Continuously verified Compliance: Achieving and Maintaining Network Security

Claas Bernhard Lorenz

27. Mai 2026
Version: Nachforderungen

Continuously verified Compliance: Achieving and Maintaining Network Security

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

angefertigt am
Institut für Informatik und Computational Science,

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

und
verteidigt am 30.04.2026

von
Claas Bernhard Lorenz, M.Sc.

Potsdam, den 27. Mai 2026

Claas Bernhard Lorenz

Continuously verified Compliance:
Achieving and Maintaining Network Security

Reviewers:

Prof. Dr. B. Schnor,
Prof. Dr. G. Carle, and
Prof. Dr. J. Posegga

Supervisors:

Prof. Dr. B. Schnor and
Prof. Dr. C. Kreitz

Universität Potsdam
Lehrstuhl für Betriebssysteme und Verteilte Systeme
Institut für Informatik und Computational Science
An der Bahn 2
14476 Potsdam
Deutschland/Germany

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate, Inhalte anderer Verfasser oder kollaborativen Arbeiten wurden ausdrücklich als solche gekennzeichnet.

Hereby, I declare that I authored this thesis on my own and did not use any resources other than those indicated in the thesis. All scientifically used texts, citations, other author's contents, or collaborative works have been marked as such explicitly. Further, I did not submit it in candidature for any degree to any other faculty or university.

Potsdam, 27. Mai 2026

Claas Bernhard Lorenz

Danksagung

Ich widme diese Arbeit meinen Töchtern Amalia und Aurelia sowie
meiner Frau Warinka.

Ich möchte meiner Frau und meiner Familie für den unfreiwilligen Verzicht auf viel gemeinsame Zeit danken. Auch musstet ihr meine geistige Abwesenheit trotz körperlicher Anwesenheit ertragen und ich hoffe, dass ich in Zukunft wohl in familienkompatibleren Gedanken versunken sein werde.

Liebe Bettina, dir möchte ich für die fachlich wie menschlich herausragende Betreuung, die jahrzehntlange Unterstützung und das Vertrauen danken. Du hast dich immer wieder für mein Thema begeistern können und mir oft den richtigen Impuls für Fortschritte und schließlich den Abschluss gegeben. Ohne dein Engagement wäre ich den Weg vielleicht nicht zu Ende gegangen.

Auch möchte ich meinem Arbeitgeber genua – und insbesondere meinen Vorgesetzten Alexander und Simon – danken für die Möglichkeit der nebenberuflichen Promotion sowie für das Vertrauen und die gezeigte Flexibilität. Darüber hinaus haben mich viele kompetente Kollegen begleitet. Lieber Andreas, lieber Sven und lieber Cuong, ich bedanke mich für die fachlich anregenden Diskussionen, die offenen Ohren und die vielen gemeinsamen – auch akademischen – Arbeiten. Auch meinen diversen Koautorinnen und Koautoren gebührt mein Dank. Gemeinsam und gegenseitig haben wir unsere Forschung besser gemacht und konnten so vor dem kritischen Blick der wissenschaftlichen Öffentlichkeit bestehen. Und schlussendlich auch vielen Dank an die vielen weiteren, ungenannten Kollegen und Wegbegleiter, die mich inspiriert und herausgefordert haben.

Ich freue mich, diesen Weg nun abzuschließen und ein neues Kapitel mit vielen spannenden Entdeckungen öffnen zu können.

Abstract[1]

In this work, we[2] address the complexity of large enterprise networks as one of the central challenges of IT security. Increasingly complex IT landscapes as well as historically evolved organizational structures, network infrastructures, and security configurations make it difficult to ensure security and regularly lead to security incidents with immense negative consequences for the affected organizations. In general, countermeasures can be taken at both organizational and technical levels and are regularly applied within the framework of organizational *Information Security Management* (ISM). Ensuring **compliance** between organizational security requirements and the network's configuration is a challenging task and lacks proper tooling support. This work aims to facilitate the reduction and control of complexity at organizational and technical levels. For this purpose, we support security officials and network administrators with practicable security process building blocks, workflows, and tooling based on highly performant and scalable formal methods.

On the organizational level, measures are often implemented within the framework of ISM using a *Plan-Do-Check-Act* (PDCA) cycle. This cycle approaches the desired state of security, i.e., *compliance*, by repeated planning, implementing, validating, and adjusting of security measures. We accompany this process with accessible, broadly automatible, and scalable formal tools. First, they support the migration of existing, complex network infrastructures towards compliance and, second, they ensure continuous compliance maintenance in PDCA cycles. For this purpose, we define three process phases: an *understanding* phase (UNDERSTAND), an *adaptation* phase (ADJUST), and a *control* phase (CONTROL). Each phase is based on the rapid repetition of two tool-supported workflows: compliance verification and the detection of firewall anomalies. In the UNDERSTAND phase, the status quo of network security is determined by verifying conformance with the desired state of network security and by detecting firewall anomalies. Any differences and detected anomalies are reported and provide the necessary insights for the further migration. Next, in the ADJUST phase, these incompliances and anomalies can be addressed and reverification is conducted. If there are still incompliances and firewall anomalies the process of adjustment and reverification is repeated until the network is compliant and comprises no firewall anomalies. Finally, in the CONTROL phase, the desired

[1] For the Abstract in German language, see below.

[2] Despite me being the sole author of this work, I was supported by many people whom I discussed my ideas with and who challenged my approaches. This group includes the co-authors of my publications as well as others. I would arrogate their support by writing this thesis as "I" and therefore, I use the pronoun "we". Further, this improves readability in comparison to a passive sentencing.

state of network security compliance is maintained troughout the PDCA cycle. This is achieved by generating a significant portion of the critical security configuration automatically and by verifying overall compliance continuously.

Our approach involves the specification of security rules and the modeling of network as well as security configurations by security officials and network administrators. Formal verification is then applied to check the model's compliance with the security rules. This approach raises three central research questions:

1. How to describe the desired state of network security?
2. How to model the network configuration?
3. How to achieve performance that enables continuity?

We address the first research question with our user-friendly **FaVe Policy Language** (FPL), which enables the specification of organizational roles and services, their hierarchical structuring, and security interactions. FPL allows network administrators to specify the desired state of security in a concise and auditable manner – even without extensive academic expertise.

We address the second research question with the conceptualization and prototypical implementation of **FaVe** – our fast verification framework. FaVe enables a simple modeling of complex networks using predefined device models, e.g., switches, routers, or stateful packet filters, and configuration parsers, e.g., Cisco IOS or IPTables/Netfilter. FaVe's modeling is based on our **Domain-oriented Header Space Algebra** (DHSA), which allows for human-readable models and can be embedded into the classical *Header Space Algebra* (HSA) using an injective homomorphism. FaVe provides an efficient implementation thereof and offers further support for the modeling and verification of dynamic protocols – particularly IPv6. FaVe transforms DHSA models to HSA and initiates their verification for compliance with FPL policies. For this purpose, we extended the public HSA implementation *NetPlumber* and leverage its high verification speed.

Our performance evaluation of FaVe answers the third research question. FaVe enables the modeling and verification of a complex university campus network in under 36 seconds. Also, it outperforms the state-of-the-art by a factor of 41 for a large, real-world firewall rule set. Despite its higher user-friendliness and simple, human-readable modeling, FaVe's pace and scalability are comparable to those of NetPlumber. FaVe adds only a reasonably low overhead, and its short total runtimes hence enable a frequent repetition of compliance verification. This way, migration processes can be executed quickly and effectively and the compliance verification

can be conducted continuously throughout PDCA cycles. Further, another increase in efficiency and a reduction of complexity can be achieved after reaching the desired state of security by generating the security configuration directly from the FPL policies. With the help of this automation, administrators can overcome manual configuration which is a major source of errors.

With the ability to detect **firewall anomalies**, FaVe offers further tooling for the practical reduction of complexity in firewall rule sets. We describe common firewall anomalies with DHSA and extended NetPlumber for their efficient detection. Our evaluation reveals the superiority of our domain oriented approach to network modeling in comparison with traditional, more generic approaches. FaVe analyzes a large, real-world IPTables/Netfilter rule set in under half a second and outperforms the state-of-the-art with a factor of 115. Further, FaVe scales to enormous rule sets of up to 15,000 rules.

With FPL, FaVe, and DHSA, this work provides a user-friendly and performant approach for verifying security compliance and detecting firewall anomalies. Our tools provide support for all phases of the migration to a PDCA cycle and its subsequent maintenance within the framework of an ISM. Therefore, we help security officials and network administrators to reduce complexity and keep security manageable in the long run.

Zusammenfassung

In dieser Arbeit adressieren wir[3] die schwer beherrschbare Komplexität großer Unternehmensnetze als eine der zentralen Herausforderungen der IT-Sicherheit. Immer komplexere IT-Landschaften sowie historisch gewachsene organisatorische Strukturen und Netzinfrastrukturen erschweren deren sicherheitliche Beherrschung und führen regelmäßig zu Sicherheitsvorfällen mit immensen negativen Folgen für die betroffenen Organisationen. Übliche Gegenmaßnahmen können auf organisatorischer sowie auf technischer Ebene ergriffen werden und kommen regelmäßig strukturiert im Rahmen eines organisationellen Informationssicherheitsmanagements (ISM) zur Anwendung. Die Sicherstellung der **Konformität** (Compliance) von organisatorischen Sicherheitsanforderungen mit der Konfiguration der Netzinfrastrukturen ist eine enorme Herausforderung und es fehlt bislang an geeigneter Toolunterstützung. Diese Arbeit zielt auf das Ermöglichen der Reduktion und Kontrolle von Komplexität auf der organisatorischen sowie technischen Ebene ab. Dafür unterstützen wir Sicherheitsverantwortliche und Netzadministratoren mit praktischen Sicherheitsprozessbausteinen, Vorgehensweisen und hochperformanten sowie skalierbaren technischen Werkzeugen basierend auf formalen Methoden bei der Beherrschung und Reduktion sicherheitlicher Komplexität.

