# Can my Firewall System Enforce this Policy?

Lorenzo Ceragioli[a], Pierpaolo Degano[a,b], Letterio Galletta[b,c]

[a] *Dipartimento di Informatica, Università di Pisa, Pisa, Italy*
[b] *IMT School for Advanced Studies Lucca, Lucca, Italy*
[c] *CINI Cybersecurity National Laboratory, Rome, Italy*

**Abstract**

Firewalls are a fundamental tool for managing and protecting computer networks. They behave according to a configuration that specifies the desired policy, i.e., which packets are allowed to enter a network, possibly with modified addresses. Several tools allow the user to specify policies in various high level languages, and to compile them into different target configuration languages, as well as to automatically migrate a configuration from a system to another. Often, these tools implicitly assume that the target system can enforce any desired policy. Unexpectedly, we find that this is not always the case. In particular, we show that the most common UNIX firewall systems, i.e., `iptables`, `ipfw`, `pf`, are not equally expressive, in that some policies can be implemented in one system but not in another. Here, we formally investigate the expressive power of these firewall systems using techniques from programming language semantics, and set up a formal model to precisely characterize their relationships. Based on this formal model we then present *F2F*, a prototypical tool that predicts when a policy cannot be expressed in a given system. Our prototype gives detailed information about the unexpressible parts of a policy and provides administrators with hints for fixing the detected problems.

*Keywords:* Formal Security Models, Language-based Security, Configuration Languages, Semantics of Firewalls, Automatic Analysis of Firewalls

## 1. Introduction

Firewalls are one of the most used tools for protecting computer networks and enforcing access control policies. They grant control on which packets can enter a network and how they are transformed by the so-called network address translation (NAT). The firewall behavior is specified by a configuration that implements the wanted policy. Roughly, a configuration is a list of rules that transforms and filters the incoming packets. Each firewall system comes with its own configuration language, with a specific syntax and a different way of evaluating and applying the rules.

Nowadays, network administrators exploit policy-based management systems, e.g., the open source tools FirewallBuilder [1] and Google Capirca [2], that provide high-level and user-friendly policy languages that are then compiled to configurations of a target system (e.g., `iptables` on Linux). Typically, these tools support different target systems and implicitly assume that *any* policy can be compiled in each of them.

*Email addresses:* `lorenzo.ceragioli@phd.unipi.it` (Lorenzo Ceragioli), `pierpaolo.degano@unipi.it` (Pierpaolo Degano), `letterio.galletta@imtlucca.it` (Letterio Galletta)

| dstIP | dstPort | srcIP | srcPort | DNAT | SNAT |
|---|---|---|---|---|---|
| any | 53 | any | any | 9.9.9.9 : id | id : id |
| any | not 53 | any | any | id : id | id : id |

Table 1: A policy expressible in `iptables` but not in `pf` nor in `ipfw`.

However, this is not true as shown by the following example. Consider the simple firewall policy $\tau$ in Table 1, expressed in a tabular form in the spirit of FirewallBuilder. This policy accepts all the packets, redirecting those received on port 53 (DNS protocol) to a remote host at 9.9.9.9. A possible compilation of $\tau$ into `iptables` would roughly give the configuration in Figure 1. Instead, $\tau$ is neither expressible in `pf` nor in `ipfw`, because no packets leaving the firewall itself can undergo a transformation on the destination address. This example is minimal and captures in isolation the problem of expressivity. However, similar situations can happen in configurations with thousands of rules, where this problem is concealed and hard to detect, paving the way to a possibly dangerous misconfiguration. This may occur, e.g., when updating legacy configurations or when migrating *manually* a configuration from a system to another because the tool support is lacking or insufficient [3, 4].

Different solutions exist for this kind of problems. A largely used mechanism is marking specific packets with an internal identifier, called *tag*, that can later be used in identifying certain packet flows and in refining filter and translation rules [5]. Alternatively, administrators resort to special rules, e.g., `NFQUEUE` in `iptables` [6], that transfer packet management to external tools to complement the behavior of the firewall [7]. However, such "patches" receive little attention by policy-based management systems (e.g., by [1]) or even no attention (e.g., by [2]), while a reliable system should at least warn administrators of such problems. Otherwise, some implemented access policies might differ from those the administrator has in mind, e.g., the

not applied transformations illustrated above. It is therefore crucial to know the actual expressive power of firewall configuration languages, and to have mechanisms to check when a given policy cannot be expressed in a given system.

In this paper, we investigate the expressive power of firewall configuration languages using techniques from programming language semantics. In particular, we focus our investigation on `iptables`, `ipfw` and `pf` that are the most common UNIX firewall systems and that are used as building blocks of many network appliances on the market, e.g., pfSense [8]. Indeed, often these appliances run customized Unix distributions that use these firewall systems for filtering/translating packets, and that resort to other tools such as Suricata [9] and Snort [10] to provide advanced features, e.g., deep packet inspection.

Our goal is to precisely characterize the relationships between these firewall configuration languages, and for that we set up a formal model. It is based on IFCL, an intermediate language for firewall configurations [11, 12], which subsumes the most common firewall languages.

Our first contribution is a denotational semantics for IFCL, which represents a firewall configuration as a function from packets to transformations on them.

Based on this denotational semantics, we then introduce two different notions of expressivity that associate each language with the set of policies it can express. The first notion is called *individual expressivity*, as it considers each packet in isolation. Roughly, it captures policies that can be implemented in a given system, possibly relying on tags. We prove that IFCL and `iptables` are universal according to the notion of individual expressivity, and that `ipfw` is as powerful as `pf` (Theorem 4); this result is detailed in Table 2. The second type of expressivity is called *function expressivity*, and discriminates more, because it takes care of all the packets at the same time and the way transformations change them. Firewalls evaluate packets in steps, where each step applies a given transformation. It may happen that a given step makes two different packets indistinguishable, because of the transformations applied in the previous steps. A clash exists if the policy dictates that the two packets must instead be treated differently (see Example 7). Looking at clashes, we establish a finer hierarchy where IFCL strictly dominates all the others, `iptables` and `ipfw` are incomparable, and only `ipfw` strictly dominates `pf` (Theorem 6). Actually, this notion characterizes policies that can be implemented without resorting to ad hoc mechanisms.

Our model supports the definition of two algorithms that compute and check both kinds of expressivity. To check individual expressivity, the first algorithm concisely enumerates the pairs $(p, t)$, where $p$ is a packet and $t$ is the transformation it can undergo. In this way one determines which pairs are expressible and which are not. The second algorithm discovers if the policy requires the firewall to transform a packet in two incompatible ways. We show how tags can solve this clash.

Last but not least, we describe the tool *F2F*, available online at [13]. It supports the administrators in a number of ways First, it verifies if a certain policy is expressible in a given language, so facilitating the choice of the more appropriate system and of its supporting tools. When the policy is not expressible,

```
1  *nat
2  :PREROUTING ACCEPT [0:0]
3  :INPUT ACCEPT [0:0]
4  :OUTPUT ACCEPT [0:0]
5  :POSTROUTING ACCEPT [0:0]
6
7  -A PREROUTING -p udp --dport 53 -j DNAT --to 9.9.9.9
8  -A OUTPUT -p udp --dport 53 -j DNAT --to 9.9.9.9
9
10 COMMIT
11
12 *filter
13 :INPUT ACCEPT [0:0]
14 :FORWARD ACCEPT [0:0]
15 :OUTPUT ACCEPT [0:0]
16
17 COMMIT
```

Figure 1: An `iptables` configuration expressible neither in `pf` nor in `ipfw`.

*F2F* gives detailed information about the reasons why, and provides administrators with hints for fixing the detected problems. For example, it signals which packets require to be handled by special mechanisms provided by the target system. In this way, our tool supports administrators to check that their policy-based management system, e.g., FirewallBuilder [1], creates correct configurations when compiling to different targets. Also, *F2F* predicts if a configuration is portable from a system to another using an automatic tool like FWS [11, 14, 12]. Although still prototypical, and far from being optimized, our tool has an acceptable performance when analyzing real configurations available on-line. Remarkably, the correctness of the given results is guaranteed by our formal model and our algorithms.

Briefly, our tool answers the questions "what can I do without mechanisms external to my firewall system? And what without tags?" Answering them is important because most of the tools for the management and the analysis of firewall configurations do not support such extensions [15, 16, 17, 18]. Thus, *F2F* integrates and refines the functionalities and the results of these tools, helping administrators to reduce misconfigurations that may impact security.

*Plan of the Paper*

The next section briefly surveys `iptables`, `ipfw`, `pf`. The basics of IFCL and its new denotational semantics are in Section 3, together with the encoding of the considered firewall languages into IFCL. Section 4 presents illustrative examples of a policy expressible in iptables, but not in ipfw and pf, and one that is non expressible in iptables. Section 5 defines the two notions of expressivity and the theorems that compare the expressive power of the considered languages. In Section 6 we present our tool for checking expressivity. In particular we describe the compact representation of the firewall semantics, the main algorithms of the tool and its work-flow. In Section 7 we evaluate its effectiveness on a case of study and the performance of *F2F* on real-world cases. The last section compares our proposal with the literature, and draws some conclusions. Table 5 summarizes the notation and the symbols used in the paper. Further material, the proofs, and auxiliary lemmata are in the Appendices.

## 2. Background: Linux and Unix Firewall Systems

Typically, firewalls are implemented either as proprietary devices, or as software tools running on general purpose operating systems. Independently of their actual implementations, they are usually characterized by a set of rules that determine which packets have permission to reach the different hosts of a network, and how they are translated.

This section briefly introduces the main firewall systems used in Linux and Unix environments, the expressivity of which we study in this paper and check through our tool.

In particular, we describe the constructs that administrators need to use to write configurations for these firewall systems. We also give an intuition on how these system process packets, sketching out their decision algorithms that we formalize in terms of control diagrams in Section 3. Moreover, since hereafter we only consider new connections, we omit any detail about constructs related to the state of connections. We refer the reader to the relevant manuals for additional details.

*`iptables`.* It is the default tool for packet filtering in Linux and it operates on top of Netfilter, the framework for packets processing of the Linux kernel [19].

An `iptables` configuration is built on *tables* and rulesets, called *chains*. Intuitively, each table has a specific purpose and is made of a collection of chains. The most commonly used tables are: `Filter` for packet filtering; `Nat` for network address translation; `Mangle` for packet alteration. Chains are ordered lists of rules that are inspected to find the first one matching the packet under evaluation. There are five predefined chains, provided by the system, and users can define theirs. Chains are inspected by the Linux kernel at specific moments of the packet life cycle [6] to decide the destiny of a packet $p$: `PreRouting`, when $p$ reaches the firewall host; `Forward`, when $p$ is routed through the host; `PostRouting`, when $p$ is about to leave the firewall host; `Input`, when $p$ is routed to the host; `Output`, when $p$ is generated by the host. The inspection order depends on different conditions that a packet satisfies and we represent it through the control diagram of Figure 2a. For example, a packet entering the firewall that needs to be forwarded to another host is processed in turn by the chains `PreRouting`, `Forward` and `PostRouting`; instead a packet for the host running the firewall is processed by the chains `PreRouting` and `Input`. Tables do not necessarily contain all the predefined chains, but only a specific subset (see [19] for a full account details). For example, the table `Filter` contains the chains `Forward`, `Input`, and `Output`. Chains are inspected top-down to find a rule applicable to the packet under elaboration. Each rule specifies a *condition* and an action: if the packet matches the condition then it is processed according to the specified target. The most commonly used targets are: ACCEPT, to accept the packet; DROP, to discard the packet; DNAT, to perform destination NAT, i.e., a translation of the destination address and port; SNAT, to perform source NAT, i.e., a translation of the source address and port. There are also targets that allow implementing mechanisms similar to jumps and procedure calls. One can also mark packets with a tag and use the tag value later on to better drive rule application. Built-in chains have a user-configurable default policy (ACCEPT or DROP) to be applied when the evaluation reaches the end of a built-in chain.

*`ipfw`.* It is the standard firewall for FreeBSD [20]. A configuration consists of a single ruleset that is inspected twice, when the packet enters the firewall and when it exits. The way `ipfw` processes packets is described by Figure 2b that we comment in the next section. By using the keywords `in` and `out`, it is possible to specify when a certain rule has only to be applied in either case. Similar to `iptables`, rules are inspected sequentially from top to bottom until the first match occurs and the corresponding action is taken. The most common actions in `ipfw` are the following: `allow`/`deny` to accept/reject packets; `nat` to change the destination/source address of incoming/outgoing packets via NAT.

The packet is dropped if there is no matching rule. The sequential order of inspection is altered by special targets that jump to a rule that follows the current one. Packet marking is supported also by `ipfw`: if a rule containing the `tag` keyword is applied, the packet is marked with the specified identifier and then processed according to the rule action.

*`pf`.* This is the standard firewall of OpenBSD [21] and macOS since version 10.7. A `pf` configuration has a single ruleset, inspected when the packet enters and exits the host. Figure 2c formalizes the algorithm used by `pf` to process a packet, and we better describe it in the next section. Similar to the other systems, each rule consists of a condition and a target that specifies how to process the packets matching the condition. The most common targets are: `pass` and `block` to accept and reject packets, respectively; `rdr` and `nat` to perform destination and source NAT, respectively. The ruleset is sequentially inspected as in the other systems, but the rule to apply is the *last matched rule* (unless otherwise specified by `quick` rules). Moreover, when a packet enters the host, DNAT rules are examined first and filtering is performed after address translation. Similarly, when a packet leaves the host, its source address is translated by the relevant SNAT rules, and then the resulting packet is filtered.

## 3. The intermediate language IFCL

We present here the Intermediate Firewall Configuration Language IFCL [12] that is used to put the systems surveyed above in a common formal setting. The language has the most common targets, including those for altering the control flow, like ACCEPT, DROP, NAT, JUMP, CALL and RETURN. Here, we omit those affecting the control flow because an unfolding procedure can remove them producing an equivalent firewall. The interested reader can find a detailed presentation of the encoding and of the unfolding procedure, together with the proof of their correctness, in [12]. The semantics of IFCL defined below is more abstract than the one of [12] because a ruleset is associated with a function from packets to transformations, rather than with a list of rules. Intuitively, a transformation says if a packet is either dropped, or it is accepted with the applied translations. This representation simplifies the usage of IFCL as a framework

for the formal development of Section 5 and the algorithms implemented in our tool.

### 3.1. Firewalls in IFCL

A firewall in IFCL consists of a *control diagram* and an *assignment of functions* to its nodes. A control diagram is a graph that deterministically describes the order in which functions (that abstractly represent firewall rules) are applied by the network stack of the operating system. The control diagram $C_{\mathcal{L}}$ is specific to the given firewall configuration language $\mathcal{L}$ and captures its decision algorithm, i.e., the algorithm that decides if a packet is accepted or not: every node $q$ thus stands for a processing step and it is associated with a function that represents the operations to perform when the control reaches $q$ (roughly, the application of the configuration rules to packets). Arcs are labeled with predicates that encode routing decisions taken by the firewall, e.g., for checking if a packet comes from the external world. Intuitively, a packet $p$ is accepted if there exists a path in $C_{\mathcal{L}}$ (i.e., a sequence of nodes $\pi = q_i \cdot q \cdot \ldots \cdot q_f$ with $\cdot$ as the juxtaposition operator) from the initial node $q_i$ to the final node $q_f$ such that $p$ passes the checks, and is possibly transformed by the functions associated with the nodes of $\pi$.

Some notation is in order. *A packet $p$* is a tuple of packet fields $p_w \in D_w$, where the index $w \in W$ identifies the element of the tuple. The set $\mathbb{P}$ includes all IP packets and is formally the Cartesian product of domains $D_w$, one for each field. Here, we do not fix the (finite) set $W$, and we only assume it to contain the fields *dIP*, *dPort*, *sIP* and *sPort* that are the destination (source) IP and port, respectively. Also, we let the functions $d(p)$ and $s(p)$ return the pairs of addresses $p_{dIP} : p_{dPort}$ and $p_{sIP} : p_{sPort}$, respectively.

A firewall either leaves a packet unchanged, or it modifies some of its fields. We formalize this activity by the transformation functions associated with the nodes of the control diagram. A *transformation $t$ of a field $w$* is either the identity function $id$ (the value of $w$ is left unchanged) or the constant function $\lambda_{a'}$ returning the value $a' \in D_w$ (the value of $w$ is now $a'$). A *packet transformation* $t = (t_{dIP} : t_{dPort}, t_{sIP} : t_{sPort}) \in \mathcal{T}_{\mathbb{P}}$ is a quadruple of transformations, two for the destination IP and port fields, and two for the source fields. Packet transformations are applied and composed component-wise.

For convenience, we extend $\mathbb{P}$ with the distinguished element $\perp$ to represent the dropped packets. Thus, transformations are blankly extended, assuming $t(\perp) = \perp$. Also, we denote with $\lambda_{\perp}$ the transformation that always drops packets.

**Example 1.** *Let the packet $p$ have $d(p) = 8.8.8.8\!:53$ and $s(p) = 192.168.0.8 : 50000$, and let $t = (id\!:\!id, 151.15.1.5 : id)$ be the transformation that changes the source IP (performing a SNAT); then $p' = t(p)$ has $d(p') = 8.8.8.8 : 53$ and $s(p') = 151.15.1.5 : 50000$.*

The following definition introduces the representation in IFCL of the decision logic of a firewall configuration language. It borrows some details from the one in [12].

