which they are implemented (among them) in L^{man}_{PRE} and L^{man}_{INP} is irrelevant; similarly for S_A (L^{man}_{INP}) where the target is always ACCEPT. For S_{DNAT} and S_{SNAT} it is trickier as the firewall semantics presented in Table II is non-deterministic in the presence of overlapping NATs. We will address this problem later in this Section.

Finally, we would like to make a simple remark in terms of efficiency of the generated rules. It is clear from the

translation that we are overpopulating the lists of $\mathcal{F}$. As an example, notice that we place all the rules of S_A in L_{INP}^{man}, but only packets with destination $dst \in \mathcal{L}$ will ever flow through the `Input` chain. So, one could effectively "restrict" the rules in L_{INP}^{man} to those which the destination has a non-empty intersection with $\mathcal{L}$ and obtain the same behavior, as the removed rules would never be triggered. Similarly for the other lists. We adopted to overpopulate the lists and do this "pruning" afterwards as it would simplify both the translation and the proofs. We can however state the following proposition.

Proposition 3. *Let $\mathcal{F}$ be a Netfilter firewall and L_{ch}^{tab} and T_{ch}^{tab} respectively a basic rule list and a translation rule list of chain ch and table tab. Define*

(i) $\overline{L_{\text{INP}}^{tab}} = \{r \mid r = (n_1, n_2, \phi, t) \in L_{\text{INP}}^{tab} \wedge n_2 \cap \mathcal{L} \neq \emptyset\}$;

(ii) $\overline{L_{\text{OUT}}^{tab}} = \{r \mid r = (n_1, n_2, \phi, t) \in L_{\text{OUT}}^{tab} \wedge n_1 \cap \mathcal{L} \neq \emptyset\}$;

(iii) $\overline{L_{\text{FOR}}^{tab}} = \{r \mid r = (n_1, n_2, \phi, t) \in L_{\text{FOR}}^{tab} \wedge (n_1 \cup n_2) \cap \mathcal{L} = \emptyset\}$

(iv) Similarly for T_{INP}^{nat} and T_{OUT}^{nat}.

Then for $ch = \text{INP}, \text{OUT}$ one has that $p, s \models_{T_{ch}^{tab}} t$ iff $p, s \models_{\overline{T_{ch}^{tab}}} t$. For $ch = \text{INP}, \text{OUT}, \text{FOR}$ one has that $p, s \models_{L_{ch}^{tab}} t$ iff $p, s \models_{\overline{L_{ch}^{tab}}} t$.

The presence of two NATs with different targets, for example, $n_1\ [n_t] > n_2 \mid \phi$ and $n_1\ [n_t'] > n_2 \mid \phi$, or $n_1 > [n_2]\ n_t \mid \phi$ and $n_1 > [n_2]\ n_t' \mid \phi$, is a source of non-determinism as in the first case we do not know which source address to exhibit, and in the second we do not know which destination to send the packet to.

Fortunately in real Netfilter policies the behavior is deterministic, as the order in which the system administrator writes the rules reflects the option he is willing to take. With this in mind we can force our configurations also to be deterministic (keeping the overlapping part only in the option that we want to enforce). A configuration is said *deterministic* if for every packet p, state s, if $p, s \models_C \text{DNAT}(n_t)$ and $p, s \models_C \text{DNAT}(n_t')$ then $n_t = n_t'$. Similarly for SNAT.

These conditions require that whenever there are two matching NAT rules of the same type, they have the same target. Similarly to enforcing NAT-safety, this can be syntactically enforced (although more restrictively) by checking that either $n_1 \cap n_1' = \emptyset$ or $n_2 \cap n_2' = \emptyset$ or $n_t = n_t'$, whenever rules $n_1 > [n_2]\ n_t \mid \phi$ and $n_1' > [n_2']\ n_t' \mid \phi'$ are in C (analogous for rules $n_1\ [n_t] > n_2 \mid \phi$ and $rn_1'\ [n_t'] > n_2' \mid \phi'$).

