Semantics-Preserving Simplification of Real-World Firewall Rule Sets

Cornelius Diekmann, Lars Hupel, and Georg Carle

Technische Universität München

Abstract. The security provided by a firewall for a computer network almost completely depends on the rules it enforces. For over a decade, it has been a well-known and unsolved problem that the quality of many firewall rule sets is insufficient. Therefore, there are many tools to analyze them. However, we found that none of the available tools could handle typical, real-world *iptables* rulesets. This is due to the complex chain model used by *iptables*, but also to the vast amount of possible match conditions that occur in real-world firewalls, many of which are not understood by academic and open source tools.
In this paper, we provide algorithms to transform firewall rulesets. We reduce the execution model to a simple list model and use ternary logic to abstract over all unknown match conditions. These transformations enable existing tools to understand real-world firewall rules, which we demonstrate on four decently-sized rulesets. Using the Isabelle theorem prover, we formally show that all our algorithms preserve the firewall's filtering behavior.

Keywords: Computer Networks, Firewalls, Isabelle, Netfilter Iptables, Semantics

1 Introduction

Firewalls are a fundamental security mechanism for computer networks. Several firewall solutions, ranging from open source [2,28,29] to commercial [3,13], exist. Operating and managing firewalls is challenging as rulesets are usually written manually. While vulnerabilities in the firewall software itself are comparatively rare, it has been known for over a decade [32] that many firewalls enforce poorly written rulesets. However, the prevalent methodology for configuring firewalls has not changed. Consequently, studies regularly report insufficient quality of firewall rulesets [7,12,18,21,27,31,33,34].

Therefore, several tools [18–22,25,30,33] have been developed to ease firewall management and reveal configuration errors. However, when we tried to analyze real-world firewalls with the publicly available tools, none of them could handle our firewall rules. We found that the firewall model of the available tools is too simplistic.

In this paper, we address the following fundamental problem: Many tools do not understand real-world firewall rules. To solve the problem, we transform and simplify the rules such that they are understood by the respective tools.

```
Chain INPUT (policy ACCEPT)
target      prot source         destination
DOS_PROTECT all  0.0.0.0/0      0.0.0.0/0
ACCEPT      all  0.0.0.0/0      0.0.0.0/0   state RELATED,ESTABLISHED
DROP        tcp  0.0.0.0/0      0.0.0.0/0   tcp dpt:22
DROP        tcp  0.0.0.0/0      0.0.0.0/0   multiport dports ↩
                                     21,873,5005,5006,80,548,111,2049,892
DROP        udp  0.0.0.0/0      0.0.0.0/0   multiport dports ↩
                                                    123,111,2049,892,5353
ACCEPT      all  192.168.0.0/16 0.0.0.0/0
DROP        all  0.0.0.0/0      0.0.0.0/0

Chain DOS_PROTECT (1 references)
target      prot source         destination
RETURN      icmp 0.0.0.0/0      0.0.0.0/0   icmptype 8 limit: ↩
                                                        avg 1/sec burst 5
DROP        icmp 0.0.0.0/0      0.0.0.0/0   icmptype 8
RETURN      tcp  0.0.0.0/0      0.0.0.0/0   tcp flags:0x17/0x04 ↩
                                                 limit: avg 1/sec burst 5
DROP        tcp  0.0.0.0/0      0.0.0.0/0   tcp flags:0x17/0x04
RETURN      tcp  0.0.0.0/0      0.0.0.0/0   tcp flags:0x17/0x02 ↩
                                           limit: avg 10000/sec burst 100
DROP        tcp  0.0.0.0/0      0.0.0.0/0   tcp flags:0x17/0x02
```

Fig. 1. Linux *iptables* ruleset of a Synology NAS (network attached storage) device

To demonstrate the problem by example, we decided to use *ITVal* [19] because it natively supports *iptables* [28], is open source, and supports calls to user-defined chains. However, ITVal's firewall model is representative of the model used by the majority of tools; therefore, the problems described here also apply to a vast range of other tools. Firewall models used in related work are surveyed in Sect. 2. For this example, we use the firewall rules in Fig. 1, taken from an NAS device. The ruleset reads as follows: First, incoming packets are sent to the user-defined `DOS_PROTECT` chain, where some rate limiting is applied. Afterwards, the firewall allows all packets which belong to already established connections. This is generally considered good practice. Then, some services, identified by their ports, are blocked. Finally, the firewall allows all packets from the local network 192.168.0.0/16 and discards all other packets. We used ITVal to partition the IP space into equivalence classes (i.e. ranges with the same access rights) [20]. The expected result is a set of two IP ranges: the local network 192.168.0.0/16 and the "rest". However, ITVal erroneously only reports one IP range: the universe. Removing the first two rules (in particular the call in the `DOS_PROTECT` chain) lets ITVal compute the expected result.

We identified two main problems which prevent tools from "understanding" real-world firewalls. First, calling and returning from custom chains, due to the possibility of complex nested chain calls. Second, more seriously, most tools do not understand the firewall's match conditions. In the above example, the rate limiting is not understood. The problem of unknown match conditions cannot simply be solved by implementing the rate limiting feature for the respective tool. The major reason is that the underlying algorithm might not be capable of dealing with this special case. Additionally, firewalls, such as *iptables*, support numerous match conditions and several new ones are added in every release.[1] We expect even more match conditions for nftables [29] in the future since they can be written as simple userspace programs [17]. Therefore, it is virtually impossible to write a tool which understands all possible match conditions.

In this paper, we build a fundamental prerequisite to enable tool-supported analysis of *real-world* firewalls: We present several steps of semantics-preserving ruleset simplification, which lead to a ruleset that is "understandable" to subsequent analysis tools: First, we unfold all calls to and returns from user-defined chains. This process is exact and valid for arbitrary match conditions. Afterwards, we process unknown match conditions. For that, we embed a ternary-logic semantics into the firewall's semantics. Due to ternary logic, all match conditions not understood by subsequent analysis tools can be treated as always yielding an unknown result. In a next step, all unknown conditions can be removed. This introduces an over- and underapproximation ruleset, called upper/lower closure. Guarantees about the original ruleset dropping/allowing a packet can be given by using the respective closure ruleset.

To summarize, we provide the following novel contributions:

1. a formal semantics of *iptables* packet filtering (Sect. 4),
2. chain unfolding: transforming a ruleset in the complex chain model to a ruleset in the simple list model (Sect. 5),
3. an embedded semantics with ternary logic, supporting arbitrary match conditions, introducing a lower/upper closure of accepted packets (Sect. 6), and
4. normalization and translation of complex logical expressions to an *iptables*-compatible format, discovering a meta-logical firewall algebra (Sect. 7).