**Definition 1** (Control diagram)**.** *Let $\mathcal{P}$ be a set of predicates over packets, assuming that for all $\psi \in \mathcal{P}$ and for all $p \in \mathbb{P}$, $\psi(p) = \bigwedge_{w \in W} \psi_w(p_w)$.*

*Given a firewall configuration language $\mathcal{L}$, its IFCL* control diagram *is a graph $C_{\mathcal{L}} = (Q, A, q_i, q_f)$ where*

- *$Q$ is the set of nodes;*
- *$A \subseteq Q \times \mathcal{P} \times Q$ is the set of arcs, such that whenever $(q, \psi, q'), (q, \psi', q'') \in A$ and $q' \neq q''$ then $\neg(\psi \wedge \psi')$;*
- *$q_i, q_f \in Q$ are special nodes denoting the start of elaboration of an incoming packet $p$ and the end, if $p$ is accepted.*

*We also let $\delta$ be the* transition function *of $C_{\mathcal{L}}$, such that $\delta(q, p) = q'$ if $(q, \psi, q') \in A \wedge \psi(p)$.*

Now we represent in IFCL a firewall written in the language $\mathcal{L}$, by associating the decisions to be taken with a node $q$ of its control diagram when a packet reaches $q$. Formally:

**Definition 2** (Firewall)**.** *Given a firewall configuration language $\mathcal{L}$ with control diagram $C_{\mathcal{L}} = (Q, A, q_i, q_f)$, an IFCL* firewall configuration *is a map $f\colon Q \to \mathbb{P} \to \mathcal{T}_{\mathbb{P}}$.*
*In the following we will write $(C_{\mathcal{L}}, f)$ to concisely denote an IFCL firewall configuration of $\mathcal{L}$.*

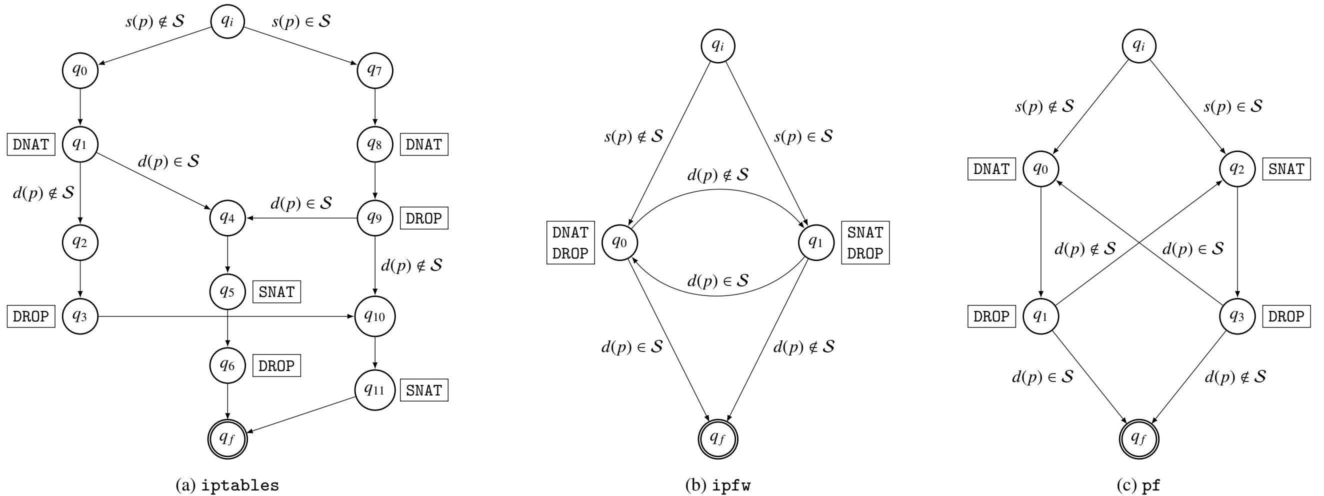
### 3.2. Modeling `iptables`, `ipfw` and `pf` in IFCL

In the following we intuitively present the IFCL encoding of the languages introduced in Section 2. The interested reader can find in [12] a formal encoding of these languages into a ruleset-based version of IFCL, and in [22] the translation of IFCL rulesets into functions from packets to transformations.

Hereafter let $\mathcal{S}$ be the set of the addresses of the firewall interfaces; and let $d(p) \in \mathcal{S}$ ($s(p) \in \mathcal{S}$, resp.) specify that $p$ is for (comes from, respectively) the firewall. In the control diagrams of Figure 2 we label arcs with predicates expressing constraints on the header of packets according to $\mathcal{S}$; arcs with no label carry implicitly "*true*."

The targets for accepting (ACCEPT), dropping (DROP), changing the destination address (DNAT) and changing the source address (SNAT) are encoded in transformations in the following way: ACCEPT as $(id : id, id : id)$, DROP as $\lambda_{\perp}$, DNAT to IP address *ip* and port address *port* as $(\lambda_{ip} : \lambda_{port}, id : id)$, finally SNAT to IP address *ip* and port address *port* as $(id : id, \lambda_{ip} : \lambda_{port})$.

*`iptables`.* Figure 2a shows the control diagram $C_{\texttt{iptables}}$ of `iptables`. The encoding associates a functional representation of the predefined chains with the nodes of the control diagram. For example, the table `Nat` contains the `PostRouting` chain that is associated with $q_{11}$. It is important that in `iptables` a DNAT is only performed in nodes $q_1, q_8$, whereas SNAT only in nodes $q_5, q_{11}$. Similarly, DROP can only be applied when the control is in nodes $q_3, q_6, q_9$. These capabilities are denoted by the labels in the boxes. To represent a chain as a function it is sufficient to translate every rule in a separate case of the function definition. The encoding of a single `iptables` condition into a predicate is trivial. Rules are composed to carefully represent the top-down evaluation order. This is done by joining



(a) `iptables` (b) `ipfw` (c) `pf`

Figure 2: Control diagrams of `iptables`, `ipfw` and `pf`.

the condition of a rule with the negation of the conditions of all the preceding rules. This also guarantees that every predicate is disjointed from the other. Common targets are translated as defined above, whereas jumps to user defined chains can be macro expanded in a preprocessing phase [12]. More precisely, a `jump` to a user defined chain $R$ is treated as the rules inside $R$ after adding the condition of the jump rule to the each rule inside $R$.

*`ipfw`.* The control diagram of `ipfw` is in Figure 2b. The encoding of [12] splits the single `ipfw` ruleset in two rulesets containing each the rules annotated with the keyword `in` and `out`, respectively (if not annotated, the rule goes in both). Both rulesets can filter packets and transform them, but only $q_0$ ($q_1$ resp.) can apply DNAT (SNAT resp). The node $q_0$ is associated with the ruleset applied when an IP packet reaches the host from the net, whereas, $q_1$ is for when the packet leaves the host. The encoding of the conditions is the same of `iptables`, since in `ipfw` the evaluation is top-down. Common targets are translated as usual, whereas `skipto` can be macro expanded in a preprocessing phase [12]. More precisely, a rule $r_i$ with condition $p$ and target `skipto` $r_j$ is treated as a list of rules $r_j \dots r_{end}$, after adding $p$ to each condition, followed by $r_{i+1} \dots r_{end}$ where not $p$ is added to each condition.

*`pf`.* Figure 2c displays the control diagram of `pf`, where the nodes $q_0$ and $q_1$ are associated with the rules applied to packets that reach the firewall, while $q_2$ and $q_3$ are for when they leave the firewall. Also in this case, the single ruleset of `pf` is split in different rulesets. A source NAT can only be applied to packets leaving the firewall, and destination NAT only to those reaching it. Importantly, NAT rules (in nodes $q_0$ and $q_2$) are evaluated *before* filtering (in nodes $q_1$ and $q_3$). The encoding of single conditions is trivial. For the composition we have to keep in mind that evaluation is top-down, but that the last matching rule is taken, with the exception of `quick` rules, that are applied immediately. Hence quick rules conditions are in conjunction with the negation of the ones of all preceding quick rules, whereas common rules conditions are in conjunction with the negation of all the conditions of quick rules and of the following common rules.

### 3.3. *Our running example*

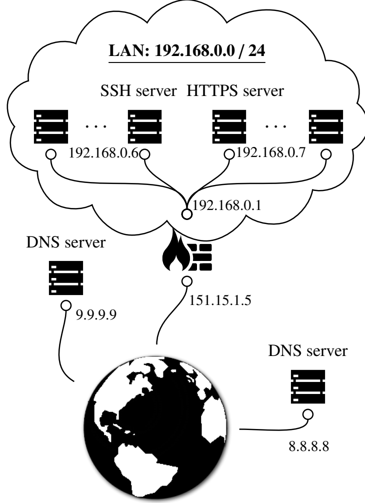
Consider a typical network of a small company depicted in Figure 3a that will be used as a scenario for the examples of the paper. Asssume that the firewall at the adresses $\mathcal{S} = \{192.168.0.1, 151.15.1.5\}$ is the only connection point between the Internet and the Local Area Network (LAN). The LAN private addresses range over 192.168.0.0/24; the internal hosts at 192.168.0.6 and 192.168.0.7 run an SSH and an HTTPS server. On the Internet, two DNS servers are hosted at 8.8.8.8 and 9.9.9.9.

Figure 3b shows a `pf` configuration for this scenario. Rule at line 2 implements SNAT and translates packets from local machines directed to the Internet, replacing the source *IP* with that of the external interface of the firewall. Rule at line 3 realizes DNAT redirecting SSH (port 22) connections on the external interface of the firewall to the internal SSH server. Traffic is allowed only among internal machines (line 11), from internal machines to the Internet for web-browsing (line 14) and from the Internet to the LAN for accessing the SSH server (line 16).

The IFCL configuration $f$ of Figure 3c encodes the `pf` configuration in Figure 3b. For example, in node $q_3$ we accept as they are packets belonging to communications among internal machines ($p_{sIP} \in 192.168.0.0/24 \wedge p_{dIP} \in 192.168.0.0/24$) or directed to SSH server ($d(p) = 192.168.0.6 : 22$). Other packets are dropped. Note that NAT is managed by the nodes $q_0$ and $q_2$, according to how `pf` deals with translations.

### 3.4. *The denotational semantics of IFCL*

We now define the denotational semantics of the firewalls expressed according to Definition 2. This new semantics is a function from packets to transformations, called *fw-function*, that precisely implements the policy that the firewall has to enforce. The idea is to compose the transformations applied to a packet $p$ in each node of the path it follows in the control diagram.




(a) The network of our examples.

```
1  ### NAT rules ###
2  nat from 192.168.0.0/24 to ! 192.168.0.0/24 -> 151.15.1.5
3  rdr proto tcp to 151.15.1.5 port 22 -> 192.168.0.6
4
5  ### Filtering rules ###
6  # Default drop policy
7  block all
8  # Allow outgoing connections by the firewall
9  pass out from 151.15.1.5 to any
10 # Allow arbitrary traffic among intranets
11 pass from 192.168.0.0/24 to 192.168.0.0/24
12
13 # Allow HTTP/HTTPS outgoing traffic
14 pass out proto tcp to port {80, 443}
15 # Allow SSH/HTTPS incoming traffic to the corresponding hosts
16 pass proto tcp to 192.168.0.6 port 22
```

(b) A simple pf configuration file.

$$f(q_i)(p) = (q_f)(p) = (id : id, id : id)$$

$$f(q_0)(p) = \begin{cases} (\lambda_{192.168.0.6} : id, id : id) & \text{if } d(p) = 151.15.1.5 : 22 \\ (id : id, id : id) & \text{otherwise} \end{cases}$$

$$f(q_2)(p) = \begin{cases} (id : id, \lambda_{151.15.1.5} : id) & \text{if } p_{sIP} \in 192.168.0.0/24 \land \\ & \quad p_{dIP} \notin 192.168.0.0/24 \\ (id : id, id : id) & \text{otherwise} \end{cases}$$

$$f(q_1)(p) = \begin{cases} (id : id, id : id) & \text{if } (p_{sIP} \in 192.168.0.0/24 \land p_{dIP} \in 192.168.0.0/24) \\ & \quad \lor d(p) = 192.168.0.6 : 22 \\ \lambda_\bot & \text{otherwise} \end{cases}$$

$$f(q_3)(p) = \begin{cases} (id : id, id : id) & \text{if } p_{sIP} = 151.15.1.5 \lor \\ & \quad (p_{sIP} \in 192.168.0.0/24 \land p_{dIP} \in 192.168.0.0/24) \\ & \quad \lor p_{dPort} \in \{80, 443\} \\ & \quad \lor d(p) = 192.168.0.6 : 22 \\ \lambda_\bot & \text{otherwise} \end{cases}$$

(c) The corresponding IFCL configuration.

Figure 3: The network of our examples, and an example of firewall.

**Definition 3.** *Let $\mathbb{P} \to \mathcal{T}_\mathbb{P}$ be the set of* fw-functions*, and let $\mathcal{F} = (C, f)$ be a firewall. The* semantics *of $\mathcal{F}$ is*

$$(\!| \mathcal{F} |\!) = \odot^{\mathcal{F}}_{\{q_i\}} q_i : \mathbb{P} \to \mathcal{T}_\mathbb{P}$$

*where, for any $I \subseteq Q$*

$$\odot^{(C,f)}_I (q)(p) = \begin{cases} \odot^{(C,f)}_{I \cup \{q'\}}(q')(p') \circ t & \text{if } q \neq q_f \land t \neq \lambda_\bot \land q' \notin I \\ t & \text{if } q = q_f \lor t = \lambda_\bot \\ \lambda_\circlearrowleft & \text{if } q' \in I \end{cases}$$

*with $t = f(q)(p)$, $p' = t(p)$ and $q' = \delta(q, p')$.*

Note that in the definition above, while traversing the nodes of the control diagram, we not only apply the transformations to packets, but we also accumulate and compose them. Indeed, our goal is to characterize the overall transformation a packet undergoes, rather than simply determine whether it is accepted or not. Recall that in each step the packet $p$ can change and that $\lambda_\bot$ immediately drops it. The special transformation $\lambda_\circlearrowleft$ is applied as soon as a loop is detected, and for that the index $I$ accumulates the nodes already visited. According to our experiments with the real firewall systems described in Section 2, cycling packets are usually dropped, i.e. $\lambda_\circlearrowleft = \lambda_\bot$, but no official documentations state this in clear. Thus, best practice suggests to avoid considering configurations where packets cycle. For this reason in the following we only consider firewalls $\mathcal{F}$ where no packets cycle: $(\!| \mathcal{F} |\!)(p) \neq \lambda_\circlearrowleft$ for any packet $p$.

**Example 2.** *Consider a packet $p$ such that $s(p) = 1.1.1.1 : 4444$ and $d(p) = 151.15.1.5 : 22$ entering the firewall described in Subsection 3.3. Its processing starts in node $q_i$ and follows the path $q_0 \cdot q_1 \cdot q_f$. The initial and final nodes accept the packets as it is. Since $s(p) \notin \mathcal{S}$, i.e., it is not a firewall interface, $p$ goes to $q_0$ where it matches the first condition of $f(q_0)$ and the transformation $(id : id, \lambda_{192.168.0.6} : id)$ is applied, replacing the destination IP with 192.168.0.6 so obtaining $p'$ (leaving unchanged the other fields of the packet). The packet $p'$ reaches $q_1$, where it is accepted as it is (with transformation $(id : id, id : id)$) by the first case of $f(q_1)$, because $d(p') = 192.168.0.6 : 22$. Finally, since $d(p') \in \mathcal{S}$, the packet reaches $q_f$ where it is accepted. Thus, the overall transformation associated to the packet by the firewall is $(id : id, id : id) \circ (id : id, id : id) \circ (id : id, \lambda_{192.168.0.6} : id) \circ (id : id, id : id) = (id : id, \lambda_{192.168.0.6} : id)$.*

## 4. Examples of Inexpressible Policies

There are some policies that are not expressible in all configration languages, unless one resorts to external tools or tags. Here, we present some examples of policies that are not expressible in some Unix firewall languages. We describe why

```
1  *nat
2  :PREROUTING ACCEPT [0:0]
3  :INPUT ACCEPT [0:0]
4  :OUTPUT ACCEPT [0:0]
5  :POSTROUTING ACCEPT [0:0]
6
7  -A PREROUTING -p udp --dport 53 -j DNAT --to 9.9.9.9
8  -A OUTPUT -p udp --dport 53 -j DNAT --to 9.9.9.9
9  -A POSTROUTING ! -d 192.168.0.0/24 -j SNAT --to 151.15.1.5
10 -A INPUT ! -d 192.168.0.1 -j SNAT --to 151.15.1.5
11
12 COMMIT
13
14 *filter
15 :INPUT DROP [0:0]
16 :FORWARD DROP [0:0]
17 :OUTPUT DROP [0:0]
18
19 -A OUTPUT -p udp -d 9.9.9.9 --dport 53 -j ACCEPT
20 -A OUTPUT -p tcp -s 192.168.0.1 ! -d
21              192.168.0.0/24 --dport 80 -j ACCEPT
22 -A INPUT -p udp -d 9.9.9.9 --dport 53 -j ACCEPT
23 -A INPUT -p tcp -s 192.168.0.0/24 ! -d
24              192.168.0.1 --dport 80 -j ACCEPT
25 -A FORWARD -p udp -d 9.9.9.9 --dport 53 -j ACCEPT
26 -A FORWARD -p tcp -s 192.168.0.0/24 ! -d
27              192.168.0.0/24 --dport 80 -j ACCEPT
28
29 COMMIT
```

Figure 4: An `iptables` configuration not expressible in `pf` and in `ipfw`.

$$\tau(p) = \begin{cases} (id:id, \lambda_{151.15.1.5}:id) & \text{if } p_{sIP} \in 192.168.0.0/24 \\ & \quad \wedge\, p_{dIP} \in \mathit{Internet} \\ (\lambda_{192.168.0.6}:id, id:id) & \text{if } p_{sIP} \in \mathit{Internet} \\ & \quad \wedge\, p_{dIP} = 151.15.1.5 \\ & \quad \wedge\, p_{dPort} = 22 \\ \lambda_{\perp} & \text{if } p_{sIP} \in \mathit{Internet} \\ & \quad \wedge\, p_{dIP} \in 192.168.0.0/24 \\ (id:id, id:id) & \text{otherwise} \end{cases}$$

Figure 5: fw-function not expressible in `pf` and `iptables` (*Internet* stands for any public address not in the protected network).

these expressivity limitations arise and how they impact on configurations.

As a first example, take the `iptables` configuration of Figure 4. The firewall applies a SNAT to each packet leaving the LAN to change its source address to 151.15.1.5, and permits connections to only start from inside the LAN. Also, the new connections using the DNS protocol (port 53) are redirected to the server with IP 9.9.9.9. Finally, every other connection from the LAN to the Internet is prevented, except for HTTP (port 80).

Now, consider a packet $p$ with $d(p) = 8.8.8.8 : 53$ and $s(p) = 192.168.0.1 : 50000$. The configuration above transforms $p$ into $p'$ with $d(p') = 9.9.9.9 : 53$ and $s(p') = 151.15.1.5 : 50000$. However, no configuration in `pf` behaves the same, because it cannot apply a DNAT to $p$. Roughly, in the control diagram of `pf`, $p$ only follows the path $q_i \cdot q_2 \cdot q_3 \cdot q_f$ and cannot apply a DNAT, as highlighted by the labels in Figure 2c For similar reasons, neither `ipfw` can express such a behavior.

As a second example assume the administrator wants to implement a policy enforcing the following behavior. The connection requests *(i)* from the local network to the Internet are allowed and subject to SNAT, *(ii)* from the Internet to the firewall on port 22 are redirected via DNAT to the internal host 192.168.0.6, *(iii)* from the Internet to internal hosts are denied, while *(iv)* all the other connection requests are allowed (between internals, among firewall and internals and so on).

The fw-function $\tau$ representing this behavior is in Figure 5, where *Internet* is the set of all the public IP addresses. This function is expressible neither in `pf`, nor in `iptables`, unless one resorts to the additional mechanism of tags. By the encoding of `iptables` and `pf` in Subsection 3.2 and by their control diagrams, we know that they both apply DNAT to packets coming from the Internet *before* checking other rules. Thus, they cannot discriminate between two packets $p_1$ and $p_2$ that come from the Internet but that are directed both on port 22 to the firewall and to the internal host 192.168.0.6, respectively. Neither `pf` nor `iptables` treat these two packets differently, hence they cannot express $\tau$, since $\tau(p_1) = (\lambda_{192.168.0.6} : id, id : id)$ and $\tau(p_2) = \lambda_{\perp}$.

It is worth noting that combining the two examples above one can write a policy expressible by neither `iptables`, nor `ipfw`, nor `pf`. Yet IFCL can express it. More precisely, no policy expressible by the first three firewall languages comply with the following behavior. The connection requests *(i)* from the local network to the Internet are allowed and subject to SNAT, *(ii)* from the Internet to the firewall on port 22 are redirected via DNAT to the internal host 192.168.0.6, *(iii)* from the Internet to internal hosts are denied, *(iv)* on port 53 are redirected via DNAT to the server 9.9.9.9; while *(v)* all the other connection requests are allowed.