This notion of deterministic configurations can be extended to intermediate firewalls. We say that an intermediate firewall $\mathcal{F}_\mathcal{I}$ is *deterministic* if for any packet p, state s, and rules $r, r' \in S_{\text{DNAT}}$ with $r = (n_1, n_2, \phi, t)$, $r' = (n_1', n_2', \phi', t')$, if $p, s \models r$ and $p, s \models r'$ then $t = t'$; analogous for $r, r' \in S_{\text{SNAT}}$.

Proposition 4. *Let C be a configuration and $\mathcal{F}_\mathcal{I}$ the intermediate firewall obtained by the translation presented in Table IV.*

Then C is deterministic iff $\mathcal{F}_\mathcal{I}$ is deterministic.

Lemma 5. *Let $\mathcal{F}_\mathcal{I}$ be an intermediate firewall and $\mathcal{F}$ the Netfilter firewall obtained by the translation presented in*

Definition 10. Then for any state s and packets $p, \tilde{p}, p'$ we have that:

(i) for $\delta = \text{PRE}, \text{OUT}$, if $(s, p) \downarrow_{ll}^{\delta} \tilde{p}$ then $(s, p) \downarrow_{il}^{DNAT} \tilde{p}$;

(ii) for $\sigma = \text{POST}, \text{INP}$, if $(s, p, \tilde{p}) \downarrow_{ll}^{\sigma} p'$ then $(s, p, \tilde{p}) \downarrow_{il}^{SNAT} p'$

   Moreover, if $\mathcal{F}_\mathcal{I}$ is deterministic we also have that:

(iii) for $\delta = \text{PRE}, \text{OUT}$, if $(s, p) \downarrow_{il}^{DNAT} \tilde{p}$ then $(s, p) \downarrow_{ll}^{\delta} \tilde{p}$;

(iv) for $\sigma = \text{POST}, \text{INP}$, if $(s, p, \tilde{p}) \downarrow_{il}^{SNAT} p'$ then $(s, p, \tilde{p}) \downarrow_{ll}^{\sigma} p'$

We can now state our Theorem.

Theorem 6. *Let $\mathcal{F}_\mathcal{I}$ be an intermediate firewall and $\mathcal{F}$ the Netfilter firewall obtained by the translation presented in Definition 10. Then for any states s, s' and packets p, p' we have that:*

(i) if $s \xrightarrow{p,p'}_{ll} s'$ then $s \xrightarrow{p,p'}_{il} s'$;

(ii) if $\mathcal{F}_\mathcal{I}$ is deterministic and $s \xrightarrow{p,p'}_{il} s'$ then $s \xrightarrow{p,p'}_{ll} s'$.

Notice that the translation given in Definition 10 is always sound ($ll \Rightarrow il$) like the one presented in Table IV ($il \Rightarrow hl$). This implies that all the real Netfilter behaviors are captured by the semantics of Table II.

VI. THE MIGNIS TOOL

MIGNIS, "murus ignis" ("wall of fire" in latin) is a publicly available Python program of about 1500 lines of code, under active development.[1] The aim of MIGNIS is to take a set of rules written in the firewall specification language and to translate them into real `iptables` rules.

A. Conflicts and redundancies

In Section V we have seen that the completeness of the translation from the firewall language to the Netfilter model requires a number of conditions. These conditions are syntactically checked by MIGNIS, namely *NAT-safety*, *NAT-consistency*, no local SNAT rules, and *Determinism*. If one of these conditions does not hold, MIGNIS returns an error pointing out the inconsistency or non-determinism in the specification, ruling out very efficiently any ill-formed policy. There are also cases of redundancies (when, for example, two ACCEPT rules overlap) that are harmless and can be correctly translated. In this case MIGNIS reports the redundant fragment of the policy so to allow administrators to remove these overlaps and simplify the specification.

B. Translation

In the previous sections we have shown how to translate the firewall language into the Netfilter model in a way that preserves the semantics. It is thus enough to map the rules in the Netfilter model into the actual `iptables` syntax.

