Continuous Verification of Network Security Compliance

Claas Lorenz, Vera Clemens, Max Schrötter, and Bettina Schnor

***Abstract*—Continuous verification of network security compliance is an accepted need. Especially, the analysis of stateful packet filters plays a central role for network security in practice. But the few existing tools which support the analysis of stateful packet filters show runtimes in the order of minutes to hours making them unsuitable for continuous compliance verification.**

In this work, we address these challenges and present a solution which is based on the application of formal methods. First, we introduce the formal language *FPL* that enables a high-level human-understandable specification of the desired state of network security. Second, we demonstrate the instantiation of a compliance process using a verification framework that analyzes the configuration of complex networks and devices - including stateful firewalls - for compliance with FPL policies. Our evaluation results show the scalability of the presented approach for the well known Internet2 and Stanford benchmarks as well as for large firewall rule sets where it outscales state-of-the-art tools by a factor of over 41.

***Index Terms*—Network, Security, Compliance, Formal Verification**

I. Introduction

ASSESSING the state of network security compliance is a recurring, error-prone, and expensive task for many organizations. Real world networks evolve over time and technical configurations of network security policies like firewall rule sets tend to become increasingly complex. This makes it hard to reconstruct whether the actual network configuration still fulfills the organization's compliance rules. Further, compliance rules are often not stated explicitely or not in a form which makes an automatic verification possible. Additionally, the ongoing transition towards state-of-the-art networking with IPv6 poses a challenge for formal approaches. The introduction of control protocols like neighbor discovery or router advertisement and dynamic extension header chains leads to an increase of the complexity of firewall rule sets and consequently, the state space.

Since IPv6 traffic has been growing continuously over the last years - currently, Google monitors a share of more than 35 %[1] - it is getting more and more important to support IPv6. This is also reflected in several research papers ([1], [2], [3], etc.). Hence, we started the FaVe project with the motivation to support IPv6 right away. We demonstrated how to reuse the fast verification engine NetPlumber [4] to model IPv6 packets as well as the concept of firewalls [5]. But the support for compliance checking was still lacking and the support of stateful firewalls was rudementary. Our former approach for FaVe was based on an online monitoring of state changes at runtime covering only snapshots instead of all possible states. In this work, we replace this approach with a static analysis of the firewall configuration to determine and model all possible states efficiently.

Our approach leverages formal methods to automate compliance verification and it offers accessible tooling for expressing policies and modeling network topologies as well as complex network devices like stateful packet filters or routers. For this purpose, at first, security officials express compliance rules in a high-level, semi-natural, and language compatible to common *Role Based Access Control* (RBAC). Second, the network configuration is formally modeled and checked by our verification system *FaVe* against the security rules. Finally, a human-understandable report is generated. Given a sufficiently fast verification, this process can be repeated often and hence, it reduces the obstacles refraining administrators from changing security related configurations like firewall rules.

As depicted in Figure 1 the compliance verification workflow consists of three major parts - the specification of the security policy, the network modeling, and the compliance verification. At first, the security official needs to specify the *security policy* (1a) using abstract terms of the network, e.g., reachability of services or subnets.

The network administrator describes an *inventory* (1b) which maps these abstract terms to their technical implementation in the network. The policy is used to generate formal *security invariants* (step (2a)) that are instrumented in a *verification engine*. Currently, FaVe reuses and extends the verification engine NetPlumber [4]. In the second part, the *network topology* and *configurations* of entities like routers, firewalls, or switches need to be modeled using the original device configurations as input, e.g., firewall rule sets or switch configurations. This model covers the forwarding behaviour of the network and is instrumented in the verification engine as well.

Finally, after verifying the model against the security policy (3) the results are used to report the state of security compliance to the security official (4). All steps except for the policy and inventory definitions can be automated which allows a *continuous reverification* of network security either periodically or after configuration changes.

In this work, we make the following major contributions:

- FaVe Policy Language (FPL): A formal language featuring hierarchical roles and services for expressing security policies which are concise but remain understandable for non-technical auditors and can be integrated into RBAC frameworks. (see Section III)
- We extend FaVe with support for stateful packet filter

The authors are with the Department of Computer Science, University of Potsdam, Germany, e-mails: {cllorenz, clemens}@uni-potsdam.de, {schroetter, schnor}@cs.uni-potsdam.de

[1] https://www.google.com/intl/en/ipv6/statistics.html

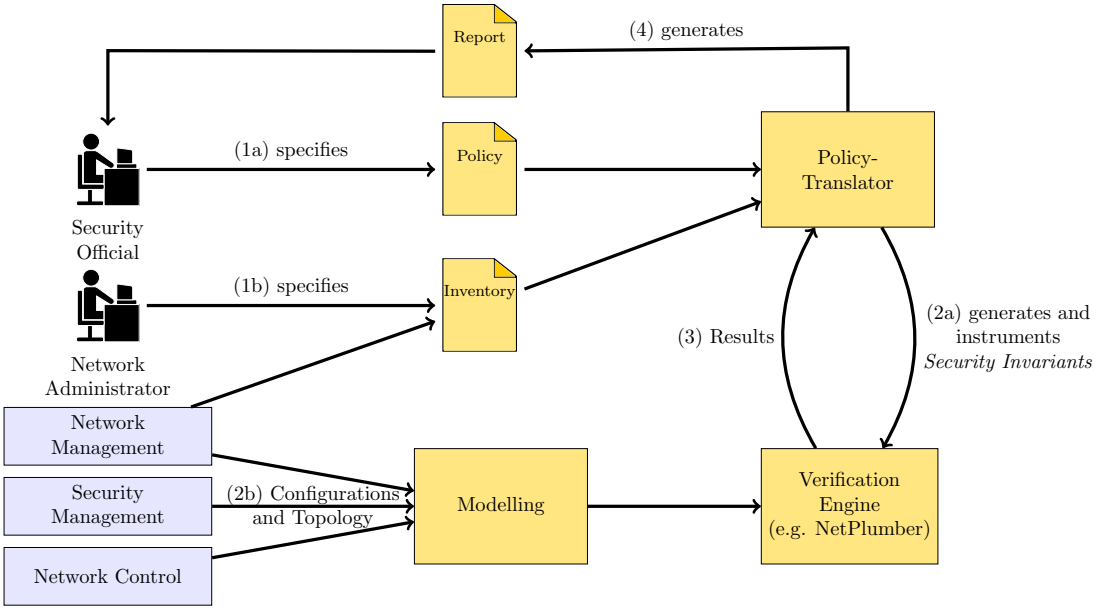

Fig. 1. Compliance verification workflow (yellow boxes) and its integration with existing management systems (blue boxes).

firewalls by a technique called *state shell interweaving*. (see Section IV)

- An evaluation of FaVe's prototype implementation with the prominent Stanford and Internet2 benchmarks [4] shows the scalability of the presented approach. Further, results with the TUM benchmark [2], a large real world stateful packet filter rule set, show that FaVe outperforms other state-of-the-art tools more than 41-fold. (see Section VI)

This work is structured as follows. After giving an overview of the current state-of-the-art in Section II, we introduce the FaVe Policy Language in Section III. Then, we provide a formal modeling technique for stateful packet filters in Section IV. This is followed by an overview over the verification framework in Section V, and finally in Section VI, we evaluate the prototypical implementation of FPL and FaVe concerning their scalability and performance against the state-of-the-art.

II. Overview of related Verification Tools

There have been numerous approaches to apply formal methods in the fields of networks and network security. On the one hand, tools for *firewall verification* put an emphasis on models for this feature rich device class which is central for network security. On the other hand, tools for *data plane* and *control plane analysis* focus on models for whole networks and typically they offer far greater performance results than firewall verification tools. In the following, we give an overview of these verification approaches, highlight notable tools, and compare them to FaVe.

1) Firewall Verification Tools: Tools to analyze firewalls follow a long tradition and leverage *Finite Automata* [6], *Binary Decision Diagrams* (BDD) [7], *Firewall Decision Diagrams* (FDD) [8], *Boolean Satisfiability* (SAT) [9], [1], *theorem prover* [2], or *Satifiability Modulo Theories* (SMT) [10], [11] as verification techniques to discover anomalies and insecurities in packet filter rule sets and networks. Also, support for IPv6 addresses [3] and extension header chains [1] was shown. Later, analysis of mutable datapaths for stateful firewalls and other middleboxes has been presented featuring *virtual state tags* [2], *state oracles and SMT solving* [12], *predicate transformers* [13], *symbolic model checking* [14], and *liveness verification* [15].

Diekmann et altera [2] have presented an approach to verify stateful `iptables` rule sets where their tool `fffuu` itself is fully proven by the theorem prover *Isabelle/HOL*. The tool calculates service matrices which partition the address space concerning a pair of fixed source and destination ports. In contrast, FaVe covers all possible port combinations in a single run. While being more general FaVe offers a much better performance than `fffuu` (cf. Section VI-C). Additionally, the IPv6 version `fffuu6` does not support extension header chains. Instead, the tool handles unknown fields by an approximation strategy leading to imprecise reachability analyses.

A recent approach to the verification of stateful network functions is NetSMC [14]. The tool represents network device models as state machines which are composed based on the network's topology. The state machines are encoded in terms of a symbolic representation for a custom model checker presented by the authors. For their evaluation, they use a model for the *pfSense* firewall learned with their tool *Alembic* [16]. NetSMC supports an LTL dialect to specify policies. While being powerful, LTL can hardly be considered suitable for network administrators or security officials. FPL on the other hand was designed to be understood by other people than computer scientists (see Section III). Additionally, NetSMC lacks support of IPv6.

These firewall verification tools have in common that they run quite slow in the range of minutes to hours for larger firewall rule sets as found in many corporate networks. Hence, the ability to scale to large networks is rather limited for

these model checking and theorem prover based approaches. In addition, the support for IPv6 is often treated with low priority.