Auf organisatorischer Ebene werden Maßnahmen im Rahmen des ISM oft durch einen *Plan-Do-Check-Act*-Zyklus (PDCA) realisiert. Mit diesem soll ein sicherheitlicher Sollzustand sichergestellt werden (Compliance), indem wiederholt Sicherheitsmaßnahmen geplant, implementiert, bezüglich ihrer Effektivität validiert und gegebenenfalls Sofortmaßnahmen durchgeführt werden. Mit unseren zugänglichen, weitgehend automatisierbaren und skalierbaren formalen Werkzeugen unterstützen wir bei der Migration bestehender, komplexer Netzinfrastrukturen hin zu einem dauerhaft Compliance-konformen PDCA-Zyklus. Wir bedienen uns dabei dreier Prozessphasen – einer *Erfassungsphase* (UNDERSTAND), einer *Anpassungsphase* (ADJUST) und einer *Beherrschungsphase* (CONTROL). Jede dieser Phasen basiert auf einer schnell wiederholten Anwendung zweier, werkzeuggestützter Vorgehensweisen: der Compliance-Verifikation sowie der Detektion von Firewall-Anomalien. In der UNDERSTAND-Phase wird der sicherheitliche Istzustand mit dem Sollzustand abgeglichen und es werden Abweichungen sowie Firewall-Anomalien aufgezeigt. Im

[3] Obwohl ich der einzige Autor dieser Arbeit bin, wurde ich durch viele weitere Menschen unterstützt, mit denen ich Ideen diskutiert habe und die meine Ansätze hinterfragt haben. Diese Gruppe umfasst die Ko-Autoren meiner Veröffentlichungen aber auch weitere Menschen. Ich würde ihre Unterstützung unterschlagen, wenn ich in dieser Arbeit in der Ich-Form schreiben würde und nutze daher die Wir-Form. Darüber hinaus empfinde ich diese Form als lesbarer als einen Schreibstil im Passiv.

Anschluss werden diese Abweichungen und Anomalien in der ADJUST-Phase behoben, das Netz auf Konformität mit dem Sollzustand überprüft und gegebenenfalls noch vorhandene Abweichungen und Anomalien dargestellt. Dieser Vorgang wird so lange wiederholt bis Compliance vorliegt. Schließlich wird in der CONTROL-Phase ein gewichtiger Teil der Sicherheitskonfiguration automatisiert generiert und das Netz im Rahmen des PDCA stetig auf Compliance überprüft, sodass der sicherheitliche Sollzustand stets gewährleistet bleibt.

Unser Ansatz sieht zunächst die Spezifikation von Sicherheitsregeln und die Modellierung von Netz- sowie Sicherheitskonfigurationen durch die Sicherheitsverantwortlichen und Netzadministratoren vor. Anschließend wird das Modell durch formale Verifikation auf Konformität mit den Sicherheitsregeln überprüft. Aus diesem Vorgehen ergeben sich drei zentrale Forschungsfragen:

1. Wie kann der sicherheitliche Sollzustand spezifiziert werden?

2. Wie können Netz- und Sicherheitskonfigurationen modelliert werden?

3. Wie kann eine Kontinuitäts-garantierende Performance erreicht werden?

Wir adressieren die erste Forschungsfrage mit unserer nutzerfreundlichen **FaVe Policy Language** (FPL), welche die Spezifikation organisatorischer Rollen und Dienste, deren hierarchischer Strukturierung und ihrer sicherheitlichen Interaktionen ermöglicht. Mit FPL kann der sicherheitliche Sollzustand durch oft nicht akademisch ausgebildete Netzadministratoren knapp und einfach auditierbar spezifiziert werden.

Die zweite Forschungsfrage adressieren wir durch die Konzeptionierung und prototypische Implementierung unseres schnellen Verifikationsframeworks **FaVe**. FaVe ermöglicht eine einfache Modellierung komplexer Netze mit Hilfe vordefinierter Gerätemodelle – wie beispielsweise Switche, Router oder zustandsbehaftete Paketfilter – und mit Konfigurationsparsern für beispielsweise Cisco IOS oder IPTables/Netfilter. Die Modellierung in FaVe basiert auf unserer *Domain-oriented Header Space Algebra* (DHSA), welche menschenverständliche Modelle ermöglicht und mittels injektivem Homomorphismus in die klassische *Header Space Algebra* (HSA) eingebettet werden kann. FaVe liefert hierfür eine effiziente Implementierung und bietet darüber hinaus Unterstützung für die Modellierung und Verifikation dynamischer Protokolle – insbesondere IPv6. FaVe überführt DHSA-Modelle nach HSA und initiiert die Konformitätsverifikation mit FPL-Policies. Für diesen Zweck haben wir die öffentlich verfügbare HSA-Implementierung *NetPlumber* erweitert und profitieren von dessen hoher Verifikationsgeschwindigkeit.

Die Performance-Evaluation von FaVe beantwortet schließlich die dritte Forschungsfrage. FaVe ermöglicht die Modellierung und Verifizierung eines komplexen Universitätsnetzes in unter 36 Sekunden. Außerdem übertrifft es andere bekannte Werkzeuge mit einem Faktor von 41 für einen großen, realen Firewallregelsatz. Trotz der hohen Nutzerfreundlichkeit und einfachen, menschenverständlichen Modellierung entsprechen die Geschwindigkeit und Skalierungsfähigkeiten von FaVe denen von NetPlumber. FaVe fügt lediglich einen überschaubaren Overhead hinzu und die kurzen Gesamtlaufzeiten ermöglichen folglich eine schnelle und häufige Wiederholung der Compliance-Verifikation. Dadurch können die Migrationsprozesse schnell und effektiv durchgeführt werden und die Compliance-Verifikation kontinuierlich im Rahmen des PDCA-Zyklus eingesetzt werden. Eine weitere Effizienzsteigerung und Komplexitätsreduktion kann darüber hinaus nach Erreichen des sicherheitlichen Sollzustands durch die Generierung von Sicherheitskonfigurationen aus der FPL-Spezifikation erreicht werden. Durch die Automatisierung wird die manuelle Konfiguration als häufige Fehlerquelle überwunden.

Mit der Fähigkeit, **Firewall-Anomalien** detektieren zu können, bietet FaVe ein weiteres Rüstzeug zur praktischen Reduktion der Komplexität von Firewall-Regelsätzen. Wir beschreiben gängige Firewall-Anomalien mit DHSA und implementieren ihre effiziente Erkennung in NetPlumber. Unsere Evaluation zeigt die Überlegenheit unseres domänenspezifischen Verifikationsansatzes. FaVe analysiert einen großen, realen IPTables/Netfilter-Regelsatz in unter einer halben Sekunde und übertrifft andere bekannte Werkzeuge um einen Faktor von 115. Außerdem wird die Skalierbarkeit von FaVe für enorm große Regelsätze mit bis zu 15.000 Regeln gezeigt.

Diese Arbeit liefert mit FPL, FaVe und DHSA einen nutzerfreundlichen und performanten Ansatz zur Verifikation sicherheitlicher Konformität sowie der Erkennung von Firewall-Anomalien. Unsere Werkzeuge bieten Unterstützung in allen Phasen der Migration hin zu einem PDCA-Zyklus und dessen Aufrechterhaltung im Rahmen eines ISM. Folglich helfen sie Sicherheitsverantwortlichen und Netzadministratoren, Komplexitäten zu reduzieren und Sicherheit beherrschbar zu halten.

Contents

1 Introduction

"Security's worst enemy is complexity." Ferguson and Schneier [43]

Originally, this famous quote targeted the standardization of the IPsec protocol suite but also bears some universal insight: the more complex a technical system becomes, the harder it gets to ensure its secure functioning. In practice, this insight has been confirmed several times – anecdotically [105, 71, 135] and empirically [64, 101, 87, 143, 85, 32]. In addition, IT landscapes have grown even more complex throughout the recent years. Trends like *Mobile Computing*, *Cloud Computing*, or the *Internet of Things* introduced new opportunities for many if not most organizations and their adoption ultimatively increased the complexity of these organizations' IT landscapes [130, 99]. This thesis aims at the practical reduction of complexity for security officials and administrators to effectively and efficiently handle security on the organizational as well as on the technical level.

1.1 Motivation

Growing complexity stems from several reasons – technical as well as organizational – and they manifest in the organizations' network configurations. In this thesis, we[1] focus on two sources of complexity: manual maintenance and organizational evolution. Manual maintenance of the network and security configurations occurs in organizations of any size and frequently amounts in complexity [139]. Over time, the configuration grows, people change, and, at some point, the manual controllability diminishes and, eventually, gets lost. The result is increasing anomalous behaviour and growing maintenance costs. Without proper tooling and automation, maintenance and transformation is infeasible. In this work, we provide formal tooling that enable administrators to overcome this challenge and, hence, reduce the configurations' complexity.