We evaluate applicability on large real-world firewalls in Sect. 8. All proofs are machine-verified with Isabelle [24] (Sect. 3). Therefore, the correctness of all obtained results only depends on a small and well-established mathematical kernel and the *iptables* semantics (Fig. 2).

2 Firewall Models in the Literature and Related Work

Packets are routed through the firewall and the firewall needs to decide whether to allow or deny a packet. A firewall ruleset determines the firewall's filtering behavior. The firewall inspects its ruleset for each single, arbitrary packet to determine the action to apply to the packet. The ruleset can be viewed as a list of rules; usually it is processed sequentially and the first matching rule is applied.

The literature agrees on the definition of a single firewall rule. It consists of a predicate (the match expression) and an action. If the match expression applies to a packet, the action is performed. Usually, a packet is scrutinized by several rules. Zhang et al. [34] specify a common format for packet filtering rules. The action is either "allow" or "deny", which directly corresponds to the firewall's filtering decision. The ruleset is processed strictly sequentially. Yuan et al. [33] call this the *simple list model.* ITVal also supports calls to user-defined chains as an action. This allows "jumping" within the ruleset without having a final filtering decision yet. This is called the *complex chain model* [33]. Zhang et al. [34] support matching on the following packet header fields: IP source and destination address, protocol, and port on layer 4. This model is commonly found in the literature [4, 5, 25, 33, 34]. ITVal extends these match conditions with flags (e.g. `TCP SYN`) and connection states (`INVALID`, `NEW`, `ESTABLISHED`, `RELATED`). The state matching is treated as just another match condition.[2] This model is similar to Margrave's model for IOS [21]. When comparing these features to the simple firewall in Fig. 1, it becomes obvious that none of these tools supports that firewall.

We are not aware of any tool which uses a model fundamentally different than those described in the previous paragraph. Our model enhances existing work in that we use ternary logic to support arbitrary match conditions. To analyze a large *iptables* firewall, the authors of Margrave [21] translated it to basic Cisco IOS access lists [3] by hand. With our simplification, we can automatically remove all features not understood by basic Cisco IOS. This enables translation of any *iptables* firewall to a basic Cisco access lists which is guaranteed to drop no more packets than the original *iptables* firewall. This opens up all tools available only for Cisco IOS access lists, e.g. Margrave [21] and Header Space Analysis [15].[3]

3 Formal Verification with Isabelle

We verified all proofs with Isabelle, using its standard Higher-Order Logic (HOL). The corresponding theory files are publicly available. An interested reader may consult the detailed (100+ pages) proof document.

Notation. We use pseudo code close to SML and Isabelle. Function application is written without parentheses, e.g. $f\ a$ denotes function f applied to parameter a. We write $::$ for prepending a single element to a list, e.g. $a :: b :: [c, d] = [a, b, c, d]$, and $:::$ for appending lists, e.g. $[a, b] ::: [c, d] = [a, b, c, d]$. The empty list is written as $[]$. $[f\ a.\ a \leftarrow l]$ denotes a list comprehension, i.e. applying f to every element a of list l. $[f\ x\ y.\ \ x \leftarrow l_1,\ y \leftarrow l_2]$ denotes the list comprehension where f is applied to each combination of elements of the lists l_1 and l_2. For $f\ x\ y = (x, y)$, this returns the cartesian product of l_1 and l_2.

4 Semantics of *iptables*

We formalized the semantics of a subset of *iptables*. The semantics focuses on access control, which is done in the `INPUT`, `OUTUT`, and `FORWARD` chain. Thus packet modification (e.g. NAT) is not considered (and also not allowed in these chains).

Match conditions, e.g. `source 192.168.0.0/24` and `protocol TCP`, are called *primitives*. A primitive matcher γ decides whether a packet matches a primitive. Formally, based on a set X of primitives and a set of packets P, a primitive matcher γ is a binary relation over X and P. The semantics supports arbitrary packet models and match conditions, hence both remain abstract in our definition.

In one firewall rule, several primitives can be specified. Their logical connective is conjunction, for example `src 192.168.0.0/24` *and* `tcp`. Disjunction is omitted because it is neither needed for the formalization nor supported by *iptables*; this is consistent with the model by Jeffrey and Samak [14]. Primitives can be combined in an algebra of *match expressions* M_X:

$$\mathit{mexpr} \quad = \quad x \quad \text{for } x \in X \quad \mid \quad \neg\, \mathit{mexpr} \quad \mid \quad \mathit{mexpr} \wedge \mathit{mexpr} \quad \mid \quad \texttt{True}$$

For a primitive matcher γ and a match expression $m \in M_X$, we write $m \vartriangleright_\gamma p$ if a packet $p \in P$ matches m, essentially lifting γ to a relation over M_X and P, with the connectives defined as usual. With completely generic P, X, and γ, the semantics can be considered to have access to an oracle which understands all possible match conditions.

Furthermore, we support the following *actions*, modeled closely after *iptables*: `Accept`, `Reject`, `Drop`, `Log`, `Empty`, `Call` c for a chain c, and `Return`. A *rule* can be defined as a tuple (m, a) for a match expression m and an action a. A list (or sequence) of rules is called a *chain*. For example, the beginning of the `DOS_PROTECT` chain in Fig. 1 is $[(\texttt{icmp} \wedge \texttt{icmptype 8 limit: } \ldots, \texttt{Return}), \ldots]$.

A set of chains associated with a name is called a *ruleset*. Let Γ denote the mapping from chain names to chains. For example, Γ `DOS_PROTECT` returns the contents of the `DOS_PROTECT` chain. We assume that Γ is well-formed that means, if a `Call` c action occurs in a ruleset, then the chain named c is defined in Γ. This assumption is justified as the Linux kernel only accepts well-formed rulesets.

The semantics of a firewall w.r.t. to a given packet p, a background ruleset Γ, and a primitive matcher γ can be defined as a relation over the currently active chain and the state before and the state after processing this chain. The semantics is specified in Fig. 2. The expression $p \vdash \langle rs,\ t \rangle \Rightarrow t'$ states that starting with state t, after processing the chain rs, the resulting state is t'. For a packet p, our semantics focuses on firewall filtering decisions. Therefore, only the following three states are necessary: The firewall may allow (✓⃝) or deny (⊗) the packet, or it may not have come to a decision yet (?⃝).