## 5. Expressivity of Firewall Configuration Languages

Here, we introduce two formal notions of expressivity that characterize the set of policies expressible by each language. We provide a logical definition of these notions that will be turned in an operational style and then checked by our tool *F2F* in the next section. The first notion is *individual expressivity* in which we assume that the firewall manages single packets in isolation. The second is *function expressivity* that characterizes what is expressible taking care also of how a transformation on a packet may affect that of another packet. Actually, these notions only consider the *legal configurations* of a language, i.e., those meeting the constrains of the language in hand. We provide also a formal characterization of those constraints. We only use below the basic targets ACCEPT, DROP and NAT and avoid ad hoc constructs, e.g., tagging packets. This choice is driven by the empirical evidence that most of the configurations freely available use these targets only [23]. Note that control flow instructions are implicitly considered in the following, because they are macro-expanded in IFCL. Also, we formally relate the expressivity of the languages considered so far, according to these two notions.

### 5.1. Legal Configurations

Assume that a firewall language $\mathcal{L}$ has $C_{\mathcal{L}}$ as control diagram that represents its decision algorithm for accepting or dropping a packet. We require that each node $q$ of $C_{\mathcal{L}}$ also carries information on which operations can be applied to packets

when in node $q$. For example, in the encoding of pf, the rules rdr are only assigned to the node $q_0$ (see Section 3). Hence, a configuration associating $\lambda_\perp$ with some packets in $q_0$ is not valid for pf, yet it is for IFCL.

We represent this additional information in a handy manner by decorating each node $q$ with *cap-labels* representing the transformations allowed in $q$. Formally, we are making explicit that languages put constraints by restricting the image of the function that the configuration $f$ associates with the nodes of the control diagram. Figure 2 shows the cap-labels for the languages we are considering, within rectangles (we assume ID, representing the transformation *id*, be always present, therefore omitted). Hereafter, let *IP* and *Port* denote the set of IP addresses and ports, respectively.

**Definition 4** (Cap-labels and allowed transformations)**.** *Given a control diagram $C$ with nodes $Q$, and the set of cap-labels $\mathbb{L} = \{\mathit{ID}, \mathit{DNAT}, \mathit{SNAT}, \mathit{DROP}\}$, a* cap-label assignment *is a function $V : Q \to 2^{\mathbb{L}}$.*
*Furthermore, we define the mapping $\varepsilon$ that associates a cap-label $l \in \mathbb{L}$ with the set of allowed transformations as follows:*

$$\varepsilon(\mathit{ID}) = \{(id : id\,,\ id : id)\} \qquad \varepsilon(\mathit{DROP}) = \{\lambda_\perp\}$$
$$\varepsilon(\mathit{DNAT}) = \Lambda \times \{id : id\} \qquad \varepsilon(\mathit{SNAT}) = \{id : id\} \times \Lambda$$

*where $\Lambda = \{\lambda_a : \lambda_{a'} \mid a \in IP, a' \in Port\}$.*

*We extend $\varepsilon$ to homomorphically operate on sets $L \subseteq \mathbb{L}$ of labels, i.e. returning the union of $\varepsilon(l)$, for $l \in L$.*

We call *legal* those configurations that respect the cap-label assignment. Formally,

**Definition 5** (Legal configuration)**.** *A configuration $f : Q \to \mathbb{P} \to \mathcal{T}_{\mathbb{P}}$, for a control diagram $C_{\mathcal{L}}$ with nodes $Q$ is* legal for *a cap-label assignment $V$ if and only if for each node $q \in Q$, $\varepsilon(V(q))$ includes the image of $f(q)$.*

Given a language $\mathcal{L}$, let hereafter $V_{\mathcal{L}}$ be the cap-label assignment for $\mathcal{L}$. Note that $V_{\mathcal{L}}$ induces a direct connection between $\mathcal{L}$ and its legal configurations. Consequently, the expressivity of a firewall language can be established by just examining its legal configurations, and thus our two notions of expressivity only consider legal configurations.

### 5.2. Individual Expressivity

The *individual expressivity* of a firewall language describes which transformations a legal configurations can apply to packets. Below we first define it intensionally. We then characterize it constructively in terms of traces, i.e., labeled paths along the control diagram of the language, so providing the bases of the algorithms of Section 6.

#### 5.2.1. Expressible Pairs

We start with an intensional characterization of the legal transformations that can be applied to each single packet.

**Definition 6.** *The set of* expressible pairs *of a language $\mathcal{L}$ is*

$$E_{\mathcal{L}} = \{\, (p,t) \mid \exists f.\, f \text{ is legal for } V_{\mathcal{L}} \wedge (\!|(C_{\mathcal{L}}, f)|\!)(p) = t \,\}$$

We now give a constructive characterization of this expressivity through the notion of trace, i.e., a pair composed by a "complete" acyclic path that a packet may traverse in a control diagram and by the sequence of cap-label of the traversed nodes. A complete path starts from the initial node and either ends in the final node or in one dropping the packet; in addition, for each node $q$ the cap-label is among the permitted ones $V(q)$. Formally,

**Definition 7.** *A* trace *of a language $\mathcal{L}$ is a pair $h = (\pi, v)$ where $\pi = q_0 \cdot q_1 \ldots \cdot q_n$, $v = l_0 \cdot l_1 \ldots \cdot l_n$, with $q_0 = q_i$, are such that $\forall i, j.\ i \neq j \Rightarrow q_i \neq q_j$, and $\forall j. (q_j, \psi, q_{j+1}) \in A$ for some $\psi$, $l_j \in V_{\mathcal{L}}(q_j)$, $\forall j \neq n. (q_j \neq q_f \wedge l_j \neq \mathit{DROP})$ and either $q_n = q_f$ or $l_n = \mathit{DROP}$. Finally, let $\mathcal{H}_{\mathcal{L}}$ be the set of the traces of $\mathcal{L}$.*

By abuse of notation, given a trace $(\pi, v)$ we call a configuration $f$ *legal* for $v$ when $\forall p \in \mathbb{P}, q_j \in \pi.\ f(q_j)(p) \in \varepsilon(l_j)$.

**Example 3.** *The following are traces of* pf*:*

$$(q_i \cdot q_2 \cdot q_3 \cdot q_f, \mathit{ID} \cdot \mathit{ID} \cdot \mathit{ID} \cdot \mathit{ID}), \quad (q_i \cdot q_0 \cdot q_1, \mathit{ID} \cdot \mathit{DNAT} \cdot \mathit{DROP})$$

*Instead the following two are* not *traces of* pf*:*

$$(q_i \cdot q_2 \cdot q_3 \cdot q_f, \mathit{ID} \cdot \mathit{DNAT} \cdot \mathit{ID} \cdot \mathit{ID}), \quad (q_i \cdot q_2 \cdot q_3, \mathit{ID} \cdot \mathit{SNAT} \cdot \mathit{ID})$$

*The first is not because of the second cap-label, $\mathit{DNAT} \notin V_{pf}(q_3)$, the second because it is not complete: neither the path ends with $q_f$ nor the sequence of its labels ends with $\mathit{DROP}$.*

We now define the *capability of a trace* $h = (\pi, v)$ that describes how a packet may be transformed by the nodes of $\pi$. Intuitively, we consider all the possible compositions of the transformations allowed by the cap-labels. Recall that the arcs of control diagrams are guarded by predicates that must be satisfied when a packet flows through them. These predicates determine which packets and transformations must be considered. Given a trace $h = (\pi, v)$, let $\psi_j$ be the predicates labelling the arcs outgoing $q_j$ in the path $q_i \cdot q_1 \ldots \cdot q_n \in \pi$.

**Definition 8.** *The* capability of a trace $h = (\pi, v)$ *is the set $\widetilde{E}_h \subseteq \mathbb{P} \times \mathcal{T}_{\mathbb{P}}$, containing the pairs $(p, t)$ such that*

$$\exists\, t_1, \ldots t_n.\ t_n \circ \cdots \circ t_1 = t \ \wedge \forall j.\ t_j \in \varepsilon(l_j) \ \wedge\ j < n \to \psi_j(p_j)$$

*where $\forall j.\ p_j = (t_j \circ \cdots \circ t_1)(p)$.*
*Finally, the* capability of the traces of a language $\mathcal{L}$ *is*

$$\widetilde{E}_{\mathcal{L}} = \bigcup_{h \in \mathcal{H}_{\mathcal{L}}} \widetilde{E}_h$$

**Example 4.** *Consider the trace of* pf

$$h = (\pi, v) = (q_i \cdot q_2 \cdot q_3 \cdot q_f,\ \mathit{ID} \cdot \mathit{SNAT} \cdot \mathit{ID} \cdot \mathit{ID})$$

*and, assuming that a packet is fully specified by its destination and source addresses, let*

$$p_1 = (8.8.8.8 : 22,\ 192.168.0.1 : 50000)$$
$$t_1 = (id : id,\ \lambda_{151.15.1.5} : \lambda_{50000})$$

*Then the pair* $(p_1, t_1)$ *is in* $\widetilde{E}_h$*, because the transformations*

$$id \cdot (id : id,\ \lambda_{151.15.1.5} : \lambda_{50000}) \cdot id \cdot id$$

*verify the conditions of Definition 8.*
*Instead, the pair* $(p_2, t_2)$ *where*

$$p_2 = (9.9.9.9 : 22,\ 192.168.0.1 : 50000)$$
$$t_2 = (\lambda_{8.8.8.8} : \lambda_{22},\ id : id)$$

*is not in* $\widetilde{E}_h$*, because no* DNAT *occurs in* $v = \texttt{ID} \cdot \texttt{SNAT} \cdot \texttt{ID} \cdot \texttt{ID}$*.*
*Finally, the pair* $(p_3, t_3)$ *where*

$$p_3 = (192.168.0.1 : 80,\ 23.23.23.23 : 50000)$$
$$t_3 = (\lambda_{192.168.0.8} : \lambda_{80},\ id : id)$$

*is not in* $\widetilde{E}_h$*, because* $s(p_3) \notin \mathcal{S}$*.*

The following theorem assures that a pair (packet, transformation) is expressible by $\mathcal{L}$ if and only if it is expressible by one of its traces $h$; in other words, the capability of the traces of a language coincides with the set of its expressible pairs. More precisely, Definition 6 and 8 are equivalent.

**Theorem 1.** $E_{\mathcal{L}} = \widetilde{E}_{\mathcal{L}}$.

*5.2.2. Algorithmically Characterizing Individual Expressivity*

The operational characterization of individual expressivity is based on Algorithm 1 and on Theorems 2 and 3 below.

Algorithm 1 checks whether a packet $p$ can follow a given trace $h = (\pi, v)$ by following the path $\pi$ and verifying if $p$ satisfies all the predicates $\psi_j$ labelling the arc outgoing $q_j$. It uses the auxiliary functions *length*, *head* and *tail* on sequences with the usual meaning, and the following *Ext* predicate (recall from Definition 1 that $\psi_w$ is the predicate $\psi$ restricted on the field $w$)

$$Ext(\psi, L) = \bigwedge_{w \notin \bigcup_{l \in L} \gamma(l)} \psi_w \text{ where}$$

$$\gamma(l) = \begin{cases} \{dIP, dPort\} & \text{if } l = \texttt{DNAT} \\ \{sIP, sPort\} & \text{if } l = \texttt{SNAT} \\ W & \text{if } l = \texttt{DROP} \\ \emptyset & \text{if } l = \texttt{ID} \end{cases}$$

A few comments are in order. The cap-labels encountered along the trace $h$ play an important role, and are accumulated in the auxiliary set *CL* (line 6). Actually, the function *Ext* weakens the predicate $\psi_j$ by removing the conditions affected by the elements of *CL*; the resulting predicate is then applied to $p$ to verify if the arc outgoing $q_j$ can be taken (line 8). As soon as the algorithm traverses a node $q$ with cap-label DNAT (SNAT respectively), all the predicates on the destination (source, respectively) addresses are not checked against $p$ any longer. This is because, after leaving $q$, the original values of the destination (source, respectively) fields of $p$ have been changed, thus the relevant predicates will be evaluated on the updated fields. The arc predicates are also accumulated into $\overline{\psi}$ (line 7) and checked

**Algorithm 1** Checks if a packet $p$ may follow the trace $h = (\pi, v)$

```
1: function CHECK_FLOW(p, (π, v))
2:     CL ← ∅
3:     ψ̄ ← true
4:     while length(π) > 1 do
5:         (q_j, l_j) ← (head(π), head(v))
6:         CL ← CL ∪ {l_j}
7:         ψ̄ ← Ext(ψ̄, l_j) ∧ ψ_j
8:         if ¬(Sat(ψ̄) ∧ Ext(ψ_j, CL)(p)) then return false
9:         (π, v) ← (tail(π), tail(v))
10:    return true
```

for consistency through a call to a *SAT* procedure (line 8). Intuitively, in a node labeled by SNAT the packet source can always be associated with a value satisfying the constraints of the next arcs in the trace, unless some predicates are contradictory. Since Algorithm 1 considers each node and label of the trace only once, its complexity is linear in the length of the trace, because in the worst case, it visits the nodes in $h$ only once.

The individual expressivity of a language can be characterized by the following theorem, where the function CHECK_FLOW is computed by Algorithm 1, and the function REV($h$) reverses the trace $h$. Note that the packet $p$ traversing the trace $h$ must become $t(p)$ and keep satisfying the predicates on the arcs also after some of its fields have been transformed. If different from those in $t(p)$, the new values of a field can thus be arbitrary chosen, provided that the predicates are satisfied, because the values of the destination (source, respectively) fields between two DNAT (SNAT) are immaterial. Therefore, it suffices checking the destination fields (source, respectively) from the last DNAT (SNAT) label onward. This is actually obtained by reversing the trace $h$ and applying again Algorithm 1. Below, we use the function $\hat{\varepsilon}$ that extends $\varepsilon$ to the second component of traces $v$ as follows:

$$\hat{\varepsilon}(l \cdot v) = \hat{\varepsilon}(v) \circ \varepsilon(l)$$

Note that $\hat{\varepsilon}(v) = \lambda_\perp$ if the last element of $v$ is DROP.

**Theorem 2.** *Given a trace* $h = (\pi, v)$*, the pair* $(p, t)$ *is in* $\tilde{E}_h$ *iff*

$$t \in \hat{\varepsilon}(v) \wedge \mathit{CHECK_FLOW}(h, p) \wedge \mathit{CHECK_FLOW}(\mathit{REV}(h), t(p))$$

**Example 5.** *Consider the pair* $(p_1, t_1)$ *and the trace h of Example 4. To make sure that* $(p_1, t_1) \in E_{\mathcal{H}}(h)$*, we first check that* $t_1 \in \hat{\varepsilon}(v) = \varepsilon(\texttt{SNAT})$*; then, CHECK_FLOW(h,* $p_1$*) returns* true *because* $s(p_1) \in \mathcal{S}$ *and* $d(p_1) \notin \mathcal{S}$*; finally, CHECK_FLOW(REV(h),* $p_1$*) is also* true *because the fields changed by* $t_1$ *do not appear on the arcs of the path of* $p_1$*.*

*The pair* $(p_3, t_3)$ *instead is not in* $E_{\mathcal{H}}(h)$*, because* $s(p_3) \notin \mathcal{S}$*.*

*5.2.3. Comparing the Individual Expressivity of Languages*

To compare the individual expressivity of different languages, we finitely enumerate their expressible pairs. For that, we use the intuition behind Algorithm 1 and Theorem 2 to partition the

set of pairs $(p, t)$ in equivalence classes $\omega$ so that if $(p_1, t_1) \in \omega$ is expressible (not expressible, respectively), then all the pairs in $\omega$ are expressible (not expressible, respectively) as well. Two pairs $(p, t)$ and $(p', t')$ belong to the same equivalence class when they follow "similar" paths, in a sense made precise below. As we will see, a language can have different such equivalence classes, which can be efficiently determined. Since their number is small, one can easily enumerate the expressible pairs of different languages and compare them. Formally:

**Definition 9.** *Let* $\mathbb{T} = \{\varepsilon(\textit{ID}), \varepsilon(\textit{DNAT}), \varepsilon(\textit{SNAT}), \varepsilon(\textit{DROP}), \Lambda \times \Lambda\}$ *be a partition of the set of transformations* $\mathcal{T}_{\mathbb{P}}$*; let* $\Psi$ *be the set of the predicates labeling the arcs of the control diagram of a given language* $\mathcal{L}$*; and let* $g \colon \mathbb{P} \to 2^{\Psi}$ *be defined as*

$$g(p) = \{\psi \in \Psi \mid p \textit{ satisfies } \psi\}$$

*Then, for any* $X_1$ *and* $X_2$ *subsets of* $\Psi$ *and for any* $Y \in \mathbb{T}$*, let* $\Omega$ *contain the following sets of pairs* $(p, t)$

$$\omega_{X_1,Y,X_2} = \{(p, t) \mid g(p) = X_1 \wedge (t(p) \neq \bot \Rightarrow g(t(p)) = X_2) \wedge t \in Y\}$$

Two pairs $(p, t)$ and $(p', t')$ belong to an equivalence class $\omega_{X_1,Y,X_2}$ whenever they fulfill the three conditions roughly described below. First, if $p$ satisfies the set $X_1$, so must $p'$, therefore both packets will each follow a path in the same set. Analogously, when $p$ and $p'$ are not dropped, the transformation $t(p)$ satisfies $X_2$ if and only if $t'(p')$ does, so guaranteeing that these transformations share the possible next steps. Finally, the same set of cap-labels $Y$ is associated with both transformations $t$ and $t'$.

**Example 6.** *The two pairs* $(p_1, t_1)$ *and* $(p_2, t_2)$ *where*

$$\begin{aligned} p_1 &= (8.8.8.8 : 80,\ 192.168.0.1 : 50000) \\ t_1 &= (\lambda_{7.7.7.7} : \lambda_{80},\ id : id) \\ p_2 &= (9.9.9.9 : 80,\ 192.168.0.1 : 50000) \\ t_2 &= (\lambda_{151.15.1.5} : \lambda_{80},\ id : id) \end{aligned}$$

*are not in the same equivalence class.*
*Indeed, even though* $g(p_1) = g(p_2) = \{s(p) \in \mathcal{S}, d(p) \notin \mathcal{S}\}$*, and* $t_1, t_2 \in \varepsilon(\textit{DNAT})$*, we have that* $t_1(p_1)$ *verifies* $d(p) \notin \mathcal{S}$ *whereas* $t_2(p_2)$ *does not, hence* $g(t_1(p_1)) \neq g(t_2(p_2))$*.*

We have the following theorem:

**Theorem 3.** $\Omega$ *is a partition of* $\mathbb{P} \times \mathcal{T}_{\mathbb{P}}$ *such that the elements of* $\omega_{X_1,Y,X_2}$ *are either all expressible or all not expressible.*

Using the equivalence classes we succinctly enumerate and compare the expressible pairs of firewall languages, summarized in Table 2. The first three columns show the equivalence class under consideration; the last columns describe whether the pair representative for each class is expressible or not by a language. The details for selecting a representative pair of a given class are in Appendix A.

