Verified iptables Firewall Analysis

Cornelius Diekmann, Julius Michaelis, Maximilian Haslbeck, and Georg Carle
Technische Universität München
Email: {diekmann|carle}@net.in.tum.de, {michaeli|haslbecm}@in.tum.de

***Abstract*—We present a fully verified firewall ruleset analysis framework. Ultimately, it computes minimal service matrices, i.e. graphs which partition the complete IPv4 address space and visualize the allowed accesses between partitions for a fixed service. Internally, we are working with a simplified firewall model and a core contribution is the translation of complex real-world iptables firewall rules into this model. The presented algorithms and translation are formally proven correct with the Isabelle theorem prover. A real-world evaluation demonstrates the applicability of our tool. Both the `iptables-save` datasets and the Isabelle formalization are publicly available.**

I. Introduction

Firewall rulesets are inherently difficult to manage. It is a well-studied but unsolved problem that many rulesets show several configuration errors [1]–[3]. Tools were designed to help uncover configuration errors and verify a ruleset. We focus on tools for the static analysis of rulesets. They have the benefit that the analysis can be carried out offline, without any negative effects on the network. In contrast to testing, static analysis can achieve a full coverage (e.g. the results hold for all packets) and thus are able to uncover all errors and give strong guarantees for the absence of certain classes of errors. However, in practice, static ruleset analysis tools fail for various reasons: They do not support the vast amount of firewall features, they require the administrator to learn a complex query language which might be more complex than the firewall language itself, the analysis algorithms do not scale to large firewalls, and the output of the verification tools itself cannot be trusted.

To overcome these issues and to foster static analysis and verification of real-world firewall rulesets, we present the first fully verified and large-scale tested Linux/netfilter iptables firewall analysis and verification tool. In detail, our contributions are:

- A simple firewall model, designed for mathematical beauty and ease of static analysis (Section III)
- A series of translation steps to translate real-world firewall rulesets into this simple model (Section IV)
- Static and automatic firewall analysis methods, based on the simple model, featuring
  - IP address space partitioning (Section V)
  - Minimal service matrices (Section VI)
- Full formal and machine-verifiable proof of correctness (Section Availability)
- Evaluation on large real-world data set (Section VII)

The Linux iptables firewall is wide-spread, has evolved over a long time, and is well-known for its vast amount of features. In addition, in production networks, huge, complex, and legacy firewall rulesets have evolved over time. Therefore, iptables poses a particular challenge. Naturally, our methodology can also be applied to firewalls with simpler semantics, or younger technology with yet fewer features, e.g. Cisco IOS Access Lists or OpenFlow.

ISBN 978-3-901882-83-8 © 2016 IFIP

We outline related work in Section II. The real-world and simplified firewall models are presented in Section III. We detail on the translation between these models in Section IV. Afterwards, we present the IP address space partitioning (Section V) and service matrices (Section VI). In Section VII, we evaluate our algorithms on a large set of real-world iptables rulesets.

II. Related Work

We will call the features a firewall can use to match on packets *primitives*. For example, among others, iptables supports the following primitives: src IP address, layer 4 port, inbound interface, conntrack state, entries and limits in the `recent` list, ...

Popular tools for static firewall analysis include FIREMAN [4], Capretta et al. [5], and the Firewall Policy Advisor [6]. They support the following primitives: IP addresses, ports, and protocol. This corresponds to (a subset of) our simple firewall model, hence, these tools would not be applicable to most firewalls from our evaluation. The tools focus on detecting conflicts between rules and can consequently not offer service matrices. The work most similar to our IP address space partitioning is ITVal [7]: It supports a large set of iptables features and can compute an IP address space partition [8]. Unfortunately, ITVal is not formally verified and its implementation has several errors. For example, ITVal produces spurious results if the number of significant bits in IP addresses in CIDR notation [9] is not a multiple of 8. It does not consider logical negations which may occur when `RETURN`ing prematurely from user-defined chains, which leads to wrong interpretation of complement sets. It does not support abstracting over unknown primitives but simply ignores them, which also leads to spurious results. For rulesets with more than 1000 rules, ITVal requires tens of gigabytes of RAM. Finally, ITVal neither proves the soundness nor the minimality of its IP address range partitioning. Nevertheless, ITVal demonstrates the need for and the use of IP address range partitioning and has demonstrated that its implementation works well on rulesets which do not trigger the aforementioned errors. Building on the ideas of ITVal (but with a different algorithm), we overcome all presented issues.

Exodus [10] translates existing device configurations to a simpler model, similar to our translation step. It translates router configurations to a high-level SDN controller program, which is implemented on top of OpenFlow. Exodus supports

many Cisco IOS features. The translation problem solved by Exodus is comparable to this paper's problem of translating to a simple firewall model: OpenFlow 1.0 only supports a limited set of features (comparable to our simple firewall) whereas IOS supports a wide range of features (comparable to iptables); A complex language is ultimately translated to a simple language, which is the 'hard' direction.

Complementary to our verification tool, and well-suited for debugging, is Margrave [11]. It can be used to query firewalls and to troubleshoot configurations or to show the impact of ruleset edits. Margrave can find scenarios, i.e. it can show concrete packets which violate a security policy. Our framework does not show such information. Margrave's query language (which should be learned by a potential user) is based on first-order logic.

III. Firewall Semantics

All facts presented in this work are formally verified with the Isabelle theorem prover [12]. All executable algorithms are also implemented in Isabelle and formally proven correct.

Isabelle is an LCF-style theorem prover: A proposition is only accepted by Isabelle if it can be explained to its mathematical inference kernel. That kernel is very small and well-understood by the formal methods community which makes it very unlikely that Isabelle allows proving false statements. The last 20 years of Isabelle in practice underline this statement. In general, the formal methods community treats facts machine-verified with Isabelle as well-founded truth. Also, the real-world firewall reference model we will use in this work (Section III-B) has been previously evaluated by said community [3]. Our formalization, implementation, and proofs are publicly available (cf. Section Availability). An interested reader can replay the proofs and results of the evaluation on her system. For brevity, in this paper, we omit all technical proof details and only outline the intuition of the correctness proofs. For further mathematical details, we refer the interested reader to our proof document. We use Isabelle's standard Higher-Order Logic (HOL). This means, all proofs can be reduced to the axioms of HOL. We stick closely to the formalization and do not sweep any assumption under the carpet.