[1] Despite me being the sole author of this work, I was supported by many people whom I discussed my ideas with and who challenged my approaches. This group includes the co-authors of my publications as well as others. I would arrogate their support by writing this thesis as "I" and therefore, I use the pronoun "we". Further, this improves readability in comparison to a passive sentencing.

Organizational evolution, as a second source of complexity, roots in the business processes that form an organization's core functioning. Since the IT realizes business processes, it is a central or even existential asset for any organization. Securing IT, therefore, means securing the business processes as well as the processed data with respect to business and regulatory requirements, e.g., business continuity, risk management, or laws for data protection. For instance, government agencies [50], companies under espionage regulation [7], or critical infrastructures [51] are required by regulators to implement certain security measures[2]. Further, industry norms and process standards like the IEC 62443 [67] for industrial security or the ISO 27001 [68] drive the implementation of security measures. In this context, network segmentation through firewalls, encryption of data in transit, or intrusion detection are examples of common security measures.

Additionally, recurring changes to business processes and regulation also imply the need to frequently update security measures, i.e., IT security itself needs to be implemented as a business process. For example, introducing some new service, e.g., a CRM service, to be used in regular business processes requires changes to firewall configuration in order to enable access to the service over the network. In practice, the ISO 27001 helps to cope with the growing *complexity* on the *organizational* level by introducing an *Information Security Management* (ISM)[3]. Due to the fact that the ISO 27001 remains rather abstract concerning its implementation, other standards like the German BSI's *Grundschutz* [69], the US-american NIST's *Security and Privacy Controls* [131] or the ISF's *Standard of Good Practice for Information Security* [141] provide frameworks and more practical orientation.

The Grundschutz suggests a life cycle for the implementation of the ISM's security process that covers the planning of security measures, their implementation, assessment, and improvement. This PDCA cycle (*P*lan, *D*o, *C*heck, *A*ct, [98])[4] is shown in Figure 1.1 and comprises the following phases:

Plan In this phase, a security concept that provides processual or technical measures for the organizational requirements is created and its implementation is planned. Also, in case of major changes of the requirements, the concept needs to be updated and the reimplementation needs to be planned.

[2] Our examples mostly cover the current situation in Germany as of the writing of this thesis. Other countries have similar regulations, e.g., the FISMA regulation [42] for US government agencies.

[3] Note that the terms ISM and *Information Security Management System* (ISMS) are closely related and often used interchangeably. While the ISM refers to an organizational management practice, the ISMS is an organizational framework for the implementation of an ISM. An ISMS like the ISO 27001 or the BSI-Grundschutz typically covers processes, tools, and data. For the sake of brevity, we use ISM favorably and only use ISMS in order to highlight its framework character.

[4] The ISF defines the *Information Risk Management Business Cycle* with a similar purpose but with differently named phases: *D*efine, *I*mplement, *E*valuate, *E*nhance (DIEE) [141].

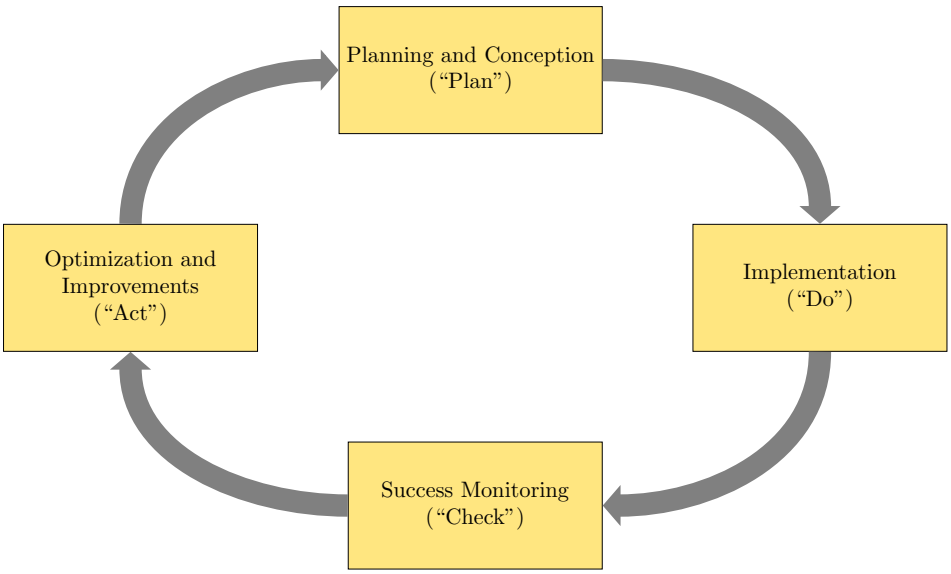


Fig. 1.1.: The PDCA process for managing security in an ISM (from [98, p. 18]).

Do All security measures need to be implemented according to the planned concept which is performed in this phase.

Check After implementation, the effectiveness and efficiency of the conducted measures need to be assessed and evaluated.

Act Given the assessment reveals minor or urgent incompliances, then immediate countermeasures need to be performed in this phase. If the discrepancies are effectively too extensive, a replanning in the next cycle of the PDCA is necessary.

Nevertheless, these standards' implications on the technical implementation are vague and there remains a semantical gap between the security measures on the organizational and on the technical level. I.e., continuously assuring *compliance* between organizational *security requirements* and the technical *security enforcement* remains a great challenge.

In this thesis, we provide means that support security officials and administrators to overcome this gap and which integrate seamlessly with the PDCA. For this purpose, we define *security compliance* as follows[5]:

> Security compliance is a state where organizational security requirements and their technical enforcement align.

[5]Since we focus on networked systems and, hence, network security, we use the terms *network security compliance* and *security compliance* interchangably for the sake of brevity.

Laws, norms, and risk management introduce compliance requirements on organizations, their business processes, and their IT systems. Additionally, the requirements change over time, and, therefore, also the IT systems and their security measures must evolve to comply with the requirements. These changes come on top of the constant progression of requirements which stems from business process evolution. Hence, the *security policy*'s[6] implementation on the technical level tends to become more and more complex. This fact becomes even more severe when compliance rules are governed by non-technically skilled personnel while security configuration is implemented by technical administrators. This – combined with growing complexity and manual implementation of the security process – leads to a situation where security compliance cannot be assessed manually.

To overcome this challenge, in this work we propose an approach that supports all phases of the PDCA. Our approach narrows the semantic gap, establishes an insight of the state of network security compliance, and helps to keep its maintenance in the presence of complex requirement evolution through a continuous application of formal methods.

1.2 General Approach

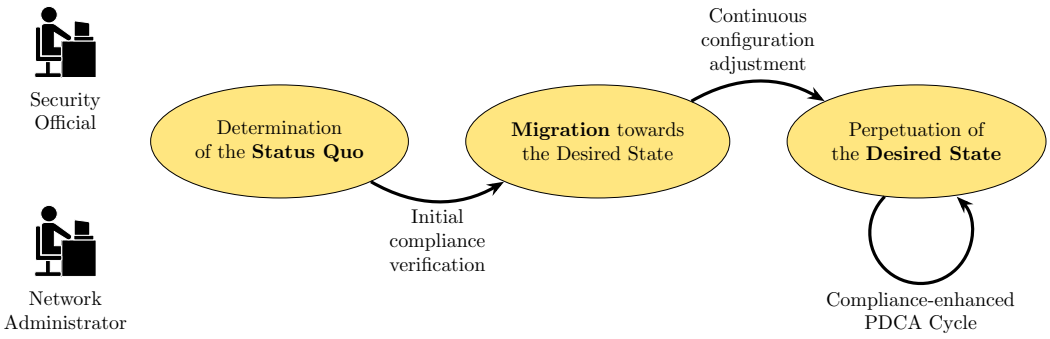


Fig. 1.2.: To fulfil *network security compliance* it is necessary for security officials to gain insights into the status quo, to develop the network towards a desired state, and, finally, maintain it in the presence of constant change.

For most organizations, the implementation of a PDCA process, e.g., as part of an ISM introduction in the course of an ISO 27001 certification, is a challenging venture. Since the ISM is implemented on top of an existing IT and IT security landscape, a

[6] We use the term *security policy* when we refer to a manifestation of the security requirements on the organizational level for a particular organization. We use the term *security specification* for a formally specified security policy.

migration strategy is needed to fulfil the ISM's requirements. All in all, it is required to:

1. first, determine the status quo of network security,
2. then, migrate towards the desired state as required by the ISM, and
3. finally, maintain the desired state wrt. the PDCA.

This process is depicted in Figure 1.2. Our approach supports the transition of existing IT security landscapes towards an ISM compliant PDCA process throughout all phases. First, the *status quo* is determined by an initial compliance verification that is enabled through the specification of the *desired state* of network security and an analysis of the current network configuration. Afterwards, the migration towards the desired state is supported by a continuous reverification of configuration changes. Finally, after achieving the desired state, the PDCA cycle is enhanced in two ways. On the one hand, compliant security configurations can be generated and enrolled upon changes of the security specification. On the other hand, overall compliance is reverified upon other changes of the network's configuration.

In particular, as depicted in Figure 1.3, we propose a management process with three phases that support the transition from the status quo to the desired state and the following perpetuation thereof – namely the phases UNDERSTAND, ADJUST, and CONTROL:

UNDERSTAND The first phase starts with the specification of the desired state of security. A formal yet human understandable *security policy*, i.e., the *security specification*, should be used for an *initial verification* of the network and security configurations which, at this point, are managed manually by the network administrator. Particularly, security policies may be specified in terms of organizational entities *reaching* other entities as well as networked assets like offered services.
In addition, *firewall anomalies* can be detected as well, e.g., a *shadowed* rule where all matching packets have been handled by higher prioritized rules. This analysis can be performed independently from the verification of compliance. This phase is essential to overcome complexity stemming from large and historically grown configurations as seen in many organizations.