Note that the predicates on the arcs of the control diagrams of `iptables`, `ipfw` and `pf` only check the *IP* addresses (see 3.2). A packet can traverse an arc depending on whether the source or destination of the packet belongs to $\mathcal{S}$. Therefore, we simply use the set $\Psi = \{d(p) \in \mathcal{S}, d(p) \notin \mathcal{S}, s(p) \in \mathcal{S}, s(p) \notin \mathcal{S}\}$, the satisfiable subsets of which follow:

$$\begin{array}{ll} \{d(p) \in \mathcal{S}, s(p) \in \mathcal{S}\} & \{d(p) \in \mathcal{S}, s(p) \notin \mathcal{S}\} \\ \{d(p) \notin \mathcal{S}, s(p) \in \mathcal{S}\} & \{d(p) \notin \mathcal{S}, s(p) \notin \mathcal{S}\} \end{array}$$

The theorem below states that only IFCL and `iptables` express all the pairs, while `pf` and `ipfw` have the same expressive power. Its proof directly follows by inspecting Table 2.

**Theorem 4.** $E_{\texttt{pf}} = E_{\texttt{ipfw}} \subsetneq E_{\texttt{iptables}} = E_{IFCL} = \mathbb{P} \times \mathcal{T}_{\mathbb{P}}$

### 5.3. Function Expressivity

Individual expressivity only considers packets in isolation. However, a firewall handles many different packets at the same time, each subject to different transformations. The interaction of the transformations applied to different packets must thus be considered, because packets can become related depending on the transformations their undergo. This affects the policies that the various languages can express, as shown by Example 7 below. We first say that a fw-function $\tau$ is expressible by a language if the semantics of one of its firewalls is $\tau$.

**Definition 10.** *Given a language* $\mathcal{L}$ *with control diagram* $C_{\mathcal{L}}$ *and a cap-label assignment* $V_{\mathcal{L}}$*, a fw-function* $\tau \colon \mathbb{P} \to \mathcal{T}_{\mathbb{P}}$ *is* $\mathcal{L}$-expressible *iff* $\exists f$ *legal for* $V_{\mathcal{L}}$ *such that* $(\!|(C_{\mathcal{L}}, f)|\!) = \tau$*.*
*Call* $T_{\mathcal{L}}$ *the set of the fw-functions expressible by* $\mathcal{L}$*.*

Note that this notion differs from individual expressivity because it considers *all* the pairs $(p, \tau(p))$ at the same time. The following theorem gives a necessary condition for function expressibility.

**Theorem 5.** *Given a language* $\mathcal{L}$ *and a fw-function* $\tau$

$$\tau \in T_{\mathcal{L}} \textit{ only if } \forall p \in \mathbb{P}. (p, \tau(p)) \in E_{\mathcal{L}}.$$

The following example shows that the condition above is not a sufficient one.

**Example 7.** *The fw-function* $\tau$ *defined below is not* `pf`*-expressible, although all the pairs* $(p, \tau(p))$ *are expressible in* `pf`*.*

$$\tau(p) = \begin{cases} (\lambda_{8.8.8.8} : id, \lambda_{151.15.1.5} : id) & \textit{if } p_{sIP} = 192.168.0.8 \wedge \\ & \quad p_{dIP} = 6.6.6.6 \\ (\lambda_{8.8.8.8} : id, id : id) & \textit{if } p_{sIP} = 192.168.0.8 \wedge \\ & \quad p_{dIP} = 7.7.7.7 \\ \lambda_{\bot} & \textit{otherwise} \end{cases}$$

*All the pairs* $(p, \tau(p))$ *are expressible in* `pf` *(in Section 6 we present an algorithm to check that this is actually the case). To show that function* $\tau$ *is not expressible by* `pf`*, we assume by contradiction that it is. Take two packets* $p$ *and* $p'$*, both with source IP 192.168.0.8, that only differ on their destination IP which are 6.6.6.6 for* $p$ *and 7.7.7.7 for* $p'$*. They must traverse the nodes* $q_i, q_0, q_1, q_2, q_3$ *and* $q_f$*, and both are transformed by* DNAT *in* $q_0$ *into* $p''$ *with destination IP 8.8.8.8. When* $p$ *and* $p'$ *arrive in node* $q_2$ *they have been already transformed in* $p''$*, and when applying* SNAT *the two cannot be taken apart. But* $\tau$ *says that one has to be subject to* SNAT *and the other has not.*

| | $X_1$ | | $Y$ | $X_2$ | | $(p,t)$ | $E_{\mathcal{L}}$ | |
|---|---|---|---|---|---|---|---|---|
| | $d(p)$ | $s(p)$ | | $d(t(p))$ | $s(t(p))$ | $((d(p), s(p)),\ t)$ | `pf`/`ipfw` | `iptables` |
| 1 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{ID})$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, a:r), id)$ | ✓ | ✓ |
| 2 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{ID})$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, b:r), id)$ | ✓ | ✓ |
| 3 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{ID})$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, a:r), id)$ | ✓ | ✓ |
| 4 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{ID})$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, b:r), id)$ | ✓ | ✓ |
| 5 | – | $\in \mathcal{S}$ | $\varepsilon(\texttt{ID})$ | – | $\notin \mathcal{S}$ | $\square$ | | |
| 6 | $\in \mathcal{S}$ | – | $\varepsilon(\texttt{ID})$ | $\notin \mathcal{S}$ | – | $\square$ | | |
| 7 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, a:r), (\lambda_a:\lambda_r, id:id))$ | ✓ | ✓ |
| 8 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, a:r), (\lambda_b:\lambda_r, id:id))$ | ✗ | ✓ |
| 9 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, b:r), (\lambda_a:\lambda_r, id:id))$ | ✓ | ✓ |
| 10 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, b:r), (\lambda_b:\lambda_r, id:id))$ | ✓ | ✓ |
| 11 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, a:r), (\lambda_a:\lambda_r, id:id))$ | ✗ | ✓ |
| 12 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, a:r), (\lambda_b:\lambda_r, id:id))$ | ✗ | ✓ |
| 13 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, b:r), (\lambda_a:\lambda_r, id:id))$ | ✓ | ✓ |
| 14 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, b:r), (\lambda_b:\lambda_r, id:id))$ | ✓ | ✓ |
| 15 | – | $\in \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | – | $\notin \mathcal{S}$ | $\square$ | | |
| 16 | – | $\notin \mathcal{S}$ | $\varepsilon(\texttt{DNAT})$ | – | $\in \mathcal{S}$ | $\square$ | | |
| 17 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, a:r), (id:id, \lambda_a:\lambda_r))$ | ✓ | ✓ |
| 18 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, a:r), (id:id, \lambda_b:\lambda_r))$ | ✓ | ✓ |
| 19 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, b:r), (id:id, \lambda_a:\lambda_r))$ | ✗ | ✓ |
| 20 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, b:r), (id:id, \lambda_b:\lambda_r))$ | ✗ | ✓ |
| 21 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, a:r), (id:id, \lambda_a:\lambda_r))$ | ✓ | ✓ |
| 22 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, a:r), (id:id, \lambda_b:\lambda_r))$ | ✓ | ✓ |
| 23 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, b:r), (id:id, \lambda_a:\lambda_r))$ | ✓ | ✓ |
| 24 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{SNAT})$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, b:r), (id:id, \lambda_b:\lambda_r))$ | ✓ | ✓ |
| 25 | $\in \mathcal{S}$ | – | $\varepsilon(\texttt{SNAT})$ | $\notin \mathcal{S}$ | – | $\square$ | | |
| 26 | $\notin \mathcal{S}$ | – | $\varepsilon(\texttt{SNAT})$ | $\in \mathcal{S}$ | – | $\square$ | | |
| 27 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, a:r), (\lambda_a:\lambda_r, \lambda_a:\lambda_r))$ | ✓ | ✓ |
| 28 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, a:r), (\lambda_a:\lambda_r, \lambda_b:\lambda_r))$ | ✓ | ✓ |
| 29 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, a:r), (\lambda_b:\lambda_r, \lambda_a:\lambda_r))$ | ✗ | ✓ |
| 30 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, a:r), (\lambda_b:\lambda_r, \lambda_b:\lambda_r))$ | ✗ | ✓ |
| 31 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, b:r), (\lambda_a:\lambda_r, \lambda_a:\lambda_r))$ | ✗ | ✓ |
| 32 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, b:r), (\lambda_a:\lambda_r, \lambda_b:\lambda_r))$ | ✗ | ✓ |
| 33 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((a:r, b:r), (\lambda_b:\lambda_r, \lambda_a:\lambda_r))$ | ✓ | ✓ |
| 34 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((a:r, b:r), (\lambda_b:\lambda_r, \lambda_b:\lambda_r))$ | ✓ | ✓ |
| 35 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, a:r), (\lambda_a:\lambda_r, \lambda_a:\lambda_r))$ | ✗ | ✓ |
| 36 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, a:r), (\lambda_a:\lambda_r, \lambda_b:\lambda_r))$ | ✗ | ✓ |
| 37 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, a:r), (\lambda_b:\lambda_r, \lambda_a:\lambda_r))$ | ✗ | ✓ |
| 38 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, a:r), (\lambda_b:\lambda_r, \lambda_b:\lambda_r))$ | ✗ | ✓ |
| 39 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, b:r), (\lambda_a:\lambda_r, \lambda_a:\lambda_r))$ | ✗ | ✓ |
| 40 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, b:r), (\lambda_a:\lambda_r, \lambda_b:\lambda_r))$ | ✗ | ✓ |
| 41 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $((b:r, b:r), (\lambda_b:\lambda_r, \lambda_a:\lambda_r))$ | ✓ | ✓ |
| 42 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\Lambda \times \Lambda$ | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $((b:r, b:r), (\lambda_b:\lambda_r, \lambda_b:\lambda_r))$ | ✓ | ✓ |
| 43 | $\in \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{DROP})$ | – | – | $((a:r, a:r), \bot)$ | ✓ | ✓ |
| 44 | $\in \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{DROP})$ | – | – | $((a:r, b:r), \bot)$ | ✓ | ✓ |
| 45 | $\notin \mathcal{S}$ | $\in \mathcal{S}$ | $\varepsilon(\texttt{DROP})$ | – | – | $((b:r, a:r), \bot)$ | ✓ | ✓ |
| 46 | $\notin \mathcal{S}$ | $\notin \mathcal{S}$ | $\varepsilon(\texttt{DROP})$ | – | – | $((b:r, b:r), \bot)$ | ✓ | ✓ |

Table 2: The expressible pairs of `iptables`, `pf` and `ipfw`. The first two columns contain the predicates in the subset $X_1$ (the – stands for both $\in$ and $\notin \mathcal{S}$); the third column contains the set of transformations $Y$; the fourth and fifth columns contain the predicates in the subset $X_2$; the sixth column contains the representative pair $(p,t)$ for the given class or $\square$ if the class is empty; the other columns have ✓ if the class containing the pair $(p,t)$ is expressible or ✗ if not.

**Algorithm 2** Check function expressivity of $\mathcal{L}$

```
1:  function CHECK_FUNCTION(τ, C, V)
2:      for all q ∈ Q do g(q) ← ∅
3:      for all (P, t) ∈ τ do
4:          h ← COMPUTE_TRACE(C, V, ([P], t))
5:          if h = Null then print (P, t) not expressible
6:          else g ← CHECK_PAIR(h, (P, t), g)

7:  function COMPUTE_TRACE(C, V, (p, t))
8:      for all h ∈ H_L do
9:          if t ∈ Ê(v) ∧ CHECK_FLOW(h, p)∧
               CHECK_FLOW(REV(h), t(p)) then return h
10:     return Null

11: function CHECK_PAIR(h, (P, t), g)
12:     (P_@, t_▹, t_◃) ← (P, t, (id : id, id : id))
13:     for all (q, l) ∈ h do
14:         (t_@, t_▹) ← SPLIT(t_▹, l)
15:         for all ((P̃_@, t̃_◃, t̃_@), (P̃, t̃)) ∈ g(q) s.t. t̃_@ ≠ t_@ do
16:             P^×_@ ← P_@ ∩ P̃_@
17:             if P^×_@ ≠ ∅ then
18:                 P^× ← P ∩ t_◃^{-1}(P^×_@)
19:                 P̃^× ← P̃ ∩ t̃_◃^{-1}(P^×_@)
20:                 print (P^×, t), (P̃^×, t̃) clash in q : (P^×_@, t_@, t̃_@)
21:         g(q) ← g(q) ∪ {((P_@, t_◃, t_@), (P, t))}
22:         t_◃ ← t_@ ∘ t_◃
23:         P_@ ← t_@(P_@)
24:     return g
```

Function expressivity enables us to further compare `iptables`, `ipfw` and `pf` that originate a partial order, the top of which is IFCL that can express all the fw-functions.

**Theorem 6.**

- $T_{\texttt{pf}} \subsetneq T_{\texttt{ipfw}} \subsetneq T_{IFCL}$
- $T_{\texttt{iptables}} \subsetneq T_{IFCL}$
- $T_{\texttt{pf}} \nsubseteq T_{\texttt{iptables}}$, $T_{\texttt{iptables}} \nsubseteq T_{\texttt{pf}}$
- $T_{\texttt{ipfw}} \nsubseteq T_{\texttt{iptables}}$, $T_{\texttt{iptables}} \nsubseteq T_{\texttt{ipfw}}$

## 6. Mechanically Checking Expressivity with *F2F*

In this section we describe how *F2F* checks if a policy can be expressed by a given firewall language. In particular, we first present the data structure used to represent the semantics of a firewall and then the algorithm which our tool relies on. Finally, we describe the workflow of our tool.

### *6.1. Effective Representation of Firewall Semantics*

In set theoretical terms, the semantics of a configuration, i.e., a fw-function $\tau$, is a set of pairs $(p, t)$. To concisely represent these pairs, we first group in a set $P$ all the packets subject to the same transformation $t$, and we present the function $\tau$ as a set of $\tau$-*pairs* $(P, t)$. We then efficiently represent fw-functions using multi-cubes [24]. A multi-cube generalizes the notion of cube, and can be considered as the cartesian product of the union of intervals. A multi-cube compactly represents a set of packets, and each union specifies the interval to which the values of the corresponding field of a packet belong. We then represent transformations as functions from multi-cubes to multi-cubes that change the intervals of their argument.

**Example 8.** *Consider the following multi-cube:*

$$( [192.198.0.2, 192.198.0.10] : [8080, 8080], \quad [192.198.0.1, 192.198.0.1] : [0, 22] \cup [25, 100] )$$

*It represents the set of packets with destination address in* $[192.198.0.2, 192.198.0.10]$ *and port* $8080$*, and source address* $192.198.0.1$ *with source port in the intervals* $[0, 22]$ *or* $[25, 100]$.

Since a set $P$ in a $\tau$-pair not always forms a multi-cube, we need an intermediate step. In such a case, $P$ is partitioned in a set of $P_i$ each yielding a multi-cube. In addition, we make sure that every $\tau$-pair $(P_i, t)$ is such that all $(p, t)$ with $p \in P_i$ belong to the same equivalence class of $\Omega$. This further partition of $\tau$-pairs is based on the fact that each subset $\omega_{X_1,Y,X_2} \in \Omega$ can be represented as $\{(P_{X_1,t,X_2}, t) | t \in Y\}$ lifting Definition 9 to sets of packets and taking a single transformation $t$. We can then componentwise split each $(P_i, t)$ into its intersections with the pairs $(P_{X_1,t,X_2}, t)$.

After the steps above, given a $\tau$-pair $(P, t)$ we can pick up a single packet $p \in P$, denoted by $[P]$, with the guarantee that any other packet in $P$ is transformed in the same manner and follows the same trace.

**Example 9.** *Consider the following fw-function $\tau$, where Internet stands for any public address not in the protected LAN.*

$$\tau(p)=\begin{cases}(\lambda_{192.168.0.6} : id, id : id) & \text{if } p_{sIP} \in Internet \wedge \\ & \quad p_{dIP} = 151.15.1.5 \wedge p_{dPort} = 22 \\ \lambda_\perp & \text{if } p_{sIP} \in Internet \wedge \\ & \quad (p_{dIP} \in 192.168.0.0/24 \vee \\ & \quad p_{dPort} \neq 22) \\ (id : id, id : id) & \text{otherwise}\end{cases}$$

*The set of packets to be dropped is not a multi-cube, and is represented as the union of* $P_{\perp 1} = 192.168.0.0/24 : _ \times Internet : _$ *and* $P_{\perp 2} = _ : _ \times Internet : [0 - 21] \cup [23 - 65536]$*, where* $_$ *stands for "any value." Note also that* $(P_{\perp 1}, \lambda_\perp)$ *is not included in any equivalence class, and thus it is split into* ([192.168.0.2, 192.168.0.255] : $_ \times Internet : _,\ \lambda_\perp$) *and* ({192.168.0.1} : $_ \times Internet : _,\ \lambda_\perp$) *(we omit here the invalid address* 192.168.0.0*).*

### *6.2. An Algorithm for Checking Expressivity*

We now describe Algorithm 2, which is the core of *F2F*. Roughly, Algorithm 2 iterates on all the $\tau$-pairs $(P, t)$ representing a fw-function $\tau$. For each pair, it creates an *extended configuration* $g$ that associates each node $q$ with the required transformation $t_q$ (occurring in the cap-labels of $q$). In doing so, the

algorithm checks that no *clash* occurs. Intuitively, a configuration has clashes when in a given node of the control diagram it prescribes to apply "incompatible" transformations to a packet, e.g., drop and transform it at the same time, or change its destination field to two different addresses. Formally,

**Definition 11** (Clashes)**.** *Given a language $\mathcal{L}$, two pairs $(p, t)$ and $(\tilde{p}, \tilde{t})$ collide in a node $q \in C_{\mathcal{L}}$ with clash $(p_@, t_@, \tilde{t}_@)$ iff, for all configurations $f$ legal for $V_{\mathcal{L}}$, $\tilde{t}_@ \neq t_@$ and*

$$(\!|(C_{\mathcal{L}}, f)|\!)(p) = t \Rightarrow f(q)(p_@) = t_@$$
$$(\!|(C_{\mathcal{L}}, f)|\!)(\tilde{p}) = \tilde{t} \Rightarrow f(q)(p_@) = \tilde{t}_@$$

*Also, we say that two $\tau$-pairs $(P, t)$ and $(\tilde{P}, \tilde{t})$ collide in $q \in C_{\mathcal{L}}$ with clash $(P_@, t_@, \tilde{t}_@)$ iff, for all configurations $f$ legal for $V_{\mathcal{L}}$*

$$(\ \forall p \in P.\ (\!|(C_{\mathcal{L}}, f)|\!)(p) = t\ ) \Rightarrow \forall p_@ \in P_@.\ f(q)(p_@) = t_@$$
$$(\ \forall \tilde{p} \in \tilde{P}.\ (\!|(C_{\mathcal{L}}, f)|\!)(\tilde{p}) = \tilde{t}\ ) \Rightarrow \forall p_@ \in P_@.\ f(q)(p_@) = \tilde{t}_@$$

*and $\tilde{t}_@ \neq t_@$.*

The CHECK_FUNCTION in Algorithm 2 takes as input an fw-function $\tau$ and a firewall language $\mathcal{L}$, represented by its control diagram $C$ and the cap-label assignment $V$. First, the function initializes the extended configuration $g$ we want to build as "empty" and then iterates the following steps over all $\tau$-pairs $(P, t)$ of $\tau$. For each $\tau$-pair $(P, t)$ we look for a trace $h$ of $C$ that can express it. If there is none, the transformation $t$ cannot be associated with the packets represented by $P$ (line 5). Otherwise, two cases may arise. The first is when two packets processed in a node $q$ of $C$ clash, as detailed below (line 17); the second example of Section 4 illustrates a clash. The other possible case is when the configuration $g$ can be correctly updated with the new $\tau$-pair $(P, t)$.