Our notation is close to Isabelle, Standard ML, or Haskell: Function application is written without parentheses, e.g. $\mathsf{f}\ a$ denotes function f applied to parameter a. For lists, we denote cons and append by '$\mathrel{::}$' and '$@@$', e.g. '$x \mathrel{::} [y, z] @@ [a]$'. Linux shell commands are set in `typewriter` font. Executable functions are set in sans serif font. We will write firewall rules as tuple $(m,\ a)$, where m is a match expression and a is the action the firewall performs if m matches for a packet. The firewall has two possibilities for the filtering decision: it may accept (✓) the packet or deny (✗) the packet. There is also an intermediate state (?) in which the firewall did not come to a filtering decision. Note that iptables firewalls always have a default policy and the ? case cannot occur as final decision.

A. Simple Firewall

First, we present a very simple firewall model. This model was designed to feature nice mathematical properties but it is too simplistic to mirror the real world. Therefore, we will afterwards present a model for real-world firewalls. Section IV will show how rulesets can be translated between these two models. This preprocessing step simplifies all future static firewall analysis. The model is a simple recursive function. The first parameter is the ruleset the firewall iterates over, the second parameter is the packet.

$$\begin{aligned}
&\mathsf{simple\text{-}fw}\ \ [\,] && p = && ?\\
&\mathsf{simple\text{-}fw}\ \ ((m,\ \texttt{Accept}) \mathrel{::} rs)\ p = \\
&\qquad \text{if } \mathsf{match}\ m\ p \text{ then } ✓ \text{ else } \mathsf{simple\text{-}fw}\ rs\ p\\
&\mathsf{simple\text{-}fw}\ \ ((m,\ \texttt{Drop}) \mathrel{::} rs) && p = \\
&\qquad \text{if } \mathsf{match}\ m\ p \text{ then } ✗ \text{ else } \mathsf{simple\text{-}fw}\ rs\ p
\end{aligned}$$

A function match tests whether a packet p matches the match condition m. The match condition is an 7-tuple, consisting of the following primitives:

$$(\text{in},\ \text{out},\ \text{src},\ \text{dst},\ \text{protocol},\ \text{src ports},\ \text{dst ports})$$

In contrast to iptables, negating matches is not supported. In detail, the following is supported:

- in/out interface, including support for the '+' wildcard
- src/dst IP address range in CIDR notation, e.g. 192.168.0.0/24
- protocol (Any, tcp, udp, icmp, or any numeric protocol identifier)
- src/dst interval of ports, e.g. 0:65535

For example, we obtain an empty match (a match that does not apply to any packet) *iff* an end port is greater than the start port. The match which matches any packet is constructed by setting the interfaces to "+", the ips to 0.0.0.0/0, the ports to 0:65535 and the protocol to Any. With this type of match expression, it is possible to implement a function conj which takes two match expressions m_1 and m_2 and returns exactly one match expression being the conjunction of both.

Theorem 1 (Conjunction of two simple match expressions)**.**

$$\mathsf{match}\ m_1\ p\ \wedge \mathsf{match}\ m_2\ p\ \longleftrightarrow \mathsf{match}\ (\mathsf{conj}\ m_1\ m_2)\ p$$

Computing the conjunction of the individual match expressions for port intervals and single protocols is straightforward. The conjunction of two intervals in CIDR notation is either empty or the smaller of both intervals. The conjunction of two interfaces is either empty if they don't share a common prefix, otherwise it is the longest of both interfaces (non-wildcard interfaces dominate wildcard interfaces).

The type of match expressions was carefully designed such that the conjunction of *two* match expressions is only *one* match expression. If features were added to the match expression, for example negated interfaces, this would no longer be possible. Of all common features found in a firewall, we only found that it would further be possible to add TCP flags to the match expression without violating the aforementioned conjunction property.

B. Semantics of Iptables

We now outline the model of a real-world iptables firewall. Most firewall analysis is concerned with the access control rules of a firewall, therefore the model focuses on the `filter`

table. This implies, packet modification (e.g. NAT, which must not occur in this table) is not considered in this work. We rely on our previous work [3]. The model supports the following common actions: `Accept`, `Drop`, `Reject`, `Log`, `Call`ing to and `Return`ing from user-defined chains, as well as the "empty" action. The model is defined as an inductive predicate with the following syntax:

$$\Gamma, \gamma, p \vdash \langle rs,\ s\rangle \Rightarrow t$$

The ruleset of the firewall is rs and the packet under examination is p. The states s and t are in $\{\text{✓}, \text{✗}, \text{?}\}$. The starting state of the firewall is s, usually ?. The filtering decision after processing rs is t, usually ✓ or ✗. User-defined chains are stored in Γ, which corresponds to the background ruleset. A primitive matcher γ (a boolean function which takes a primitive and the packet as parameters) decides whether a certain primitive matches for a packet. Note that the model and all algorithms on top of it are proven correct for an arbitrary γ, hence, this model supports *all* iptables matching features. Obviously, there is no executable code for an arbitrary γ. However, the algorithms which transform rulesets are executable.

We make use of these algorithms, in particular: An algorithm which unfolds all calls to and returns from user-defined chains and rewriting of further actions. This leaves a ruleset where only the following actions occur: `Accept` and `Drop`. Thus, a large step for translating the real-world model to the simple firewall model is already accomplished. Translating the match expressions remains. The real-world model allows a match expression to be an arbitrary propositional logic expression. However, iptables only accepts match expressions in *negation normal form* (NNF). A Boolean formula is in NNF *iff* all occurring negations are on primitives, i.e. there are no nested negated expressions. For example, iptables can load `-s 10.0.0.0/8 ! -p tcp` but not `! (-s 10.0.0.0/8 -p tcp)`. However, such negated expressions may occur as a result of the unfolding algorithm. An algorithm to translate a ruleset to a ruleset where all match conditions are in NNF is already available [3].[1] However, there is an additional constraint imposed by iptables, not solved by the algorithm: A primitive must only occur at most once. This problem will be addressed in this paper.

We have implemented a subset of γ, namely for all primitives supported by the simple firewall and some further primitives, detailed in Section IV. Previous work provides an algorithm to abstract over all 'unknown' primitives which are not understood by our subset implementation of γ. This algorithm leads to an approximation of the ruleset. It can either be an overapproximation which results in a more permissive ruleset, or an underapproximation, which results in a stricter ruleset. For the sake of example, we will only consider the overapproximation in this paper, the underapproximation is analogous and can be found in our formalization.

Since firewalls usually accept all packets which belong to an `ESTABLISHED` connection, the interesting access control rules in a ruleset only apply to `NEW` packets. We only consider `NEW` packets, i.e. `--ctstate NEW` and `--syn` for TCP packets. Our first goal is to translate a ruleset from the real-world model to the simple model. We have proven that the set of new packets accepted by the simple firewall is a superset (overapproximation) of the packets accepted by the real-world model. This is a core contribution and we detail on the translation in the following section.