ADJUST After understanding the status quo, the network configuration can be adjusted manually while security compliance is *reverified continuously*. This asserts a high confidence that the configuration changes actually provide progress towards the desired state of network security.

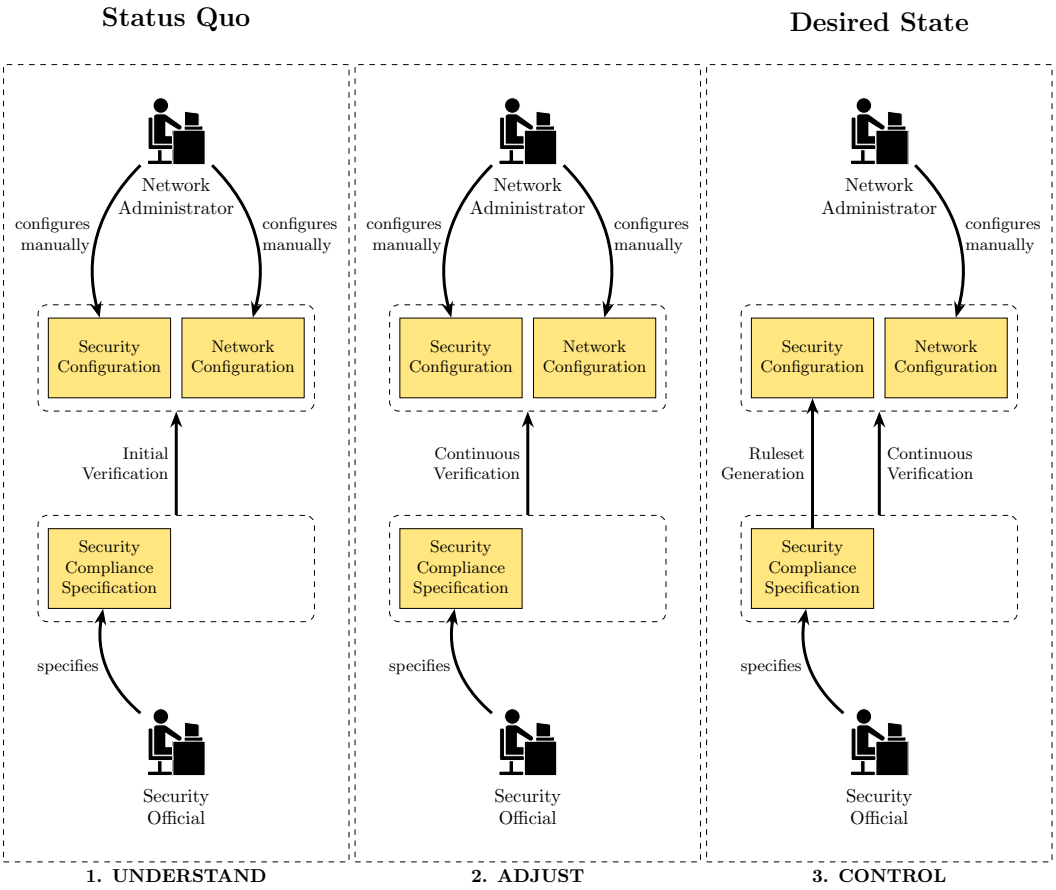


Fig. 1.3.: The *compliance management process* comprises of three phases – namely UNDERSTAND, ADJUST, and CONTROL – which enable security officials and network administrators to assess the status quo of network security, to transfer it to a desired state, and, finally, to maintain the latter.

This phase is integral for an effective migration of the organization's network and security configurations towards a compliant state where the PDCA can take over.

CONTROL Once the network and security configuration comply with the formally specified policies, the third phase enables *formally safeguarded maintenance* of both – policies and configuration. First, major parts of the security configuration – namely firewall rule sets – can now be *generated* from the security specification which relieves the network administrator from this often manual and error-prone task. Ultimatively, this removes a major source of complexity. And second, the continuous reverification of the overall configuration enables more confident changes to the network.

This phase seamlessly integrates with the PDCA cycle. First, the security specification may be changed in the *Plan* phase. The generated security configuration

can, then, be enrolled in the *Do* phase. In addition, in the *Check* phase, the overall network configuration, i.e., the generated security configuration and other manual configuration, can be verified for overall security compliance. Finally, the compliance of small configuration changes can be quickly verified in the *Act* phase.

In this work, we provide the necessary concepts, formal methodology, and tooling to realize this management process. We explore and enhance technologies for its implementation and evaluate their suitability thereof.

1.3 Security Management Workflows

To realize the three phases of the compliance management process, we define two security management workflows – namely the *Compliance Verification* (CV) workflow and the *Automatic Anomaly Detection* (AAD) workflow. Both workflows are applied repeatedly and help the security officials and administrators while progressing through the phases, i.e, the initial determination of the status quo, the migration towards the desired state, and its maintenance in the compliance-enhanced PDCA.

Compliance Verification Workflow

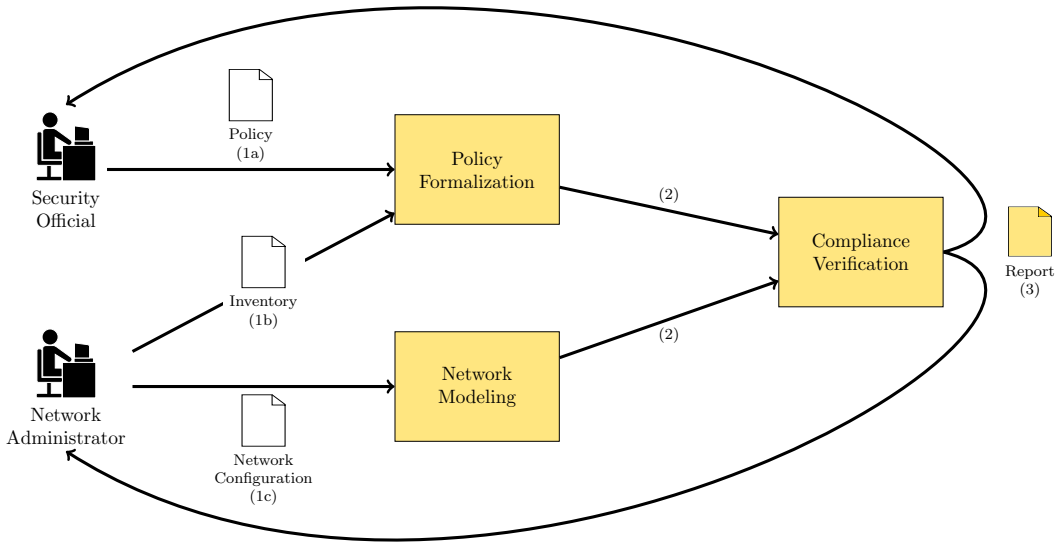


Fig. 1.4.: The Compliance Verification workflow.

As depicted in Figure 1.4, the CV 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). In this specification, access control is defined through *reachabilities* between organizational entities, e.g., services or network segments, that have been described in an *inventory*[7] (1b) which is provided by the network administrator. The inventory maps the organizational terms, e.g., the organization's ERP service or the research department's network segment, to their technical implementation, e.g., a segment's IP address range or a service's protocol and port. Also, the network administrator provides the network's configurations, e.g., firewall rule sets or routing configuration, that are to be modeled (1c). Then, the network model is verified for conformance with the security specification (2) and, finally, the result is reported (3) for further utilization by the security official and the network administrator.

The CV workflow is central for the implementation of the UNDERSTAND and ADJUST phases as it provides the security specification which describes the desired state. The CV workflow is executed once for the initial assessment of the status quo in UNDERSTAND and, then, repeatedly throughout ADJUST while the administrator changes the configuration towards the desired state. In CONTROL, the security specification is used to generate security configurations. Also, the CV workflow can be applied to continuously monitor the state of the whole network's security compliance. This is especially relevant in non-traditional network architectures, e.g., when using SDN/NFV, where bypasses of security enforcement points like firewalls could exist due to misconfiguration and the lack of physical separation. In these cases, the CV workflow offers broader insights beyond the analysis of single security entities since it takes the configuration of up to the whole network into consideration, i.e., possibly down to the level of switch configurations.

Automatic Anomaly Detection Workflow

The AAD workflow allows the detection of firewall anomalies stemming from complex dependencies between rules, e.g., rule *shadowing*, and is depicted in Figure 1.5. First, the administrator specifies the file with the rule set (1), for example written as IPTables rules. Then, the anomaly detection itself consists of two steps: The firewall rule set needs to be modeled formally and, then, it is analysed for anomalies (2). Both – modeling and analysis – can be automated. Finally, the administrator receives a report that shows all rules that are affected by an anomaly (3). This can be used to safely adjust the firewall rule set and, therefore, reduces its complexity.