The auxiliary function COMPUTE_TRACE returns a trace that expresses $(p, t)$, if any, when $p = [P]$ is one of the packets of $P$ (cf. Theorem 2).

The auxiliary function CHECK_PAIR is called with a trace $h$ returned by COMPUTE_TRACE; a $\tau$-pair; and an (incomplete) extended configuration $g$. An extended configuration maps each node to a pair and intuitively extends a configuration with the transformations annotated by the $\tau$-pair in which they occur. Roughly, CHECK_PAIR visits sequentially each node $q$ of $h$; computes the transformations that are applied in it; and updates the configuration $g$ accordingly. More precisely, CHECK_PAIR computes the transformations $t_{\triangleleft}$ and $t_@$, and a new multi-cube $P_@$ for each node $q$. The transformation $t_{\triangleleft}$ describes how the packets have been rewritten along the sub-path of $h$ from $q_i$ to $q$. Whereas $t_@$ is the transformation to be applied in $q$ to packets in the multi-cube $P_@$, obtained from the initial one $P$ applying $t_{\triangleleft}$. For each node, $t_@$ is extracted from the transformation $t_{\triangleright}$ that records the part of $t$ still to be considered. We get $t_@$ through the SPLIT function, a sort of inverse of function composition, that given a $t_{\triangleright}$ and a cap-label $l$ returns the transformation satisfying $l$ and removes it from $t_{\triangleright}$. Summing up, each node $q$ is associated with pairs of the form $((P_@, t_{\triangleleft}, t_@), (P, t))$. Suppose now that a node is associated with two such pairs, one with $t_@$ to be applied to $P_@$ and the other with a different transformation $\tilde{t}_@$ to be applied to $\tilde{P}_@$. A clash occurs if the two multi-cubes have non-empty intersection (line 16). Indeed, the packets in the intersection will be transformed in two conflicting ways by $t_@$ and $\tilde{t}_@$, and the clash is reported to user (line 17). The lines 18 and 19 recover the packets of $P$ and $\tilde{P}$ clashing in $q$, by computing the pre-image of $P^{\times}_@$ under the transformation applied from $q_i$ to $q$. The last three lines update $g$, $t_{\triangleleft}$ and $P_@$, respectively.

**Example 10.** *Consider the following two $\tau$-pairs, where Internet stands for any public address not in the protected LAN, and _ stands for "any port"*

$$(P_1, t_1) = \\ (\{151.15.1.5\} : \{22\} \times \mathit{Internet} : _,\ (\lambda_{192.168.0.6} : id, id : id))$$
$$(P_2, t_2) = \\ ([192.168.0.2, 192.168.0.255] : _ \times \mathit{Internet} : _,\ \lambda_\perp)$$

*Choose $[P_1] = (151.15.1.5 : 22 \times 6.6.6.6 : 55)$ as the representative for $P_1$. It follows the trace $([q_i, q_0, q_1, q_f], [\mathit{ID}, \mathit{DNAT}, \mathit{ID}, \mathit{ID}])$. After evaluating $q_1$, the call to CHECK_PAIR results in the extended configuration $g$ where $\overrightarrow{id}$ stands for $(id : id, id : id)$*

$$g(q_i) = (P_1, \overrightarrow{id}, \overrightarrow{id}), (P_1, t_1)$$
$$g(q_0) = (P_1, \overrightarrow{id}, t_1), (P_1, t_1)$$
$$g(q_1) = ((\{192.168.0.6\} : \{22\} \times \mathit{Internet} : _), t_1, \overrightarrow{id}), (P_1, t_1)$$

*In the same way, choose $[P_2] = (192.168.0.3 : 11 \times 7.7.7.7 : 44)$ that follows the trace $([q_i, q_0, q_1], [\mathit{ID}, \mathit{ID}, \mathit{DROP}])$. When $q_0$ is reached, the call to CHECK_PAIR results in updating $g$ as follows*

$$g(q_i) = g(q_i) \cup (P_2, \overrightarrow{id}, \overrightarrow{id}), (P_2, t_2)$$
$$g(q_0) = (P_2, \overrightarrow{id}, \overrightarrow{id}), (P_2, t_2)$$

*When $(q_1, \mathit{DROP})$ is considered, $P^{\times}_@ = (P_@ \cap \tilde{P}_@) = \tilde{P}_@ \neq \emptyset$, where $P_@ = P_2$ and $\tilde{P}_@ = (\{192.168.0.6\} : \{22\} \times \mathit{Internet} : _)$, making $(P_1, t_1)$ and $((\{192.168.0.6\} : \{22\} \times \mathit{Internet} : _), t_2)$ to clash in $q_1$.*

To compute the cost of Algorithm 2 we consider the specific policy $\tau$ on which it runs. More prcisely, in the formula below we take care of the cardinality: of $\mathcal{H}_{\mathcal{L}}$, the traces of $\mathcal{L}$; of $Q_{\mathcal{L}}$, the nodes in the control diagram of $\mathcal{L}$; of the $\tau$-pairs of $\tau$; and of the intervals in the field $w \in W$ of $P$. For each $\tau$-pair $(P, t)$, the algorithm inspects the traces of $\mathcal{L}$ to select $h$, the one that expresses $(P, t)$. For that, Algorithm 1 is invoked that requires $Q_{\mathcal{L}}$ iterations at most. Then, for each node of $h$ (containing all the nodes of $Q_{\mathcal{L}}$ in the worst case) the intersections in lines 16, 18, and 19 are computed between the actual pair and those that visit the same node (all the other $\tau$-pairs in the worst case). Finally, the intersection between two multicubes $P$ and $\tilde{P}$ requires intersecting the intervals $P_w$ and $\tilde{P}_w$ for each field $w \in W$. Summing up, we obtain the following formula:

$$|\tau| \cdot |Q_{\mathcal{L}}| \cdot (|\mathcal{H}_{\mathcal{L}}| + |\tau| \cdot |W| \cdot max\{|P_w| \mid (P, t) \in \tau\}).$$

Algorithm 2 is correct, as stated below. However, its correctness relies on two assumptions that are satisfied by all the

languages that we have studied so far, including iptables, pf and ipfw. The first is that each pair $(p, t)$ can be expressed by no more than one trace $h \in \mathcal{H}_{\mathcal{L}}$. The only exception is when $t = \lambda_{\perp}$. We choose to only keep the traces $(q_i \ldots q_n), (l_i \ldots \texttt{DROP})$ such that $\forall j < n. \texttt{DROP} \notin V(q_j)$ and $l_j = \texttt{ID}$ (recall that ID is associated with every node of $C$). Roughly, these traces are the shortest that drop a packet without transforming any of its fields. Taking the shortest traces does not affect the expressivity of a language, and there is no point in changing discarded packets. The other assumption is that the trace $h$ contains no repeated DNAT or SNAT labels, and makes the extraction of $t_@$ from $t_\triangleright$ well-defined, in particular when $t = \lambda_{\perp}$.

**Theorem 7.** *For each firewall language $\mathcal{L}$ and fw-function $\tau$, the Algorithm 2 is correct because it prints all and only*

1. *the $\tau$-pairs $(P, t)$ not expressible by $\mathcal{L}$;*
2. *the $\tau$-pairs $(P, t)$ and $(\tilde{P}, \tilde{t})$ that clash on some node $q$.*

This theorem has some important practical consequences, as granted by the following corollary.

**Corollary 1.** *A fw-function $\tau$ is expressible by $\mathcal{L}$ if and only if Algorithm 2 prints nothing.*

Thus, if the administrators solve all the inexpressible pairs and the reported clashes, then they obtain a configuration for the desired system. The inexpressible pairs can be simply removed if irrelevant, otherwise the administrator can patch the configuration through calls to external code, e.g., NFQUEUE target in iptables [6]. There are two different ways to solve clashes, depending on whether the intended behavior of the system is implemented. One is selecting the more appropriate transformation $t_@$ and $\tilde{t}_@$ for every clashing $\tau$-pairs. Actually, acting on the transformations may change the semantics of the firewall. The other solution is semantics-preserving: one may use other features of the language to distinguish between the two clashing sets of packets, e.g., using tags or external code. In subsection 7.1 we show on an example that *F2F* alerts an administrators when this is the case, and proposes a tag-based solution.

### 6.3. Workflow of the Tool

We implement a tool, called *F2F* [13] that applies Algorithm 2. To support the user in analyzing and migrating real-world configurations, the tool gets as input a policy expressed in one of the configuration languages of Section 2 or as a fw-function in a tabular form. Also, the user chooses the wanted target language. When the user provides a configuration, the tool computes its semantics as a preprocessing step, thus obtaining a fw-function. Then, it checks if the policy is expressible by the target language, and notifies the user with exact information if this is not the case.

Figure 6 sketches the workflow of *F2F* (from left to right) when converting a configuration from ipfw to pf. In the upper, left part there is the source configuration that is then encoded in IFCL, from which the tool extracts the semantics as a fw-function. The bottom part depicts the control diagram and the label assignment for pf, and the step computing its expressivity. The tool than checks whether the semantics of the source configuration is expressible in pf, and if this is not the case, it produces a report with the inexpressible and clashing pairs.

For the computation of the semantics we rely on a tailored version of *FWS* [12], a tool based on IFCL that computes an abstract representation of the semantics of the given firewall configuration.

## 7. Evaluation

### 7.1. A Case Study: FirewallBuilder

Consider again the scenario in Figure 3a showing a typical network of a small company, and assume the administrator produces a configuration for pf using the policy management system FirewallBuilder. Below we show how *F2F* may fit a policy creation and management workflow; how it may help the administrator in both understanding the limitations of FirewallBuilder and of pf; and in supporting possible fixes of the detected problems. We also consider iptables and ipfw and we show that they have similar weaknesses.

*FirewallBuilder.* FirewallBuilder is a well-known tool for Unix-based systems that supports the administrator to write firewall policies in a tabular form and then compiles them to the most common firewall languages. Two separate tables are given, the first one for network address translation and the second one for packet filtering. Since FirewallBuilder does not come with a clear definition of its semantics, the user not always knows exactly how these tables are used to determine the destiny of packets. Also, the tables may contain rules that are superfluous or conflicting depending on which table is used first, e.g., when a packet $p$ is transformed with $t$ in the translation table, and $p$ or $t(p)$ is discarded in the filtering table. This may lead to clashes when producing the target configuration.

*Policy Requirements and Specification.* We start by putting forward the requirements that the firewall policy of the scenario of Figure 3a must meet:

1. LAN hosts freely communicate with each other;
2. LAN hosts access the firewall via SSH (port 22), only;
3. The company hosts (a LAN host or the firewall) can freely send packets to the Internet;
4. Packets from the Internet are discarded, if not directed to the public IP of the firewall with port 22 or 443;
5. Packets from the Internet directed to port 22 or 443 are redirected to the internal SSH server (at address 192.168.0.6) or to the HTTPS server (at address 192.168.0.7);
6. When a company host tries to connect to a DNS service on the Internet, on port 53, the packet is redirected to 8.8.8.8;

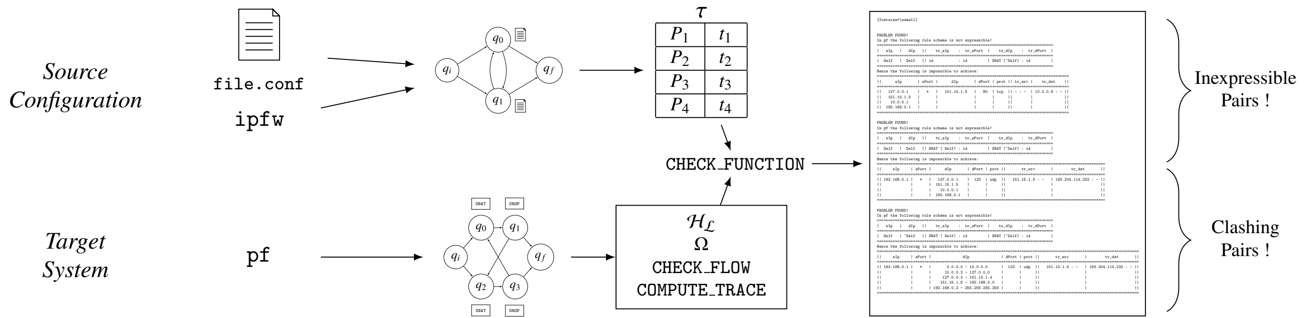



Figure 6: Schema of *F2F*.

| dstIP | dstPort | srcIP | srcPort | DNAT | SNAT |
|---|---|---|---|---|---|
| not (151.14.1.5 or 192.168.0.0/24) | 53 | 192.168.0.0/24 | any | 8.8.8.8 : id | 151.15.1.5 : id |
| not (151.14.1.5 or 192.168.0.0/24) | 53 | 151.15.1.5 | any | 8.8.8.8 : id | id : id |
| not (151.14.1.5 or 192.168.0.0/24) | not 53 | 151.15.1.5, 192.168.0.0/24 | any | id : id | 151.15.1.5: id |
| 192.168.0.1, 151.15.1.5 | 22 | 192.168.0.2 - 192.168.0.255 | any | id : id | id : id |
| 192.168.0.2 - 192.168.0.255 | any | 192.168.0.2 - 192.168.0.255 | any | id : id | id : id |
| 151.15.1.5 | 443 | not (151.14.1.5 or 192.168.0.0/24) | any | 192.168.0.7 : id | id : id |
| 151.15.1.5 | 22 | not (151.14.1.5 or 192.168.0.0/24) | any | 192.168.0.6 : id | id : id |

Table 3: An example policy for a typical network.

7. The source address of all the packets leaving a company host towards the Internet is replaced with the public IP of the firewall.

These requirements give rise to the policy represented in Table 3, expressed in a declarative form as the list of the accepted packets and their transformations.

*Implementing the Policy and Detecting Problems.* Typically, defining a policy using FirewallBuilder is straightforward but there may be some cases where the administrator needs to check how the translation and filtering tables are used to manage packets, because the documentation is not always clear. For example, consider the last line of Table 3 that conflicts with the implicit indication to discard the packets from the Internet and with destination 192.168.0.6. If the administrator assumes that the translation table is checked before the filtering one, the packets directed to 192.168.0.6 must be accepted, thus obtaining the `pf` configuration in Figure 8 (which is actually the output of the tool, with a little of maquillage for legibility). However, the configuration fails in encoding the desired policy since Requirement (4) is not met. If instead the filtering table is inspected first, the last line of the policy is simply ignored. Summing up, in neither cases FirewallBuilder implements correctly the policy in Table 3, and in addition no warning is notified.

*F2F* instead signals a clash on the considered configuration. The output in Figure 7 shows that in node $q_1$ the packets from the Internet directed to the internal server in 192.168.0.6 are indistinguishable from the ones originally directed to the firewall because of the DNAT in node $q_0$. Our tool suggest the administrator to use tags in the node $q_0$ for distinguishing the two clashing (multi-cubes of) packets. As a matter of fact, there are two clashes, and the tool suggests tags to fix both.

*F2F* signals other problems in the analyzed configuration, because there are also two sets of inexpressible pairs. The first set is $(P, t)$, where $P$ contains the packets from the external interface of the firewall towards the Internet on port 53, and $t$ is a DNAT to the address 8.8.8.8. In other words, the firewall can access any DNS server on the Internet, not only the one prescribed by the policy. Even though the inexpressible pairs are considered by FirewallBuilder when producing its output configuration, the resulting rules are simply ignored by `pf`. Note that the administrator gets no warning about this problem and could wrongly think that the configuration enforces them (as it seems at first sight). Of course, this misconfiguration is possibly dangerous for security. Moreover, `pf` cannot apply the translations $t$ above to the packets in $P$, thus the administrator can only fix this problem resorting to external tools. The second set of inexpressible pairs is quite similar to what just described.

*Problems with `ipfw` and `iptables`.* Similar problems arise if one compiles the policy in Table 3 to `ipfw` and to `iptables`. In more detail, the generated `ipfw` configuration suffers from no clashes but from the same inexpressible pairs. Whereas, the `iptables` configuration presents the same clashes, but no inexpressible pairs.