In contrast to this work, none of these tools offer an accessible compliance workflow. The only exception is given by Microsoft's *SecGuru* [10] which scales well and puts some effort into verifying compliance to support Azure cloud network engineers by offering *Cloud Contracts* in first order logic. Yet, the approach is tightly tailored to the well structured environment of cloud data centers. Both, the compliance specification and the verification technique are not easily generalizable to arbitrary networks.

2) Data Plane Verification Tools: These tools analyze snapshots of the data plane leveraging smart, very fast, and domain specialized data structures and algorithms including *Header Space Analysis* (HSA) [17], [4], packet *Equivalence Classes* [18], and *Atomic Predicates* [19], [20]. These approaches scale to networks with a couple of 100k forwarding rules and some support incremental updates [4], [18] which allow reverification times in the range of milliseconds. Further improvements have been made by dividing snapshots into smaller but closed portions [21], [22] which are easier to analyze and allow massive parallelization [23]. Additional improvements can be made by limiting on IP routed networks instead of an arbitrary amount of header fields [24]. All in all, some tools like Libra scale to complex networks that contain some 100m forwarding rules [21]. Despite being enormously fast and scalable, these data plane approaches do not support verification of networks containing stateful middleboxes like firewalls as stated in the conclusion of a recent survey [25]. FaVe leverages the HSA based *NetPlumber* engine [4] as verification backend and introduces capabilities to model stateful packet filters (cf. Section IV). NetPlumber offers simple reachability policy descriptions based on regular expressions and is meant to be instrumented by FML [26] - a Datalog like management language for packet flows. Both seem to be more suitable for technical experts rather than security officials. FPL and FaVe offer a more accessible way to describe and verify security compliance which can be integrated into standard RBAC workflows. AP Verifier [19] has shown to be even faster than NetPlumber, but missed to support incremental updates. This absent feature is added by the approach of APKeep [20] which makes it a very promising canidate, too, as it offers modular modeling, very fast runtimes, and incremental updates.

A differing approach to verify network data planes is offered by SymNet [27] which is based on symbolic execution. It verifies equivalence between a network model consisting of device models and an abstract policy model. The models of packet processing devices are expressed with the imparative *Symbolic Execution Friendly Language* (SEFL) [28] and the authors ship a variety of prebuild models for different network devices. However, unlike FPL, SymNet does not offer accessible means to express security policies. Also, the tool does not scale very well for large firewall rule sets as shown in the evaluation in Section VI-C.

3) Tools for Control Plane Verification: Tools like *Batfish* [29], *Minesweeper* [30], or *Tiramisu* [31] infer forwarding behaviour from control plane configuration in routed networks. By doing so they are able to analyze several data plane incarnations at once instead of single snapshots. Depending on the tool, routing protocols like BGP, OSPF, IGP, and others may be considered. Since they do not support firewalls or other complex devices, we do not consider these tools to be suitable for network security verification.

III. The *FaVe Policy Language*

The approaches to verify network security like NetPlumber or AP Verifier focus on the efficiency and scaling of the verification process while putting less effort into the definition of accessible security policies. Yet, to achieve an auditable security compliance both aspects are of equal importance. Current approaches to describe policies fall short for practical usage (see Section III-E). Therefore, in this section we present the declarative *FaVe Policy Language* (FPL) that enables the definition of essential yet auditable policies. It is based on hierarchical roles and also allows the specification of stateful reachability policies.

The main idea of FPL is to separate functional roles from their technical implementation in the network and define policy rules for these roles. The benefit of this approach is similar to the benefit of RBAC for user access - moreover, FPL may be integrated into RBAC frameworks that follow NIST standardization [32] as discussed later in Section III-D. A separation of roles and configuration details helps security officials and administrators to focus on their strengths and enables an independent evolution of compliance rules and their actual implementation. I.e., on the one hand the security policy does not require any changes when the network configuration changes, e.g., when all web servers are migrated to another address space or from IPv4 to IPv6. On the other hand, if compliance mandates changes of the security policy, e.g., due to risk management, the report shows whether any changes to the network configuration are necessary to comply with the new requirements.

A. Inventory Description

Before describing the security policy it is necessary to define the network inventory which maps abstract names to technical details. FPL's approach for the inventory description is inspired by network administration tools like Ansible [33] and Puppet [34] which are widely used in practise. FPL offers two language features to express and order terms of the network - *roles* and *services*. Services describe transport layer protocols and ports, e.g., HTTPS on TCP port 443 or DNS on UDP port 53. Roles represent network entities like single machines, groups of hosts, or even complete subnets. They may offer services and can hold attributes like IP addresses or domain names. Roles can be aggregated by abstract roles called *super roles* which, in turn, behave like normal roles as they can offer services or hold attributes themselves. When describing policies one may refer to roles, super roles, and optionally their offered services.

We define the relation between super roles and sub roles in terms of an acyclic directed graph where a super role points to

an arbitrary amount of sub roles. Loop-freeness allows us to define an unambigious downstream resolution mechanism for services and attributes. As a role may belong to multiple super roles there might be several root nodes, i.e., nodes without incoming edges. Attribute resolution is performed by starting at the root nodes and by collecting all attributes and services along a path until a leaf is reached. If an attribute supersedes another, the more specific is propagated, e.g., when comparing IP address ranges. Later, when verifying compliance, these leaf roles will serve as communication endpoints and their collected attributes will characterize traffic they emit.

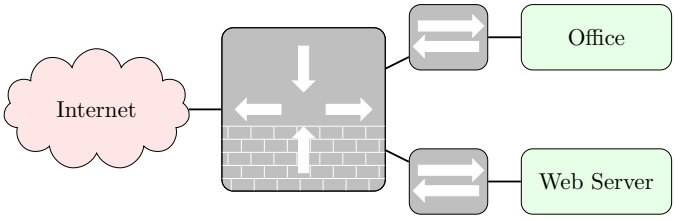

Fig. 2. The example network comprising of a firewall, two switches and two hosts.

Throughout this paper, we demonstrate several aspects of the formal verification process by utilizing the example network depicted in Figure 2. It comprises of two switched subnets separated by a perimeter firewall from the Internet. One subnet represents an internal network with office machines whereas the other subnet offers a DMZ with a public web server. The inventory for the example network is the following:

```
describe service HTTP
    protocol = 'tcp'
    port = 80
end

describe service SSH
    protocol = 'tcp'
    port = 22
end

describe role Office
    description = 'Hosts of the office network.'
    ipv6 = '2001:db8::200/120'
end

describe role WebServer
    description = 'Public web servers.'
    ipv6 = '2001:db8::100/120'
    offers HTTP
    offers SSH
end

describe role AllClients
    description = 'Internal and external clients.'
    includes Internet
    includes Office
end
```

There are the two TCP based services defined - *SSH* and *HTTP* - which listen on their default ports. Further, there are two basic roles defined representing the *Office* machines and the *WebServer*. Also, the web server offers HTTP and SSH. Finally, there is a super role *AllClients* which contains the Office role and the *Internet*. For comfort purposes FPL includes the builtin role *Internet* which represents external clients or machines that offer all sorts of services.

B. Policy Specification

In FPL, policies are described as lists of reachability rules between roles. Policies may either follow an *allow* listing or a *deny* listing approach but cannot mix them. This avoids a major source of confusion and conflicts when auditing or writing security policies in practice. All rules are of the form:

```
Subject Operator Object[.Service|.*]
```

where *subject* and *object* are roles and the *operator* may be one of the following:

- `--->` simple reachability
- `<-->` bidirectional simple reachability
- `<->>` stateful reachability

Simple reachability A `--->` B means that traffic flows from A to B. As purely unidirectional policies are seen rather rarely in practice, e.g., when using data diodes, FPL offers the bidirectional operator A `<-->` B for comfort purposes. The same semantics can be achieved by stating two rules A `--->` B and B `--->` A.

Finally, the stateful operator `<->>` allows the specifications of policies where the traffic flows are subject to stateful communication patterns. The initiator (left hand of the operator) reaches the recipient (right hand) which in turn only sends traffic reactively. The recipient must not initiate communication on its own. This maps to protocols with stateful behaviour, e.g., TCP based protocols, and may be seen in practice most frequently.

Reachability policies may be expressed more precisely by specifying a target service offered by the destination role which is denoted by a dot, i.e., $A.S$ where a role A offers a service S. If multiple services should be reached, one needs to specify a reachability rule for each target service. For comfort purposes FPL allows to specify that all services offered by a role can be reached: $A.^*$. During verification all traffic flows must be covered by at least one explicit reachability rule. Otherwise, compliance is not given and a violation will be reported. Also, unreached targets that should be reached according to a rule are reported as this indicates possible disruptions of business continuity.

For the example from Figure 2, a security policy may look like this:

```
describe policy (default: deny)
    AllClients <->> WebServer.HTTP
    Office <->> WebServer.SSH
    Office <->> Internet
end
```

The benefits of the grouping by super roles already appears in this small example as there are only three policy rules necessary. Without the *AllClients* super role, HTTP access for the web server would require two separate rules - one for the *Internet* and one for the *Office* role. The gain is even more evident for larger examples like the policy for the UP benchmark given in Appendix B with only 33 policy rules in comparison to 1035 `iptables` rules. The policy rules are much easier to understand and analyze. The UP benchmark will be used in the evaluation section (cf. Section VI).

C. Limitations

Currently, FPL does not support confidentiality policies, e.g., VPN or TLS based data encryption between roles. Conceptually, extending FPL's policy specification and FaVe's verification is possible and it is subject to future work. Also, NAT is considered a security feature in many real world networks. We omitted NAT in FPL since a) its masquerading resembles more a technical realization detail rather than a matter of security policy, and b) IPv6 offers a tremendous address space that obsoletes auxiliary functions like NAT in the foreseeable future. Conceptually, it is possible to add support to model NAT with FaVe by introducing rewriting rules into the designated tables of the packet filter model.