Recall that we have two kinds of rules that are translated differently depending on the list of rules they belong to:

Basic rules

$(IP1{:}p1, IP2{:}p2, \phi, t) \in L_{chain}^{tab}$ is translated into

```
iptables -t tab -A chain -s IP1 --sport p1 -d IP2
                                --dport p2 φ -j t
```

---

[1] Available for download at the address https://github.com/secgroup/Mignis

Translation rules

(IP1:p1, IP2:p2, ϕ, nt) $\in T_\delta^{nat}$ is translated into

```
iptables -t nat -A δ -s IP1 --sport p1 -d IP2 --dport p2
                    φ -j DNAT --to-destination nt
```

(IP1:p1, IP2:p2, ϕ, nt) $\in T_\sigma^{nat}$ is translated into

```
iptables -t nat -A σ -s IP1 --sport p1 -d IP2 --dport p2
                    φ -j SNAT --to-source nt
```

There are a few special cases that we discuss below:

(1) We allow to specify the protocol out of ϕ as this is more readable and handy when specifying real policies. For example `lan [proxy] > *:443 tcp` specifies that lan can connect to any other host on https port 443 via tcp protocol. This is also useful to more precisely detect possible overlaps of rules as it allows MIGNIS to discriminate on the filtered protocols. Protocol specifications are translated as `-p protocol` and are placed before any occurring filter on ports (`--sport` or `--dport`). This is required in `iptables` syntax since filters on ports are protocol-specific;

(2) `iptables` does not allow one to specify arbitrary sets of addresses. In MIGNIS we have added the possiblity of defining lists of addresses and protocols that are pre-compiled into sets of MIGNIS rules which, in turn, are translated into `iptables`, as specified above.

   We have also developed a 'smart' translation of set exclusion which is not treated in the formal model yet but might be accounted for in future work. For example, it is possible to specify lan \ h1 to match all hosts in lan except h1. Intuitively, this case is translated as if the address were just lan, and the obtained translation is added into a user defined chain c. We then insert a rule at the beginning of this chain c that simply exits the chain whenever the packet comes from h1. This excludes h1 from any subsequent rule match in the chain c. In fact, what we do for packets coming from h1 is that we jump into c and then exit immediately. This way the translated rule will never hit packets from h1 but will process packets from any other host in lan.

(3) the usage of $orig(p) \not\vdash_s$ and $orig(p) \vdash_s$ respectively in L_{PRE}^{man} and L_{FOR}^{fil} was just for convenience in the exposition as these could be written respectively as `-m state --state NEW` and `-m state --state ESTABLISHED`. If one of the address components is a wildcard $*$ (e.g., no source port is specified) the corresponding option is removed;

(4) In SNAT, when there is no translation address we use the target `MASQUERADE` that dynamically picks the address for the interface through which the packets are going out. Rule (IP1:p1, IP2:p2, ϕ, ϵ) $\in T_\sigma^{nat}$ is translated into

```
iptables -t nat -A σ -s IP1 --sport p1 -d IP2 --dport p2
                    φ  -j MASQUERADE
```

(5) MIGNIS additionally allows users to specify interfaces instead of addresses. In this case the translation is done

```
INTERFACES
lan        eth0        10.0.0.0/24
ext        eth1        0.0.0.0/0

ALIASES
mypc                    10.0.0.2
router_ext_ip           1.2.3.4
malicious_host          5.6.7.8

FIREWALL
lan [.] > ext
lan / malicious_host
ext > [router_ext_ip:8888] mypc:8888 tcp
* > local:22 tcp
```

TABLE V
EXAMPLE OF A FIREWALL CONFIGURATION.

using options `-i` (input interface) and `-o` (output interface), as it is shown later on, rather than `-s` and `-d`.

C. Firewall configuration

In Table V we present a simple example of a configuration file. It is composed of different sections. We describe the most relevant ones:

   `INTERFACES`: the list of the router interfaces and subnets mapping. It has 3 columns: interface alias (used in the rules), real interface name, and subnet in the `XXX.XXX.XXX.XXX/N` form;

   `ALIASES`: generic aliases used to avoid repeating IP addresses;

   `FIREWALL`: rules written in the firewall language;

The configuration presented in Table V is common amongst home users:

   `lan [.] > ext` specifies that from the lan it is allowed to access the Internet, using a SNAT with dynamic address (masquerade);

   `lan / malicious_host` forbids any pc in the lan to communicate with the malicious host;

   `ext > [router_ext_ip:8888] mypc:8888 tcp` allows incoming `tcp` connections on port 8888 that are forwarded to `mypc`, i.e., `10.0.0.2`;

   `* > local:22 tcp` allows for ssh connections to the local host.