We will now discuss the most important rules. The Accept rule describes the following: if the packet p matches the match expression m, then the firewall with no filtering decision (?⃝) processes the singleton chain $[(m, \texttt{Accept})]$ by switching to the allow state. Both the Drop and Reject rules deny a packet; the difference is only in whether the firewall generates some informational message, which does not influence filtering. The NoMatch rule specifies that if the firewall has not come to a filtering decision yet, it can process any non-matching rule without changing its state. The Decision rule specifies that as soon as the firewall made

$$\textsc{Skip}\quad \frac{}{p \vdash \big\langle [],\ t \big\rangle \Rightarrow t} \qquad\qquad \textsc{Accept}\quad \frac{m\ \rhd_\gamma p}{p \vdash \big\langle [(m,\ \texttt{Accept})],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{\checkmark}}$$

$$\textsc{Drop}\quad \frac{m\ \rhd_\gamma p}{p \vdash \big\langle [(m,\ \texttt{Drop})],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{\times}} \qquad\qquad \textsc{Reject}\quad \frac{m\ \rhd_\gamma p}{p \vdash \big\langle [(m,\ \texttt{Reject})],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{\times}}$$

$$\textsc{NoMatch}\quad \frac{\neg\ m\ \rhd_\gamma p}{p \vdash \big\langle [(m,\ a)],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{?}} \qquad\qquad \textsc{Decision}\quad \frac{t \neq \textcircled{?}}{p \vdash \big\langle rs,\ t \big\rangle \Rightarrow t}$$

$$\textsc{Seq}\quad \frac{p \vdash \big\langle rs_1,\ \textcircled{?} \big\rangle \Rightarrow t \qquad p \vdash \big\langle rs_2,\ t \big\rangle \Rightarrow t'}{p \vdash \big\langle rs_1 ::: rs_2,\ \textcircled{?} \big\rangle \Rightarrow t'}$$

$$\textsc{CallResult}\quad \frac{m\ \rhd_\gamma p \qquad\qquad p \vdash \big\langle \mathit{\Gamma}\ c,\ \textcircled{?} \big\rangle \Rightarrow t}{p \vdash \big\langle [(m,\ \texttt{Call}\ c)],\ \textcircled{?} \big\rangle \Rightarrow t}$$

$$\textsc{CallReturn}\quad \frac{m\ \rhd_\gamma p \qquad \mathit{\Gamma}\ c = rs_1 ::: (m',\ \texttt{Return}) :: rs_2 \qquad m'\ \rhd_\gamma p \qquad p \vdash \big\langle rs_1,\ \textcircled{?} \big\rangle \Rightarrow \textcircled{?}}{p \vdash \big\langle [(m,\ \texttt{Call}\ c)],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{?}}$$

$$\textsc{Log}\quad \frac{m\ \rhd_\gamma p}{p \vdash \big\langle [(m,\ \texttt{Log})],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{?}} \qquad\qquad \textsc{Empty}\quad \frac{m\ \rhd_\gamma p}{p \vdash \big\langle [(m,\ \texttt{Empty})],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{?}}$$

(for any primitive matcher γ and any well-formed ruleset $\mathit{\Gamma}$)

Fig. 2. Big Step semantics for *iptables*

a filtering decision, it does not change its decision. The Seq rule specifies that if the firewall has not come to a filtering decision and it processes the chain rs_1 which results in state t and starting from t processes the chain rs_2 which results in state t', then both chains can be processed sequentially, ending in state t'. The CallResult rule specifies that if a matching `Call` to a chain named "*c*" occurs, the resulting state t is the result of processing the chain $\mathit{\Gamma}\ c$. Likewise, the CallReturn rule specifies that if processing a prefix rs_1 of the called chain does not lead to a filtering decision and directly afterwards, a matching `Return` rule occurs, the called chain is processed without result.[4] The Log rule does not influence the filtering behavior. Similarly, the Empty rule does not result in a filtering decision. An Empty rule, i.e. a rule without an action, occurs if *iptables* only updates its internal state, e.g. updating packet counters.[5]

The subsequent parts of this paper are all based on these semantics. Whenever we provide a procedure P to operate on chains, we proved that the firewall's filtering behavior is preserved, formally:

$$p \vdash \big\langle P\ rs,\ t \big\rangle \Rightarrow t' \quad \textit{iff} \quad p \vdash \big\langle rs,\ t \big\rangle \Rightarrow t'$$

All our proofs are machine-verified with Isabelle. Therefore, once the reader is convinced of the semantics as specified in Fig. 2, the correctness of all subsequent theorems follows automatically – without any hidden assumptions or limitations.

The rules in Fig. 2 are designed such that every rule can be inspected individually. However, considering all of them together, it is not immediately clear whether the result depends on the order of their application to a concrete ruleset and packet. Theorem 1 states that the semantics is deterministic, i. e. only one uniquely defined outcome is possible.

Theorem 1 (Determinism).

$$\textit{If} \quad p \vdash \langle rs,\ s \rangle \Rightarrow t \quad \textit{and} \quad p \vdash \langle rs,\ s \rangle \Rightarrow t' \quad \textit{then} \quad t = t'$$

5 Custom Chain Unfolding

In this section, we present algorithms to convert a ruleset from the complex chain model to the simple list model.

The function `pr` ("process return") iterates over a chain. If a `Return` rule is encountered, all subsequent rules are amended by adding the `Return` rule's negated match expression as a conjunct. Intuitively, if a `Return` rule occurs in a chain, all following rules of this chain can only be reached if the `Return` rule does not match.

$$\begin{aligned}
&\texttt{add-match}\ m'\ rs && = [(m \wedge m',\ a).\ (m,\ a) \leftarrow rs] \\
&\texttt{pr}\ [] && = [] \\
&\texttt{pr}\ ((m,\ \texttt{Return}) :: rs) && = \texttt{add-match}\ (\neg m)\ (\texttt{pr}\ rs) \\
&\texttt{pr}\ ((m,\ a) :: rs) && = (m,\ a) :: \texttt{pr}\ rs
\end{aligned}$$

The function `pc` ("process call") iterates over a chain, unfolding one level of `Call` rules. If a `Call` to the chain c occurs, the chain itself (i. e. $\Gamma\ c$) is inserted instead of the `Call`. However, `Return`s in the chain need to be processed and the match expression for the original `Call` needs to be added to the inserted chain.

$$\begin{aligned}
&\texttt{pc}\ [] && = \ [] \\
&\texttt{pc}\ ((m,\ \texttt{Call}\ c) :: rs) && = \ \texttt{add-match}\ m\ \ (\texttt{pr}\ \ (\Gamma\ c)) ::: \texttt{pc}\ rs \\
&\texttt{pc}\ ((m,\ a) :: rs) && = \ (m,\ a) :: \texttt{pc}\ rs
\end{aligned}$$

The procedure `pc` can be applied arbitrarily many times and preserves the semantics. It is sound and complete.