[7] Definition: "A network inventory is a collection of data for network devices and their components managed by a specific management system." (cf. [152])

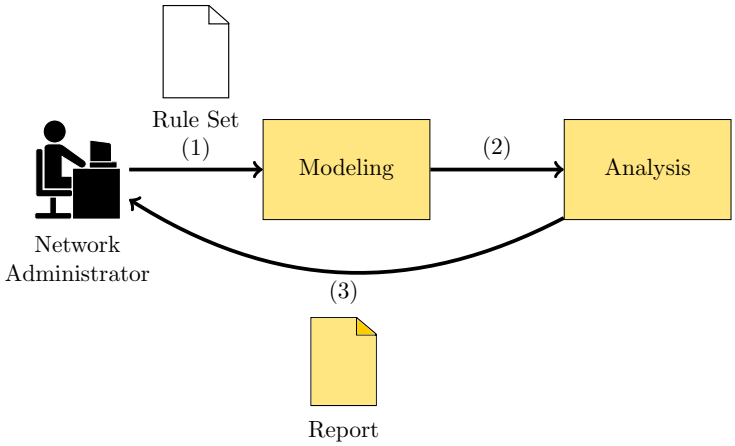


Fig. 1.5.: The Automatic Anomaly Detection workflow.

The AAD workflow is used primarily in the UNDERSTAND and ADJUST phases. First, in UNDERSTAND, finding all firewall anomalies is part of the determination of the *status quo* and allows insights into complex firewall configurations. Second, in ADJUST, the anomaly detection is repeated whenever the administrator adjusts firewall configurations. In this phase, the removal of anomalies reduces the configurations' complexity and continuous checking reveals any newly introduced anomalies and, hence, unnecessary increases in complexity. Therefore, the administrator gets direct feedback and can correct these issues in a very timely manner. In CONTROL, the generated security configuration excludes firewall anomalies by construction and, hence, checking for anomalies is not necessary anymore. Nevertheless, anomalies can also occur in other, complex network configurations like large routing tables. To support administrators in these regards, a slight adoption of the AAD workflow that incorporates other configuration than firewall rule sets would be necessary. Even though our prototype implementation is capable to check other kinds of configuration, e.g., routers or switches, this has not been evaluated explicitly.

1.4 Preliminary Study

In order to choose technology for the implementation of the compliance verification and anomaly detection, we conducted a preliminary study. Our prototype *ad6* followed an approach for firewall anomaly detection with traditional SAT-based model checking [73], i.e., the model and invariants are encoded in terms of a boolean satisfiability problem which is, then, solved by a SAT solver. The implementation

and evaluation was conducted in my Master's thesis [91] and the results have been published in [94].

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

Tab. 1.1.: Specification of the measurement environment.

We evaluated the SAT-based approach through two experiments using IPv6 workloads: *end-to-end reachability* and *firewall anomaly detection*. To achieve comparability of all measurements of the preliminary study as well as all other evaluations in this thesis, we used a common measurement environment as specified in Table 1.1. Also, we remeasured the firewall anomaly detection and achieved similar results.

The end-to-end experiment is a simplified version of the experiments for compliance verification in Chapter 7 and is conducted to set a baseline that helped us to decide upon the further continuation with the SAT-based approach, i.e., if this approach does not work well for a simplified scenario, we need to continue differently. The simplification lies in the policy being analysed which is limited to simple pairwise reachabilities, i.e., is there some packet emitted by some host A that reaches some host B? The evaluation in Chapter 7 covers a much more sophisticated policy, e.g., including stateful reachabilities.

The experiment is based on the UP campus network – a synthetic workload that represents a medium sized network with one central firewall, 23 switches, and 130 hosts. The overall amount of firewall rules is 3,396 with 1,035 rules in the central firewall. More details on the workload can be found in Section 7.2. We measured the pairwise reachabilities beween the hosts which results in $\frac{130 \cdot (130-1)}{2} = 8,385$ individual checks. The analysis took nearly three hours (∼170 min).

These results clearly show that the SAT-based approach is too slow for continuous verification – particularly, the end-to-end reachability analysis. Especially, throughout the ADJUST phase, it is necessary to reverify quickly to give administrators continuous feedback in order to enhance the *status quo* towards the *desired state*. Neither the anomaly detection nor the end-to-end reachability analysis yield the necessary performance.

This observation aligns with the results presented in literature. Approaches based on general model checking tend to be much slower than domain-specific algorithms and data structures. This is true for network analysis as well as for firewall anomaly

detection. More details on this matter are discussed in the related work section (Chapter 3).

To conclude, we need another approach that yields a much better performance to continuously verify the state of network security.

1.5 Research Questions

The prime challenges to realize workflows for security officials and network administrators to achieve network security compliance and to reduce overall complexity lie in the specification of security, the conceivability of the network's configuration, and fast repeatability of the workflows. Therefore, in this work, we deal with three research questions:

Q1 How to describe the desired state of network security?

Q2 How to model the network configuration?

Q3 How to achieve performance that enables continuity?

Previous approaches fell short in answering these questions as seen in the preliminary study in Section 1.4 and as discussed in the related work section (Chapter 3) – especially, due to their lack of performance.

1.6 Contributions

To overcome the performance limitations of general model checking approaches, to realize the phases of the compliance management process, and to answer the research questions, in this work, we make the following major contributions:

Specification We empower security officials and administrators to specify security policies in terms of an abstract language – namely the *FaVe Policy Language* (FPL). The FPL is highly usable as it operates on a level of abstraction suitable also for non-academically skilled personnel, e.g., many network administrators. FPL provides capabilities to concisely specify organizational structures in a hierarchical model using roles, groups of roles, and services. This way these entities are abstracted from their technical implementation, e.g., with VLANs, IP addresses, or ports. Policies are specified using these abstract terms which

makes them concise and easily auditable for organizational compliance. FPL is presented in Chapter 4.

Modeling and Verification In order to offer administrators an accessible and comprehensive way to model their networks, in Chapter 5, we introduce DHSA – a human-readable yet efficient formalism based on the notion of *Header Spaces*. In Chapter 6, we prototypically implemented DHSA in *FaVe* – our fast verification framework – which ships with a variety of predefined model templates for network devices and configurations including firewalls, routers, and switches. This way, modeling becomes an easy venture and further knowledge of DHSA is only needed when introducing new or altering existing device templates. Our implementation provides an efficient mapping of DHSA's data structures onto classical HSA [78] bitvector structures to benefit from the latter's solving performance. In addition, we published FaVe as open source.

Statefulness FaVe's fast compliance verification of FPL policies includes support for stateful packet filters. Statefulness poses a common verification challenge as it raises computational complexity. In Chapter 7, we present *State Shell Interweaving* – a modeling technique which efficiently projects stateful behaviour in DHSA models onto stateless capabilities. This allows networks that include stateful devices to be analysed efficiently. We evaluate our approach's effectiveness and performance using a variety of well known as well as custom benchmarks.

Anomaly Detection To enable the detection of firewall anomalies, we implemented a fast and scalable approach based on DHSA in FaVe as described in Chapter 8. We show its effectiveness and scalability with well known as well as custom benchmarks with rule sets consisting of up to 15,000 rules.

Dynamic Protocols In order to support state-of-the-art networking, we offer and evaluate IPv6 support including extension header chains. Their dynamic nature challenge the applicability of existing high speed verification approaches that operate on fixed size data models like HSA. The ability to dynamically extend the model at runtime is the key to successfully verify advanced security properties for complex IPv6 networks. We remain fully compatible with the still widely used IPv4 (cf. Chapter 5).

These contributions enable security officials and administrators to gain and manage compliant network security as previously described in the management process in Figure 1.3.

1.7 Publications

A major part of the above contributions has been peer-reviewed and published whereas [96] forms the main contribution of this thesis.

[92] Claas Lorenz, Sebastian Kiekheben, and Bettina Schnor. "FaVe: Modeling IPv6 Firewalls for Fast Formal Verification". In: *Proceedings of the International Conference on Networked Systems (NetSys)*. pp. 1–8. Göttingen, Germany, 2017.

We published FaVe's use cases, modeling approach, and architecture in this paper. In addition, we described, implemented, and evaluated the support of dynamic protocols, i.e., IPv6 with extension header chains. These topics are mainly covered in Chapter 6.

[96] Claas Lorenz, Vera Clemens, Max Schrötter, and Bettina Schnor. "Continuous Verification of Network Security Compliance". In: *IEEE Transactions on Network and Service Management (TNSM)*. Vol. 19.2 (2022), pp. 1729–1745. 2022.

In this article, we introduced and evaluated the compliance verification. This includes FPL, DHSA, and state shell interweaving which, in this thesis, are covered in Chapter 4, Chapter 5, and Chapter 7.

[93] Claas Lorenz and Bettina Schnor. "Firewall Management: Rapid Anomaly Detection". In: *IEEE 24th International Conference on High Performance Computing (HPCC)*, pp. 1465–1472. 2022.

We published our approach on firewall anomaly detection with FaVe in this paper. The concept, implementation, and evaluation is covered in Chapter 8.

Further (Co-)Authorships: I participated in further peer-reviewed publications which are out of the scope of this thesis: [49, 114, 40, 54, 45, 95, 44, 38, 142].

1.8 Scope

In this work, we investigate on the continuous improvement of network security. Yet, there are related issues that might be interesting but lay beyond the scope of this work. In the following, we discuss these matters.

Confidentiality, Authenticity, and Integrity Policies For our compliance verification, we focus on Network Access Policies in terms of host, network, and service reachabilities. I.e., who is legitimately able to communicate with whom and who is not? This does not cover other major security goals like confidentiality, integrity, and authenticity. E.g., is information disclosed only to legitimate users or entities? Has the information been changed by some unauthorized party?