Theorem 2 (Translation to simple firewall model)**.**

$$\{p.\ \ \mathsf{new}\ p \wedge \Gamma, \gamma, p \vdash \langle rs,\ \text{?}\rangle \Rightarrow \text{✓}\}$$
$$\subseteq$$
$$\{p.\ \ \mathsf{new}\ p \wedge \mathsf{simple\text{-}fw}\ (\mathsf{translate\text{-}oapprox}\ rs) = \text{✓}\}$$

Any packet dropped by the translated, overapproximated simple firewall ruleset is guaranteed to be dropped by the real-world firewall, for arbitrary γ, Γ, rs. Similar guarantees for certainly accepted packets can be given by considering the translated underapproximation. Given the simple and carefully designed model of the simple-fw, it is much easier to write algorithms to analyze and verify the translated rulesets.

Example: We consider a `FORWARD` chain with a default policy of `DROP` and a user-defined chain `foo`.

```
-P FORWARD DROP
-A FORWARD -s 10.0.0.0/8 -j foo
-A foo ! -s 10.0.0.0/9 -j DROP
-A foo -p tcp -j ACCEPT
```

This ruleset, though it only consist of three rules and a default policy, is complicated to analyze. Our translation algorithm translates it to the simple firewall model, where the ruleset becomes remarkably simple. We use $\star$ to denote a wildcard:

```
(*,*,10.128.0.0/9,*, * ,*,*)  DROP
(*,*, 10.0.0.0/8 ,*,TCP,*,*)  ACCEPT
(*,*,      *      ,*, * ,*,*)  DROP
```

No over- or underapproximation occurred since all primitives could be translated. Note the 10.128.0.0/9 address.

IV. Translating Primitives

A firewall has the same behavior for two rulesets rs_1 and rs_2 *iff* for all packets, the firewall computes the same filtering decision for rs_1 and rs_2. Formally,

$$\forall p\ s\ t.\ \ \Gamma, \gamma, p \vdash \langle rs_1,\ s\rangle \Rightarrow t \ \ \longleftrightarrow \ \ \Gamma, \gamma, p \vdash \langle rs_2,\ s\rangle \Rightarrow t$$

In this section, we present algorithms to transform an arbitrary rs_1 to rs_2 without changing the behavior of the firewall. In the resulting rs_2, all primitives will be normalized such that the translation to the simple-fw is obvious. We continue by describing the normalization of all common primitives found in iptables rulesets.

A. IPv4 Addresses

"Modeling IP addresses efficiently is challenging." [11] First, we present a datatype to efficiently perform set operations on intervals of machine words, e.g. 32-bit integers. We will use this type for IPv4 addresses, but it can be generalized to machine words of arbitrary length, e.g. IPv6 addresses or L4 ports. We call it word interval (wi), and WI $start\ end$

[1] NNF normalizing may create additional rules.

describes the interval with $start$ and end inclusive. The Union of two wis is defined recursively.

$$\text{datatype } wi = \mathsf{WI}\ word\ word \mid \mathsf{Union}\ wi\ wi$$

Let set denote the interpretation into mathematical sets, then wi has the following semantics: $\text{set}\ (\mathsf{WI}\ start\ end) = \{start..end\}$ and $\text{set}\ \mathsf{Union}\ wi_1\ wi_2 = \text{set}\ (wi_1) \cup \text{set}\ (wi_2)$.

An IP address in CIDR notation or IP addresses specified by e.g. `-m iprange` can be translated to one WI. We have implemented and proven the common set operations: '$\cup$', '$\{\}$', '$\setminus$', '$\cap$', '$\subseteq$', and '$=$'. These operations are linear in the number of Union-constructors. The result is optimized by merging adjacent and overlapping intervals and removing empty intervals. We can also represent 'UNIV' (the universe of all IP addresses). Since most rulesets use IP addresses in CIDR notation or intervals in general, the wi datatype has proven to be very efficient. Recall that the intersection of two intervals, constructed from addresses in CIDR notation, is either empty or the smaller of both intervals.

wi is an internal representation and for the simple firewall, the result needs to be represented in CIDR notation. For this direction, one WI may correspond to several CIDR ranges. We describe an algorithm to split off one CIDR range from an arbitrary word interval r. The output is a CIDR range and r', the remainder after splitting off this CIDR range. split is implemented as follows: Let a be the lowest element in r. If this does not exist, then r corresponds to the empty set and the algorithm terminates. Otherwise, we construct the list of CIDR ranges $[a/0, a/1, ..., a/32]$. The first element in the list which is well-formed (i.e. all bits after the network prefix must be zero) and which is a subset of r is the wanted element. Note that this element always exists. It is subtracted from r to obtain r'. To convert r completely to a list of CIDR ranges, this is applied recursively until it yields no more results. This algorithm is guaranteed to terminate and the resulting list in CIDR notation corresponds to the same set of IP addresses as represented by r. Formally, $\bigcup \text{map set}\ (\mathsf{split}\ r) = \text{set}\ r$.

For example, $\mathsf{split}\ (\mathsf{WI}\ 10.0.0.0\ 10.0.0.15) = [10.0.0.0/28]$ and $\mathsf{split}\ (\mathsf{WI}\ 10.0.0.1\ 10.0.0.15) = [10.0.0.1/32, 10.0.0.2/31, 10.0.0.4/30, 10.0.0.8/29]$.

With the help of these functions, arbitrary IP address ranges can be translated to the format required by the simple firewall. The following is applied to matches on src and dst IP addresses: First, the IP match expression is translated to a word interval. If the match on an IP range is negated, we compute $\text{UNIV} \setminus wi$. All matches in one rule can be joined to a single word interval, using the $\cap$ operation. The resulting word interval is translated to a set of non-negated CIDR ranges. Using the NNF normalization, at most one match on an IP range in CIDR notation remains. We have proven that this process preserves the firewall's filtering behavior.

We conclude with a simple, synthetic worst-case example. The evaluation shows that this worst-case does not prevent successful analysis: `-m iprange --src-range 0.0.0.1-255.255.255.254`. Translated to the simple firewall, this one range blows up to 62 ranges in CIDR notation. A similar blowup may occur for negated IP ranges.

B. Conntrack State

If a packet p is matched against the stateful match condition `ESTABLISHED`, conntrack looks up p in its state table. When the firewall comes to a filtering decision for p, if the packet is not dropped and the state was `NEW`, the conntrack state table is updated such that the flow of p is now `ESTALISHED`. Similarly, other conntrack states are handled.