### 7.2. Performance on real configurations

We evaluated the *F2F* effectiveness against real world configurations [13]. The experiments are performed on a desktop computer with an i7-7700 processor (3.60GHz) and 8Gb RAM, running Ubuntu 20.04.3 LTS. The results are in Table 4: the first column reports the name of the configuration; the second one the number of lines of the configuration; the third one the time taken by *F2F* to compute the IFCL-configuration, to extract its $\tau$-function, and to check both kinds of expressivity; finally the last one is the time for checking the expressivity only. Performance is acceptable for all the configurations, and the time

```
$ sudo ./f2f table Example/interfaces Example/table.conf pf

!!! Inexpressible Pair Found !!!
=======================================================================================================
||     sIp     | sPort |              dIp              | dPort | prot ||      tr_src      |    tr_dst    ||
=======================================================================================================
|| 192.168.0.1 |   *   |       0.0.0.0 - 151.15.1.4    |   53  |  *   || 151.15.1.5 : id | 8.8.8.8 : id ||
||             |       |  151.15.1.6 - 192.167.255.255 |       |      ||                  |              ||
||             |       | 192.168.1.0 - 255.255.255.255 |       |      ||                  |              ||
=======================================================================================================

!!! Inexpressible Pair Found !!!
==============================================================================================
||    sIp     | sPort |              dIp              | dPort | prot ||  tr_src |    tr_dst    ||
==============================================================================================
|| 151.15.1.5 |   *   |       0.0.0.0 - 151.15.1.4    |   53  |  *   || id : id | 8.8.8.8 : id ||
||            |       |  151.15.1.6 - 192.167.255.255 |       |      ||         |              ||
||            |       | 192.168.1.0 - 255.255.255.255 |       |      ||         |              ||
==============================================================================================

!!! Clashing Pairs Found !!!
(P1, t1):
=============================================================================
||              sIp              | sPort |     dIp     | dPort | prot ||  tr  ||
=============================================================================
||       0.0.0.0 - 151.15.1.4    |   *   | 192.168.0.7 |  443  |  *   || DROP ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||
=============================================================================
(P2, t2):
==================================================================================================
||              sIp              | sPort |    dIp     | dPort | prot ||  tr_src |      tr_dst      ||
==================================================================================================
||       0.0.0.0 - 151.15.1.4    |   *   | 151.15.1.5 |  443  |  *   || id : id | 192.168.0.7 : id ||
||  151.15.1.6 - 192.167.255.255 |       |            |       |      ||         |                  ||
|| 192.168.1.0 - 255.255.255.255 |       |            |       |      ||         |                  ||
==================================================================================================
in node q1:
with [P@ || t1@ || t2@]:
===================================================================================================
||              sIp              | sPort |     dIp     | dPort | prot || tr1  || tr2_src | tr2_dst ||
===================================================================================================
||       0.0.0.0 - 151.15.1.4    |   *   | 192.168.0.7 |  443  |  *   || DROP || id : id | id : id ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||         |         ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||         |         ||
===================================================================================================
Hint: Apply tags to P1 in node q0 and use them to choose the transformation in node q1

!!! Clashing Pairs Found !!!
(P1, t1):
=============================================================================
||              sIp              | sPort |     dIp     | dPort | prot ||  tr  ||
=============================================================================
||       0.0.0.0 - 151.15.1.4    |   *   | 192.168.0.6 |   22  |  *   || DROP ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||
=============================================================================
(P2, t2):
==================================================================================================
||              sIp              | sPort |    dIp     | dPort | prot ||  tr_src |      tr_dst      ||
==================================================================================================
||       0.0.0.0 - 151.15.1.4    |   *   | 151.15.1.5 |   22  |  *   || id : id | 192.168.0.6 : id ||
||  151.15.1.6 - 192.167.255.255 |       |            |       |      ||         |                  ||
|| 192.168.1.0 - 255.255.255.255 |       |            |       |      ||         |                  ||
==================================================================================================
in node q1:
with [P@ || t1@ || t2@]:
===================================================================================================
||              sIp              | sPort |     dIp     | dPort | prot || tr1  || tr2_src | tr2_dst ||
===================================================================================================
||       0.0.0.0 - 151.15.1.4    |   *   | 192.168.0.6 |   22  |  *   || DROP || id : id | id : id ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||         |         ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||         |         ||
===================================================================================================
Hint: Apply tags to P1 in node q0 and use them to choose the transformation in node q1
```

Figure 7: *F2F* output when checking the example policy for `pf`.

```
1  rdr from <NotCompany> to 151.15.1.5 port 443 -> 192.168.0.7
2  rdr from <NotCompany> to 151.15.1.5 port 22 -> 192.168.0.6
3  nat from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany> port 54 -> 151.15.1.5
4  nat from { 151.15.1.5 , 192.168.0.0/24 } to { 6.6.6.6, 8.8.8.8 } -> 151.15.1.5
5  nat from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany> -> 151.15.1.5
6  nat from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany> port 54 -> 151.15.1.5
7  rdr from 192.168.0.0/24 to <NotCompany> port 53 -> 8.8.8.8
8  nat from 192.168.0.0/24 to 8.8.8.8 port 53 -> 151.15.1.5
9  rdr from 151.15.1.5 to <NotCompany> port 53 -> 8.8.8.8
10
11 pass quick from <NotCompany> to 192.168.0.7 port 443
12 pass quick from <NotCompany> to 192.168.0.6 port 22
13 pass quick from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany>
14 pass quick from 192.168.0.2 - 192.168.0.255 to 192.168.0.2 - 192.168.0.255
15 pass quick from 192.168.0.2 - 192.168.0.255 to { 151.15.1.5 , 192.168.0.1 } port 22
16 block quick  from any  to any
```

Figure 8: FirewallBuilder output when compiling the example policy for `pf`, where `<NotCompany>` is a table containing all the IP addresses that are not used by the company.

| Configuration | Lines | Total time (*s*) | Checking time (*s*) |
|---|---|---|---|
| ticket_openwrt | 128 | 7.00 | 0.05 |
| sqrl_shorewall | 106 | 190.10 | 0.48 |
| random_srv | 16 | 7.43 | 0.05 |
| memphis_testbed | 46 | 6.51 | 0.05 |
| medium sized company | 639 | 75.54 | 0.20 |
| kornwall | 88 | 54.73 | 0.54 |
| home router | 130 | 20.67 | 0.21 |
| github_myiptables | 53 | 5.35 | 0.03 |
| eduroam_laptop | 57 | 7.97 | 0.10 |
| blog_a | 51 | 11.03 | 0.13 |

Table 4: Experimental results of *F2F* against real-world configurations.

for checking expressivity is negligible. When checking `pf` and `ipfw`, we found two inexpressible pairs in the configuration `eduroam_laptop`: in both pairs a DNAT is applied to a packet from an address in $\mathcal{S}$ to one not in $\mathcal{S}$ (lines 11 and 12 of Table 2). For the same systems, we also found two clashes in the configuration `medium sized company`, between packets to be both translated and dropped.

## 8. Related Work and Concluding Remarks

### *8.1. Related Work*

To the best of our knowledge, this is one of the first papers that investigate the expressive power of firewall systems by using programming language-based techniques. Some preliminary work of ours on this topic is in [22, 25]. Section 3 extends and improves the early version of the denotational semantics of [22]. The same paper also has a first version of the algorithm for only checking individual expressivity that was used in [25] as a guideline for a preliminary version of *F2F*. Here, we introduce a new algorithm that also checks function expressivity, and we prove its correctness; exploiting it, *F2F* additionally verifies if a policy can be expressed without ad hoc mechanisms, also informing the administrator when tags solve the expressivity problems. A further novelty with our previous work is the assessment of the effectiveness of *F2F* through an experimental evaluation on real configurations, in particular to detect some misbehaviors of FirewallBuilder [1] discussed in sub-section 7.1.

The literature reports on many proposals for modeling and analyzing firewall configurations, e.g., see [26, 27, 28, 1]. However, these papers differ from ours because they do not rely on a formal semantics and because they are used for different purposes than ours. Typically, they either compile from a high level language to a low level one or they verify whether a configuration complies with a given specification.

Since our approach heavily relies on a formal semantics to precisely compare the firewall behavior, below we only focus on those papers that do the same.

Diekmann et al. [18] propose a semantics for a subset of `iptables` and mechanize it using Isabelle/HOL. Their semantics only focuses on filtering, and does not consider packet modifications as NAT. Furthermore, they define and prove correct a simplification procedure that aims to make configurations easier to be analyzed by automatic tools. Differently, our work is mainly focused on studying the expressivity of a policy and on comparing the capabilities of different systems. Moreover, we provide a denotational semantics for different firewall systems, not only `iptables`, and we also deal with NAT.

Adão et al. [15] introduce Mignis, a high level firewall language for specifying abstract filtering policies, which is then compiled into `iptables`. The paper formalizes the semantics of Mignis and provides an operational semantics of `iptables`. This semantics considers packet filtering and NAT and it is used to prove that the Mignis compiler is correct. Successively, Adão et al. [29] propose a denotational semantics for Mignis that maps a configuration into a packet transformer, representing all the accepted packets with the corresponding translations. This semantics is similar to the one we give in Section 3 for IFCL, where the meaning of a configuration is a function from packets to transformations. The main outcome of this line of work is a specific high-level firewall configuration language with a formal semantics. Instead, we formally model existent systems and offer a semantic based tool to support administrators in the definition of their policies. Moreover, we believe that the compiler of Mignis could benefit from the algorithms presented in Section 6 to support more target systems.

Anderson et al. [30] introduce NetKAT, a language for programming a collection of network switches. It is equipped with a denotational and with an axiomatic semantics, both based on

Kleene algebra with tests. Although the primitives provided by NetKAT are similar to those of IFCL, i.e., for filtering, modifying, and transmitting packets, its focus is different from ours. Indeed, their goal is modeling an entire network leaving out the details of the single firewalls, expressed in commercial languages. Whereas in our formal model we focus on the problem of expressivity and we consider policies of a single host, to be implemented in widely adopted configuration languages.

### *8.2. Conclusion*

We have considered `iptables`, `ipfw` and `pf`, the main firewall systems used in Linux, FreeBSD, OpenBSD and MacOS. We have developed a tool based on a rigorous basis, which can be used to check if a given policy is expressible in either language, and reports on the reasons when it cannot be. Our investigation is based on IFCL [12], an intermediate general language for configuring firewalls, which provided us with a formal and common framework in which all the above mentioned firewall systems can be encoded.

We only focus on the common targets provided by all the firewall systems that have a clear semantics and implicitly on the targets for altering the control flow, which however can be macro-expanded [12]. This choice is supported by empirical evidence: most of the configurations available in the wild [23] use these targets only. In addition, most analysis tools [31, 32, 33, 15, 24] only focus on the same set of targets we consider. For example, tags are always neglected, mainly because they are handled in quite incompatible and scarcely documented ways by the various firewall systems, even by different versions of the same system. Also, their usage is subject to specific and sometimes obscure constraints. However, tags are a safe mechanism to solve a specific class of expressivity problems, and our work helps in identifying when they are actually needed to keep packets apart. Also we disregard specific constructs for describing the status of connections, because they not always have a precise, intuitive meaning, and thus we only focus on new connections.

We gave a new denotational semantics to IFCL and thus to the languages it encodes. Using it we have defined two notions of expressivity. The first considers the network address transformations applied to a *single*, arbitrary packet when it is accepted. In this setting, we have proved that `iptables` and IFCL are universal, in that they can express every possible transformation on packets, while not all of them can be defined using `ipfw` and `pf` that turned out to be equally powerful (Theorem 4). The second notion considers the transformations that an *entire* configuration applies to *all* the packets. The later notion is stronger than the first one, because it considers further constraints about the packets that clash in some node, i.e., are indistinguishable but require different transformations.

Our tool efficiently checks both kinds of expressivity. Running the tool we found some policies expressible in `ipfw` but not in `pf`. Also there are policies expressible in `ipfw` but not in `iptables` and viceversa, but them all are expressible in IFCL (Theorem 6).

Our work finds application in building semantics-based tools to support network administrators. As a test, we have used it to assess a quite standard configuration generated by FirewallBuilder, a commonly used policy management system in Linux environment; also we used it within the automatic transcompilation pipeline presented in [11, 14, 34, 12] that supports the analysis, migration and refactoring of real UNIX firewall configurations. In particular, our tool has been used to predict when a policy cannot be expressed, so pointing out the cases when ad hoc extensions to the used firewall language or another language or another mechanism are in order. Remarkably, when the administrator resolves the notified clashes, and removes the non expressible cases, the policy is ready to be implemented and deployed. Summing up, our work answers the question "can I do this without language extensions, like tags?" precisely characterizing in which cases you cannot.

Future work includes considering different firewall systems, like Cisco-IOS, which is particularly challenging because the control diagram is also affected by routing choices. Another research direction is extending our work to languages with tags. Although adapting the denotational semantics is lengthy but straightforward, enhancing the expressivity analysis requires a deep understanding of tag systems and a non trivial re-working of our algorithms.

| *Notation* | *Description* |
|---|---|
| $\cdot$ | juxtaposition operator |
| $dstIP/srcIP$ | destination/source IP address |
| $dstPort/srcPort$ | destination/source port address |
| $p \in \mathbb{P}$ | a packet in the set of IP packets |
| $p_w \in D_w$ | the field of $p$ in the domain $D_w$ |
| $t = (t_{dIP} : t_{dPort}, t_{sIP} : t_{sPort})$ | componentwise transformation of $p$ |
| $\bot \in \mathbb{P}$ | dropped packet, with $t(bot) = \bot$ |
| $\mathcal{L}$ | firewall configuration language |
| | with control diagram $C_{\mathcal{L}}$ with nodes $q$ |
| $\mathcal{F} = (C_{\mathcal{L}}, f)$ | firewall of $\mathcal{L}$ with configuration $f$ |
| $(\!\lvert \mathcal{F} \rvert\!) = \tau$ | fw-function, semantics of $\mathcal{F}$ |
| $\tau(p)$ | behavior of the firewall $\mathcal{F}$ on $p$ |
| $\mathcal{S}$ | addresses of the firewall interfaces |
| $l \in \{\mathtt{ID}, \mathtt{DNAT}, \mathtt{SNAT}, \mathtt{DROP}\}$ | cap-labels associated by $\epsilon$ with nodes $q$ |
| $E_{\mathcal{L}} = \{ (p, t) \}$ | expressible pairs, s.t. |
| | $(\!\lvert (C_{\mathcal{L}}, f) \rvert\!)(p) = t$, and $f$ legal (cf. Def. 5) |
| $h = (\pi, v)$ | trace, with $\pi$ list of nodes ($v$ of cap-labels) |
| $\widetilde{E}_h$ | capability of trace h |
| $\omega_{X_1,Y,X_2}$ | set only containing |
| | expressible / non expressible pairs |
| $\mathcal{L}$-expressible $\tau$ | if $\exists f$ legal s.t. $(\!\lvert (C_{\mathcal{L}}, f) \rvert\!) = \tau$ |
| $(p, t), (\tilde{p}, \tilde{t})$ collide in $q$ | if $f$ applies incompatible transformations |

Table 5: Notation and symbols


## Acknowledgement

The authors wish to warmly thank the anonymous referees for their careful and helpful suggestions.

This work was partially supported by the MIUR project PRIN 2017FTXR7S *IT MATTERS*.


## References


[1] FirewallBuilder, `http://fwbuilder.sourceforge.net/` (2000).

[2] Capirca — Multi-platform ACL Generation System), `https://github.com/google/capirca` (2021).

[3] Migrating from iptables to pf, a love story, http://daemonforums.org/showthread.php?t=7775 (2013).
[4] How to go from iptables to pf?, https://serverfault.com/questions/228313/how-to-go-from-iptables-to-pf (2013).
[5] PF - Packet Tagging (Policy Filtering), https://www.openbsd.org/faq/pf/tagging.html (2020).
[6] Netfilter, https://www.netfilter.org/ (2019).
[7] Queueing to userspace, https://wiki.nftables.org/wiki-nftables/index.php/Queueing_to_userspace (2016).
[8] pfSense — World's Most Trusted Open Source Firewall, https://www.pfsense.org/ (2021).
[9] Suricata, https://www.suricata.io/ (2021).
[10] Snort — Network Intrusion Detection & Prevention System, https://www.snort.org/ (2021).
[11] C. Bodei, P. Degano, L. Galletta, R. Focardi, M. Tempesta, L. Veronese, Language-independent synthesis of firewall policies, in: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, 2018, pp. 92–106.
[12] C. Bodei, L. Ceragioli, P. Degano, R. Focardi, L. Galletta, F. L. Luccio, M. Tempesta, L. Veronese, FWS: analyzing, maintaining and transcompiling firewalls, J. Comput. Secur. 29 (1) (2021) 77–134. doi:10.3233/JCS-200017.
URL https://doi.org/10.3233/JCS-200017
[13] F2F tool, https://github.com/lceragioli/F2F (2021).
[14] C. Bodei, P. Degano, R. Focardi, L. Galletta, M. Tempesta, Transcompiling firewalls, in: L. Bauer, R. Küsters (Eds.), Principles of Security and Trust - 7th International Conference, POST 2018, Vol. 10804 of LNCS, 2018, pp. 303–324.
[15] P. Adão, C. Bozzato, G. Dei Rossi, R. Focardi, F. L. Luccio, Mignis: A Semantic Based Tool for Firewall Configuration, in: proc. of the 27th IEEE CSF, 2014, pp. 351–365.
[16] W. T. Hallahan, E. Zhai, R. Piskac, Automated repair by example for firewalls, in: 2017 Formal Methods in Computer Aided Design (FMCAD), 2017, pp. 220–229. doi:10.23919/FMCAD.2017.8102263.
[17] Y. Bartal, A. J. Mayer, K. Nissim, A. Wool, Firmato: A novel Firewall Management Toolkit, ACM Transactions on Computer Systems 22 (4) (2004) 381–420.
[18] C. Diekmann, L. Hupel, J. Michaelis, M. P. L. Haslbeck, G. Carle, Verified iptables firewall analysis and verification, J. Autom. Reasoning 61 (1-4) (2018) 191–242.
[19] R. Russell, Linux 2.4 Packet Filtering HOWTO, http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html (2002).
[20] The IPFW Firewall, https://www.freebsd.org/doc/handbook/firewalls-ipfw.html (2017).
[21] Packet Filter (PF), https://www.openbsd.org/faq/pf/ (2019).
[22] L. Ceragioli, P. Degano, L. Galletta, Are all firewall systems equally powerful?, in: Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, PLAS'19, ACM, 2019, p. 1–17.
[23] C. Diekmann, net-network: Public Collection of firewall dumps, https://github.com/diekmann/net-network (2017).
[24] K. Jayaraman, N. Bjørner, G. Outhred, C. Kaufman, Automated Analysis and Debugging of Network Connectivity Policies, Tech. rep., Microsoft (2014).
[25] L. Ceragioli, P. Degano, L. Galletta, Checking the Expressivity of Firewall Languages, in: M. Alvim, K. Chatzikokolakis, C. Olarte, F. Valencia (Eds.), The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy, Vol. 11760 of LNCS, Springer Nature, 2019.
[26] F. Cuppens, N. Cuppens-Boulahia, T. Sans, A. Miège, A Formal Approach to Specify and Deploy a Network Security Policy, in: proc. of 2nd IFIP FAST, 2004, pp. 203–218.
[27] S. M. Perez, J. Cabot, J. García-Alfaro, F. Cuppens, N. Cuppens-Boulahia, A Model-Driven Approach for the Extraction of Network Access-Control Policies, in: Proceedings of the Workshop on Model-Driven Security Workshop, MDsec 2012, 2012.
[28] S. N. Foley, U. Neville, A Firewall Algebra for OpenStack, in: Proceedings of the 3rd IEEE Conference on Communications and Network Security, CNS 2015, 2015, pp. 541–549.
[29] P. Adão, R. Focardi, J. D. Guttman, F. L. Luccio, Localizing firewall security policies, in: proc. of the 29th IEEE CSF, Lisbon, Portugal, June 27 - July 1, 2016, pp. 194–209.
[30] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, D. Walker, NetKAT: Semantic Foundations for Networks, in: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, ACM, 2014, pp. 113–126.
[31] A. J. Mayer, A. Wool, E. Ziskind, Fang: A Firewall Analysis Engine, in: proc. of the 21st IEEE S&P 2000, 2000, pp. 177–187.
[32] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, P. Mohapatra, FIREMAN: A Toolkit for FIREwall Modeling and ANalysis, in: Proceedings of the 27th IEEE Symposium on Security and Privacy, S&P 2006, 2006, pp. 199–213.
[33] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, S. Krishnamurthi, The Margrave Tool for Firewall Analysis, in: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010, 2010.
[34] L. Ceragioli, L. Galletta, M. Tempesta, From firewalls to functions and back, in: P. Degano, R. Zunino (Eds.), Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019., Vol. 2315 of CEUR Workshop Proceedings, CEUR-WS.org, 2019.


## Appendix A. Computing the Representative Pairs

We now show how to efficiently build a pair $(p, t)$, representative of a whole $\omega_{X_1,Y,X_2} \neq \emptyset$ without actually computing this equivalence class. One can thus effectively check whether $\omega_{X_1,Y,X_2}$ is expressible by applying Theorem 2 to the pair $(p, t)$. The algorithm relies on two assumptions that hold in all the firewall languages we have considered. The first requires that each predicate on the arcs is expressible as a predicate on a single field of the packet.[1] The second assumption says that given a satisfiable predicate one can mechanically build a packet satisfying it.