Theorem 2 (Soundness and Completeness).

$$p \vdash \langle \texttt{pc}^n\ rs,\ t \rangle \Rightarrow t' \quad \textit{iff} \quad p \vdash \langle rs,\ t \rangle \Rightarrow t'$$

In each iteration, the algorithm unfolds one level of `Call`s. The algorithm needs to be applied until the result no longer changes. Note that the semantics

$$\begin{aligned}&[(\neg\,(\texttt{icmp} \wedge \texttt{icmptype 8 limit:}\ldots) \wedge \texttt{icmp} \wedge \texttt{icmptype 8}, \texttt{Drop}),\\ &\ (\neg\,(\texttt{icmp} \wedge \texttt{icmptype 8 limit:}\ldots) \wedge \neg\,(\texttt{tcp} \wedge \texttt{tcp flags:0x17/0x04 limit:}\ldots) \wedge\\ &\quad \texttt{tcp} \wedge \texttt{tcp flags:0x17/0x04}, \texttt{Drop}),\ \ldots,\ \ (\texttt{src 192.168.0.0/16}, \texttt{Accept}),\ \ldots]\end{aligned}$$

Fig. 3. Unfolded Synology Firewall

allows non-terminating rulesets; however, the only rulesets that are interesting for analysis are the ones actually accepted by the Linux kernel.[6] Since it rejects rulesets with loops, both our algorithm and the resulting ruleset are guaranteed to terminate.

Corollary 1. *Every ruleset (with only* `Accept`*,* `Drop`*,* `Reject`*,* `Log`*,* `Empty`*,* `Call`*,* `Return` *actions) accepted by the Linux kernel can be unfolded completely while preserving its filtering behavior.*

In addition to unfolding calls, the following transformations applied to any ruleset preserve the semantics:

- Replacing `Reject` actions with `Drop` actions,
- Removing `Empty` and `Log` rules,
- Simplifying match expressions which contain `True` or ¬ `True`.
- For some given primitive matcher, specific optimizations may also be performed, e.g. rewriting `src 0.0.0.0/0` to `True`.

Therefore, after unfolding and optimizing, a chain which only contains `Allow` or `Drop` actions is left. In the subsequent sections, we require this as a precondition. As an example, recall the firewall in Fig. 1. Its `INPUT` chain after unfolding and optimizing is listed in Fig. 3. Observe that the computed match expressions are beyond iptable's expressiveness. An algorithm to normalize the rules to an *iptables*-compatible format will be described in Sect. 7.

6 Unknown Primitives

As we argued earlier, it is infeasible to support all possible primitives of a firewall. Suppose a new firewall module is created which provides the `ssh_blacklisted` and `ssh_innocent` primitives. The former applies if an IP address has had too many invalid SSH login attempts in the past; the latter is the opposite of the former. Since we made up these primitives, no existing tool will support them. However, a new version of *iptables* could implement them or they can be provided as third-party kernel modules. Therefore, our ruleset transformations must take unknown primitives into account. To achieve this, we lift the primitive matcher γ to ternary logic, adding `Unknown` as matching outcome. We embed this new "approximate" semantics into the semantics described in the previous sections. Thus, it becomes easier to construct matchers tailored to the primitives supported by a particular tool.

6.1 Ternary Matching

Logical conjunction and negation on ternary values are as before, with these additional rules for Unknown operands (commutative cases omitted):

$$\texttt{True} \wedge \texttt{Unknown} = \texttt{Unknown} \qquad \texttt{False} \wedge \texttt{Unknown} = \texttt{False} \qquad \neg\, \texttt{Unknown} = \texttt{Unknown}$$

These rules correspond to Kleene's 3-valued logic [16] and are well-suited for firewall semantics: The first equation states that, if one condition matches, the final result only depends on the other condition. The next equation states that a rule cannot match if one of its conditions does not match. Finally, by negating an unknown value, no additional information can be inferred.

We demonstrate this by example: the two rulesets $[(\texttt{ssh\_blacklisted}, \texttt{Drop})]$ and $[(\texttt{True}, \texttt{Call}\ c)]$ where $\Gamma\, c = [(\texttt{ssh\_innocent}, \texttt{Return}), (\texttt{True}, \texttt{Drop})]$ have exactly the same filtering behavior. After unfolding, the second ruleset collapses to $[(\neg\, \texttt{ssh\_innocent}, \texttt{Drop})]$. Both the ssh_blacklisted and the ssh_innocent primitives are Unknown to our matcher. Thus, since both rulesets have the same filtering behavior, a packet matching Unknown in the first ruleset should also match $\neg$ Unknown in the second ruleset matches.

6.2 Closures

In the ternary semantics, it may be unknown whether a rule applies to a packet. Therefore, the matching semantics are extended with an *"in-doubt"-tactic*. This tactic is consulted if the result of a match expression is Unknown. It decides whether a rule applies.

We introduce the *in-doubt-allow* and *in-doubt-deny* tactics. The first tactic forces a match if the rule's action is Accept and a mismatch if it is Drop. The second tactic behaves in the opposite manner. Note that an unfolded ruleset is necessary, since no behavior can be specified for Call and Return actions.[7]

We denote the exact Boolean semantics with "$\Rightarrow$" and embedded ternary semantics with an arbitrary tactic α with "$\Rightarrow_\alpha$". In particular, $\alpha = \mathit{allow}$ for *in-doubt-allow* and $\alpha = \mathit{deny}$ analogously.

"$\Rightarrow$" and "$\Rightarrow_\alpha$" are related to the in-doubt-tactics as follows: considering the set of all accepted packets, *in-doubt-allow* is an overapproximation, whereas *in-doubt-deny* is an underapproximation. In other words, if "$\Rightarrow$" accepts a packet, then "$\Rightarrow_{\text{allow}}$" also accepts the packet. Thus, from the opposite perspective, the *in-doubt-allow* tactic can be used to guarantee that a packet is certainly dropped. Likewise, if "$\Rightarrow$" denies a packet, then "$\Rightarrow_{\text{deny}}$" also denies this packet. Thus, the *in-doubt-deny* tactic can be used to guarantee that a packet is certainly accepted.