We present an alternative model for this behavior: Before the firewall starts processing the ruleset for p, the conntrack state table is consulted for the state of the connection of p. This state is added as a (phantom) tag to p. Therefore, ctstate can be modeled as just another header field of p. When processing the ruleset, it is not necessary to inspect the conntrack table but only the virtual state tag of the packet. After processing, the state table is updated accordingly.

We have proven that both models are equivalent. The latter model is simpler for analysis purposes since the conntrack state can be considered an ordinary packet field.[2]

In Theorem 2, we are only interested in `NEW` packets. In contrast to previous work, there is no longer the need to manually exclude `ESTABLISHED` rules from a ruleset. The alternative model allows us to consider only `NEW` packets: all state matches can be removed (by being pre-evaluated for an arbitrary `NEW` packet) from the ruleset without changing the filtering behavior of the firewall.

C. Layer 4 Ports & TCP Flags

Translating singleton ports or intervals of ports to the simple firewall is straightforward. A challenge remains for negated port ranges and the `multiport` module. However, the word interval type is also applicable to 16 bit machine words and solves these challenges. For ports, there is no need to translate an interval back to CIDR notation.[3]

Iptables can match on a set of L4 flags. To match on flags, a $mask$ selects the corresponding flags and c declares the flags which must be present. For example, the match `--syn` is a synonym for $mask$ = `SYN,RST,ACK,FIN` and c = `SYN`. For a set f of flags in a packet, matching can be formalized as $(f \cap mask) = c$. If c is not a subset of $mask$, the expression cannot match; we call this the empty match. We proved that two matches $(mask_1, c_1)$ and $(mask_2, c_2)$ are equal if and only if $(\text{if } c_1 \subseteq mask_1 \wedge c_2 \subseteq mask_2 \text{ then } c_1 = c_2 \wedge mask_1 = mask_2 \text{ else } (\neg c_1 \subseteq mask_1) \wedge (\neg c_2 \subseteq mask_2))$ holds. We also proved that the conjunction of two matches is exactly $(\text{if } c_1 \subseteq mask_1 \wedge c_2 \subseteq mask_2 \wedge mask_1 \cap mask_2 \cap c_1 = mask_1 \cap mask_2 \cap c_2 \text{ then } (mask_1 \cup mask_2, c_1 \cup c_2) \text{ else } \texttt{empty})$. If we assume `--syn` for a packet, we can remove all matches which are equal to `--syn` and add the `--syn` match as conjunction to all other matches on flags and remove empty matches. Some matches on flags may remain, e.g. `URG`, which need to be abstracted over later.

[2] This holds because the semantics does modify a packet during filtering.

[3] As a side note, OpenFlow (technically, the Open vSwitch) defines CIDR-like matching for L4 ports. With the small change of converting ports to CIDR-like notation, our simple firewall can be directly converted to OpenFlow and we have the first (almost) fully verified translation of iptables rulesets to SDN.

D. Interfaces

The simple firewall model does not support negated interfaces, e.g. `! -i eth+`. Therefore, they must be removed. We first motivate the need for abstracting over negated interfaces.

For whitelisting scenarios, one might argue, that negated interfaces is bad practice anyway. This is because new (virtual) interfaces might be added to the system at runtime and a match on negated interfaces might now also include these new interfaces. Therefore, it can be argued that negated interfaces correspond to blacklisting, which is not recommended for most firewalls. However, the main reason why negated interfaces are not supported by our model is of technical nature: Let set denote the set of interfaces that match an interface expression. For example, set `eth0` = {`eth0`} and set `eth+` is the set of all interfaces that start with the prefix `eth`. If the match on `eth+` is negated, then it matches all strings in the complement set: UNIV ∖ (set `eth+`). The simple firewall model requires that a conjunction of two primitives is again at most one primitive. This can obviously not be achieved with such sets. In addition, working with negated interfaces can cause great confusion. Note that the interface match condition '+' matches any interfaces. Also note that '+' ∈ UNIV ∖ (set `eth+`). In the second equation, '+' is not a wildcard character but the name of an interface. The confusion introduced by negated interfaces becomes more apparent when one realizes that '+' can occur as both wildcard character and normal character. Therefore, it is not possible to construct an interface match condition which matches exactly on the interface '+', because a '+' at the end of an interface match condition is interpreted as wildcard.[4]

Correlating with IP Ranges: Later, in Section V, we will compute an IP address space partition. For best clarity, this partition must not be 'polluted' with interface information. Therefore, for the partition, we will assume that no matches on interfaces occur in the ruleset. In this subsection, we describe a method to get rid of both, negated and non-negated input interfaces while preserving their relation to IP address ranges.

Interfaces are usually assigned an IP range of valid source IPs which are expected to arrive on that interface. Let $ipassmt$ be a mapping from interfaces to an IP address range. This information can be obtained by `ip route` and `ip addr`. We will write $ipassmt[i]$ to get the corresponding IP range of interface i. For the following examples, we assume

$$ipassmt = [\texttt{eth0} \mapsto \{10.8.0.0/16\}]$$

The goal is to rewrite interfaces with the corresponding IP range. For example, we would like to replace all occurrences of `-i eth0` with `-s 10.8.0.0/16`. This idea can only be sound if there are no spoofed packets; we only expect packets with a source IP of `10.8.0.0/16` to arrive at `eth0`. Once we have assured that the firewall blocks spoofed packets, we can assume in a second step that there are no spoofed accepted packets left. By default, the Linux kernel offers reverse path filtering, which blocks spoofed packet automatically. In this case we can assume that no spoofed packets occur. In some complex scenarios, reverse path filtering needs to be disabled and spoofed packets should be blocked manually with the help of the firewall ruleset. In previous work [13], we presented an algorithm to verify that a ruleset correctly blocks spoofed packets. This algorithm is integrated in our framework, proven sound, works on the same $ipassmt$ and does not need the simple firewall model (i.e. supports negated interfaces). If some interface i should accept arbitrary IP addresses (essentially not providing spoofing protection), it is possible to set $ipassmt[i]$ = UNIV. Therefore, we can verify spoofing protection according to $ipassmt$ at runtime and afterwards continue with the assumption that no spoofed packets occur.

Under the assumption that no spoofed packets occur, we will now present two algorithms to relate an input interface i to $ipassmt[i]$. Both approaches are valid for negated and non-negated interfaces. Approach one provides better results but requires stronger assumptions (which can be checked at runtime), whereas approach two is applicable without further assumptions. These approaches could be generalized to output interfaces (`-o`), which requires the routing table instead of $ipassmt$. Because a routing table may change frequently, even triggered by external malicious routing advertisements, we refrain from this rewriting in this work.