D. Compatibility with Role Based Access Control

The role concept in FPL shows similarities to RBAC and, indeed, FPL is compatible with NIST RBAC [32]. RBAC regulates the relations between users and access permissions through roles. User access to roles and session management are beyond FPL's scope and can be omitted in this discussion. Concerning RBAC's notion of roles the `Subject Operator Object` triples in FPL correspond to the *permission assignment* relation in *Core RBAC* enabling compatibility. Super role relations in FPL are realized as a DAG and therefore, they can form general role hierarchies as specified by *Hierarchical RBAC*. FPL's resolution mechanism corresponds to the *heritage* relation and the *authorized permission* function in Hierarchical RBAC. Finally, *Constrained RBAC* only deals with role access regulation and therefore, it is out of FPL's scope. Hence, FPL is fully compatible with NIST RBAC and can be integrated into standard RBAC frameworks.

E. Related Policy Languages

The *Flow Management Language* (FML) [26] was designed for specifying security policies for reactive controllers in Software Defined Networks (SDN). FML offers a Datalog-like flow-centric syntax and the order of rules does not matter. In contrast, FPL offers abstractions to separate security policy from technical details which improves readability as well as maintainability. Also, FPL is designed to exclude the specification of conflicting policies. In consequence, the order of policy rules is not important at all which helps to avoid subtle insecurities in practice. In comparison to FML which relies on a conflict detection and resolving mechanism, FPL avoids rule conflicts by design.

Another approach is given by *ForestFirewall* [35] which is a tool set to generate firewall rule sets for SCADA systems. They decouple policies from technical details by offering a simplified *zone-conduit* model following the IEC 62443 standards to group assets and specify their allowed reachabilities. Therefore, they propose an operator similar to FPL's stateful reachability operator, but they do not cover simple reachability.

Finally, Cisco's *TrustSec* [36] offers a matrix to describe reachability policies between fine grained segments for their *Software Defined Access* products. While helping administrators to keep an overview of their policies, TrustSec lacks the ability to consisely describe policies for groups of segments. FPL, on the other hand, provides hierarchies to group roles which significantly reduces the policies' sizes. Another approach is offered by the open source tool *FirewallBuilder* [37]. Reachability policies can be described based on a flexible description system featuring named network objects which encapsule technical details. Yet, the tool is limited to generate rule sets only for a single central firewall and does not support complex networks.

IV. Modeling Stateful Behaviour

As seen in Section II, most of the existing verfication tools only support the modeling of stateless behaviour. Especially, the fast data plane approaches like NetPlumber or AP-Verifier are limited in their expressiveness concerning stateful policies. On the other hand, previous efforts on firewall verification with *ad6* which is based on a model checking approach yielded poor perfomance results ranging in the tens of minutes [1].

FaVe overcomes these shortcomings. The basic idea is to model stateful behaviour by deriving a virtual rule set called the *state shell* which only consists of stateless rules. These rules are constructed according to a state behaviour function and woven into the packet filter's tables with respect to the conservation of overall filter semantics.

This approach allows FaVe to re-use fast data plane engines for verification while covering stateful behaviour as well.

Before going into detail of formalizing stateful packet filter behaviour, we start with a description of the internals of Linux `iptables/netfilter` [38][2]. We chose `iptables` for our formalization and prototype implementations for two reasons. First, it is one of the most common packet filter implementations found in practice. And second, it is possible to transcompile other firewall configurations, e.g., for OpenBSD's `pf` or FreeBSD's `ipfw`, to an `iptables` configuration as shown in [11]. Therefore, our formalization for `iptables` serves as a *generic packet filter model* suitable for a large variety of real world scenarios.

A. Stateful Semantics of Linux `iptables`

Any packet, which enters the `netfilter` framework within Linux' network stack, traverses the `conntrack` table first. Here, it is checked whether the packet is related to a known connection. If so, the connection's state is checked, updated, and the packet is marked with the current state, e.g., `NEW, ESTABLISHED`, in its meta data. Later, when traversing filtering tables, e.g., `FORWARD` or `INPUT`, rules may match the state data to make filtering decisions. If the packet does not belong to a known connection, a shadow entry in `conntrack` is created which is activated once the packet leaves the `netfilter` framework, i.e., being accepted in the `INPUT` or `POSTROUTING` chain.

The following example demonstrates that `iptables` offers a great degree of freedom to administrators to implement their policies.

[2]In the following, we refer to `iptables/netfilter`'s frontend as `iptables` and to its in-kernel framework as `netfilter`.

```
(0) ip6tables -P FORWARD DROP
(1) ip6tables -A FORWARD -p tcp --dport 22 \
    -j ACCEPT
(2) ip6tables -A FORWARD -s 2001:db8::2 -j DROP
(3) ip6tables -A FORWARD -m conntrack \
    --cstate ESTABLISHED -j ACCEPT
(4) ip6tables -A FORWARD -d 2001:db8::1 -p tcp \
    --dport 80 -j ACCEPT
```

Initial SSH packets are handled by rule (1), and initial HTTP packets by rule (4). Rule (3) guarantees that the backward traffic of SSH and HTTP connections is accepted. While subsequent HTTP packets belonging to this connection will be handled by rule (3), the forward SSH traffic will still be handled by rule (1) and the backward traffic by rule (3).

`netfilter`'s concept to tag packets with a state and check it like a normal header field results in an arbitrary amount of possible state checking rules in `conntrack`. In a sense, a new *virtual* rule is introduced to the rule set for each established connection. These virtual rules may appear at several points in the rule set depending on the given state checking rules and the inter-rule dependencies within the rule set.

A straightforward approach to model this behaviour would introduce a copy with reversed directions right in front of any state producing rule and mark it with a *backward flag*. This approach falls short as it does not consider all dependencies between state producing and state checking rules. For instance, rule (2) from our example drops packets from host `2001:db8::2`. If we add a virtual rule with reversed direction for rule (1) in front of rule (1), this rule would accept SSH packets originating from that address whereas in reality these packets would have been dropped before reaching the state checking rule (3). Hence, a more sophisticated method to model stateful behaviour of `iptables` is needed and it must take into account all inter-rule dependencies.

B. Foundations of Formalization

After explaining the stateful semantics of `iptables`, we turn towards its formalization. We do so by applying the concept of *Headerspaces* [17] where tuples of packet header fields are used to express flows and rule matches.

A header field tuple t^h consists of a header field $h \in \mathcal{H}$ and its respective value set $v_h \subseteq \mathcal{V}_h$:

$$t^h = (h, v_h)$$

For instance, the header field tuple $(dport, \{80\})$ describes the subset of packets with destination port 80.

To model stateful behaviour, we introduce new *virtual* header fields: the *backwards flag* $\mathcal{V}_{\texttt{back}}$ (0 or 1), the *state* $\mathcal{V}_{\texttt{state}}$ (NEW or ESTABLISHED), as well as the *ingress* $\mathcal{V}_{\texttt{iif}}$ and *egress interfaces* $\mathcal{V}_{\texttt{oif}}$. These are no explicit protocol header fields, but belong to the packet filter's internal meta data. Therefore, we call these header fields *virtual*.

Throughout the rest of this section we use the following rule set for illustration which implements the FPL policy example from Section III-B

```
(0) ip6tables -P FORWARD DROP
(1) ip6tables -A FORWARD --in-interface eth0 \
      -s 2001:db8::0/32 -j DROP
(2) ip6tables -A FORWARD -m conntrack --ctstate \
      ESTABLISHED -j ACCEPT
(3) ip6tables -A FORWARD --out-interface eth0 \
      -s 2001:db8::200/120 -j ACCEPT
(4) ip6tables -A FORWARD -d 2001:db8::101 -p tcp \
      --dport 80 -j ACCEPT
(5) ip6tables -A FORWARD -s 2001:db8::200/120 \
      -d 2001:db8::101 -p tcp --dport 22 -j ACCEPT
```

Rule (1) is a widely used sanity check against spoofing and not derived from the policy directly.

The Headerspace for this example has the header fields *IPv6 source* and *destination address*, *protocol*, *source* and *destination port*, and the virtual fields *backwards*, *state*, and *ingress* and *egress interface*. Therefore, the corresponding Headerspace is given by (addresses in IPv6 short notation):

$$\begin{aligned}
\mathcal{H} = \{&h_1 = \texttt{iif}, h_2 = \texttt{oif}, h_3 = \texttt{sip},\ h_4 = \texttt{dip},\\
&h_5 = \texttt{proto},\ h_6 = \texttt{sport},\ h_7 = \texttt{dport},\\
&h_8 = \texttt{state},\ h_9 = \texttt{back}\}\\
\mathcal{V}_{\texttt{iif}} = \mathcal{V}_{\texttt{oif}} &= \{\texttt{eth0}, \texttt{eth1}, \texttt{eth2}\}\\
\mathcal{V}_{\texttt{sip}} = \mathcal{V}_{\texttt{dip}} &= \{\texttt{0::0/0}\}\\
\mathcal{V}_{\texttt{proto}} &= \{0, ..., 255\}\\
\mathcal{V}_{\texttt{sport}} = \mathcal{V}_{\texttt{dport}} &= \{0, ..., 65535\}\\
\mathcal{V}_{\texttt{state}} &= \{NEW, ESTABLISHED\}\\
\mathcal{V}_{\texttt{back}} &= \{0, 1\}\\
\mathcal{V}_{\mathcal{H}} = &\{\mathcal{V}_{\texttt{iif}}, \mathcal{V}_{\texttt{oif}}, \mathcal{V}_{\texttt{sip}}, \mathcal{V}_{\texttt{dip}}, \mathcal{V}_{\texttt{proto}}, \mathcal{V}_{\texttt{sport}}, \mathcal{V}_{\texttt{dport}},\\
&\mathcal{V}_{\texttt{state}}, \mathcal{V}_{\texttt{back}}\}
\end{aligned}$$

In order to improve readability we introduce the following short notations for state values: $v_{new} = \{NEW\}$ and $v_{est} = \{ESTABLISHED\}$.