For example, the unfolded firewall of Fig. 1 contains rules which drop a packet if a limit is exceeded. If this rate limiting is not understood by γ, the *in-doubt-allow* tactic will never apply this rule, while with the *in-doubt-deny* tactic, it is applied universally.

We say that the Boolean and the ternary matchers agree iff they return the same result or the ternary matcher returns Unknown. Interpreting this definition, the ternary matcher may always return `Unknown` and the Boolean matcher serves as an oracle which knows the correct result. Note that we never explicitly specify anything about the Boolean matcher; therefore the model is universally valid, i.e. the proof holds for an arbitrary oracle.

If the exact and ternary matcher agree, then the set of all packets allowed by the *in-doubt-deny* tactic is a subset of the packets allowed by the exact semantics, which in turn is a subset of the packets allowed by the *in-doubt-allow* tactic. Therefore, we call all packets accepted by $\Rightarrow_{\text{deny}}$ the *lower closure*, i.e. the semantics which accepts at most the packets that the exact semantics accepts. Likewise, we call all packets accepted by $\Rightarrow_{\text{allow}}$ the *upper closure*, i.e. the semantics which accepts at least the packets that the exact semantics accepts. Every packet which is not in the upper closure is guaranteed to be dropped by the firewall.

Theorem 3 (Lower and Upper Closure of Allowed Packets).

$$\{p.\ p \vdash \langle rs,\ \textcircled{?}\rangle \Rightarrow_{\text{deny}} \textcircled{\checkmark}\} \subseteq \{p.\ p \vdash \langle rs,\ \textcircled{?}\rangle \Rightarrow \textcircled{\checkmark}\} \subseteq \{p.\ p \vdash \langle rs,\ \textcircled{?}\rangle \Rightarrow_{\text{allow}} \textcircled{\checkmark}\}$$

The opposite holds for the set of denied packets.

For the example in Fig. 1, we computed the closures (without the `RELATED, ESTABLISHED` rule, see Sect. 6.4) and a ternary matcher which only understands IP addresses and layer 4 protocols. The lower closure is the empty set since rate limiting could apply to any packet. The upper closure is the set of packets originating from 192.168.0.0/16.

6.3 Removing Unknown Matches

In this section, as a final optimization, we remove all unknown primitives. We call this algorithm `pu` ("process unknowns"). For this step, the specific ternary matcher and the choice for the in-doubt-tactic must be known.

In every rule, top-level unknown primitives can be rewritten to `True` or $\neg$ `True`. For example, let m_u be a primitive which is unknown to γ. Then, for in-doubt-allow, (m_u, `Accept`) is equal to (`True`, `Accept`) and (m_u, `Drop`) is equal to ($\neg$ `True`, `Drop`). Similarly, negated unknown primitives and conjunctions of (negated) unknown primitives can be rewritten.

Hence, the base cases of `pu` are straightforward. However, the case of a negated conjunction of match expressions requires some care.The following equation represents the De Morgan rule, specialized to the in-doubt-allow tactic.

$$\texttt{pu}\ (\neg\,(m_1 \wedge m_2),\ a) \quad = \quad \begin{cases} \texttt{True} & \text{if } \texttt{pu}\ (\neg\,m_1,\ a) = \texttt{True} \\ \texttt{True} & \text{if } \texttt{pu}\ (\neg\,m_2,\ a) = \texttt{True} \\ \texttt{pu}\ (\neg\,m_2,\ a) & \text{if } \texttt{pu}\ (\neg\,m_1,\ a) = \neg\,\texttt{True} \\ \texttt{pu}\ (\neg\,m_1,\ a) & \text{if } \texttt{pu}\ (\neg\,m_2,\ a) = \neg\,\texttt{True} \\ \neg\,(\neg\,\texttt{pu}\ (\neg\,m_1,\ a) \wedge \neg\,\texttt{pu}\ (\neg\,m_2,\ a)) & \text{otherwise} \end{cases}$$

The $\neg$ `Unknown` = `Unknown` equation is responsible for the complicated nature of the De Morgan rule. Fortunately, we machine-verified all our algorithms. For example, during our research, we wrote a seemingly simple (but incorrect) version of pu and everybody agreed that the algorithm looks correct. In the early empirical evaluation, with yet unfinished proofs, we did not observe our bug. Only because of the failed correctness proof did we realize that we introduced an equation that only holds in Boolean logic.

Theorem 4 (Soundness and Completeness).

$$p \vdash \langle [\mathtt{pu}\ r.\ \ r \leftarrow\ \ rs],\ t \rangle \Rightarrow_{\text{allow}} t' \quad \textit{iff} \quad p \vdash \langle rs,\ t \rangle \Rightarrow_{\text{allow}} t'$$

Theorem 5. *Algorithm* pu *removes all unknown primitive match expressions.*

An algorithm for the in-doubt-deny tactic (with the same equation for the De Morgan case) can be specified in a similar way. Thus, $\Rightarrow_\alpha$ can be treated as if it were defined only on Boolean logic with only known match expressions.

As an example, we examine the ruleset of the upper closure of Fig. 1 (without the `RELATED,ESTABLISHED` rule, see Sect. 6.4) for a ternary matcher which only understands IP addresses and layer 4 protocols. The ruleset is simplified to [(`src 192.168.0.0/16`, `Accept`), (`True`, `Drop`)]. ITVal can now directly compute the correct results on this ruleset.

6.4 The `RELATED,ESTABLISHED` Rule

Since firewalls process rules sequentially, the first rule has no dependency on any previous rules. Similarly, rules at the beginning have very low dependencies on other rules. Therefore, firewall rules in the beginning can be inspected manually, whereas the complexity of manual inspection increases with every additional preceding rule.

It is good practice [9] to start a firewall with an `ESTABLISHED` (and sometimes `RELATED`) rule. This also happens in Fig. 1 after the rate limiting. The `ESTABLISHED` rule usually matches most of the packets [9],[8] which is important for performance; however, when analyzing the filtering behavior of a firewall, it is important to consider how a connection can be brought to this state. Therefore, we remove this rule and only focus on the connection setup.

The `ESTABLISHED` rule essentially allows packet flows in the opposite direction of all subsequent rules [6]. Unless there are special security requirements (which is not the case in any of our analyzed scenarios), the `ESTABLISHED` rule can be excluded when analyzing the connection setup [6, Corollary 1].[9] If the `ESTABLISHED` rule is removed and in the subsequent rules, for example, a primitive `state NEW` occurs, our ternary matcher returns `Unknown`. The closure procedures handle these cases automatically, without the need for any additional knowledge.