Approach One: In general, it is considered bad practice [1], [14] to have zone-spanning interfaces. Two interfaces are zone-spanning if they share a common, overlapping IP address range. Mathematically, absence of zone-spanning interfaces means that for any two interfaces in $ipassmt$, their assigned IP range must be disjoint. Our tool emits a warning if $ipassmt$ contains zone-spanning interfaces. If absence of zone-spanning interfaces is checked, then all input interfaces can be replaced by their assigned source IP address range. This preserves exactly the behavior of the firewall. The idea is that in this case a bidirectional mapping between input interfaces and source IPs exists. Interestingly, our proof does not need the assumption that $ipassmt$ maps to the complete IP universe.

Approach Two: Unfortunately, though considered bad practice, we found many zone-spanning interfaces in many real-world rulesets and hence cannot apply the previous algorithm. First, we proved that correctness of the described rewriting algorithm implies lack of zone-spanning interfaces. This leads to the conclusion that it is impossible to perform rewriting without this assumption. Therefore, we present an algorithm which adds the IP range information to the ruleset (without removing the interface match), thus constraining the match on input interfaces to their IP range. The algorithm computes the following: Whenever there is a match on an input interface i, the algorithm looks up the corresponding IP range of that interface and adds `-s` $ipassmt[i]$ to the rule. To prove correctness of this algorithm, no assumption about zone-spanning interfaces is needed, $ipassmt$ may only be defined for a subset of the interfaces, and the range of $ipassmt$ may not cover the complete IP universe. Consequently, there is no need for a user to specify $ipassmt$, but having it may yield more accurate results.

E. Abstracting Over Primitives

Some primitives cannot be translated to the simple model. Previous work already provides the function pu which removes all unknown match conditions [3]. This leads to an approximation and is the main reason for the '⊆' relation in Theorem 2. We found that we can also rewrite any known primitive *at*

[4] We greatly discourage the use of "`ip link set eth0 name +`" in production. Please fix your VM startup scripts with untrusted input now!

any time to an unknown primitive. This can be used to apply additional knowledge during preprocessing. For example, since we understand flags, we know that the following condition is false, hence rules using it can be removed: `--syn` $\wedge$ `--tcp-flags RST,ACK RST`. After this optimization, all remaining flags can be treated as unknowns and abstracted over afterwards. This allows to easily add additional knowledge and optimization strategies for further primitive match conditions without the need to adapt any algorithm which works on the simple firewall model. We proved soundness of this approach: The '$\subseteq$' relation in Theorem 2 is preserved.

V. IP Address Space Partition

In the following sections, we will work on rulesets translated to the simple-fw model. In this section, we will compute a partition of the IPv4 address space. All IP addresses in the same partition must show the same behavior w.r.t the firewall ruleset. We do not require that the partition is minimal. Therefore, the following would be a valid solution: $\{\{0\}, \{1\}, \ldots, \{255.255.255.255\}\}$. However, we will need the partition as starting point for a further algorithm and a partition of size 2^{32} is too large for this purpose. In this section, we will present an algorithm to compute a partition which behaves roughly linear in the number of rules for real-world rulesets. First, we motivate the partitioning idea with the following observation.

Lemma 1. *For an arbitrary packet p, we write $p(src \mapsto s)$ to fix the src IP address to s. Let X be the set of all src IP matches specified in rs, i.e. X is a set of CIDR ranges. If*

$$\forall A \in X.\ B \subseteq A \vee B \cap A = \{\}$$

then let $s_1 \in B$ and $s_2 \in B$ then

$$\text{simple-fw}\ rs\ p(src \mapsto s_1)\ =\ \text{simple-fw}\ rs\ p(src \mapsto s_2)$$

Reading the lemma backwards, it states that all packets with arbitrary source IPs picked from B are treated equally by the firewall. Therefore, B is a member of an IP address range partition. The condition imposed on B is that for all src CIDR ranges specified in the ruleset (called A in the lemma), B is either a subset of the range or disjoint. The lemma shows that this condition is sufficient for B, therefore we will construct an algorithm to compute B. For an arbitrary set X, this condition is purely set-theoretic and we can solve it independently from the firewall theory.

For simplicity, we use finite sets and lists interchangeably. We will write an algorithm part and reuse the common list algorithm from functional programming foldr. For X, the following algorithm computes a partition: foldr part X $\{UNIV\}$. In addition, it is guaranteed that the union of the resulting partition is equal to the universe. For our scenario, this means that the partitioning covers the complete IPv4 space. The algorithm part is implemented as follows: The first parameter is a set $S \in X$, the second parameter TS is a set of sets and corresponds to the remaining set which will be partitioned. In the first call $TS = \{UNIV\}$. For a fixed S, part S TS iterates over TS and splits the set such that the precondition of Lemma 1 holds: Written as recursive function: part S $(\{T\} \cup TS) = (S \cap T) \cup (T \setminus S) \cup (\text{part}\ (S \setminus T)\ TS)$

The result size of calling part once can be up to two times the size of TS. This means, the partition of a complete firewall ruleset is in $O(2^{|rules|})$. However, the empirical evaluation shows that the resulting size for real-world rulesets is much better. This is because IP address ranges may overlap in a ruleset, but they do not overlap in the worst possible way for all pairs of rules. Consequently, at least one of the sets $S \cap T$ or $T \setminus S$ is usually empty and can be optimized away. For example, for our largest firewall, the number of computed partitions is 10 times smaller than the number of rules. Table I confirms that the number of partitions is usually less than the number of rules.

Our algorithm fulfills the assumption of Lemma 1 for arbitrary X. Because IP addresses occur as source and destination in a ruleset, we use our partitioning algorithm where X is the set of all IPs found in the ruleset. The result is a partition where for any two IPs in the same partition, setting the src or dst of an arbitrary packet to one of the two IPs, the firewall behaves equally. This results in a stronger version of Lemma 1, which holds without any assumption and also holds for both src and dst IPs simultaneously. In addition, the partition covers the complete IPv4 address space.

VI. Service Matrices

The IP address space partition may not be minimal. That means, two different partitions may exhibit exactly the same behavior. Therefore, for manual firewall verification, these partitions may be misleading. Marmorstein elaborates on this problem [8]. ITVal's solution is to minimize the partition. We suggest to minimize the partition for a fixed service. The evaluation shows that the result is smaller and thus more clear. A fixed service corresponds to a fixed packet with arbitrary IPs. For example, we can define ssh as TCP, dport 22, arbitrary sport $\geq$ 1024. A service matrix describes the allowed accesses for a specific service over the complete IPv4 address space. It can be visualized as graph, for example Figure 1. The matrix is minimal if it cannot be compressed any further.