For tuples of the same header field h, the intersection is defined as follows:

$$t_1^h \cap t_2^h = (h, v_{t_1}) \cap (h, v_{t_2}) = (h, v_{t_1} \cap v_{t_2})$$

We define a rule r at index i to have a matching part m_i and an action a_i:

$$r_i : m_i \rightarrow a_i$$

In general, the match m_i consists of a set of header field-value tuples

$$m = \{(h_i, v_{h_i}) \mid \forall i = 1...n :\ h_i \in \mathcal{H}, v_{h_i} \subseteq \mathcal{V}_{h_i}\}$$

with unique header field identifiers. Note, that for any match all headers are initialized by default which conforms with real world rule specifications. Those express specific matching criteria for network packets leaving all unspecified header fields as not relevant for that particular rule. Here, we adapt this semantics by initializing those fields with the full set, i.e., $v_h = \mathcal{V}_h$.

Using this formalization, rule (4) in the `iptables` example is written as

$$\begin{aligned}
r_4 : \{\ &(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{state}, \mathcal{V}_{\texttt{state}})\\
&(\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::101}\}),\\
&(\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}),\\
&(\texttt{back}, \{0, 1\})\\
\}\ &\rightarrow accept
\end{aligned}$$

To intersect matches, for each header field simply the corresponding tuples are intersected:

$$m_1 \cap m_2 = \{t_1^{h_i} \cap t_2^{h_i} \mid \forall i = 1...n :\ h_i \in \mathcal{H}, t_1^{h_i} \in m_1, t_2^{h_i} \in m_2\}$$

Since all headers are initialized either explicitly or implicitly by definition this operation is complete.

Finally, a filtering rule set R is defined as an ordered set of rules:

$$\begin{aligned} R = \{ & \\ & r_1 : m_1 \rightarrow a_1, \\ & r_2 : m_2 \rightarrow a_2, \\ & \quad \dots \\ & r_n : m_n \rightarrow a_n \\ \} & \end{aligned}$$

where the rule with the lowest index has the highest priority.

C. Modeling State using State Shell Interweaving

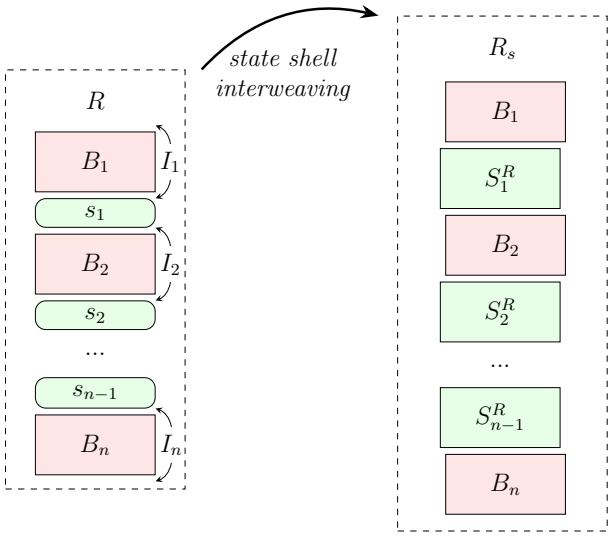

Fig. 3. Principle of interweaving the state shell with the original rule set.

As seen before, `iptables` permits rule set configurations with complex state introduction and checking dependencies. The original rule set, as depicted in Figure 3, may have an arbitrary mix of state *checking* rules s_i and blocks of state *introducing* rules B_i. This complex pattern poses a central challenge for formal verification and can be found in practice as seen in 5 out of 39 real world rule sets from [39].

Our central idea to cope with this challenge is the following:

1) We derive a virtual rule set which covers all states that could have been introduced by the original rule set - we call this derivate the *general reverse state shell* S^R.
2) Then, for any state checking rule in the original rule set, we calculate a *conditional reverse state shell* S_i^R that incorporates only states relevant for that rule.
3) Finally, we *interweave* the conditional reverse state shells into the rule set while preserving any dependencies between state introducing and state checking rules. The resulting rule set R_S consists only of stateless rules, but models the same header space as before.

Firstly, we convert our example `iptables` rule set to a filtering rule set as defined in Section IV-B. Note that in `iptables` the first rule (with an index of 0) defines the default policy which applies if no other rule matches. Therefore, it becomes the final rule r_6 in the converted rule set. Then, before deriving the general reverse state shell, we collect the set of state checking rules $S \subseteq R$ which will also be used later for the conditional reverse state shells:

$$S = \{r_i : m_i \rightarrow a_i \mid (state, v_{est}) \in m_i,\ r_i : m_i \rightarrow a_i \in R\}$$

Concerning our example, rule (2) is the only state checking rule. Hence $S = \{r_2 : \{..., (\texttt{state}, v_{est}), ...\} \rightarrow accept\}$.

1. General Reverse State Shell Derivation: We define the *tuple reverse function* ρ as a helper function that swaps fields which determine a packet's source resp. destination information:

$$\rho : \mathcal{H} \times \mathcal{V}_{\mathcal{H}} \rightarrow \mathcal{H} \times \mathcal{V}_{\mathcal{H}}$$

$$\rho(t^h) = \begin{cases} (\texttt{sip}, v_{\texttt{dip}}), \text{ if } h = \texttt{dip} \\ (\texttt{dip}, v_{\texttt{sip}}), \text{ if } h = \texttt{sip} \\ (\texttt{sport}, v_{\texttt{dport}}), \text{ if } h = \texttt{dport} \\ (\texttt{dport}, v_{\texttt{sport}}), \text{ if } h = \texttt{sport} \\ (\texttt{iif}, v_{\texttt{oif}}), \text{ if } h = \texttt{oif} \\ (\texttt{oif}, v_{\texttt{iif}}), \text{ if } h = \texttt{iif} \\ t^h, \text{ else} \end{cases}$$

Now, we can derive the general reverse state shell S^R by reversing the matches of all non-state-checking rules in R and by marking them with the virtual `back` flag to point in backward direction. Therefore, we introduce m_{bck} which has only the backwards flag set to 1.

$$\begin{aligned} S^R = \{ & \\ & r_i : \{\rho(t^h) \mid t^h \in m_i\} \cap m_{bck} \rightarrow a_i \mid r_i : m_i \rightarrow a_i \in R \backslash S \\ \} & \end{aligned}$$

The general reverse state shell keeps the relative order of the rules that produce state as well as non-state-producing rules, i.e., those with all directional fields set to their respective value domain. Therefore, the dependencies of the original rule set remain intact.

Concerning our example, rule (4) produces r_4 in the general reverse state shell:

$$\begin{aligned} r_4 : \{ \quad & (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{state}, \mathcal{V}_{\texttt{state}}) \\ & (\texttt{sip}, \{\texttt{2001:db8::101}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}) \\ & (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\ & (\texttt{back}, \{1\}) \\ \} \quad & \rightarrow accept \end{aligned}$$

2. Conditional Reverse State Shell Calculations: `iptables` allows multiple distinct state checking rules in a rule set that match different sets of packets. We model this behaviour by filtering the general reverse state shell to contain only those rules matching packets that are relevant for a particular state checking rule. Later, these *conditional reverse state shells* replace the state checking rules in the original rule set.

For this purpose, we calculate a set of numbered intervals I that mark the boundaries of the rule blocks B_i as a helper by saving the index before and after each block (as indicated by the arrows in Figure 3). An interval (i, k, l) represents the i-th block which includes the rules from index $k-1$ to index $l+1$. In our example, the rule set is split into two blocks by the

state checking rule (2). Thus $I = \{(1,0,2),(2,2,7)\}$. These boundaries are needed to determine the relative positions of rules in the new rule set for both - the conditional reverse state shells and the rule blocks.

The following function creates one conditional reverse state shell for every state checking rule s_l in S. The general reverse state shell is specialized to fit the particular subset of packets handled by this state checking rule, i.e., $m_j \cap m_l$ for every $r_j : m_j \rightarrow a_j$ from S^R. Only rules with non-empty matches form the conditional reverse state shells S_i^R shown as green boxes in Figure 3.

$$
\begin{aligned}
S_i^R = \{\ & r_{(2i+1)\cdot|R|+j} : m_j \cap m_l \rightarrow min(a_l, a_j) \mid \\
& r_j : m_j \rightarrow a_j \in S^R, \\
& \forall (h, v_h) \in m_j \cap m_l : v_h \neq \emptyset \\
\}&
\end{aligned}
$$

Each state shell entry is specialized concerning the state checking rules' match by intersection. If any header field is empty, there cannot exist any matching packet and the entry can be omitted. E.g., if the state checking rule only applies to TCP then UDP or ICMP states are not relevant for this rule. In addition, it needs to be considered that state checking rules may drop packets which applies to all packets in reverse direction as well. For this purpose, we define a total order for rule actions, i.e., $drop < accept$, and use a minimum function to determine the right action for each rule in the conditional reverse state shell.

Finally, the new rule's index is set to a value that simplifies interweaving later while preserving the relative order within the conditional reverse state shell.

Since there is only one state checking rule in our example rule set, we obtain only one conditional reverse state shell[3]:

$$
\begin{aligned}
S_1^R = \quad & \{r_{(2\cdot1+1)\cdot6+1=19} : m_1^{S^R} \cap m_2 \rightarrow drop, \\
& r_{(2\cdot1+1)\cdot6+3=21} : m_3^{S^R} \cap m_2 \rightarrow accept, \\
& \cdots \\
& \}
\end{aligned}
$$

3. State Shell Interweaving: Before we may weave the conditional reverse state shells into the original rule set, we need to reindex the rules within the blocks B_i to make space for the new reverse rules.