Given $X_1, Y, X_2$, the Algorithm 3 computes such a representative or fails if there are none, i.e., if $\omega_{X_1,Y,X_2} = \emptyset$. In it, we let $w$ range over the fields of a packet and of a transformation, including $\{dIP, dPort, sIP, sPort\}$. Recall that we denote the fields of a packet $p$ by $p_w$ and those of a transformation $t$ by $t_w$. In the same way, we split a set of predicates $X$ in the components on the fields, typically $X = X_{dIP} \wedge X_{dPort} \wedge X_{sIP} \wedge X_{sPort}$, where $\wedge$ operates homomorphically. For each $X_w$, the function $\texttt{TAKE_ONE}(X_w)$ returns an address whatsoever if $X_w = \emptyset$, or an address satisfying all the predicates in $X_w$ (second assumption above). Note that $X_w$ might be unsatisfiable, so making $\texttt{TAKE_ONE}(X_w)$ and the whole algorithm fail.

The Algorithm 3 scans the fields $w$ of the packet headers and generates an address satisfying the $w$ component of the predicates in $X_1$. Then it takes a transformation $t'$ in $Y$, if any, that changes the field $w$. In such a case, an address $a$ is taken that satisfies $X_{2_w}$, and the $w$ component of the output transformation $t$ is set to $\lambda_a$. If no $t' \in Y$ changes the field $w$, the transformation $t$ on $w$ is $id$; also, the algorithm fails when the predicates $X_1$ and $X_2$ differ on $w$.

The correctness of Algorithm 3 is guaranteed by the following theorem (which is proved in [22]).

**Theorem 8.** *Given a triple $X_1, Y, X_2$, the Algorithm 3 either returns a pair $(p, t) \in \omega_{X_1,Y,X_2} \neq \emptyset$ or it fails.*

[1] Since one can add new nodes without loosing expressive power, this is the same of asking each predicate on the arcs to be expressible as conjunction of one predicate for each field, possibly *true*.

**Algorithm 3** Build a pair $(p, t) \in \omega_{X_1,Y,X_2}$, if any.

```
1:  if Y = {λ⊥} then t = λ⊥
2:  for all w ∈ W do
3:      p_w ← TAKE_ONE(X1_w)
4:      if Y ≠ {λ⊥} ∧ w ∈ {dIP, dPort, sIP, sPort} then
5:          if ∃t' ∈ Y.t'_w ≠ id then
6:              a ← TAKE_ONE(X2_w)
7:              t_w ← λ_a
8:          else if X1_w ≠ X2_w then FAIL
9:          else t_w ← id
10: return (p, t)
```

## Appendix B. Proofs

**Lemma 1** (IFCL universality)**.** *For each function $\tau : \mathbb{P} \to \mathcal{T}_{\mathbb{P}}$ there exists an IFCL firewall $\mathcal{F}$ such that $(\!|\mathcal{F}|\!) = \tau$.*

*Proof.* Trivial: just take a control diagram with a single node $q$ and a configuration such that $f(q) = \tau$. □

As stated in Section 2, we only consider firewalls where no packets cycle. Then we define the following function, that simplifies $\odot$ by removing cycle detection.

$$\otimes^{(C,f)}(q)(p) = \begin{cases} \otimes^{(C,f)}(q')(p') \circ t & \text{if } q \neq q_f \wedge t \neq \lambda_\perp \\ t & \text{if } q = q_f \vee t = \lambda_\perp \end{cases}$$

with $t = f(q)(p)$, $p' = t(p)$ and $q' = \delta(q, p')$.

Now we state the following general property.

**Lemma 2.** *Let $(C, f)$ be a firewall, then*

$$\forall p \in \mathbb{P}.\ \odot^{(C,f)}_{\{q_i\}}(q)(p) = \lambda_{\circlearrowleft} \vee \odot^{(C,f)}_{\{q_i\}}(q)(p) = \otimes^{(C,f)}(q)(p)$$

*Proof.* We prove the more general statement

$$\forall I \subseteq Q. \forall p \in \mathbb{P}.\ \odot^{(C,f)}_{I}(q)(p) = \lambda_{\circlearrowleft} \vee \odot^{(C,f)}_{I}(q)(p) = \otimes^{(C,f)}(q)(p)$$

The proof proceed by induction on $\odot$. If $q = q_f$ or $t = \lambda_\perp$, then the thesis trivially holds. Assume $q \neq q_f$ and $t \neq \lambda_\perp$, let $t' = f(q)(p)$, $p' = t'(p)$ and $q' = \delta(q, p')$, if $q' \in I$ then the premise is false and the thesis holds. The last case is when $q' \notin I$ and $q \neq q_f$, then by definition $\otimes^{(C,f)}(q)(p) = \otimes^{(C,f)}(q')(p') \circ t'$ and $\odot^{(C,f)}_{I}(q)(p) = \odot^{(C,f)}_{I\cup\{q'\}}(q')(p') \circ t'$. By induction hypothesis, $\forall J \subseteq Q$, $\forall p \in \mathbb{P}$, $\odot^{(C,f)}_{J}(q')(p) = \lambda_{\circlearrowleft} \vee \odot^{(C,f)}_{J}(q')(p) = \otimes^{(C,f)}(q')(p)$. Take $J = I \cup \{q'\}$, if $\odot^{(C,f)}_{I\cup\{q'\}}(q')(p') = \lambda_{\circlearrowleft}$, then $\odot^{(C,f)}_{I}(q)(p) = \lambda_{\circlearrowleft}$ and the thesis follows. Otherwise, $\otimes^{(C,f)}(q')(p') = \odot^{(C,f)}_{I}(q')(p')$ and the thesis follows. □

To prove Theorem 1 we use the following lemma.

**Lemma 3.** *Let $C = (Q, A, q_i, q_f)$ be a control diagram, let $V$ be a cap-label assignment, then the two following condition are equivalent*

$$(i)\quad \exists f \text{ legal for } V.\ \otimes^{(C,f)}(q)(p) = t$$

$$\begin{aligned}(ii)\quad & \exists(\pi, \nu).\ \pi = q_1 \cdot \ldots q_n \wedge \nu = l_1 \cdot \ldots l_n \\ & \wedge\ \forall j. \exists \psi. (q_j, \psi, q_{j+1}) \in A \wedge l_j \in V(q_j) \\ & \wedge\ \forall j \neq n.\ q_j \neq q_f \wedge l_j \neq \texttt{DROP} \\ & \wedge\ \forall i, j.\ i \neq j \Rightarrow q_i \neq q_j \\ & \wedge\ q_1 = q \wedge (q_n = q_f \vee l_n = \texttt{DROP}) \wedge \\ & \exists t_1, t_2, \ldots t_n. \forall j.\ t_j \in \varepsilon(l_j) \wedge \\ & \qquad t_n \circ t_{n-1} \cdots \circ t_1 = t \wedge \\ & \qquad \forall j < n.\ \psi_j(p_j)\end{aligned}$$

*Proof.* We prove separately $(i) \Rightarrow (ii)$ and $(ii) \Rightarrow (i)$.
For $(i) \Rightarrow (ii)$ we proceed for induction on the calls of function $\otimes$. We assume the premise and consider the following exhaustive cases. If $q = q_f$ then $t \in \epsilon l$ for some $l \in V(q_f)$ and we take $\pi = q_f$ and $\nu = l$: $(ii)$ trivially holds. Otherwise if $f(q)(p) = \lambda_\perp$, since $f$ is legal for $V$ then $\texttt{DROP} \in V(q)$; we take $\pi = q$ and $\nu = \texttt{DROP}$ for which $(ii)$ trivially holds. Finally, assume $q \neq q_f$, $f(q)(p) = t' \neq \lambda_\perp$ and $\delta(q, p') = q'$ where $p' = t'(p)$, then $\otimes^{(C,f)}(q)(p) = t$ is equal to $\otimes^{(C,f)}(q')(p') \circ t'$. From the induction hypothesis we have that exists a pair $(\pi', \nu')$ such that all the conjuncts in $(ii)$ hold for it. Since $f$ is legal for $V$ we know that a cap-label $l \in V(q)$ must be such that $f(q)(p) \in \varepsilon(l)$. We build $\pi = q \cdot \pi'$ and $\nu = l \cdot \nu'$. By the induction hypothesis and by construction $\forall j. \exists \psi. (q_j, \psi, q_{j+1}) \in A$, since $\delta(q, p') = q'$; also it holds that $l_j \in V_{\mathcal{L}}(q_j)$. The condition $\forall j \neq n.\ q_j \neq q_f \wedge l_j \neq \texttt{DROP}$ holds by hypothesis on $q$ and by the induction hypothesis for the rest of $\pi$. For construction $q_1 = q$ and by the induction hypothesis $(q_n = q_f \vee l_n = \texttt{DROP})$. $\psi(p')$ holds because of $\delta(q, p') = q'$. Finally $t_n \circ t_{n-1} \cdots \circ t_1 = t$ holds because the the induction hypothesis guarantee $t_n \circ t_{n-1} \cdots \circ t_2 = t''$ such that $t'' \circ t' = t$.

We then show that $(ii) \Rightarrow (i)$ holds. We take $f$ such that $\forall j.\ f'(q_j)(p) = t_j$ and $\forall q \notin \pi.\ f(q)(p) = t$ for whichever $t \in \varepsilon(l)$ with $l$ assigned to $q$. Hence, by construction $f$ is legal for $V$; $\otimes^{(C,f)}(q)(p) = t$ is now proved by induction on the length $n$ of $\pi$ and $\nu$. If $n = 1$ then either $q_1 = q_f$ or $l_1 = \texttt{DROP}$. In the first case then $t \in \epsilon l$ for some $l \in V(q_f)$ and also $\otimes^{(C,f)}(q)(p) = t$. Otherwise if $l_1 = \texttt{DROP}$ then $f(q)(p) = t_1 = \lambda_\perp$. Finally suppose that the statement holds for any $\pi' = q' \cdot \pi''$ and $\nu'$ of length $n$, and take $(\pi, \nu) = (q \cdot \pi', l \cdot \nu')$ of length $n + 1$. By the induction hypothesis and $(ii)$ we have that $\otimes^{(C,f)}(q')(p') = t''$ where $t' = f(q)(p)$, $p' = t'(p)$ and $t = t'' \circ t'$. □

**Theorem 1.** $E_{\mathcal{L}} = \widetilde{E}_{\mathcal{L}}$.

*Proof.* Follows trivially from Lemma 2 and 3. □

**Lemma 4.** *Given a trace $h = (\pi, \nu)$ and a packet $p$, the following are equivalent*

$$(i)\quad \mathit{CHECK_FLOW}(h, p)$$

$$\begin{aligned}(ii)\quad & \exists p_1, \ldots, p_n.\ p_1 = p \wedge \\ & \qquad \forall j < n - 1.\ \psi_j(p_j) \wedge \\ & \qquad \forall j < n. \exists t_j \in \varepsilon(l_j).\ t_j(p_j) = p_{j+1}\end{aligned}$$

*Proof.* Formula $(ii)$ is equivalent to the following in which packet fields are made explicit:

$$\exists p_{1_{w_1}}, \ldots, p_{n_{w_1}}, p_{1_{w_2}}, \ldots, p_{n_{w_2}}, \ldots p_{1_{w_m}}, \ldots, p_{n_{w_m}}.$$

$$p_{1_{w_1}} = p_{w_1} \wedge \dots p_{1_{w_m}} = p_{w_m} \wedge$$
$$\forall j < n-1.\, \psi_{j_{w_1}}(p_{j_{w_1}}) \wedge \dots \wedge \psi_{j_{w_m}}(p_{j_{w_m}}) \wedge$$
$$\forall j < n.\, \exists t_j \in \varepsilon(l_j).\, t_j(p_j)_{w_1} = p_{j+1_{w_1}} \wedge \dots \wedge t_j(p_j)_{w_m} = p_{j+1_{w_m}}$$

We can then substitute

$$\exists t_j \in \varepsilon(l_j).\, t_j(p_j)_{w_1} = p_{j+1_{w_1}} \wedge \dots \wedge t_j(p_j)_{w_m} = p_{j+1_{w_m}}$$

with $\forall w \notin \gamma(l_j).\, p_{j_w} = p_{j+1_w}$

We omit the constraints on $w \in \gamma(l_j)$ because any value can be arbitrarily chosen by $t_j$ for the fields in $\gamma(l_j)$. Substitution for constraints on $w \in \gamma(l_j)$ is legal because, for every label $l$, every transformation $t \in \varepsilon(l)$ and every field $w \in \gamma(l_j)$, $t_w = id$.

We can then replace $p$ for every occurrence of $p_1$ and remove the existential quantification on its fields, since $p = p_1$. Finally, for each pair of packet fields such that $p_{i_w} = p_{j_w}$ with $i < j$ we instantiate $p_{j_w}$ to $p_{i_w}$, removing the existential quantification on $p_{j_w}$. We repeat the last step until we reach a fixpoint where no further reduction is possible. The formula obtained is the conjunction of a predicate on the packet $p$ ($(ii)_A$) and an existentially quantified predicate on the fields of some intermediate packets ($(ii)_B$).

The lemma holds because, for any iteration, $(ii)_A$ is true iff $Ext(\psi_j, CL)(p)$ at line 8 of CHECK_FLOW$(h, p)$ is true, and $(ii)_B$ is true iff $Sat(\overline{\psi})$ at line 8 of CHECK_FLOW$(h, p)$ is true. The first coimplication trivially holds because, for every $j$, the use of $Ext$ with $CL$ excludes all and only the conjuncts $\psi_{j_w}$ of $\psi_j$ that predicate on existentially quantified field values that do not relate to $p$. To show that the second coimplication holds, consider $(ii)_B$. Note that for each field $w$, the instantiation above partitions the constraints on existentially quantified field-values in disjoint intervals of indexes. Those constraints are the same accumulated by $\overline{\psi}$, hence the thesis holds since asking for $\overline{\psi}$ to be satisfiable is exactly the same as asking for constraint-satisfying field-values to exists. □

**Theorem 2.** *Given a trace $h = (\pi, v)$, the pair $(p, t)$ is in $\tilde{E}_h$ iff*

$$t \in \hat{\varepsilon}(v) \wedge \mathit{CHECK_FLOW}(h, p) \wedge \mathit{CHECK_FLOW}(\mathit{REV}(h), t(p))$$

*Proof.* $(p, t) \in \tilde{E}_h$ implies by definition $t \in \hat{\varepsilon}(v)$. We then establish the following, assuming $t \in \hat{\varepsilon}(v)$

$$\begin{aligned}(a)\ &\exists p_1, \dots, p_{n+1}.\, p_1 = p \wedge \\ &\forall j < n.\, \psi_j(p_j) \wedge \\ &\forall j \le n.\, \exists t_j \in \varepsilon(l_j).\, t_j(p_j) = p_{j+1} \wedge \\ (b)\ &\exists p'_1, \dots, p'_{n+1}.\, p'_{n+1} = t(p) \wedge \\ &\forall j < n.\, \psi_j(p'_j) \wedge \\ &\forall j \le n.\, \exists t'_j \in \varepsilon(l_j).\, t'_j(p'_j) = p'_{j+1}\end{aligned}$$

is equivalent to

$$\begin{aligned}(c)\ &\exists p''_1, \dots, p''_{n+1}.\, p''_1 = p \wedge p''_{n+1} = t(p) \wedge \\ &\forall j < n.\, \psi_j(p''_j) \wedge \\ &\forall j \le n.\, \exists t''_j \in \varepsilon(l_j).\, t''_j(p''_j) = p''_{j+1}\end{aligned}$$

$(c) \Rightarrow (a) \wedge (b)$ holds trivially. Now assume $(a)$ and $(b)$ and we build $p''_1, \dots, p''_{n+1}.\, p''_1 = p \wedge p''_{n+1} = t(p)$ satisfying $(c)$. It is sufficient to take, for each $j$, $t''_{j_w} = t'_{j_w}$ if $t_w \neq id$ and $id$ otherwise. Then, the thesis follows trivially by Lemma 4. □

Before proving Theorem 3 we prove two auxiliary lemmata.

**Lemma 5.** *For any trace $h$ and any pair of packets $p$, $p'$ that satisfy the same set of predicates, i.e. such that $g(p) = g(p')$, the following holds*

$$\mathit{CHECK_FLOW}(h, p) \Leftrightarrow \mathit{CHECK_FLOW}(h, p')$$

*Proof.* The statement follows trivially by definition since the $p$ parameter of CHECK_FLOW in Algorithm 1 is only used when checking if it verifies the predicates of the trace. □

**Lemma 6.** *For each trace $(\pi, v)$, $\hat{\varepsilon}(v)$ is in $\mathbb{T}$.*

*Proof.* We will prove separately that

$$\begin{aligned}(i)\quad &\hat{\varepsilon}(l \cdot \mathtt{ID}) = \hat{\varepsilon}(\mathtt{ID} \cdot l) = \varepsilon(l) \\ (ii)\quad &\hat{\varepsilon}(\mathtt{SNAT} \cdot \mathtt{DNAT}) = \hat{\varepsilon}(\mathtt{DNAT} \cdot \mathtt{SNAT}) = \Lambda \times \Lambda \\ (iii)\quad &\hat{\varepsilon}(v) = \Lambda \times \Lambda \wedge l \neq \mathtt{DROP} \Longrightarrow \hat{\varepsilon}(v \cdot l) = \hat{\varepsilon}(l \cdot v) = \Lambda \times \Lambda \\ (iv)\quad &\hat{\varepsilon}(v \cdot \mathtt{DROP}) = \hat{\varepsilon}(\mathtt{DROP} \cdot v) = \varepsilon(\mathtt{DROP})\end{aligned}$$

Item $(i)$ holds trivially since $\varepsilon(\mathtt{ID}) = \{id\}$. For $(ii)$ note that $\hat{\varepsilon}(\mathtt{SNAT} \cdot \mathtt{DNAT}) = \varepsilon(\mathtt{SNAT}) \circ \varepsilon(\mathtt{DNAT})$; take $t \in \hat{\varepsilon}(\mathtt{SNAT} \cdot \mathtt{DNAT})$, $t = t' \circ t''$ with $t' \in \varepsilon(\mathtt{SNAT})$ and with $t'' \in \varepsilon(\mathtt{DNAT})$. By contradiction assume $t_{sIP} = t_{sPort} = id$, then $t'_{sIP} = t'_{sPort} = id$, hence $t' \notin \varepsilon(\mathtt{SNAT})$; same for DNAT. $(iii)$ holds because $id$ is the identity of $\circ$ and because $\lambda_a \circ \lambda_{a'} = \lambda_a$. $(iv)$ holds by definition of $\hat{\epsilon}$ and $\lambda_\perp$. □

**Theorem 3.** *$\Omega$ is a partition of $\mathbb{P} \times \mathcal{T}_{\mathbb{P}}$ such that the elements of $\omega_{X_1,Y,X_2}$ are either all expressible or all not expressible.*

*Proof.* We first prove that $\Omega$ is a partition, by separately establishing the two following statements

$$\begin{aligned}(i)\quad &\forall (p, t) \in \mathbb{P} \times \mathcal{T}_{\mathbb{P}}.\, \exists X_1, Y, X_2.\, (p, t) \in \omega_{X_1,Y,X_2} \\ (ii)\quad &(X_1, Y, X_2) \neq (X'_1, Y', X'_2) \wedge \omega_{X_1,Y,X_2} \neq \omega_{X'_1,Y',X'_2} \\ &\Rightarrow \omega_{X_1,Y,X_2} \cap \omega_{X'_1,Y',X'_2} = \emptyset\end{aligned}$$

For $(i)$, take a pair $(p, t)$. If $t = \lambda_\perp$ then the thesis follows by construction with $X_1 = g(p)$, $Y = \varepsilon(\mathtt{DROP})$ and any $X_2$. Otherwise take $X_1 = g(p)$ and $X_2 = g(t(p))$; also if $t = id$ take $Y = \varepsilon(\mathtt{ID})$; if $t_{dIP} = t_{dPort} = id$ take $Y = \varepsilon(\mathtt{SNAT})$; if $t_{sIP} = t_{sPort} = id$ take $Y = \varepsilon(\mathtt{DNAT})$; in the other cases take $Y = \Lambda \times \Lambda$.