7 Normalization

Ruleset unfolding may result in non-atomic match expressions, e.g. $\neg\ (a \wedge b)$. *iptables* only supports match expressions in *Negation Normal Form* (NNF).[10]

There, a negation may only occur before a primitive, not before compound expressions. For example, $\neg\,(\texttt{src}\ \mathit{ip})\ \wedge\ \texttt{tcp}$ is a valid NNF formula, whereas $\neg\,((\texttt{src}\ \mathit{ip})\ \wedge\ \texttt{tcp})$ is not. We normalize match expressions to NNF, using the following observations:

The De Morgan rule can be applied to match expressions, splitting one rule into two. For example, $[(\neg\,(\texttt{src}\ \mathit{ip}\ \wedge\ \texttt{tcp}),\, \texttt{Allow})]$ and $[(\neg\ \texttt{src}\ \mathit{ip},\, \texttt{Allow}),$ $(\neg\,\texttt{tcp},\, \texttt{Allow})]$ are equivalent. This introduces a "meta-logical" disjunction consisting of a sequence of consecutive rules with a shared action. For example, $[(m_1,\, a),\ (m_2,\, a)]$ is equivalent to $[(m_1 \vee m_2,\, a)]$.

For sequences of rules with the same action, a distributive law akin to common Boolean logic holds. For example, the conjunction of the two rulesets $[(m_1,\, a),\ (m_2,\, a)]$ and $[(m_3,\, a),\ (m_4,\, a)]$ is equivalent to the ruleset $[(m_1 \wedge m_3,$ $a),\ (m_1 \wedge m_4,\, a),\ (m_2 \wedge m_3,\, a),\ (m_2 \wedge m_4,\, a)]$. This can be illustrated with a situation where $a = \texttt{Accept}$ and a packet needs to pass two firewalls in a row.

We can now construct a procedure which converts a rule with a complex match expression to a sequence of rules with match expressions in NNF. It is independent of the particular primitive matcher and the in-doubt tactic used. The algorithm $\texttt{n}$ ("normalize") of type $M_X \to \mathrm{List}(M_X)$ is defined as follows:

$$\begin{array}{lll}
\texttt{n}\ \texttt{True} & = [\texttt{True}] & \\
\texttt{n}\ (m_1 \wedge m_2) & = [x \wedge y.\ \ x \leftarrow \texttt{n}\ m_1,\ y \leftarrow \texttt{n}\ m_2] & \\
\texttt{n}\ (\neg\ (m_1 \wedge m_2)) & = \texttt{n}\ (\neg m_1)\ ::\!:\ \texttt{n}\ (\neg m_2) & \\
\texttt{n}\ (\neg\neg m) & = \texttt{n}\ m & \\
\texttt{n}\ (\neg\texttt{True}) & = [] & \\
\texttt{n}\ x & = [x] & \multirow{2}{*}{$\left.\right\}$ for $x \in X$} \\
\texttt{n}\ (\neg x) & = [\neg x] &
\end{array}$$

The second equation corresponds to the distributive law, the third to the De Morgan rule. For example, $\texttt{n}\ \ (\neg\,(\texttt{src}\ \mathit{ip} \wedge \texttt{tcp}))\ =\ [\neg\,\texttt{src}\ \mathit{ip},\ \neg\,\texttt{tcp}]$. The fifth rule states that non-matching rules can be removed completely.

The unfolded ruleset of Fig. 3, which consists of 9 rules, can be normalized to a ruleset of 20 rules (due to distributivity). In the worst case, normalization can cause an exponential blowup. Our evaluation shows that this is not a problem in practice, even for large rulesets. This is because rulesets are usually managed manually, which naturally limits their complexity to a level processible by state-of-the-art hardware.

Theorem 6. $\texttt{n}$ *always terminates, all match expressions in the returned list are in NNF, and their conjunction is equivalent to the original expression.*

We show soundness and completeness w.r.t. arbitrary γ, α, and primitives. Hence, it also holds for the Boolean semantics. In general, proofs about the ternary semantics are stronger (the ternary primitive matcher can simulate the Boolean matcher).

Theorem 7 (Soundness and Completeness).

$$p \vdash \big\langle [(m', a).\ \ m' \leftarrow \mathtt{n}\ m],\, t \big\rangle \Rightarrow_\alpha t' \quad \textit{iff} \quad p \vdash \big\langle [(m, a)],\, t \big\rangle \Rightarrow_\alpha t'$$

After having been normalized by n, the rules can mostly be fed back to *iptables*. For some specific primitives, *iptables* imposes additional restrictions, e. g. that at most one primitive of a type may be present in a single rule. For our evaluation, we only need to solve this issue for IP address ranges in CIDR notation [10]. We introduced and verified another transformation which computes intersection of IP address ranges, which returns at most one range. This is sufficient to process all rulesets we encountered during evaluation.

8 Evaluation

In this section, we demonstrate the applicability of our ruleset preprocessing. Usually, network administrators are not inclined towards publishing their firewall ruleset because of potential negative security implications. For this evaluation, we have obtained approximately 20k real-world rules and the permission to publish them. In addition to the running example in Fig. 1 (a small real-world firewall), we tested our algorithms on four other real-world firewalls. We put focus on the third ruleset, because it is one of the largest and the most interesting one.

For our analysis, we wanted to know how the firewall partitions the IPv4 space. Therefore, we used a matcher γ which only understands source/destination IP addresses and the layer 4 protocols TCP and UDP. Our algorithms do not require special processing capabilities, they can be executed within seconds on a common off-the-shelf 4 GB RAM laptop.

Ruleset 1 is taken from a Shorewall [8] firewall, running on a home router, with around 500 rules. We verified that our algorithms correctly unfold, preprocess, and simplify this ruleset. We expected to see, in both the upper and lower closure, that the firewall drops packets from private IP ranges. However, we could not see this in the upper closure and verified that the firewall does indeed not block such packets if their connection is in a certain state. The administrator of the firewall confirmed this issue and is currently investigating it.

Ruleset 2 is taken from a small firewall script found online [1]. Although it only contains about 50 rules, we found that it contains a serious mistake. We assume the author accidentally confused *iptables*' `-I` (insert at top) and `-A` (append at tail) options. We saw this after unfolding, as the firewall allows nearly all packets at the beginning. Subsequent rules are shadowed and cannot apply. However, these rules come with a documentation of their intended purpose, such as "drop reserved addresses", which highlights the error. We verified the erroneous behavior by installing the firewall on our systems. The author is currently investigating this issue. Thus, our unfolding algorithm alone can provide valuable insights.