First, we describe when a firewall exhibits the same behavior for arbitrary source IPs s_1, s_2 and a fixed packet p:

$$\forall d.\ \text{simple-fw}\ rs\ p(src \mapsto s_1,\ dst \mapsto d) = \text{simple-fw}\ rs\ p(src \mapsto s_2,\ dst \mapsto d)$$

We say the firewall shows same behavior for a fixed service if, in addition, the analogue condition holds for destination IPs.

We present a function groupWIs, which computes the minimal partition for a fixed service. For this, the full access control matrix for inbound and outbound connections of each partition member is generated. This can be done by taking arbitrary representatives from each partition as source and destination address and executing simple-fw for the fixed packet with those fixed IPs. The matrix is minimized by merging partitions with equal rights, i.e. equal rows in the matrix. This algorithm is quadratic in the number of partitions. The evaluation shows that it scales surprisingly well, even for large rulesets, since the number of partitions is usually small.

Theorem 3 (Service Matrix is Sound and Minimal)**.** *For any two IPs in any member of* groupWIs*, the firewall shows the same behavior for a fixed service.*

For any two arbitrary members A and B in groupWIs*, if we can find two IPs in A and B respectively where the firewall shows the same behavior for a fixed service, then $A = B$.*

VII. Evaluation

We obtained real-world rulesets from over 15 firewalls. Some are central, production-critical devices. They are written by different authors, utilize a vast amount of different features and exhibit different styles and patterns. Publishing the complete rulesets itself is an important contribution (c.f. [1], [2]). To the best of our knowledge, this is the largest, publicly-available collection of real-world iptables rulesets. Note: some administrators wish to remain anonymous so we replaced their public IP addresses with public IP ranges of our institute, preserving all IP subset relationships.

Table I summarizes the evaluation's results. The first column (Fw) labels the analyzed ruleset. Column two (Rules) contains the number of rules (only the filter table) in the output of `iptables-save`. We work directly and completely on this real-world data. Column three describes the analyzed chain. Depending on the type of firewall, we either analyzed the `FORWARD` (FW) or the `INPUT` (IN) chain. For a host firewall, we analyzed IN; for a network firewall, e.g. on a gateway or router, we analyzed FW. In parentheses, we wrote the number of rules after unfolding the analyzed chain. The unfolding also features some generic, straight-forward optimizations, such as removing rules where the match expression is False. Column four (Simple rules) is the number of rules when translated to the simple firewall. In parentheses, we wrote the number of simple firewall rules when interfaces are removed. This ruleset is used subsequently to compute the partitions and service matrices. In column five (Use), we mark whether the translated simple firewall is useful. We will detail on the metric later. Column six (Parts) lists the number of IP address space partitions. For comparison, we give the number of partitions computed by ITVal in parentheses. In Column seven (ssh) and eight (http), we give the number of partitions for the service matrices for ssh and http. In column nine, we give the overall runtime of our analysis in seconds, minutes, or hours. For comparison, we put the runtime of the partitioning by ITVal in parentheses. When translating to the simple firewall, to accomplish support for arbitrary matching primitives, some approximations need to be performed. For every firewall, the first row states the overapproximation (more permissive), the second row the underapproximation (more strict).

In contrast to previous work, there is no longer the need to manually exclude certain rules from the analysis [3]. For some rulesets, we do not know the interface configuration. For others, there were zone-spanning interfaces. For these reasons, as proven in Section IV-D, in the majority of cases, we could not rewrite interfaces. This is one reason for the differences between over- and underapproximation.

We loaded all translated simple firewall rulesets (without interfaces) with `iptables-restore`. We used iptables directly to generate the firewall format required by ITVal (`iptables -L -n`). Our translation to the simple firewall is required because ITVal cannot understand the original complex rulesets and produces flawed results for them.

Fw	Rules	Chain	Simple rules	Use	Parts (ITVal)	ssh	http	Time (ITVal)
A	2784	FW (2376)	2381 (1920)	✓	246 (1)	13	9	14min (3h*)
	-	FW (2376)	2837 (581)	✗ r	1 (1)	1	1	3min (9h*)
A	4113	FW (2922)	3114 (2862)	✓	334 (2)	11	11	75min (27h*)
	-	FW (2922)	3585 (517)	✗ r	490 (1)	1	1	5min (8h)
A	4814	FW (4403)	3574 (3144)	✓	364 (2)	9	12	105min (46h*)
	-	FW (4403)	5123 (1601)	✗ r	1574 (1)	1	1	12min (3h*)
A	4946	FW (4887)	4004 (3570)	✓	371 (2)	9	12	94min (53h*)
	-	FW (4887)	5563 (1613)	✗ r	1585 (1)	1	1	11min (4h*)
B	88	FW (40)	110 (106)	✓	50 (4)	4	2	15s (2s)
	-	FW (40)	183 (75)	✓	40 (1)	1	1	9s (1s)
C	53	FW (30)	29 (12)	✓	8 (1)	1	1	7s (1s)
	-	FW (30)	27 (1)	✓	1 (1)	1	1	1s (1s)
	-	IN (49)	74 (46)	✓	38 (1)	1	1	6s (1s)
	-	IN (49)	75 (21)	✓	6 (1)	1	1	2s (1s)
D	373	FW (2649)	3482 (166)	✓	43 (1)	1	1	29s (3s)
	-	FW (2649)	16592 (1918)	✗	67 (1)	1	1	4min (33min*)
E	31	IN (24)	57 (27)	✓	4 (3)	1	2	4s (1s)
	-	IN (24)	61 (45)	✗ r	3 (1)	1	1	2s (1s)
F	263	IN (261)	263 (263)	✓	250 (3)	3	3	11min (2min)
	-	IN (261)	265 (264)	✓	250 (3)	3	3	10min (3min)
G	68	IN (28)	20 (20)	✓	8 (5)	1	2	1s (1s)
	-	IN (28)	19 (19)	✗	8 (2)	2	2	1s (1s)
H	19	FW (20)	10 (10)	✗	9 (1)	1	1	1s (1s)
	-	FW (20)	8 (8)	✗ r	3 (1)	1	1	1s (1s)
I	15	FW (5)	4 (4)	✓	4 (4)	4	4	1s (1s)
	-	FW (5)	4 (4)	✓	4 (4)	4	4	1s (1s)
J	48	FW (12)	5 (5)	✓	3 (2)	2	2	1s (1s)
	-	FW (12)	8 (2)	✓	1 (1)	1	1	1s (1s)
K	21	FW (9)	7 (6)	✓	3 (1)	1	1	1s (1s)
	-	FW (9)	4 (3)	✓	2 (1)	1	1	1s (1s)
L	27	IN (16)	19 (19)	✓	17 (3)	2	2	1s (1s)
	-	IN (16)	18 (18)	✓	17 (3)	2	2	1s (1s)
M	80	IN (92)	64 (16)	✓	2 (2)	1	2	2s (1s)
	-	IN (92)	58 (27)	✗	11 (1)	1	1	1s (1s)
N	34	FW (14)	12 (12)	✓	10 (6)	6	6	1s (2s)
	-	FW (14)	12 (12)	✓	10 (6)	6	6	1s (2s)
O	8	IN (7)	9 (9)	✓	3 (3)	1	2	1s (1s)
	-	IN (7)	8 (8)	✓	3 (3)	1	2	1s (1s)