For $(ii)$ assume that $X_1 \neq X'_1$, and by contradiction that $(p, t) \in \omega_{X_1,Y,X_2} \cap \omega_{X'_1,Y',X'_2}$. Then by definition of $\omega_{X_1,Y,X_2}$ one has $X_1 = g(p) = X'_1$. Instead, assume $Y \neq Y'$ and that $(p, t) \in \omega_{X_1,Y,X_2} \cap \omega_{X'_1,Y',X'_2}$, then by definition of $\omega_{X_1,Y,X_2}$ we know that $t \in Y$ and $t \in Y'$, but $\mathbb{T}$ is trivially a partition of $\mathcal{T}_{\mathbb{P}}$, hence $Y = Y'$. Finally, assume $X_2 \neq X'_2$: if $Y = \varepsilon(\mathtt{DROP})$ we get that $\omega_{X_1,Y,X_2} = \omega_{X'_1,Y',X'_2}$, contradicting the hypothesis; otherwise let $(p, t)$ be $\in \omega_{X_1,Y,X_2} \cap \omega_{X'_1,Y',X'_2}$, then $X_2 = g(t(p)) = X'_2$ holds.

Now we prove that the elements of $\omega_{X_1,Y,X_2}$ are either all expressible or all not expressible. By Theorems 1 and 2, it suffices proving the following, for each trace $h = (\pi, \nu)$, and for each $(p,t),(p',t') \in \omega_{X_1,Y,X_2}$

$$t \in \hat{\mathcal{E}}(\nu) \wedge \text{CHECK_FLOW}(h,p) \wedge \text{CHECK_FLOW}(\text{REV}(h),t(p)) \\ \Leftrightarrow \\ t' \in \hat{\mathcal{E}}(\nu) \wedge \text{CHECK_FLOW}(h,p') \wedge \text{CHECK_FLOW}(\text{REV}(h),t'(p'))$$

We prove the following stronger statements

$(a)$ $\quad t \in \hat{\mathcal{E}}(\nu) \Leftrightarrow t' \in \hat{\mathcal{E}}(\nu)$

$(b)$ $\quad \text{CHECK_FLOW}(h,p) \Leftrightarrow \text{CHECK_FLOW}(h,p')$

$(c)$ $\quad \text{CHECK_FLOW}(\text{REV}(h),t(p)) \Leftrightarrow \text{CHECK_FLOW}(\text{REV}(h),t'(p'))$

From Lemma 6 we know that $\hat{\mathcal{E}}(\nu)$ is one of the equivalence classes in $\mathbb{T}$, hence $(a)$ holds because $t,t' \in \hat{\mathcal{E}}(\nu) \Leftrightarrow Y = \hat{\mathcal{E}}(\nu)$, where $Y$ indexes the considered $\omega$ class. To prove $(b)$ ($(c)$, resp.) it suffices to apply Lemma 5 to $h$ (REV$(h)$, resp.). □

**Theorem 4.** $E_{pf} = E_{ipfw} \subsetneq E_{iptables} = E_{IFCL} = \mathbb{P} \times \mathcal{T}_{\mathbb{P}}$

*Proof.* The thesis trivially follows from Table 2. □

**Theorem 5.** *Given a language $\mathcal{L}$ and a fw-function $\tau$*

$$\tau \in T_{\mathcal{L}} \textit{ only if } \forall p \in \mathbb{P}.\,(p,\tau(p)) \in E_{\mathcal{L}}.$$

*Proof.* By contradiction assume that $\exists\, p.$ $(i)$ $\tau \in T_{\mathcal{L}}$, and $(ii)$ $(p,\tau(p)) \notin E_{\mathcal{L}}$. Item $(i)$ implies by definition that $\exists f$ legal for $V_{\mathcal{L}}$ is such that $(\!|(C_{\mathcal{L}},f)|\!) = \tau$, and hence $\forall p \in \mathbb{P}.\,(\!|(C_{\mathcal{L}},f)|\!)(p) = \tau(p)$. Finally, item $(ii)$ is equivalent to $\nexists f$ legal for $V_{\mathcal{L}}$ such that $(\!|(C_{\mathcal{L}},f)|\!)(p) = \tau(p)$. Contradiction. □

**Corollary 2.** $E_{\mathcal{L}} \nsubseteq E_{\mathcal{L}'} \Rightarrow T_{\mathcal{L}} \nsubseteq T_{\mathcal{L}'}$

**Theorem 6.**

- $T_{pf} \subsetneq T_{ipfw} \subsetneq T_{IFCL}$
- $T_{iptables} \subsetneq T_{IFCL}$
- $T_{pf} \nsubseteq T_{iptables}$, $T_{iptables} \nsubseteq T_{pf}$
- $T_{ipfw} \nsubseteq T_{iptables}$, $T_{iptables} \nsubseteq T_{ipfw}$

*Proof.* IFCL dominates all the other languages by Lemma 1. Moreover, by Theorem 4 and Corollary 2, the following holds trivially:

$$T_{\texttt{iptables}} \nsubseteq T_{\texttt{ipfw}} \qquad T_{\texttt{iptables}} \nsubseteq T_{\texttt{pf}}$$

Then we show that (i) $T_{\texttt{pf}} \subseteq T_\texttt{ipfw}$, we exhibit (ii) a function $\tau_1$ that is in $T_{\texttt{ipfw}}$ but not in $T_{\texttt{pf}}$ and (iii) a function $\tau_2$ that is in $T_{\texttt{pf}}$ but not in $T_{\texttt{iptables}}$.

For proving (i) let $f$ be a configuration legal for $V_{\texttt{pf}}$. Then we build an equivalent configuration $f'$ legal for $V_{\texttt{ipfw}}$ such that $(\!|(C_{\texttt{pf}},f)|\!) = (\!|(C_{\texttt{ipfw}},f')|\!)$, by taking $f'(q_0) = f(q_1) \circ f(q_0)$ and $f'(q_1) = f(q_3) \circ f(q_2)$ (note that in the control diagram of `ipfw`, the cap-labels of $q_1$ are the union of the cap-labels of $q_2$ and $q_3$ in the control diagram of `pf`).

To establish (ii), take the following function $\tau_1$, where $b, b' \in \mathcal{S}$ and $c \notin \mathcal{S}$.

$$\tau_1(p) = \begin{cases} (id : id, \lambda_{b'} : id) & \text{if } p_{dIP} = c \wedge p_{sIP} = b \\ \lambda_\perp & \text{if } (p_{dIP} = c \wedge p_{sIP} = b') \vee \\ & \quad p_{dIP} \in \mathcal{S} \vee p_{sIP} \notin \mathcal{S} \\ id & \text{otherwise} \end{cases}$$

The function $\tau_1$ is in $T_{\texttt{ipfw}}$, indeed $(\!|(C_{\texttt{pf}},f)|\!) = \tau_1$ for $f$ as follows.

$$f(q_0)(p) = \lambda_\perp$$

$$f(q_1)(p) = \begin{cases} (id : id, \lambda_{b'} : id) & \text{if } p_{dIP} = c \wedge p_{sIP} = b \\ \lambda_\perp & \text{if } (p_{dIP} = c \wedge p_{sIP} = b') \vee \\ & \quad p_{dIP} \in \mathcal{S} \\ id & \text{otherwise} \end{cases}$$

To show that $\tau_1 \notin T_{\texttt{pf}}$, by contradiction suppose that there is a configuration $f'$ legal for $V_{\texttt{pf}}$ and such that $(\!|(C_{\texttt{pf}},f')|\!) = \tau_1$. Consider two packets $p$ and $p'$ with source $b$ and $b'$ resp., and the same destination $c$, which both traverse the nodes $q_2$ and $q_3$ of `pf`. It must be $f'(q_2)(p) = (id : id, \lambda_{b'} : id)$, and $f'(q_2)(p') = id$, transforming the two packets in the same packet $p''$ with source $b'$ and destination $c$. Also it must be $f'(q_3)(p'') = id$ and at the same time that $f'(q_3)(p'') = \lambda_\perp$, because $\tau_1$ keeps $p$ and $p'$ apart: contradiction.

For proving (iii), take the following function $\tau_2$, where $a, a'$, $a''$, $b$ and $b'$ are in $\mathcal{S}$.

$$\tau_2(p) = \begin{cases} (\lambda_{a'} : id, \lambda_{b'} : id) & \text{if } p_{dIP} = a \wedge p_{sIP} = b \\ (\lambda_{a'} : id, id : id) & \text{if } p_{dIP} = a \wedge p_{sIP} = b' \\ (\lambda_{a'} : id, id : id) & \text{if } p_{dIP} = a'' \wedge p_{sIP} = b \\ \lambda_\perp & \text{otherwise} \end{cases}$$

The function $\tau_2$ is in $T_{\texttt{pf}}$, indeed $(\!|(C_{\texttt{pf}},f)|\!) = \tau_2$ for $f$ as follow.

$$f(q_0)(p) = \begin{cases} (\lambda_{a'} : id, id : id) & \text{if } (p_{dIP} = a \wedge p_{sIP} = b') \\ & \quad \vee (p_{dIP} = a'' \wedge p_{sIP} = b) \\ id & \text{otherwise} \end{cases}$$

$$f(q_1)(p) = \begin{cases} id & \text{if } p_{dIP} = a' \wedge \\ & \quad (p_{sIP} = b \vee p_{sIP} = b') \\ \lambda_\perp & \text{otherwise} \end{cases}$$

$$f(q_2)(p) = \begin{cases} (id : id, \lambda_{b'} : id) & \text{if } p_{dIP} = a \wedge p_{sIP} = b \\ id & \text{otherwise} \end{cases}$$

$$f(q_3)(p) = \begin{cases} id & \text{if } p_{dIP} = a \wedge p_{sIP} = b' \\ id & \text{if } p_{dIP} = a'' \wedge p_{sIP} = b \\ \lambda_\perp & \text{otherwise} \end{cases}$$

To show that $\tau_2 \notin T_{\texttt{iptables}}$, by contradiction suppose that there is a configuration $f'$ legal for $V_{\texttt{iptables}}$ and such that $(\!|(C_{\texttt{iptables}},f')|\!) = \tau_2$. Consider two packets $p$ and $p'$ with the same source $b$ and with destination $a$ and $a''$, resp. They

both traverse the nodes $q_8$ and $q_5$. It must be $f'(q_8)(p) = f'(q_8)(p') = (\lambda_{a'} : id, id : id)$, transforming the two packets in the same packet $p''$ with source $b$ and destination $a'$. Also it must be $f'(q_5)(p'') = id$ and at the same time that $f'(q_5)(p'') = (id : id, \lambda_{b'} : id)$, because $\tau_2$ keeps $p$ and $p'$ apart: contradiction. □

We formalize the assumptions of Section 6.

**Assumption 1.** *For every firewall language $\mathcal{L}$*

1. *Algorithm 2 does not consider any trace $(\pi, v)$ with $\pi = (q_1, \ldots q_n)$ and $v = (l_1, \ldots, l_n)$ where $l_n =$ DROP and $\exists i. i < n$ such that either DROP $\in V(q_i)$ or $l_i \neq$ ID;*
2. *we do not consider configurations $f$ such that, for some $p$, $(\!|(C_{\mathcal{L}}, f)|\!)(p) = \lambda_\perp$ and $\neg\chi_{q_i}(p)$, where*

$$\chi_q(p) = \begin{cases} true & \text{if } q = q_f \\ (\text{DROP} \in V_{\mathcal{L}}(q) \wedge p \neq \perp \Rightarrow t = \lambda_\perp) \wedge & \\ \quad t \in \{id, \lambda_\perp\} \wedge \chi_{\delta(q,t(p))} t(p) & o.w. \end{cases}$$

   *with $t = f(q)(p)$;*
3. *every pair $(p, t)$ is expressed by at most one trace $h$ in $\mathcal{H}_{\mathcal{L}}$.*
4. *The control diagram $C_{\mathcal{L}}$ is such that no trace $h \in \mathcal{H}_{\mathcal{L}}$ contains repetitions of SNAT or DNAT cap-labels.*

**Lemma 7.** *For any language $\mathcal{L}$, legal configuration $f$, packet $p$ and transformation $t$, (i) $(\!|(C_{\mathcal{L}}, f)|\!)(p) = t$ iff*

$$\begin{aligned} (ii)\quad & \exists (q_1, \ldots, q_n, l_1, \ldots, l_n) \in \mathcal{H}_{\mathcal{L}}. \\ & \exists t_1, \ldots t_n.\ t_n \circ \cdots \circ t_1 = t \wedge \\ & \forall j.\ f(q_j)(p_j) = t_j \ \wedge\ j < n \Rightarrow \psi_j(p_{j+1}) \end{aligned}$$

*where $p_1 = p$ and $\forall j.\ p_{j+1} = (t_j \circ \cdots \circ t_1)(p)$.*

*Proof.* Trivially follows from the definition of ⊗ and Lemma 2. As for Lemma 3, we can proceed for induction, on the calls of function ⊗ for $(i) \Rightarrow (ii)$, and on $n$ for $(ii) \Rightarrow (i)$. □

**Theorem 7.** *For each firewall language $\mathcal{L}$ and fw-function $\tau$, the Algorithm 2 is correct because it prints all and only*

1. *the $\tau$-pairs $(P, t)$ not expressible by $\mathcal{L}$;*
2. *the $\tau$-pairs $(P, t)$ and $(\tilde{P}, \tilde{t})$ that clash on some node $q$.*

*Proof.* We start with item 1), and we note that Algorithm 2 takes $([P], t)$ out of each of the $\tau$-pairs $(P, t) \subseteq \omega \in \Omega$, which by Theorem 3 faithfully represents them all. By Theorem 2, checking expressivity for a trace is the same of using CHECK_FLOW as done by function COMPUTE_TRACE. Finally, Theorem 1 reduces the expressivity of a language to that of its traces.

For proving item 2), we simplify Algorithm 2 to operate on a single pair $(p, t)$, obtaining Algorithm 4; then the statement follows trivially. Consider Lemma 7 and an expressible pair

**Algorithm 4** Single packet version of Algorithm 2.

```
 1: function CHECK_FUNCTION(τ, C, V)
 2:     for all q ∈ Q do  g(q) ← ∅
 3:     for all (p, t) ∈ τ do
 4:         h ← COMPUTE_TRACE(C, V, (p, t))
 5:         if h = Null then print (p, t) not expressible
 6:         else g ← CHECK_PAIR(h, (p, t), g)

 7: function COMPUTE_TRACE(C, V, (p, t))
 8:     for all h ∈ H_L do
 9:         if t ∈ Ê(v) ∧ CHECK_FLOW(h, p)∧
                CHECK_FLOW(REV(h), t(p)) then return h
10:     return Null

11: function CHECK_PAIR(h, (p, t), g)
12:     (p@, t▹, t◃) ← (p, t, (id : id, id : id))
13:     for all (q, l) ∈ h do
14:         (t@, t▹) ← SPLIT(t▹, l)
15:         for all ((p̃@, t̃◃, t̃@), (p̃, t̃)) ∈ g(q) s.t. t̃@ ≠ t@ do
16:             if p@ = p̃@ then
17:                 print (p, t) and (p̃, t̃) clash in q : (p@, t@, t̃@)
18:         g(q) ← g(q) ∪ {((p@, t◃, t@), (p, t))}
19:         t◃ ← t@ ∘ t◃
20:         p@ ← t@(p@)
21:     return g
```

$(p, t)$. Because of Assumption 3, we know there is only one trace $h$ such that

$$\exists t_1, \ldots t_n. \quad t_n \circ \cdots \circ t_1 = t \wedge \forall j.\ f(q_j)(p_j) = t_j \wedge (j < n \Rightarrow \psi_j(p_{j+1}))$$

It is trivial to prove that, because of Assumption 4, for each expressible pair $(p, t)$ and trace $h$, there is only one legal way of decomposing $t$ in such $t_1, \ldots t_n$. Hence $(\!|(C_{\mathcal{L}}, f)|\!)(p) = t$ iff

$$\forall j.\ f(q_j)(p_j) = t_j \ \wedge\ (j < n \Rightarrow \psi_j(p_{j+1}))$$

with $h$ returned by COMPUTE_TRACE, and $t_1, \ldots t_n$ unique legal decomposition of $t$.

Note that, as $h$ and $t_1, \ldots t_n$ are uniquely determined, we can show that $\forall j.\ f(q_j)(p_j) = t_j \Rightarrow (j < n \Rightarrow \psi_j(p_{j+1}))$. Indeed, the opposite would imply that $(p, t)$ is not expressible, leading to a contradiction. Hence, we can finally conclude that $(\!|(C_{\mathcal{L}}, f)|\!)(p) = t$ iff $\forall j.\ f(q_j)(p_j) = t_j$ with $h$ returned by COMPUTE_TRACE, and $t_1, \ldots t_n$ unique legal decomposition of $t$. To show the correctness of Algorithm 4 it is then sufficient to prove that the list of transformations $t_@$ generated by Algorithm 4 is a legal decomposition of $t$, which is true by definition of SPLIT. □

**Corollary 1.** *A fw-function $\tau$ is expressible by $\mathcal{L}$ if and only if Algorithm 2 prints nothing.*

*Proof.* We show that $\tau$ is expressible by $\mathcal{L}$ (i.e. $\tau \in T_{\mathcal{L}}$) iff

1. all pairs $(p, \tau(p))$ are expressible by $\mathcal{L}$;

2. no pairs $(p, \tau(p))$ and $(\tilde{p}, \tau(\tilde{p}))$ collide on some node $q$.

$\tau \in T_{\mathcal{L}} \Rightarrow (1), (2)$ trivially holds. Conversely, assume (1) and (2). For each pair $(p, \tau(p))$ there is only one possible trace and decomposition of $t$. Thus, we can accumulate the set of conditions on $f$, that are necessary and sufficient to have $(\!|(C_{\mathcal{L}}, f)|\!)(p) = \tau(p)$. Each of these conditions states that $f(q_@)(p_@) = t_@$ for some $q_@$, $p_@$ and $t_@$. Hence, for any $p \in \mathbb{P}$ we build a set $F_p$ of triplets $(q_@, p_@, t_@)$ such that, for any legal configuration $f$

$$(\!|(C_{\mathcal{L}}, f)|\!)(p) = \tau(p) \Longleftrightarrow \forall (q_@, p_@, t_@) \in F_p.\ f(q_@)(p_@) = t_@$$

We build then the following configuration $f$:

$$\forall q, p.\ f(q)(p) = \begin{cases} t & \text{if } (q, p, t) \in \bigcup_{p \in \mathbb{P}} F_p \\ id & \text{o.w.} \end{cases}$$

Note that $f$ is indeed a function because we assume there are no clash. $\square$