The execution of MIGNIS consists of several steps: (i) Netfilter is reset; (ii) default policies and some default rules are applied: these allow the router to initiate external connections, receive pings, and broadcast traffic which is usually considered safe (but can be switched-off if needed); (iii) firewall language rules are translated into `iptables` rules; (iv) rules that guarantee that each IP address will originate only from its assigned interface are added. This makes equivalent, from a security perspective, to specify interfaces or addresses as sources and destinations; (v) log rules are added.

Applying MIGNIS to the configuration in Table V reports that two overlapping rules have been defined: `lan / malicious_host` and `lan [.] > ext`. In fact the former aims at blacklisting `malicious_host` which is part of `ext`. Overlapping information is quite useful to point out possible redundancies and mistakes in configurations. For lack

of space, the output of MIGNIS is reported in [36] together with the technical proofs.

D. Real case-study

In this section we describe how our theory and tool have been applied to a real system, the computer network of the department of Enviromental Sciences, Informatics and Statistics of the Ca' Foscari University, Venice. Sensitive informations were abstracted out or otherwise anonymised. The network is part of a larger university network and is, in turn, composed of 3 subnetworks, each of them with different purposes and configurations. We identify the following entities:

- The whole university network, denoted by `U`, which is not under our direct control.
- The main departmental network, denoted by `D`.
- The laboratory network, denoted by `L`
- A recently-allocated network which will be used mostly by personal clients, denoted by `C`.
- The firewall `F`, which acts as a router between `D`, `L`, `C` and `U`, and is also connected to the Internet through other routers.

The are about 350 active hosts on `D`, `L` and `C`, plus a varying number of temporary or guest's hosts.

As it is common for real systems, these networks are not strictly compartmentalised according to the purpose and to the security requirements of the hosts, due to historical, economical and technical constraints. In particular, `D` is a mixed network, in which servers and faculty members' clients, which are not yet moved to `C`, coexists. Network `L` is also partially mixed, because it contains both lab clients and servers which are used mainly by students. All the aforementioned networks support multicast transmissions, and are integrated in a broader multicast system, used mainly for audio and video transmissions, which is managed by the university. This setting leads to a quite complex firewall configuration, which is rich of exceptions for single hosts or classes of those.

The original firewall configuration, as a sequence of hand-crafted `iptables` commands, was 610 lines long (counting only the commands themselves), with 5 custom chains. After a careful translation, we produced a 210 lines long MIGNIS configuration which was able to match the behaviour of the previous firewall, and which was subsequently used as our actual firewall configuration. In a second phase, using both the overlap-detecting capabilities of MIGNIS, analysing the more readable configuration, and using the information given by the annotated logs, we were able to further reduce the configuration to 141 MIGNIS lines, which were translated into 472 `iptables` commands. Some overlapping warning were ignored in order to increase the readability and the maintainability of the configuration, thus the configurations could be further reduced in size by approximately 30 lines, without changing its semantics. To this day, the department firewall runs on this latter configuration.

In Table VI we give some snippets of the actual configuration, in which some non-trivial rules are shown. Notice the

```
# Transparent proxy for lab hosts.
labs > [ext:80] proxy:8080 tcp
labs [F] > proxy tcp
# https requests should appear from the same source
labs [proxy] > *:443 tcp

# acting as a DHCP server
(D,L,C):68 <> local:67 udp

# acting as a multicast router (using xorp)
(*,local) <> multicast all
* > local (igmp,pim,udp) | -d multicast
local > * (igmp,pim,udp) | -s multicast

# as a policy, we want to notify rejection
# for unallowed outgoing tcp connections
POLICIES
(D,L,C) // ext tcp
```