Additionally to renumbering, the rules that apply explicitely on unknown connections are set to act in forward direction only. This is done by setting the *backwards* flag to 0. For this purpose, we introduce m_{fwd} (analogously to m_{bck}) which has only the backwards flag set to 0.

We define for each $(i,k,l) \in I$:

$$
\begin{aligned}
B_i = \{\ & r_{2i\cdot|R|+j} : m \rightarrow a_j \mid\ r_j : m_j \rightarrow a_j \in R, \\
& k < j < l, \\
& m = \begin{cases} m_j \cap m_{fwd}, \text{ if } (state, v_{new}) \in m_j \\ m_j, \text{ else} \end{cases} \\
\}&
\end{aligned}
$$

[3]For the sake of brevity $m_i^{S^R}$ denotes the match of the rule with the index i from the general reverse state shell S^R.

The rule blocks for our example rule set are the following:

$$
\begin{aligned}
B_1 = \quad & \{r_{2\cdot1\cdot6+1=13} : m_1 \rightarrow drop\} \\
B_2 = \quad & \{r_{2\cdot2\cdot6+3=27} : m_3 \rightarrow accept, \\
& r_{28} : m_4 \rightarrow accept, \\
& r_{29} : m_5 \rightarrow accept, \\
& r_{30} : m_6 \rightarrow drop\}
\end{aligned}
$$

Finally, we can interweave the conditional reverse state shells with the rule blocks:

$$
R_S = \bigcup_{(i,k,l)\in I} B_i \cup \bigcup_{1\leq i\leq|S|} S_i^R
$$

As we already adapted all rules' indices, we simply need to collect all rules by union all blocks and conditional reverse state shells. During these steps we also remove the virtual `state` header field as state handling is projected onto the state shells and the *backwards* flag.

For our example the interwoven state shell looks like this[4]:

$$
\begin{aligned}
R_S = \{\quad & r_{13} : m_1 \backslash \{(\texttt{state}, v_{\texttt{state}})\} \rightarrow drop, \\
& r_{19} : (m_1^{S^R} \cap m_2) \backslash \{(\texttt{state}, v_{\texttt{state}})\} \rightarrow drop, \\
& r_{21} : (m_3^{S^R} \cap m_2) \backslash \{(\texttt{state}, v_{\texttt{state}})\} \rightarrow accept, \\
& \cdots \\
& r_{27} : m_3 \backslash \{(\texttt{state}, v_{\texttt{state}})\} \rightarrow accept, \\
& r_{28} : m_4 \backslash \{(\texttt{state}, v_{\texttt{state}})\} \rightarrow accept, \\
& \cdots \\
\}&
\end{aligned}
$$

The new rule set covers the stateful behaviour of `iptables` and only consists of header space fields that can be analyzed by a fast verification engine like *NetPlumber*.

Limitations: Our approach does not support connections that are handled as `RELATED` by `conntrack`, e.g., FTP or RTP data streams. Therefore, only their management or control channels can be analyzed, e.g., SIP for RTP. In general, complex extension modules as offered by `iptables` [40] are not within the scope of this work, e.g., arbitrary pattern matches, rate limiting, or eBPF programs.

Also, we assume that stateful traffic traverses the same interfaces of the firewall in both directions. While this assumption should hold for many use cases it leaves out load balancing scenarios that include firewall interfaces.

V. FaVe's Architecture

The fast verification system *FaVe* offers a framework to model networks containing complex network devices and verify packet flows against policy rules specified with FPL (see Section III). The tool aims at supporting continuous compliance verification by offering a large degree of automation for model creation, model aggregation, and fast reverification. FaVe is based on previous work where the concepts for modeling of stateless firewalls and IPv6 support were added to NetPlumber (see [5]). In this work, the system is extended by a compliance checking component and the support to model stateful firewalls (following the approach presented in Section IV).

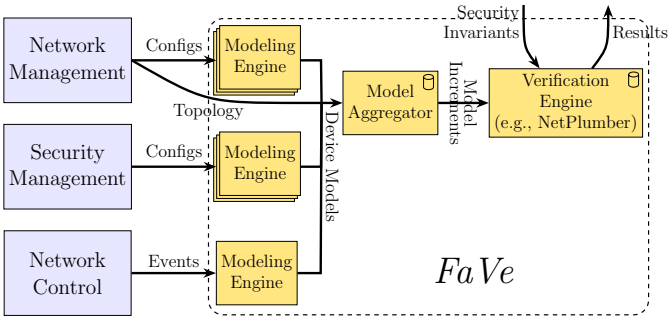

Fig. 4. Overview of FaVe's modeling pipeline.

The Model Aggregator

As depicted in Figure 4 FaVe consists of an incremental modeling pipeline, configuration modeling engines, and a verification backend. At first, the modeling engines parse network device configurations and instantiate predefined model templates. Currently, these templates include complex device classes like routers, switches, packet filters, and hosts. Then, these device models are aggregated and stored. Afterwards, the model is sent to the verification backend where it is verified against the security policy specifications. If the verification backend supports incremental updates upon a change of a device model, the aggregator calculates an increment and feeds it into the verification engine. In case of NetPlumber, this is the way how FaVe and NetPlumber interact.

Modeling devices and networks in FaVe

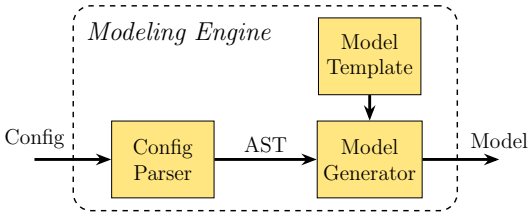

Fig. 5. Internals of a modeling engine.

For network administrators modeling network devices in FaVe is easy. Both, stateless and stateful capabilities are handled completely automatically by the modeling engines and the model aggregator. Figure 5 shows the components of a modeling engine. First, a device configuration is parsed and the resulting abstract syntax tree is used to instantiate a model template for that particular device type. Model templates consist of an inner and an outer part, e.g., as shown in Figure 6 for the packet filter model. The outer part offers ports to interconnect device models with respect to the network topology whereas the inner model consists of a branchable pipeline of consecutive tables. Each table holds a list of prioritized rules that operate on a *match-action*-semantics. During the verification process the incoming packet flows are distributed over the rule set based on the dependencies between the match parts. Then, these sub flows are processed based on the respective set of actions, e.g., forwarded, rewritten, or dropped.

Figure 6 shows how the example network from Figure 2 is modeled in FaVe. Each endpoint is represented by a host, i.e., the Internet, the Office, and the WebServer. The DMZ and Office networks each consist of a switch and the firewall is realized as a packet filter model. The internal pipeline of the packet filter comprises of six tables. At first, traffic is selected based on whether it is destined for the packet filter itself or to be forwarded. So, afterwards, packet flows are being processed by an input or forwarding filter table. They hold the respective rulesets specified in the firewall configuration as well as the interwoven state shell as explained in Section IV. For example, the `INPUT` and `FORWARD` chains from Linux' `iptables` configurations directly map onto the input and forward filter tables. Additionally, the model offers an output filter table for traffic originating from the firewall. This enables the verification of policy rules with roles that include the firewall. After filtering forwarded and outgoing traffic a routing table determines where to send flows. Finally, a post routing table emits traffic through the firewalls outer ports based on the routing decision. Also, this table ensures that egress traffic is not passed throught its ingress port which prevents loops.

Stateful Compliance Checks with FaVe

Finally, we want to check for compliance with FPL policies. For this purpose, it is necessary to simulate traffic propagation for *each FPL role* and to analyze the resulting reachability trees concerning their conformance with the FPL policy rules.

Each policy rule $A\ op\ B$ translates to a set of constraints that need to hold for all paths between A and B:

`--->`	At least one path from A to B must exist.
`<->>`	At least one path from A to B must exist. In addition, paths from B to A for backward traffic must exist which is indicated by the `backwards` flag set to 1. Also, no path with initialization traffic in backwards direction is allowed. This traffic is marked by the `backwards` flag set to 0.
`default`	There must not exist any path from A to B (default: deny).

If a service is consumed, i.e., $B.S$, or the operands hold attributes, additional packet constraints are applied for a more precise analysis.

Implementation of the Prototype

FaVe's prototype is implemented in Python and reuses the publicly available NetPlumber verification backend [41] which is implemented in C/C++.[5] We chose NetPlumber due to its natural modeling of networks and packet flows, its high pace, and its availability as open source code. Most recent approaches from literature, like APKeep [20], seem promising, too, and we are interested to compare them in the future.

[4]For more details on the particular header fields refer to Appendix A.

[5]We added several bug fixes and code enhancements to NetPlumber which will be made publicly available.

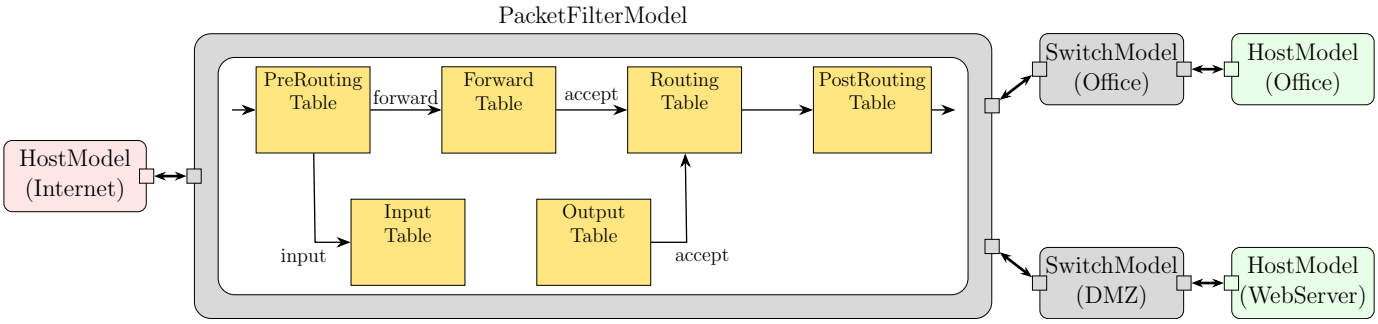

Fig. 6. Building blocks of the example network's model in FaVe with insights into the packet filter model.