* ITVal memory consumption, in order of appearance: 84GB, 96GB, 94GB, 95GB, 61GB, 98GB, 96GB, 21G

Table I. Summary of Evaluation on Real-World Firewalls

Performance: The code of our tool is automatically generated by Isabelle. Isabelle can translate executable algorithms to SML. For verifiable correctness, Isabelle also generates code for many datastructures which are already in the standard library of many programming languages. Usually, the machine-generated code by Isabelle can be quite inefficient. For example, lookups in Isabelle-generated dictionaries have linear lookup time, compared to constant lookup time of standard library implementations. In contrast, ITVal is highly optimized C++ code. We benchmarked our tool on a commodity i7-2620M laptop with 2 physical cores and 8 GB of RAM. In contrast, we gave ITVal a server with 16 physical Xeon E5-2650 cores and 128 GB RAM. The runtime measured for our tool is the complete translation to the two simple firewalls, computation of partitions, and the two service matrices. In contrast, the ITVal runtime only consists of computing one partition.

These benchmark settings are extremely 'unfair' for our tool. Indeed, exporting our tool to a standalone Haskell application, replacing some common datastructures with optimized ones from the Haskell std lib, enabling aggressive compiler optimization and parallelization, and running our tool on the

Xeon server, the runtime of our tool improves by orders of magnitude. Nevertheless, we chose the 'unfair' setting to demonstrate the feasibility of running fully verified code directly in a theorem prover. In addition, we preserve the property of full verification; even for the results of executable code.[5]

Table I shows that our tool outperforms ITVal for large firewalls. We added ITVal's memory requirements to the table if they exceeded 20GB. ITVal requires an infeasible amount of memory for larger rulesets while our tool can finish on commodity hardware. The overall numbers show that the runtime for our tool is sufficient for static, offline analysis, even for large real-word rulesets.

Quality of results: The main goal of ITVal is to compute a minimal partition while ours may not be minimal. Since a service matrix is more specific than a partition, a partition cannot be smaller than a service matrix. ITVal may produce spurious results (and it did in certain examples) while ours are provably correct. For firewall A, it can be seen that ITVals's results must be spurious. However, comparing the number of partitions for other rulesets, we can see that ITVal often computes better results. Our service matrices are provably minimal and can improve on ITVal's partition.

In column five, we show the usefulness of the translated simple firewall (including interfaces). We deem a firewall useful if interesting information was preserved by the approximation. Therefore, we manually inspected the rulesest and compared it to the original. For the overapproximation, we focused on preserved (non-shadowed) `DROP` rules. For the underapproximation, we focused on preserved (non-shadowed) `ACCEPT` rules. If the firewall features some rate-limiting for all packets in the beginning, the underapproximation is naturally a drop-all ruleset because the rate-limiting could apply to all packets. According to our metric, such a ruleset is of no use (but the only sound solution). We indicate this case with an $^{\mathrm{r}}$. The table indicates that, usually, at least one approximation per firewall is useful.

For brevity, we only elaborate on the most interesting rulesets and stories.

Firewall A: This firewall is the core firewall of our lab (Chair for Network Architectures and Services). It has two uplinks, interconnects several VLANs, hence, the firewall matches on more than 20 interfaces. It has around 500 direct users and one transfer network for an AS behind it. The traffic is usually several Mbit/s. The dumps are from Oct 2013, Sep 2014, May 2015, Sep 2015 and the changing number of rules indicates that it is actively managed. The firewall starts with some rate-limiting rules. Therefore, its stricter approximation assumes that the rate-limiting always applies and transforms the ruleset into a deny-all ruleset. The more permissive approximation abstracts over this rate-limiting and provides a very good approximation of the original ruleset. The ssh service matrix is visualized in Figure 1. The figure can be read as follows: The vast majority of our IP addresses are grouped into *internal* and *servers*. Servers are reachable from the outside, internal hosts are not. ip_1 and ip_2 are two individual IP addresses with special exceptions. There is also a

[5]There are methods to improve the performance and provably preserve correctness, which are out of the scope of this paper.

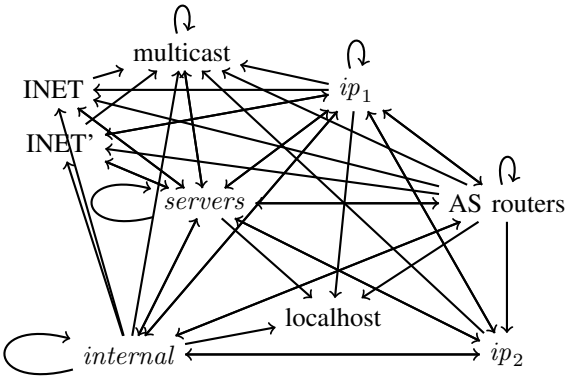

Figure 1. TUM ssh Service Matrix

group for the backbone routers of the connected AS. INET is the set of IP addresses which does not belong to us, basically the Internet. INET' is another part of the Internet. With the help of the service matrix, the administrator confirmed that the existence of INET' was an error caused by a stale rule. The misconfiguration has been fixed. Figure 1 summarizes over 4000 firewall rules and helps to easily visually verify the complex ssh setup of our firewall. The administrator was also interested in the kerberos-adm and ldap service matrices. They helped verifying the complex setup and discovered potential for ruleset cleanup.

Firewall D: This firewall was taken from a Shorewall system with 373 rules and 65 chains. It can be seen that unfolding increases the number of rules. This is due to linearizing the complex call structures generated by the user-defined chains. The transformation to the simple firewall further increases the ruleset size. This is, among others, due to rewriting several negated IP matches back to non-negated CIDR ranges and NNF normalization. However, the absolute numbers tell that this blow up is no problem for computerized analysis. The firewall basically wires interfaces together, i.e. it heavily uses `-i` and `-o`. This can be easily seen in the overapproximation. There are also many zone-spanning interfaces. As we have proven, it is impossible to rewrite interface in this case. In addition, for some interfaces, no IP ranges are specified. Hence, this ruleset is more of a link layer firewall than a network layer firewall. Consequently, the service matrices are barely of any use.