In practice, these policies are implemented using cryptography on the application layer rather than the network layer. In applications, users and data can be identified precisely and policies can be applied in a fine grained manner. Typically, network layer measures like TLS[8] or IPsec work in a much broader granularity, i.e., on the level of networks, hosts, or services. Client and service certificates as offered by TLS are the most fine grained identifiers available. Hence, authenticity and confidentiality policies are coarse grained as well and always apply in an endpoint-to-endpoint manner. In essence, on the network layer compliant access policies form a strong basis for further analysis. I.e., it is easy to check access compliant relations for endpoint-to-endpoint policies concerning authenticity and confidentiality. This is beyond the scope of this work which focuses on network access policies.

NAT Network Address Translation (NAT) is an IP address space conservation mechanism, i.e., where few public addresses represent many private addresses. Also, it is popular to establish security-by-obscurity in networks, i.e., by raising the bar for an attacker to guess hosts and services correctly in order to access these. While our approach would be able to support NAT configurations conceptually, we decided against an implementation.

The reasons are two-fold. First, the security benefits of NAT pose rather a detail on the technical level and play no role in terms of compliance. Second, IPv6 offers a tremendous address space that obsoletes auxiliary functions like NAT in the foreseeable future. Consequently, NAT was not part of the IPv6 standard initially but has been standardized later as NPTv6 (*Network Prefix Translation*, [145]) as an additional option for operating IPv6 networks. Hence, we did not implement and evaluate support for NAT/NPTv6 in our prototype.

Instead, a good practice is the use of the *IPv6 Privacy Extensions* [53] where hosts frequently generate and use temporary addresses but the network's prefix remains

[8] TLS can be arguably regarded as an application layer protocol as well. Here, we refer to the portion that is visible from the network and, hence, can be inspected and filtered by networked security devices.

stable. As is, FPL provides the necessary capabilities to specify according inventories and policies using such network prefixes.

Other Dynamic Protocols In this work, we demonstrate our approach's feasibility to support dynamic protocols concerning IPv6 extension header chains. Conceptually, other dynamic protocols like MPLS could be supported but this is beyond the scope of this work.

Complex Matching Configurations Modern packet filter implementations like IPTables support arbitrarily complex filtering mechanisms like regular expressions on strings or even custom filtering functions. While the former would be hard to model precisely, modeling the latter could be infeasible due to the Halting Problem. However, these configurations are very exotic and barely never seen in practice. For example, there were no such configurations in a public set of 39 real-world rule sets [36]. Hence, these kinds of matching configurations are beyond the scope of this work.

1.9 Outline

This work is structured as follows. In Chapter 2, we start by providing background knowledge about practical security management, the concept of *Header Space Analysis* for formal network verification, and specifics of IPv6 relevant for verification before we provide related work for network verification in Chapter 3. Then, in Chapter 4, we enable the specification of the desired state of network security by introducing a formal language for high-level access policy specifications called the *FaVe Policy Language* (FPL). In Chapter 5 we proceed by introducing a formal definition of the *Domain-oriented Header Space Analysis* (DHSA), prove its algebraic properties as a distributive lattice, and show that our mapping of DHSA onto HSA is an injective homomorphism. Afterwards, in Chapter 6, we provide an architectural overview and an implementation of our verification framework *FaVe*. FaVe enables the modeling of networks and their verification in terms of FPL policies and the detection of firewall anomalies using DHSA. Also, we show how to model dynamic protocols with FaVe by the example of IPv6 extension header chains. Next, in the Chapters 7 and 8 we model devices and networks with FaVe and thouroghly evaluate compliance verification resp. firewall anomaly detection. Finally, we conclude this thesis and discuss possible further research directions in Chapter 9.

1.10 Conventions

In this thesis, we employ a series of conventions as follows:

- We display *new* terms using an italic font, e.g., *Security Compliance*.
- Abbreviations that accompany a new term are not highlighted, e.g., *Common Criteria* (CC).
- Further, we use an italic font to highlight particularly *important* terms.
- To indicate `code` or `configuration` fragments, we use a mono-spaced font.
- `Keywords` in code listings are highlighted with a bold font, e.g.:

```
def foo(x):
    if x is not None:
        bar(x)
```

- Further, explicit references to command line tools, e.g., `iptables`, or technical frameworks, e.g., `netfilter`, are highlighted with a mono-spaced font.
- Finally, literal citations are indicated through double quotation marks and are directly followed by a reference, e.g., “This publication provides a catalog of security and privacy controls [...]” (cf. [131, p. ii]).
- For reference, we provide a list of abbreviations on p. 285, important definitions on p. 291, and an index on p. 293.

2 Background

This chapter provides background knowledge as a basis for the remainder of this thesis. Particularly, in Section 2.1, we provide some definitions and explanations on how security management is practically implemented. In Section 2.2, we introduce the concept of *Header Spaces Analysis* which was proposed by Kazemian et al. [78] as a domain specific approach for formal network verification. Also, we detail their efficient implementation thereof in the well-known tool *NetPlumber* [79]. Finally, in Section 2.3, we explain and highlight specifics of the IPv6 protocol that pose special challenges for formal verification.

2.1 Practical Security Management

As stated in Section 1.1, practical realizations of an *Information Security Management* (ISM) require the statement of *security policies* which specify rules that govern the interactions of entities within the organization's IT system. Hence, security policies are a central cornerstone for the practical implementation of any ISM and should enable security officials to specify the desired state of network security and guide administrators throughout the technical implementation. Particularly, the concept of *access control* has been studied extensively and has been adopted in security management frameworks and systems.

There is no commonly agreed definition for the term *security policy* as different standards and frameworks provide differing definitions. For instance, the German BSI states in its glossary for the *Grundschutz* framework [70, p. 108]:

> "A security policy states the security goals and general security measures concerning the official goals of an enterprise or a government agency. More detailed security measures are covered by a more extensive security concept." [1]

[1] This definition has been translated from German. The original reads: "In einer Sicherheitsrichtlinie werden Schutzziele und allgemeine Sicherheitsmaßnahmen im Sinne offizieller Vorgaben eines Unternehmens oder einer Behörde fomuliert. Detaillierte Sicherheitsmaßnahmen sind in einem umfangreicheren Sicherheitskonzept enthalten."

Policies – An Etymological Approach

The term *policy* origins in the Old Greek word $\pi o \lambda \iota \varepsilon \acute{\iota} \alpha$ (politeia) which covered the meanings of the *citizenship of a (city) state*, the *(city) state's constitution*, and the *business of a polititian*. The Latin word *politia* narrowed the meaning towards the description of the *state* itself and its *government*. Later, in Middle French the word *policie* changed again and denoted the terms *rule* or *law*. Finally, in English the word *policy* transitioned to mean *practice determined by an authority*. [24]

This definition operates on the organizational level and remains rather vague since it requires specific (organizational) measures to be stated in a separate security concept.

Another example is provided by the *Common Criteria* (CC) – an internationally normed framework for security certifications of security products – for the term *Information Flow Control Policy* [26, p. 65]:

> "This Family[2] identifies the information flow control SFPs[3] (by name) and defines the scope of control for each named information flow control SFP. This scope of control is characterised by three sets: the subjects under control of the policy, the information under control of the policy, and operations which cause controlled information to flow to and from controlled subjects covered by the policy."

In addition, a *Security Function Policy* (SFP) is defined as [26, p. 18]:

> "The SFRs[4] may define multiple Security Function Policies (SFPs) to represent the rules that the TOE[5] must enforce. Each such SFP must specify its scope of control by defining the subjects, objects, resources or information, and operations to which it applies. All SFPs are implemented by the TSF[6], whose mechanisms enforce the rules defined in the SFRs and provide necessary capabilities."

The CC definitions remain vague and oriented at technical implementations (of security products).

To conclude, both definitions fall short to provide practical requirements for security policies in order to enable security officials and administrators to define the desired state of network security. Therefore and due to the fact that, in practice, policies are

[2] Families denote groups of criteria for CC auditors
[3] Security Function Policies
[4] Security Functional Requirements
[5] Target of Evaluation
[6] TOE Security Functionality

expressed prosaically but a formal specification is necessary for formal analysis, in this work, we use a more lightweight definition for the terms *security policy* resp. *network security policy*:

> Security policies are sets of rules that govern the interaction of entities on the organizational level. Network security policies specify the conditions of communication between subjects and objects.

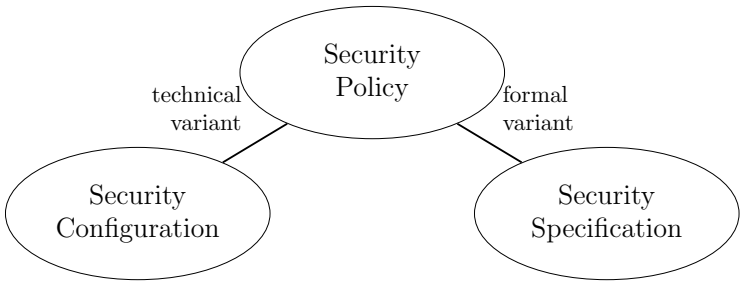


Fig. 2.1.: Distinction between the terms *security policy*, *security specification*, and *security configuration*.

I.e., security policies refer to a manifestation of the organizational security requirements for a particular organization. Also, network security policies are a subset of the security policies.[7] Our definition distincts the *security policy* from the *security specification* and the *security configuration*. As depicted in Figure 2.1, the security specification denotes a formally specified version of the security policy whereas the security configuration is a concrete implementation thereof. In this work, we enable security officials and network administrators to determine compliance between the security policy and the network's configuration by verifying the latter's conformance with the security specification.