The modeling engines are realized as small standalone tools which communicate over UNIX domain sockets with the Aggregator component and include models for routers with Cisco ACLs, switches, and stateful `ip6tables` packet filters. The Aggregator is implemented as a two-threaded daemon with a frontend thread that accepts device models from the modeling engines and a backend thread that aggregates the network model, calculates increments, and instruments NetPlumber.

NetPlumber offers building blocks to model rule tables with attached ports which can be interconnected by directed links. Additionally, flow generators to inject packets and probes to analyze traffic can be attached to ports as well. NetPlumber's tables automatically derive dependencies between rules and traffic is propagated accordingly.

As seen in Figure 6, we can use these modeling blocks to build more complex device models and networks, and to calculate reachability trees needed for policy verification.

VI. Evaluation

For the evaluation, we ran four benchmarks that show the approach's real-world usefulness and scalability, especially in comparison with state-of-the-art tools.[8] Table I provides an overview of the benchmarks' characteristics and Table II shows the specification of the machine and software which we used for our measurements.

First, the *UP* benchmark tests the applicability in IPv6 networks with stateful firewalls. It models a medium sized yet complex campus network which can be hardly inspected manually. Second, we show the scalability concerning large networks by running the well known *Internet2* and *Stanford* workloads [41]. Third, we compare FaVe against the available open source firewall verification tools `fffuu` [2] and `SymNet` [27]. For the comparison, we use the TUM workload from [39] which was already used by Diekmann et al. for the evaluation of `fffuu`. The TUM benchmark is a real world stateful packet filter rule set consisting of 3,795 rules.

[6]The original paper speaks of more than 757,000 forwarding and 1,500 ACL rules [4]. We reproduced their results which included a preprocessing step that compressed these rules down to 8,792 in about 35 seconds. Without compression the benchmark took 28,280 seconds. The scripted reproduction can be found here: https://github.com/cllorenz/hassel-reproduction.

[7]Analogously to the Stanford rules a preprocessing step compresses the original rule set of more than 126,000 rules down to 77,841.

[8]The benchmarks including policies and configurations will be shipped along with our prototype upon acceptance of this work.

A. Compliance Verification of a Campus Network

The *UP* benchmark is a synthetic representation of an university campus network with a large perimeter firewall and several subnets containing different hosts and services[9]. The firewall rule sets follow best practices and the network topology resembles real-world setups. This benchmark shows FaVe's ability to verify compliance for complex networks.

As shown in Table I, the UP benchmark consists of a central perimeter firewall and 23 switched subnets. These include a DMZ with 8 hosts, a WIFI domain, and 21 generic subnets with 6 hosts each. Each host comprises a small firewall for incoming and outgoing traffic. Together with the main firewall's ruleset of 1,035 rules, all firewall rules sum up to 3,396. This does not include the state shell yet as it is calculated within FaVe at runtime and therefore, it is not part of the configuration known to the administrator. The firewall rule sets consist of a large variety of header fields as it includes several rules that realize IPv6 specific requirements like for example filtering ICMPv6 traffic [42]. Policy checks sum up to 11,902 including 4,953 state checks.

In the experiment, we measure different phases of the verification process. First, we measure the network model *initialization* and the calculation of *reachabilities* with FaVe and NetPlumber respectively. Second, we evaluate the verification of the *compliance* checks as specified by the FPL policy using FaVe. Compliance checks go beyond NetPlumber's abilities and therefore, we can present only results for FaVe.

The initialization phase covers all steps FaVe conducts to build the model. During this phase, FaVe aggregates and transforms models, which includes the interweaving of the state shell, calculates increments, and instruments NetPlumber as its verification engine. After that, all steps to calculate reachabilities are performed, i.e., for each role a source node is connected to the network model and the propagation of packet flows is calculated.

To determine the runtime overhead of FaVe compared to NetPlumber we implemented a dumping function. Since NetPlumber is not capable to deal with complex network devices like firewalls, we use FaVe for the preprocessing step which results in an *aggregated model* and we dump the network in a form which is suitable for NetPlumber. For the measurements with NetPlumber, we load the aggregated

[9]The policy specification of the UP benchmark is given in Appendix B.

TABLE I
OVERVIEW OF THE BENCHMARKS.

Benchmark	UP	Stanford	Internet2	TUM
Network	1 firewall, 23 switches, 130 hosts, 1 dummy	16 routers	9 routers	1 firewall
Rules	3,396 (1,035 in main firewall)	8,792[6]	77,841[7]	3,795
Routing	**IPv6**	IPv4	IPv4	IPv4
Roles	71	16	9	-
Policy Checks	11,902 (w. 4,953 state checks)	256	81	-
Stateful Rules	**yes**	no	no	**yes**
Header Fields	`iif, oif, proto, sip6, dip6, sport, dport, limit, nxt_hdr, rtsegs, rttype, icmp6type, back`	`vlan, sip, dip, proto, dport, tcp flags`	`vlan, dip`	`iif, oif, vlan, sip, dip, proto, sport, dport, back`

TABLE II
SPECIFICATION OF THE MEASUREMENT ENVIRONMENT.

CPU	2x Intel Xeon E5-2650v4 à 12 Cores, 2.2 GHz				
Platform	x86_64	**Memory**	64 GB	**OS**	Debian Stretch, Linux v4.9
Software	Python v2.7, GCC v6.3.0, OpenJDK v1.8, Scala v2.11.7, GHC v8.0.1				

model directly into NetPlumber without involving FaVe. Also compliance checking is not supported by NetPlumber but performed by FaVe. Note that the actions done by NetPlumber are also performed in conjunction with FaVe and therefore, they are implicitly included in the runtime result of FaVe's measurements.

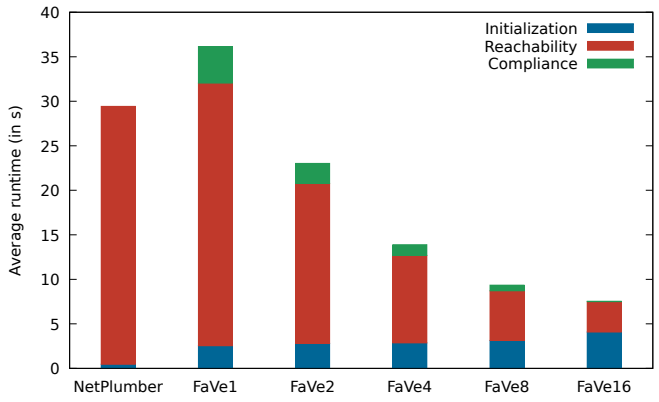

Fig. 7. Average runtimes after ten repetitions of the UP benchmark (in ms). The indices for FaVe indicate the number of backend instances as well as the number of threads used to check for compliance. For each experiment the coefficient of variation of the total run time is below 3 %.

As seen in Figure 7 the overall runtime (FaVe 1) is less than a minute (36.15 seconds) and stable with a low standard deviation of 1.18 seconds. For the medium sized UP network, FaVe's overhead including compliance checks is about 23% which allows a periodic reverification. The number of roles is important for the runtime behavior of the compliance checks since this results in a quadratic amount of conformity checks to be verified. Figure 7 shows that the runtime for 11,902 conformity checks (see the green part in the FaVe1 measurement) is about four seconds. Hence, FaVe performs well even for a large amount of roles.

In addition, we demonstrate performance gains that can be achieved by parallelization. For this purpose we introduce multiple NetPlumber backend instances that are initialized identically but calculate reachability trees for the different source nodes. The workload is distributed in a round robin manner with precalculated buckets. Also, we show that checking for compliance benefits from parallelization as each reachability tree can be checked independently. The measurements with multiple backend instances show that FaVe benefits from parallelization. Improvements for the reachability calculations range from 18 % to 66 % for each doubling of the number of instances. The overall gains sum up to a factor of 3.7 for FaVe and further support periodic reverifications.

B. Scaling to large Networks

Next, we show FaVe's scalability for large networks by measuring the well known real world *Stanford* and *Internet2* workloads [41]. As listed in Table I, these IPv4 routed networks consist of a set of routers with 8,792 resp. 77,841 rules. The original paper speaks of more than 757,000 forwarding and 1,500 ACL rules [4] for the Stanford workload. We reproduced their results which included a preprocessing step that compressed these rules down to 8,792 in about 35 seconds. Without compression the benchmark took 28,280 seconds. Analogously, the original Internet2 rule sets of more than 126,000 rules are compressed down to 77,841[10].

We augmented the benchmarks by specifying an FPL policy for the compliance verification. The policy checks pairwise reachability, i.e., A <--> B for all roles A and B. Therefore, we added an FPL role for each router to represent external adjacent networks connected to that router.

Table III shows that for the Stanford benchmark, FaVe's overall runtime is below 2.3 seconds which is still very fast. The low number of roles results in a low amount of reachabilities to be calculated and only few compliance rules to be checked. Therefore, the reachability phase is short for FaVe and NetPlumber alike. FaVe's overhead comes from the initialization which includes modeling and instrumentation of NetPlumber. A similar behaviour has been seen for the UP benchmark before.

For the even larger Internet2 benchmark, FaVe's runtime is less than 2 minutes. Since there are only few policy

[10]The scripted reproductions can be found here: https://github.com/cllorenz/hassel-reproduction.

TABLE III
MEAN RUNTIMES AFTER TEN REPETITIONS FOR THE STANFORD AND INTERNET2 BENCHMARKS (IN SECONDS). THE COEFFIENTS OF VARIATION ARE BELOW 3.3 % RESP. 1 %.