Firewall E: This ruleset was taken from a NAS device previously analyzed [3]. The ruleset first performs some rate-limiting, consequently, the underapproximation corresponds to the deny-all ruleset. In contrast to previous analysis, we obtained a more recent version of the ruleset after a system update. Our ssh service matrix reveals a misconfiguration: ssh was accidentally left enabled after the update. The service matrix for the services provided by the NAS (not listed in the table) verifies that these services are only accessible from the local network.

Firewall F: This firewall is running on a publicly accessible server. The firewall first allows everything for localhost, then blocks IP addresses which have shown malicious behavior in the past and finally allows certain services. Since most rules are devoted to blocking malicious IPs, our IP address space

partition roughly grows linear with the number of rules. The service matrices, however, reveal that there are actually only three classes of IP ranges: localhost, the blocked IPs, and all other IPs which are granted access to the services.

Firewall G: For this production server, the service matrices verified that a SQL daemon is only accessible from a local network and three explicitly-defined public IP addresses.

Firewall H: This ruleset from 2003 appears to block Kazaa filesharing traffic during working hours. In addition, a rule drops all packets with the string "X-Kazaa-User". The more permissive abstraction correctly tells that the firewall may accept all packets for all IPs (if the above conditions do not hold). Hence, the firewall is essentially abstracted to an allow-all ruleset. According to our metric, this information is not useful. However, in this scenario, this information may reveal an error in the ruleset: The firewall explicitly permits certain IP ranges, however, the default policy is `ACCEPT` and includes all these previously explicitly permitted ranges. By inspecting the structure of the firewall, we suppose that the default policy should be `DROP`. This possible misconfiguration was uncovered by the overapproximation. The underapproximation does not understand the string match on "X-Kazaa-User" in the beginning and thus corresponds to the deny-all ruleset. However, a manual inspection of the underapproximation still reveals an interesting error: The ruleset also tries to prevent MAC address spoofing for some hard-coded MAC/IP pairs. However, we could not see any drop rules for spoofed MAC addresses in the underapproximation. Indeed, the ruleset allows non-spoofed packets but forgets to drop the spoofed ones. This firewall demonstrates the worst case for our approximations: one set of accepted packets is the universe, the other is the empty set. However, manual inspection of the simplified ruleset helped revealing several errors.

VIII. Conclusion

We have demonstrated the first, fully verified, real-world applicable analysis framework for firewall rulesets. Our tool supports the Linux iptables firewall because it is widely used and well-known for its vast amount of features. It directly works on `iptables-save`. We presented an algebra on common match conditions and a method to translate complex conditions to simpler ones. Further match conditions, which are either unknown or cannot be translated, are approximated in a sound fashion. This results in a translation method for complex, real-world rulesets to a simple model. The evaluation demonstrates that, despite possible approximation, the simplified rulesets preserve the interesting aspects of the original ones.

Based on the simplified model, we presented algorithms to partition the IPv4 address space and compute service matrices. This allows summarizing and verifying the firewall in a clear manner.

The analysis is fully implemented in the Isabelle theorem prover. No additional input or knowledge of mathematics is required by the administrator. A stand-alone Haskell tool can perform the analysis automatically, only requiring the following input: `iptables-save`.

The evaluation demonstrates applicability on many real-world rulesets. For this, to the best of our knowledge, we have collected and published the largest collection of real-world iptables rulesets in academia. We demonstrated that our approach can outperform existing tools with regard to: correctness, supported match conditions, CPU time, and RAM requirements. Our tool helped to verify lack of or discover previously unknown errors in real-world, production rulesets.

Availability

The collection of 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

References

[1] A. Wool, "A quantitative study of firewall configuration errors," *Computer, IEEE*, vol. 37, no. 6, pp. 62–67, Jun. 2004.

[2] ——, "Trends in firewall configuration errors: Measuring the holes in swiss cheese," *Internet Computing, IEEE*, vol. 14, no. 4, pp. 58–65, Jul. 2010.

[3] C. Diekmann, L. Hupel, and G. Carle, "Semantics-preserving simplification of real-world firewall rule sets," in *Formal Methods (FM)*. Springer, Jun. 2015, pp. 195–212.

[4] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "FIREMAN: a toolkit for firewall modeling and analysis," in *Symposium on Security and Privacy*. IEEE, May 2006, pp. 199–213.

[5] V. Capretta, B. Stepien, A. Felty, and S. Matwin, "Formal correctness of conflict detection for firewalls," in *Workshop on Formal Methods in Security Engineering*. ACM, Nov. 2007, pp. 22–30.

[6] E. Al-Shaer and H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *INFOCOM*, vol. 4. IEEE, Mar. 2004, pp. 2605–2616.

[7] R. M. Marmorstein and P. Kearns, "A tool for automated iptables firewall analysis." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 71–81.

[8] ——, "Firewall analysis with policy-based host classification." in *Large Installation System Administration Conference (LISA)*, vol. 6. USENIX, Dec. 2006, pp. 41–51.

[9] V. Fuller and T. Li, "Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan," RFC 4632 (Best Current Practice), Internet Engineering Task Force, Aug. 2006.

[10] T. Nelson, A. D. Ferguson, D. Yu, R. Fonseca, and S. Krishnamurthi, "Exodus: Toward automatic migration of enterprise network configurations to SDNs," in *SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. ACM, 2015, pp. 13:1–13:7.

[11] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The Margrave tool for firewall analysis." in *Large Installation System Administration Conference (LISA)*. USENIX, Nov. 2010.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, last updated 2016, vol. 2283. [Online]. Available: http://isabelle.in.tum.de/

[13] C. Diekmann, L. Schwaighofer, and G. Carle, "Certifying spoofing-protection of firewalls," in *11th International Conference on Network and Service Management, CNSM*, Barcelona, Spain, Nov. 2015.

[14] A. Wool, "The use and usability of direction-based filtering in firewalls," *Computers & Security*, vol. 23, no. 6, pp. 459–468, 2004.

Acknowledgments

Lars Hupel commented on early drafts of this paper. We thank all (anonymous) administrators who donated their firewall configs. This work has been supported by the German Federal Ministry of Education and Research, project SURF, grant 16KIS0145, and by the European Commission, project SafeCloud, grant 653884.