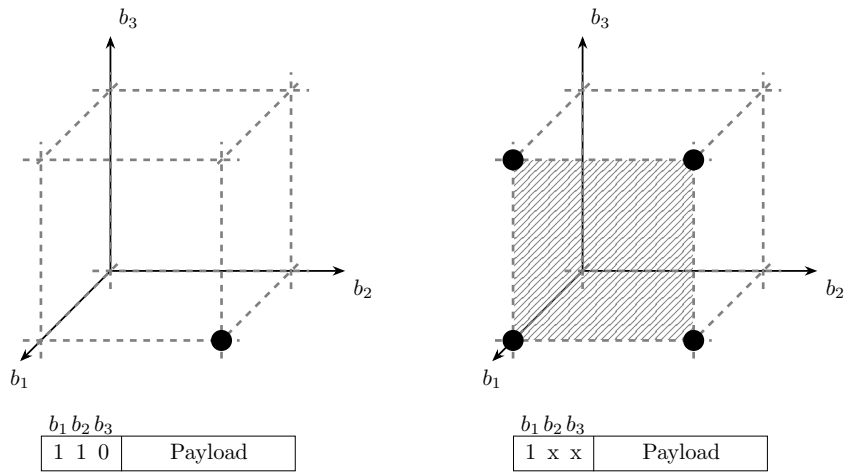
Access Control Frameworks In practice, security policies are often offered by and implemented in formal *access control frameworks*, e.g., *Discretionary Access Control* (DAC) in the UNIX file permissions [65] or *Role-based Access Control* (RBAC) in Microsoft's Active Directory [102] or in SELinux [116]. Furthermore, often also the generation of security configurations is offered by the frameworks. In Appendix A.1, we give an overview over common formal access control standards and frameworks, give a detailed description of NIST-standardized RBAC, and explain the compability of our policy specification language FPL with RBAC.

[7] Since this work focuses on network security, we use the terms interchangeably for the sake of brevity.

2.2 Network Verification with Header Space Analysis

In our preliminary study in Section 1.4, we have seen that general model checking techniques yield insufficient performance. This finding is supported by the related work and will be discussed in Chapter 3. On the other hand, approaches for verification that are tailored to the networking domain promise much better results. In this work, we build upon the the concept of *Header Space Analysis* (HSA) [78] whose open source implementation *NetPlumber* [79] by Kazemian et altera (2012) offers fast analysis. We will first introduce HSA's theoretical background – the *Header Space Algebra* (also HSA[8]) – before we detail the fast HSA implementation in the tool *NetPlumber*.

2.2.1 Header Space Algebra and Network Modeling



(a) Packet **(b)** Flow

Fig. 2.2.: Geometric representation of packets and flows in a Header Space with $l = 3$ (from [77, p. 6]).

The concept of *Header Space Analysis* (HSA) is based on two ideas: the representation of packets in terms of an *l-dimensional space* – the Header Space – and the modeling of networks as a repeated application of efficient *packet transfer functions*. The Header Space is formed over the combined bits of the header fields where every header bit is a dimension, i.e., if the set of header fields consists of l bits then the Header Space has l dimensions: $\mathcal{H} := \{0, 1\}^l$. Each bit can be set to 0 or 1 or it can

[8] Header Space Analysis and Header Space Algebra share the same acronym. Since this has already been the case in the original work by Kazemian et altera, we also follow this convention in this work. We mean *Header Space Analysis* when we elaborate on verification and we mean *Header Space Algebra* when we elaborate on the mathematical model.

be unset explicitly (denoted as *x*). For instance, the set of header fields consisting of the IPv6 destination address, the protocol field, and the destination port would require $128 + 8 + 16 = 152$ bits. Hence, the Header Space is 152-dimensional. In HSA, packets are modeled as points in the Header Space while sets of packets are represented as regions of the Header Space. Figure 2.2 depicts a geometric representation of a Header Space with $l = 3$ dimensions. A single packet with a header of `110` is a point (Figure 2.2a) in the Header Space whereas a flow of packets `1xx` is a region (Figure 2.2b).

HSA was implemented and open sourced by the original authors in the *Hassel* library which, in turn, is also used in the *NetPlumber* tool [76].

2.2.1.1. Packets and Sets of Packets

The basic building blocks are ternary bit expressions – named *Wildcard Expressions* – which can be combined to form *Header Space Objects*. In addition, several set operators are defined for wildcard expressions as well as header space objects, i.e., intersection, negation, union, and set difference. Finally, the equality and subset relations are defined. All in all, the Header Space along with the set operators *intersection*, *union*, and *complement* as well as the respective neutral elements the *all expression* and the *empty expression* form a *complete, algebraic, boolean lattice* – namely a *subset lattice* (cf. Section 2.2.1.4 for details). This can also be refered to as *Header Space Algebra* (abbreviated as HSA, too). In the following, we provide details about the wildcard expressions, header space objects, set operations, and set relations. Intuitively, wildcard expressions and their set operations represent an efficiently implementable subset of the Header Space while the more general header space objects enable the modeling of the whole Header Space.

Wildcard Expressions form the most basic building block to efficiently represent sets of packets. They have a length of l and consist of the ternary bits 0, 1, and x (don't care). In addition, bits can also become z as a result of set operations, e.g., intersection. If any z is present then the whole wildcard expression represents the empty set.

For example, the set of HTTP packets that are destined for the IPv6 address range `2001:db8::110/124` can be represented as the following wildcard expression:

$$\overbrace{\texttt{00100000,00000001,...,00000001,0001xxxx}}^{\text{IPv6 Destination: 2001:db8::110/124}},\ \overbrace{\texttt{00000110}}^{\text{Protocol: TCP}},\ \overbrace{\texttt{00000000,0101000}}^{\text{Destination Port: 80}}$$

In NetPlumber, wildcard expressions are implemented using two bits per ternary bit, i.e., 0 is encoded as 01, 1 as 10, x as 11, and z as 00. The whole expression is implemented as an array of unsigned integers which results in a dense and memory efficient encoding.

Header Space Objects are compounds built upon wildcard expressions. They are defined over a set of n wildcard expressions v_i. Formally:

$$\bigcup_{i=0}^{i<n} v_i$$

A single wildcard expression v can always be represented as a header space object with a single element: $\{v\}$. For example, let's assume that one wants to model all IPv4 packets except the address range `10.0.0.0/8`, i.e., the addresses `0.0.0.0 ... 9.255.255.255` and `11.0.0.0 ... 255.255.255.255`. The header space object can be constructed using an enumeration of IP prefixes that cover these address ranges[9]:

```
    0.0.0.0/0 − 10.0.0.0/8
= {
    7.0.0.0/5, 8.0.0.0/8, 9.0.0.0/8,
    11.0.0.0/9, 12.0.0.0/9, 13.0.0.0/9, 14.0.0.0/9, 15.0.0.0/9,
    16.0.0.0/4, 32.0.0.0/3, 64.0.0.0/2, 128.0.0.0/1
  }
:= {
    00000xxx,xxxxxxxx,xxxxxxxx,xxxxxxxx,
    00001000,xxxxxxxx,xxxxxxxx,xxxxxxxx,
    00001001,xxxxxxxx,xxxxxxxx,xxxxxxxx,
    00001011,0xxxxxxx,xxxxxxxx,xxxxxxxx,
    00001100,0xxxxxxx,xxxxxxxx,xxxxxxxx,
    00001101,0xxxxxxx,xxxxxxxx,xxxxxxxx,
    00001110,0xxxxxxx,xxxxxxxx,xxxxxxxx,
    00001111,0xxxxxxx,xxxxxxxx,xxxxxxxx,
    0001xxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx,
    001xxxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx,
    01xxxxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx,
    1xxxxxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx
  }
```

[9] An algorithm that calculates these IP prefix lists can be found here: `https://gist.github.com/stilez/1c863da87f9dc6bb3a73`

2.2.1.2. Set Operations

To form an algebra, a couple of set operations are defined over header space objects: *intersection*, *union*, *negation* resp. *complementation*, and *difference*. In the following, we introduce HSA's definitions of these operations and discuss their performance.

Intersection The intersection of two wildcard expressions is performed by bitwise application of the following table:

$\cap$	z	0	1	x
z	z	z	z	z
0	z	0	z	0
1	z	z	1	1
x	z	0	1	x

For instance, $\texttt{10xz} \cap \texttt{1111} = \texttt{1z1z}$. The implementation of wildcard expressions as arrays of unsigned integers and the two-bit encoding of the single bits allow the usage of bitwise AND operations to efficiently implement the intersection of wildcard expressions. Therefore, using this operation results in highly performant code.

Intersecting two header space objects A and B is performed by pairwise intersection of vectors from A and B and subsequent unioning of the non-empty results, i.e.,

$$A \cap B := \{a \cap b \mid a \in A, b \in B, a \cap b \neq \emptyset\}.$$

Union The union of two wildcard expressions typically yields a header space object due to the fact that it is not always possible to represent all packets from both expressions in one. For instance, $\texttt{11xx} \cup \texttt{00xx} = \{\texttt{1100}, \texttt{1101}, \texttt{1110}, \texttt{1111}\} \cup \{\texttt{0000}, \texttt{0001}, \texttt{0010}, \texttt{0011}\}$ cannot be represented using a single expression. Contrarily, two wildcard expressions must be *mergeable* in order to be combined. Mergeability is given iff the expressions differ in only one bit, e.g., $\texttt{10xx} \cup \texttt{11xx} = \texttt{1xxx}$.