Tool	Init	Reach	Compl.	Total
Stanford				
FaVe	2.08	0.11	0.08	2.28
NetPlumber	0.50	0.11	-	0.61
Internet2				
FaVe	47.85	65.00	0.55	113.39
NetPlumber	31.86	56.44	-	88,30

checks necessary, the compliance phase only takes about half a second. Again, a major part of FaVe's overhead happens during initialization. The reason for the runtime difference for the reachability phase is less obvious. Profiling revealed a better caching behaviour for NetPlumber when loading an already aggregated and dumped model instead of an instrumentation through FaVe. We opted to keep FaVe's more natural way of modeling. E.g., FaVe mandates to define device models before connecting their ports while NetPlumber does not impose such restrictions.

We conclude that FaVe scales well to large networks in accordance to NetPlumber as its underlying fast verification engine while the overhead for compliance checking is insignificant.

C. Comparison with State-of-the-Art

Finally, we compare FaVe against the public available tools `fffuu` [2] and `SymNet` [27], since both tools are able to verify stateful packet filters (cf. Section II). For the evaluation of `fffuu`, Diekmann et al. used the so-called TUM benchmark. This is a real world firewall rule set from [39] with 3,795 IPv4 rules. Since `fffuu` only supports reachability analysis for pairs of fixed source and destination ports, we limit the generated traffic in FaVe and `SymNet` to the same pairs as well.

TABLE IV
TUM BENCHMARK RESULTS AFTER TEN REPETITIONS (IN SECONDS).

Tool	Mean	Median	StdDev.
FaVe	2.42	2.42	0.05
fffuu	100.48	100.49	0.08
SymNet	oom	oom	oom

Following the methodology of Diekmann et al., first, we instrumented `fffuu` to analyze the rule set concerning the reachability from TCP port 10000 to port 80.

The measurement includes `fffuu`'s rule set transformations and a single calculation of a service reachability matrix which serves a similar purpose as FaVe's reachability trees. FaVe is instrumented with a packet filter model and the same rule set. The measurements for FaVe include the initialization phase and the calculation of a reachability tree. As shown in Table IV, FaVe outperforms `fffuu` by a factor of more than 41 (2.42 s versus 100.48 s).

Second, we compare FaVe against `SymNet`. Since its public implementation could not load the workload directly due to missing features like VLAN handling, multiport parsing, and some header fields, we enhanced the code by implementing these features[11]. `SymNet` ran out of 64 GB of memory after about 15 minutes. Additionally, we conducted measurements with the original code and a stripped down version of the TUM rule set where we removed match fields that made our modifications necessary in the first place. Again, `SymNet` ran out of memory.

VII. CONCLUSION

This work shows that an automatic verification process of security policies stated in a high-level and semi-natural language is feasable and scalable.

The policy language *FPL* was introduced which describes compliance rules in terms of an abstract inventory that is understandable by non-technical and technical people alike. Its main achievement lays in the separation of technical details from the policy description which leads to small and easily reviewable policies. Policy management with FPL is fully compatible with standardized RBAC which enables a seamless integration into common security management workflows.

For the fast verfication of stateful packet filters like `iptables`, we introduced the *state shell interweaving* - a modeling technique that transforms the *stateful* behaviour into *stateless* rules which enables the re-use of fast data plane approaches. Therefore, the prototype implementation of FaVe leverages the HSA based *NetPlumber* engine as verification backend.

The evaluation results confirm that network security verification benefits from data plane analysis - even in the presence of state, IPv6, and several header fields. For the UP benchmark which represents a university campus with 3,396 firewall rules, FaVe's runtime is 36.15 seconds, including 11,902 policy conformity checks. Further, we augmented the well known Internet2 and Stanford benchmarks with an FPL policy for compliance checking. Also for these large networks, FaVe scales well in accordance to NetPlumber as its underlying fast verification engine while the overhead for compliance checking is insignificant. In comparison to approaches from literature FaVe achieves a 41-fold speedup when verifying a large stateful packet filter rule set.

In conclusion, FPL and FaVe offer a direct benefit for security officials to continuously verify the status of the security compliance - also for complex networks.

AVAILABILITY AND ACKNOWLEDGEMENTS

Upon acceptance of this work, we intend to publish our prototype's source code as well as our benchmarks. Last but not least, we thank Sven Hager for the fruitful discussions.

REFERENCES

[1] C. Lorenz and B. Schnor, "Policy Anomaly Detection for Distributed IPv6 Firewalls," in SECRYPT, 2015.
[2] C. Diekmann, J. Michaelis, M. P. L. Haslbeck, and G. Carle, "Verified IPtables Firewall Analysis," in IFIP Networking, 2016.

[11] Modifications can be found here: https://github.com/cllorenz/iptables-sefl

[3] Y. Yin, Y. Tateiwa, Y. Wang, G. Zhang, Y. Katayama, N. Takahashi, and C. Zhang, "An Analysis Method for IPv6 Firewall Policy," in IEEE HPCC, 2019.

[4] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in NSDI, 2013.

[5] C. Lorenz, S. Kiekheben, and B. Schnor, "FaVe: Modeling IPv6 firewalls for fast formal verification," in NetSys, 2017.

[6] E. S. Al-Shaer and H. H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," in INFOCOM, 2016.

[7] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis," in IEEE S&P, 2006.

[8] A. X. Liu, E. Torng, and C. R. Meiners, "Firewall Compressor: An Algorithm for Minimizing Firewall Policies," in INFOCOM, 2008.

[9] A. Jeffrey and T. Samak, "Model Checking Firewall Policy Configurations," in POLICY, 2009.

[10] N. Bjørner and K. Jayaraman, "Checking Cloud Contracts in Microsoft Azure," in LNCS ICDCIT, 2015.

[11] C. Bodei, L. Ceragioli, P. Degano, R. Focardi, L. Galletta, F. L. Luccio, M. Tempesta, and L. Veronese, "FWS: analyzing, maintaining and transcompiling firewalls," J. Comput. Secur., vol. 29, no. 1, pp. 77–134, 2021.

[12] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying Reachability in Networks with Mutable Datapaths," in NSDI, 2017.

[13] H. Yang and S. S. Lam, "Scalable Verification of Networks With Packet Transformers Using Atomic Predicates," IEEE/ACM Transactions on Networking, vol. 25, no. 5, pp. 2900–2915, 2017.

[14] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "NetSMC: A Custom Symbolic Model Checker for Stateful Network Verification," in NSDI, 2020.

[15] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness Verification of Stateful Network Functions," in NSDI, 2020.

[16] S.-J. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang, "Alembic: Automated Model Inference for Stateful Network Functions," in NSDI, 2019.

[17] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in NSDI, 2012.

[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in NSDI, 2013.

[19] H. Yang and S. S. Lam, "Real-Time Verification of Network Properties Using Atomic Predicates," IEEE/ACM Transactions on Networking, vol. 24, no. 2, pp. 887–900, 2016.

[20] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime Verification for Real Networks," in NSDI, 2020.

[21] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks," in NSDI, 2014.

[22] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma, "Validating Datacenters at Scale," in ACM SIGCOMM, 2019.

[23] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in NSDI, 2015.

[24] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time Network Verification Using Atoms," in NSDI, 2017.

[25] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang, "A Survey on Network Verification and Testing With Formal Methods: Approaches and Challenges," IEEE Communications Surveys & Tutorials, vol. 21, no. 1, pp. 940–969, 2019.

[26] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in ACM WREN, 2009.

[27] D. Dumitrescu, R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Dataplane equivalence and its applications," in NSDI, 2019.

[28] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in SIGCOMM, 2016.

[29] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A General Approach to Network Configuration Analysis," in NSDI, 2015.

[30] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A General Approach to Network Configuration Verification," in ACM SIGCOMM, 2017.

[31] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast Multilayer Network Verification," in NSDI, 2020.

[32] "Role Based Access Control," NIST, Tech. Rep., 2004.

[33] "Ansible," https://docs.ansible.com/, 2020.

[34] "Puppet," https://puppet.com/docs/, 2020.

[35] D. Ranathunga, M. Roughan, P. Tune, P. Kernick, and N. Falkner, "ForestFirewalls: Getting Firewall Configuration Right in Critical Networks," University of Adelaine, Tech. Rep., 2018.

[36] Cisco, "SD-Access Segmentation Design Guide," 2018.

[37] S. Bakke, "fwbuilder," https://github.com/fwbuilder/fwbuilder.git, 2020.

[38] Netfilter, "Manual page for iptables," http://ipset.netfilter.org/iptables.man.html, 2019.

[39] C. Diekmann, "net-network," https://github.com/diekmann/net-network, 2016.

[40] Netfilter, "Manual page for iptables-extensions," http://ipset.netfilter.org/iptables-extensions.man.html, 2019.

[41] P. Kazemian, "Hassel," https://bitbucket.org/peymank/hassel-public/wiki/Home, 2012.

[42] E. Davies and J. Mohacsi, "Recommendations for Filtering ICMPv6 Messages in Firewalls," https://tools.ietf.org/html/rfc4890, IETF, RFC 4890, 2007.

Claas Lorenz is pursuing his Ph.D. degree at the University of Potsdam, Germany, where he received his Master's degree in Computer Science in 2015. He works at genua GmbH - a German network security company. Currently, he is a senior technology analyst after working in several nationally funded research projects. His research interests include network security, security management, and network verification.

Vera Clemens Vera Clemens received her B.Sc. degree in Informatics/Computational Science from the University of Potsdam, Germany, in 2018. She is currently pursuing her M.Sc. degree in Computer Science at the Otto von Guericke University, Magdeburg, Germany, as well as working as a Research Assistant at the Institute of Computer Science at the University of Potsdam, Germany. Her research interests are in the area of computer networks and network security.

Max Schrötter Max Schrötter received his Master's degree in Computation Science from the University of Potsdam, Germany, in 2020. Currently, he is pursuing his Ph.D. degree at the University of Potsdam where he is also working as a research assistant. His research interests include network security, programmable data planes, high performance networks, and kernel security.