TABLE VI
SNIPPETS OF CONFIGURATION FROM THE CASE STUDY.

straightforward way of expressing transparent proxy and https-passthrough rules, which translate in at least 7 `iptables` rules. In fact, since `labs` is defined as an alias for a list of aliases of 5 disjoint IP ranges, this would have required 35 lines. The rules for the DHCP server are bidirectional, in order to allow for all the steps of protocol handshaking. Another interesting example is given by the rules to allow the firewall to act as a multicast router, in which packets not going to nor coming from a local address are still seen in the INPUT and OUTPUT chain, respectively. Notice that this requires some extra options to be passed to `iptables`. The last example shows the specification of a REJECT policy for some network. The intended behaviour is to immediately notify applications running in networks behind the firewall that the desired connection is not allowed, thus sparing the need to wait for a timeout.

VII. CONCLUSIONS

In this paper we addressed the problem of simplifying the specification, management and analysis phases of a firewall system. In order to do so, we first proposed an abstract model for Netfilter, that is rich enough to capture the concepts of chains, rules, and packets. We then provided a multi-step semantics for this model that emulates real Netfilter executions.

We introduced a simple, yet expressive, declarative language to specify firewall configurations. Contrary to concrete implementations, where the order in which rules are written is crucial to determine the behavior of the system, in our language this is not relevant, which makes specifying and maintaining a firewall much simpler. We gave a formal semantics for this language, and showed that under reasonable conditions, firewall policies expressed in our language can be translated into the proposed Netfilter model in a way that preserves packet filtering and address translation.

The theoretical results of this work were implemented in MIGNIS a novel, publicly available tool which translates a high-end firewall specification expressed in our declarative language into real `iptables` rules. We also described some real-world applications of this tool, including a case study given by the actual firewall configuration of a University

department, and we showed how the static analysis capabilities of the software can help to spot errors in firewall policies and to reduce the complexity of their specifications, e.g., detecting overlaps.

While we were not the first ones presenting a language that simplifies firewall specification, to the best of our knowledge our model is the first that provides correctness guarantees about the generated configuration.

We believe this work may have impact in several communities. From a practical perspective we allow practitioners to specify firewall configurations in a simple understandable language with single-step semantics, and to generate the list of rules that implements that configuration in Netfilter. For theoreticians we propose a formalization of the behavior of a firewall that is amenable to verification of the intended security properties.

Future work: We are considering further possible optimizations of the specification language and of its translation. Since in our specification language the rule order is not relevant, it is possible to automatically rearrange the underlying Netfilter rules according to the statistics on packet matching provided by Netfilter itself. This could lead to performance improvements for firewalls managing very intense traffic. Experiments on this topic are ongoing, using the system described in the case study.

Another future work regards the extension of MIGNIS to allow for the translation into rules for firewall systems different from Netfilter but that have a similar semantics, exploiting an opportunely modified abstract model. Moreover, we are investigating the extension to networks of firewalls as addressed by Nelson et al. [37] and similarly to models that are based on network programming languages such as NetKAT [13].

Finally, it would be very interesting to extend the firewall specification language to include security goals, in the form of information-flow properties that the firewall rules are expected to achieve, in line with the approach proposed in [38].

REFERENCES

[1] A. El-Atawy, T. Samak, Z. Wali, E. Al-Shaer, F. Lin, C. Pham, and S. Li, "An automated framework for validating firewall policy enforcement," in *Proc. of the 8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'07)*. IEEE, 2007, pp. 151–160.

[2] J. Walsh, "Icsa labs firewall testing: An in depth analysis," http://bandwidthco.com/whitepapers/netforensics/penetration/Firewall%20Testing.pdf, 2004.

[3] A. Liu and M. Gouda, "Diverse Firewall Design," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 9, pp. 1237–1251, 2008.

[4] T. G. F. Mansmann and W. Cheswick, "Visual analysis of complex firewall configurations," in *Proc. of the 9th International Symposium on Visualization for Cyber Security (VizSec'12)*. ACM, 2012, pp. 1–8.

[5] S. Morrissey and G. Grinstein, "Visualizing firewall configurations using created voids," in *Proc. of the Int. Workshop on Visualization for Cyber Security*. ACM, 2009.