Ruleset 3 & 4 are taken from the main firewall of our lab (Chair for Network Architectures and Services). One snapshot was taken 2013 with 2800 rules and one snapshot was taken 2014, containing around 4000 rules. It is obvious that these rulesets have historically grown. About ten years ago, these two rulesets would have been the largest real-world rulesets ever analyzed in academia [32].

We present the analysis results of the 2013 version of the firewall. Details can be found in the additional material. We removed the first three rules. The first rule was the `ESTABLISHED` rule, as discussed in Sect. 6.4. Our focus was put on the second rule when we calculated the lower closure: this rule was responsible for the lower closure being the empty set. Upon closer inspection of this rule, we realized that it was 'dead', i.e. it can never apply. We confirmed this observation by changing the target to a `Log` action on the real firewall and could never see a hit of this rule for months. Due to our analysis, this rule could be removed. The third rule performed SSH rate limiting (a `Drop` rule). We removed this rule because we had a very good understanding of it. Keeping it would not influence correctness of the upper closure, but lead to a smaller lower closure than necessary.

First, we tested the ruleset with the well-maintained Firewall Builder [22]. The original ruleset could not be imported by Firewall Builder due to 22 errors, caused by unknown match expressions. Using the calculated upper closure, Firewall Builder could import this ruleset without any problems.

Next, we tested ITVal's IP space partitioning query [20]. On our original ruleset with 2800 rules, ITVal completed the query with around 3 GB of RAM in around 1 min. Analyzing ITVal's debug output, we found that most of the rules were not understood correctly due to unknown primitives. Thus, the results were spurious. We could verify this as 127.0.0.0/8, obviously dropped by our firewall, was grouped into the same class as the rest of the Internet. In contrast, using the upper and lower closure ruleset, ITVal correctly identifies 127.0.0.0/8 as its own class.

We found another interesting result about the ITVal tool: The (optimized) upper closure ruleset only contains around 1000 rules and the lower closure only around 500 rules. Thus, we expected that ITVal could process these rulesets significantly faster. However, the opposite is the case: ITVal requires more than 10 times the resources (both CPU and RAM, we had to move the analysis to a $>$ 40 GB RAM cluster) to finish the analysis of the closures. We assume that this is due to the fact that ITVal now understands *all* rules.

9 Conclusion

This work was motivated by the fact that we could not find any tool which helped analyzing our lab's and other firewall rulesets. Though much related work about firewall analysis exists, all academic firewall models are too simplistic to be applicable to those real-world rulesets. With the transformations presented in this paper, they became processable by existing tools. With only a small amount of manual inspection, we found previously unknown issues in four real-world firewalls.

We introduced an approximation to reduce even further the complexity of real-world firewalls for subsequent analysis. In our evaluation, we found that the approximation is good enough to provide meaningful results. In particular, using further tools, we were finally able to provide our administrator with a meaningful answer to the question of how our firewall partitions the IP space.

Our transformations can be extended for different firewall configurations. A user must only provide a primitive matcher for the firewall match conditions she wishes to support. Since we use ternary logic, a user can specify "unknown" as matching outcome, which makes definition of new primitive matchers very easy. The resulting firewall ruleset conforms to the simple list model in Boolean logic (i.e. the common model found in the literature).

Future work includes increasing the accuracy of the approximation by providing more feature-rich primitive matchers and directly implementing firewall analysis algorithms in Isabelle to formally verify them. Another planned application is to assist firewall migration between different vendors and migrating legacy firewall systems to new technologies. In particular, such a migration can be easily prototyped by installing a new firewall in chain with the legacy firewall such that packets need to pass both systems: with the assumption that users only complain if services no longer work, the formal argument in this paper proves that the new firewall with an upper closure ruleset operates without user complaints. A new fast firewall with a lower closure ruleset allows bypassing a slow legacy firewall, probably removing a network bottleneck, without security concerns.

Availability

The analyzed firewall rulesets can be found at

`https://github.com/diekmann/net-network`

Our Isabelle formalization can be obtained from

`https://github.com/diekmann/Iptables_Semantics`

Acknowledgments

A special thanks goes to Andreas Korsten for valuable discussions. We thank Julius Michaelis for contributing his Shorewall firewall. We express our gratitude to both for agreeing to publish their firewalls. In addition, Julius and Lars Noschinski contributed proofs to the formalization of the IP address space. Manuel Eberl, Lukas Schwaighofer, and Fabian Immler commented on early drafts of this paper. This work was greatly inspired by Tobias Nipkow's and Gerwin Klein's book on semantics in Isabelle [23].

This work has been supported by the German Federal Ministry of Education and Research (BMBF), EUREKA project SASER, grant 16BP12304, and by the European Commission, FP7 project EINS, grant 288021.

References

1. IPTables Example Config, `http://networking.ringofsaturn.com/Unix/iptables.php`, retrieved Sep 2014
2. PF: The OpenBSD packet filter, `http://www.openbsd.org/faq/pf/`
3. Cisco IOS firewall – configuring IP access lists. Document ID: 23602 (Dec 2007), `http://www.cisco.com/c/en/us/support/docs/security/ios-firewall/23602-confaccesslists.html`
4. Bartal, Y., Mayer, A., Nissim, K., Wool, A.: Firmato: A novel firewall management toolkit. In: Symposium on Security and Privacy. pp. 17–31. IEEE (1999)
5. Brucker, A.D., Brügger, L., Wolff, B.: Model-based firewall conformance testing. In: Testing of Software and Communicating Systems, pp. 103–118. Springer (2008)
6. Diekmann, C., Hupel, L., Carle, G.: Directed security policies: A stateful network implementation. In: Third International Workshop on Engineering Safety and Security Systems. EPTCS, vol. 150, pp. 20–34 (May 2014)
7. Diekmann, C., Posselt, S.A., Niedermayer, H., Kinkelin, H., Hanka, O., Carle, G.: Verifying security policies using host attributes. In: Formal Techniques for Distributed Objects, Components, and Systems, pp. 133–148. Springer (Jun 2014)
8. Eastep, T.M.: iptables made easy – shorewall (2014), `http://shorewall.net/`
9. Engelhardt, J.: Towards the perfect ruleset (May 2011), `http://inai.de/documents/Perfect_Ruleset.pdf`
10. Fuller, V., Li, T.: Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice) (Aug 2006), `http://www.ietf.org/rfc/rfc4632.txt`
11. Gartenmeister, M.: Iptables vs. Cisco PIX (Apr 2005), `http://lists.netfilter.org/pipermail/netfilter/2005-April/059714.html`
12. Hamed, H., Al-Shaer, E.: Taxonomy of conflicts in network security policies. IEEE Communications Magazine 44(3), 134 – 141 (Mar 2006)
13. Hewlett Packard: IP firewall configuration guide (2005), `ftp://ftp.hp.com/pub/networking/software/ProCurve-SR-IP-Firewall-Config-Guide.pdf`
14. Jeffrey, A., Samak, T.: Model checking firewall policy configurations. In: Policies for Distributed Systems and Networks. pp. 60–67. IEEE (Jul 2009)
15. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: static checking for networks. In: Networked Systems Design and Implementation. pp. 113–126. USENIX (Apr 2012)
16. Kleene, S.C.: Introduction to Metamathematics. Bibliotheca Mathematica, North-Holland, Amsterdam (1952)
17. Leblond, E.: Why you will love nftables (Jan 2014), `https://home.regit.org/2014/01/why-you-will-love-nftables/`
18. Mansmann, F., Göbel, T., Cheswick, W.: Visual analysis of complex firewall configurations. In: Proceedings of the Ninth International Symposium on Visualization for Cyber Security. pp. 1–8. VizSec '12, ACM (2012)
19. Marmorstein, R.M., Kearns, P.: A tool for automated iptables firewall analysis. In: USENIX Annual Technical Conference, FREENIX Track. pp. 71–81 (2005)