Bettina Schnor Bettina Schnor studied Mathematics and Computer Science at the TU Braunschweig, Germany, where she also received her Ph.D. degree in 1990. Since April 2000 she is head of the Department of Operating Systems and Distributed Systems at the University of Potsdam. Her research interests are distributed systems, cluster computing, and network security.

APPENDIX A
EXAMPLE OF THE STATE SHELL INTERWEAVING

The following rule set implements the policy from the introductory example in Section III (`eth0` represents the interface facing the Internet):

```
(0) ip6tables -P FORWARD DROP
    # sanity check against spoofing
    # (not derived from policy directly)
(1) ip6tables -A FORWARD --in-interface eth0 \
      -s 2001:db8::0/32 -j DROP
(2) ip6tables -A FORWARD -m conntrack --ctstate \
      ESTABLISHED -j ACCEPT
(3) ip6tables -A FORWARD --out-interface eth0 \
      -s 2001:db8::200/120 -j ACCEPT
(4) ip6tables -A FORWARD -d 2001:db8::101 -p tcp \
      --dport 80 -j ACCEPT
(5) ip6tables -A FORWARD -s 2001:db8::200/120 \
      -d 2001:db8::101 -p tcp --dport 22 -j ACCEPT
```

A quick analysis of the rule set shows that the minimal set of required header fields contains the inbound and outbound interfaces, the IPv6 source and destination addresses, the protocol field, source and destination ports, and `conntrack`'s state field. Additionally, we add the virtual *backwards* flag:

$$\mathcal{H} = \{h_1 = \texttt{iif},\ h_2 = \texttt{oif},\ h_3 = \texttt{sip},\ h_4 = \texttt{dip},\ h_5 = \texttt{proto},\ h_6 = \texttt{sport},\ h_7 = \texttt{dport},\ h_8 = \texttt{state},\ h_9 = \texttt{back}\}$$
$$\mathcal{V}_{\texttt{iif}} = \mathcal{V}_{\texttt{oif}} = \{\texttt{eth0}, \texttt{eth1}, \texttt{eth2}\}$$
$$\mathcal{V}_{\texttt{sip}} = \mathcal{V}_{\texttt{dip}} = \{\texttt{0::0/0}\}$$
$$\mathcal{V}_{\texttt{proto}} = \{0, ..., 255\}$$
$$\mathcal{V}_{\texttt{sport}} = \mathcal{V}_{\texttt{dport}} = \{0, ..., 65535\}$$
$$\mathcal{V}_{\texttt{state}} = \{\texttt{NEW}, \texttt{ESTABLISHED}\}$$
$$\mathcal{V}_{\texttt{back}} = \{0, 1\}$$
$$\mathcal{V}_{\mathcal{H}} = \{\mathcal{V}_{\texttt{iif}},\ \mathcal{V}_{\texttt{oif}},\ \mathcal{V}_{\texttt{sip}},\ \mathcal{V}_{\texttt{dip}},\ \mathcal{V}_{\texttt{proto}},\ \mathcal{V}_{\texttt{sport}},\ \mathcal{V}_{\texttt{dport}},\ \mathcal{V}_{\texttt{state}},\ \mathcal{V}_{\texttt{back}}\}$$

After parsing the rule set we obtain its formal representation R with $|R| = 6$. Note that the rule written first in the rule set is the default rule in `iptables` which applies if no other rule matches. Therefore, it needs to be put in the end of our rule list:

$$\begin{aligned}
R = \{r_1 :\ & \{(\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{drop}, \\
r_2 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
r_3 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}), \\
& (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
r_4 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
r_5 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), \\
& (\texttt{dip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
r_6 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{drop} \\
\}&
\end{aligned}$$

Next, we collect the state checking rules:

$$\begin{aligned}
S = \{r_2 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept} \\
\}&
\end{aligned}$$

1. General Reverse State Shell Derivation: First, we have to subtract the state checking rule (r_2) from the initial rule set. By reversing the directional fields and setting the *backwards* flag we obtain the general reverse state shell:

$$\begin{aligned}
S^R = \{r_1 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \{\texttt{eth0}\}), \\
& (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), (\texttt{dip}, \{\texttt{2001:db8::0/32}\}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{drop}, \\
r_3 :\ & \{(\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), \\
& (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
r_4 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::101}\}), (\texttt{dip}, \mathcal{V}_{\texttt{sip}}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
r_5 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \{22\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
r_6 :\ & \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{drop} \\
\}&
\end{aligned}$$

2. Conditional Reverse State Shell Calculations: Before filtering S^R for each state checking rule in S we calculate the block boundaries as intervals $I = \{(1, 0, 2), (2, 2, 7)\}$. Now, we can calculate the conditional reverse state shells by intersecting the general reverse state shell S^R with the state checking rule r_2 from S:

$$
\begin{aligned}
S_1^R = \{ & r_{(2\cdot1+1)\cdot6+1=19} : \{(\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \{\texttt{eth0}\}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{dip}, \{\texttt{2001:db8::0/32}\}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), \\
& (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{drop}, \\
& r_{(2\cdot1+1)\cdot6+3=21} : \{(\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), \\
& (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{(2\cdot1+1)\cdot6+4=22} : \{(\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::101}\}), (\texttt{dip}, \mathcal{V}_{\texttt{sip}}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
& (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{(2\cdot1+1)\cdot6+5=23} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \{22\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
& (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{(2\cdot1+1)\cdot6+6=24} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), \\
& (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), \\
& (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), (\texttt{state}, \{\texttt{ESTABLISHED}\}), \\
& (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{drop} \\
\}
\end{aligned}
$$

As the state checking rule accepts all known connections the original rules' actions are used for the conditional reverse state shell. The indices are projected between the first rule block (i.e., $2 \cdot 1 \cdot 6 + j = 12 + j$) and the second rule block (i.e., $2 \cdot 2 \cdot 6 + j = 24 + j$). Therefore, when interweaving the conditional reverse state shells they will replace the original state checking rule without disturbing the original filter semantics.

3. State Shell Interweaving: Before interweaving the conditional reverse state shells we need to calculate the blocks of state producing and stateless rules. The first interval $(1, 0, 2)$ consists of one rule and thus:

$$
\begin{aligned}
B_1 = \{ & r_{2\cdot1\cdot6+1=13} (\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{drop} \\
\}
\end{aligned}
$$

The second interval $(2, 2, 7)$ consists of four rules and thus:

$$
\begin{aligned}
B_2 = \{ & r_{2\cdot2\cdot6+3=27} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}), \\
& (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{2\cdot2\cdot6+4=28} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{2\cdot2\cdot6+5=29} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), \\
& (\texttt{dip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{2\cdot2\cdot6+6=30} \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{drop} \\
\}
\end{aligned}
$$

As there is no explicit `NEW` rule the *backwards* flag remains unset for all rules in these blocks.

Finally, by collecting all rules from the blocks B_i for each $(i, k, l) \in I$ as well as all rules from the conditional reverse state shells S_i^R with $1 \leq i \leq |S|$ and by removing the `state` field from each rule match we obtain the new rule set:

$$
\begin{aligned}
R_S = \{ & r_{13} : \{(\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{drop}, \\
& r_{19} : \{(\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \{\texttt{eth0}\}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{dip}, \{\texttt{2001:db8::0/32}\}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), \\
& (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{drop}, \\
& r_{21} : \{(\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{22} : \{(\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::101}\}), (\texttt{dip}, \mathcal{V}_{\texttt{sip}}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
& (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{23} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \{22\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
& (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{24} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{back}, \{1\}) \\
& \} \rightarrow \mathit{drop}, \\
& r_{27} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}), \\
& (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{28} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::101}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}), \\
& (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{29} : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::200/120}\}),
\end{aligned}
$$

$$
\begin{aligned}
&\quad (\texttt{dip}, \{\texttt{2001:db8::101}\}),\\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}),\\
&\quad (\texttt{back}, \mathcal{V}_{\texttt{back}})\\
&\} \rightarrow \mathit{accept},\\
r_{30} \quad &\{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&\quad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&\quad (\texttt{back}, \mathcal{V}_{\texttt{back}})\\
&\} \rightarrow \mathit{drop}\\
\}
\end{aligned}
$$

This rule set consists of simple rules that can be analyzed by fast verification engines.

APPENDIX B
POLICY SPECIFICATION OF THE UP BENCHMARK

In the UP policy file appear 71 different roles. Most of them represent sub-organizations, others stand for network typical components like a DMZ or a Perimeter Gateway Firewall (PGF). The corresponding `iptables` rule set consists of 1035 rules which is much harder to inspect manually.

```
describe policies(default: deny)
    All <--> DMZDNSServer

    DMZAdminConsole <->> PGF
    DMZAdminConsole <->> DMZ
    DMZ <--> DMZ

    Internet <--> DMZPublicServers
    Internet <->> SubnetsPublicServers
    SubnetClients <->> Internet
    SubnetClients <->> DMZ
    SubnetClients <->> SubnetsPublicServers

    Wifi <->> Internet
    Wifi <->> DMZ
    Wifi <->> SubnetsPublicServers
    Wifi <--> Wifi

    ApiClients <->> ApiServers
    AstaClients <->> AstaServers
    BotanClients <->> BotanServers
    ChemClients <->> ChemServers
    CsClients <->> CsServers
    GeolClients <->> GeolServers
    GeogClients <->> GeogServers
    HgpClients <->> HgpServers
    HpiClients <->> HpiServers
    InternClients <->> InternServers
    JuraClients <->> JuraServers
    LingClients <->> LingServers
    MathClients <->> MathServers
    MmzClients <->> MmzServers
    PhysikClients <->> PhysikServers
    PogsClients <->> PogsServers
    PsychClients <->> PsychServers
    SqClients <->> SqServers
    UbClients <->> UbServers
    WelcClients <->> WelcServers
end
```