[6] S. Morrissey, G. Grinstein, and B. Keyes, "Developing multidimensional firewall configuration visualizations," in *Proc. of the 2010 International Conference on Information Security and Privacy*. ISRT, 2010.

[7] T. Tran, E. Al-Shaer, and R. Boutaba, "Policyvis: Firewall security policy visualization and inspection," in *Proc. of the 21st Large Installation System Administration Conference (LISA '07)*. Usenix association, 2007, pp. 1–16.

[8] R. Russell, "Linux 2.4 packet filtering howto." http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html, 2002.

[9] M. Gouda and A. Liu, "Structured firewall design," *Computer Networks*, vol. 51, no. 4, pp. 1106–1120, Mar. 2007.

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

[11] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège, "A formal approach to specify and deploy a network security policy," in *Formal Aspects in Security and Trust (FAST'04)*, 2004, pp. 203–218.

[12] "Frenetic, a family of network programming languages." http://www.frenetic-lang.org/, 2013.

[13] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. ACM, 2014.

[14] Y. Bartal, A. Mayer, Nissim, and A. W. Firmato, "A Novel Firewall Management Toolkit," *ACM Transactions on Computer Systems*, vol. 22, no. 4, pp. 1237–1251, 2002.

[15] B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher, "Specifications of a high-level conflict-free firewall policy language for multidomain networks," in *Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*. ACM, 2007.

[16] "Netspoc: A network security policy compiler," http://netspoc.berlios.de, 2011.

[17] "High level firewall language," http://www.hlfl.org, 2003.

[18] "Uncomplicated firewall," https://help.ubuntu.com/community/UFW.

[19] "Iptables made easy, shorewall." http://www.shorewall.net/, 2014.

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

[21] "Kmyfirewall," http://www.kmyfirewall.org/, 2008.

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

[23] "Firewall builder," http://www.fwbuilder.org/, 2012.

[24] "Rule markup language," http://www.ruleml.org/, 2011.

[25] "Oasis extensible access control markup language," http://xacmlinfo.org/category/xacml-3-0/, 2013.

[26] "Chef," http://www.getchef.com/chef/, 2014.

[27] "LCFG large scale unix configuration system," http://www.lcfg.org/, 2014.

[28] "With puppet enterprise, you pull the strings," http://puppetlabs.com/, 2014.

[29] F. Cuppens, N. Cuppens-Boulahia, J. Garca-Alfaro, T. Moataz, and X. Rimasson, "Handling stateful firewall anomalies." in *SEC*, ser. IFIP Advances in Information and Communication Technology, vol. 376. Springer, 2012, pp. 174–186.

[30] S. Martínez, J. Cabot, J. Garcia-Alfaro, F. Cuppens, and N. Cuppens-Boulahia, "A model-driven approach for the extraction of network access-control policies," in *Proc. MDSec'12*. ACM, 2012, pp. 5:1–5:6.

[31] S. Pozo, R. Ceballos, and R. M. Gasca, "Afpl, an abstract language model for firewall acls," in *Proc. of the international conference on Computational Science and Its Applications, Part II*, ser. ICCSA '08. Springer-Verlag, 2008, pp. 468–483.

[32] R. M. Marmorstein, "Formal analysis of firewall policies," Ph.D. dissertation, College of William and Mary, Williamsburg, VA, May 2008.

[33] "Ipfilter," http://coombs.anu.edu.au/~avalon/, 2009.

[34] "Packet filtering," http://www.openbsd.org/faq/pf/filter.html, 2013.

[35] "pfSense, a proven open source firewall," http://www.pfsense.org/, 2014.

[36] P. Adão, C. Bozzato, G. D. Rossi, R. Focardi, and F. Luccio, "Mignis: A semantic based tool for firewall configuration (extended version)," 2014.

[37] T. Nelson, C. Barratt, D. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *Proceedings of the 24th International Conference on Large Installation System Administration (LISA'10)*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[38] J. Guttman and A. Herzog, "Rigorous automated network security management," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 29–48, 2005.