20. Marmorstein, R.M., Kearns, P.: Firewall analysis with policy-based host classification. In: Large Installation System Administration Conference. vol. 6, pp. 4–4. USENIX (Dec 2006)
21. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The margrave tool for firewall analysis. In: Large Installation System Administration Conference. USENIX (Nov 2010)
22. NetCitadel, Inc.: FirewallBuilder, `http://www.fwbuilder.org`, ver. 5.1
23. Nipkow, T., Klein, G.: Concrete Semantics. Springer (2014)
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002, last updated 2014), `http://isabelle.in.tum.de/doc/tutorial.pdf`
25. Pozo, S., Ceballos, R., Gasca, R.M.: CSP-based firewall rule set diagnosis using security policies. pp. 723–729. IEEE (Apr 2007)
26. Renard, B.: cisco-acl-to-iptables (2013), `http://git.zionetrix.net/?a=summary&p=cisco-acl-to-iptables`, retrieved Sep 2014
27. Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., Sekar, V.: Making middleboxes someone else's problem: Network processing as a cloud service. ACM SIGCOMM Computer Communication Review 42(4), 13–24 (Oct 2012)
28. The netfilter.org project: netfilter/iptables project, `http://www.netfilter.org/`
29. The netfilter.org project: netfilter/nftables project, `http://www.netfilter.org/`
30. Tongaonkar, A., Inamdar, N., Sekar, R.: Inferring higher level policies from firewall rules. In: Large Installation System Administration Conference. vol. 7, pp. 1–10. USENIX (2007)
31. Verizon Business RISK team, United States Secret Service: 2010 data breach investigations report (2010), `http://www.verizonenterprise.com/resources/reports/rp_2010-DBIR-combined-reports_en_xg.pdf`
32. Wool, A.: A quantitative study of firewall configuration errors. Computer, IEEE 37(6), 62 – 67 (6 2004)
33. Yuan, L., Chen, H., Mai, J., Chuah, C.N., Su, Z., Mohapatra, P.: FIREMAN: a toolkit for firewall modeling and analysis. In: Symposium on Security and Privacy. pp. 199–213. IEEE (May 2006)
34. Zhang, B., Al-Shaer, E., Jagadeesan, R., Riely, J., Pitcher, C.: Specifications of a high-level conflict-free firewall policy language for multi-domain networks. In: Symposium on Access Control Models and Technologies. pp. 185–194. ACM (2007)

Notes

[1] As of version 1.4.21 (Linux kernel 3.13), *iptables* supports more than 50 match conditions.

[2] Firewalls can be stateful or stateless. Most firewalls nowadays are stateful, which means the firewall remembers and tracks information of previously seen packets, e.g. the TCP connection a packet belongs to and the state of this connection. ITVal does not track the state of connections. Match conditions on connection states are treated exactly the same as matches on a packet header. In general, focusing on rulesets and not firewall implementation, matching on *iptables* conntrack states is exactly as matching on any other (stateless) condition. However, internally, not only the packet header is consulted but also the current connection tables. Note that existing firewall analysis tools also largely ignore state [21]. In our semantics, we also model stateless matching.

[3] Note that the other direction is considered easy [26], because basic Cisco IOS access lists have "no nice features" [11]. Note that there also are *Advanced* Access Lists.

[4] The semantics gets stuck if a `Return` occurs on top-level. However, this is not a problem since we make sure that this cannot happen. *iptables* specifies that a `Return` on top-level in a built-in chain is allowed and in this corner case, the chain's default policy is executed. To comply with this behavior, we always start analysis of a ruleset as follows: [(`True`, `Call` *start-chain*), (`True`, *default-policy*)], where the start chain is one of *iptables*' built-in `INPUT`, `FORWARD`, or `OUTPUT` chains with a default policy of either `Accept` or `Drop`.

[5] A rule without an action can also be used to mark a packet for later handling. This marking may influence the filtering decision. Since our primitive matchers and packets are completely generic, this case can be represented within our model: Instead of updating the firewall's internal state, an additional "ghost field" must be introduced in the packet model. Since packets are immutable, this field cannot be set by a rule but the packet must be given to the firewall with the final value of the ghost field already set. Hence, an analysis must be carried out with an arbitrary value of the ghost fields. We admit that this model is very unwieldy. However, when later embedding the more practical ternary semantics, we want to mention that all primitives which mark a packet for later processing can be considered "unknown" and are correctly abstracted by these semantics.

[6] The relevant check is in `mark_source_chains`, file `source/net/ipv4/netfilter/ip_tables.c` of the Linux kernel version 3.2.

[7] The final decision (✓ or ⊗) for `Call` and `Return` rules depends on the called/calling chain.

[8] We revalidated this observation in September 2014 and found that in our firewall, which has seen more than 15 billion packets (19+ Terabyte data) since the last reboot, more than 95% of all packets matched the first `RELATED,ESTABLISHED` rule.

[9] The same can be concluded for reflexive ACLs in Cisco's IOS Firewall [3].

[10] Since match expressions do not contain disjunctions, any match expression in NNF is trivially also in *Disjunctive Normal Form* (DNF).