So, in general, the union operation works with header space objects and since header space objects are just sets of wildcard expressions, the normal set union applies: $A \cup B$.

Hassel and NetPlumber implement header space objects as lists of wildcard expressions and their union by simply concatenating both lists without further checks for duplicates or mergeability. This improves the operation's runtime at the expense of a possibly higher memory consumption of the resulting header space object. We

extended NetPlumber to optionally improve the memory consumption of a header space object by eliminating duplicates and merge wildcard expressions if possible.

Negation/Complementation The negation of wildcard expressions typically results in a header space object and only special cases yield single wildcard expressions. The idea is to construct the set of packets that does not contain the packets represented by the expression. This is achieved by enumerating negation expressions which are are constructed by

1. fixing a single but set bit (i.e., 0 or 1)
2. negating that bit, and
3. set everything else to x.

This is done for each set bit and the resulting set of expressions represents the negation set. For example, the negation of `101x` is:

$$\overline{\texttt{101x}} = \{\texttt{0xxx}, \texttt{x1xx}, \texttt{xx0x}\}$$

The negation of a wildcard expression only yields a single wildcard expression if there is only one bit set, e.g., negating the expression `xx1x` results in `xx0x`, formally $\overline{\texttt{xx1x}} = \{\texttt{xx0x}\}$.

Our previous example where we wanted to represent all packets but `10.0.0.0/8` can be expressed using negation as follows:

$$\begin{aligned}
& \texttt{0.0.0.0/0} - \texttt{10.0.0.0/8} \\
= \quad & \overline{\texttt{10.0.0.0/8}} \\
:= \quad & \overline{\texttt{00001010,xxxxxxxx,xxxxxxxx,xxxxxxxx}} \\
= \{ \quad & \\
& \texttt{1xxxxxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx}, \\
& \texttt{x1xxxxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx}, \\
& \texttt{xx1xxxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx}, \\
& \texttt{xxx1xxxx,xxxxxxxx,xxxxxxxx,xxxxxxxx}, \\
& \texttt{xxxx0xxx,xxxxxxxx,xxxxxxxx,xxxxxxxx}, \\
& \texttt{xxxxx1xx,xxxxxxxx,xxxxxxxx,xxxxxxxx}, \\
& \texttt{xxxxxx0x,xxxxxxxx,xxxxxxxx,xxxxxxxx}, \\
& \texttt{xxxxxxx1,xxxxxxxx,xxxxxxxx,xxxxxxxx} \\
\} &
\end{aligned}$$

The negation of a header space object A is defined as

$$\overline{A} := \bigcap \{\overline{a} \mid a \in A\}$$

Intuitively, each packet set is complemented individually and the common set of packets is determined. For example, $\overline{A} := \overline{\{\texttt{1xxx}, \texttt{00xx}\}}$ represents all packets not starting with 1 and not starting with 00:

$$\begin{aligned}
\overline{A} = & \ \overline{\{\texttt{1xxx}, \texttt{00xx}\}} \\
= & \ \bigcap \{\overline{\texttt{1xxx}}, \overline{\texttt{00xx}}\} \\
= & \ \bigcap \{\{\texttt{0xxx}\}, \{\texttt{1xxx}, \texttt{x1xx}\}\} \\
= & \ \{\texttt{01xx}\}
\end{aligned}$$

Difference The difference of two header space objects A and B is defined as

$$A - B :\Leftrightarrow A \cap \overline{B}.$$

Intuitively, the resulting packet set consists of all packets from A that are not also packets in B.

2.2.1.3. Set Relations

In addition to the set operations, the *subset* and *equality* relations between header space objects are defined in HSA.

Subsets Checking the subset relation between two header space objects A and B is defined as

$$A \subseteq B :\Leftrightarrow A - B = \emptyset.$$

Equality Checking equality between two header space objects A and B is defined as

$$A = B :\Leftrightarrow A \subseteq B \wedge B \subseteq A.$$

2.2.1.4. Header Space Algebra

The Header Space along with the operators *intersection*, *union* and *complementation* as well as the neutral elements the *all expression* and the *empty expression* form a *complete boolean lattice* which is explained in the following.

But first, what is a complete boolean lattice? It is an *algebraic structure* with certain, well-defined characteristics [34, pp. 177ff]. Algebraic structures, in general, consist of a carrier set and a family of basic operations with inputs and outputs solely from and to the carrier set. Formally, an algebra $\mathcal{A}$ is defined as:

$$\mathcal{A} := (A, (f_i)_{i \in I})$$

where

A is the carrier set,

$(f_i)_{i \in I}$ is a family of indexed functions i.e., $I \subset \mathbb{N}_0$, and

f_i is an n_i-ary function that is closed under A, i.e., $f_i : A^{n_i} \to A$.

Depending on the characteristics offered by an algebraic structure, it is categorized differently, e.g., as group, ring, or lattice. For instance, the characteristics of a lattice comprise of the commutativity laws, the associativity laws, the distributivity laws, the complementary laws, idempotency, and absortion.

For example, the *boolean lattice* offers these characteristics and can be defined as:

$$\mathcal{B} := (B := \{0, 1\}; \wedge, \vee, \neg, 0, 1).$$

In particular, the boolean lattice fulfils the above characteristics as follows:

$$\begin{array}{lrcll}
\forall x, y \in B: & x \wedge y & = & y \wedge x & \text{(commutativity)} \\
\forall x, y \in B: & x \vee y & = & y \vee x & \\
\forall x, y, z \in B: & x \wedge (y \wedge z) & = & (x \wedge y) \wedge z & \text{(associativity)} \\
\forall x, y, z \in B: & x \vee (y \vee z) & = & (x \vee y) \vee z & \\
\forall x, y, z \in B: & x \wedge (y \vee z) & = & (x \wedge y) \vee (x \wedge z) & \text{(distributivity)} \\
\forall x, y, z \in B: & x \vee (y \wedge z) & = & (x \vee y) \wedge (x \vee z) & \\
\forall x \in B: & x \wedge x & = & x & \text{(idempotency)} \\
\forall x \in B: & x \vee x & = & x & \\
\forall x, y \in B: & x \wedge (x \vee y) & = & x & \text{(absortion)} \\
\forall x, y \in B: & x \vee (x \wedge y) & = & x & \\
\forall x \in B: & x \wedge \neg x & = & 0 & \text{(complementation)} \\
\forall x \in B: & x \vee \neg x & = & 1 &
\end{array}$$

In addition, the conjunction and disjunction operators have 1 resp. 0 as neutral elements: $\forall x \in B : x \wedge 1 = x$ and $\forall x \in B : x \vee 0 = x$.

The Header Space as carrier set and the intersection, union, and complementation operators along with the all-set expression as well as the empty expression form an algebraic structure, too. Namely, it is a *subset lattice*

$$\mathcal{HSA} := (2^{\mathcal{H}}; \cap, \cup, ^{-}, H_\Omega, \emptyset)$$

where $H_\Omega := \{\texttt{xx}\ldots\texttt{xx}\}$. Similar to the boolean lattice, its characteristics are the following:

$\forall x, y \in 2^{\mathcal{H}}:$	$x \cap y$	$=$	$y \cap x$	(commutativity)
$\forall x, y \in 2^{\mathcal{H}}:$	$x \cup y$	$=$	$y \cup x$	
$\forall x, y, z \in 2^{\mathcal{H}}:$	$x \cap (y \cap z)$	$=$	$(x \cap y) \cap z$	(associativity)
$\forall x, y, z \in 2^{\mathcal{H}}:$	$x \cup (y \cup z)$	$=$	$(x \cup y) \cup z$	
$\forall x, y, z \in 2^{\mathcal{H}}:$	$x \cap (y \cup z)$	$=$	$(x \cap y) \cup (x \cap z)$	(distributivity)
$\forall x, y, z \in 2^{\mathcal{H}}:$	$x \cup (y \cap z)$	$=$	$(x \cup y) \cap (x \cup z)$	
$\forall x \in 2^{\mathcal{H}}:$	$x \cap x$	$=$	x	(idempotency)
$\forall x \in 2^{\mathcal{H}}:$	$x \cup x$	$=$	x	
$\forall x, y \in 2^{\mathcal{H}}:$	$x \cap (x \cup y)$	$=$	x	(absortion)
$\forall x, y \in 2^{\mathcal{H}}:$	$x \cup (x \cap y)$	$=$	x	
$\forall x \in 2^{\mathcal{H}}:$	$x \cap \overline{x}$	$=$	$\emptyset$	(complementation)
$\forall x \in 2^{\mathcal{H}}:$	$x \cup \overline{x}$	$=$	H_Ω	

Additionally, H_Ω and $\emptyset$ serve as neutral elements for the intersection resp. union operations: $\forall x \in 2^{\mathcal{H}} : x \cap H_\Omega = x$ resp. $\forall x \in 2^{\mathcal{H}} : x \cup \emptyset = x$.

Hence, HSA is a boolean algebra and the name *Header Space Algebra* is justified.

2.2.1.5. Transfer Functions

With HSA, networks are modeled in terms of *packet transfer functions* that (optionally) transform and forward packets in the so called *Network Space* $\mathcal{N}$. The Network Space consists of the Header Space $\mathcal{H}$ and the set of network ports $\mathcal{P}$ which represent the interfaces of network devices. Formally, the Network Space is defined as $\mathcal{N} := \mathcal{H} \times \mathcal{P}$ where $\mathcal{P} := \{1, \ldots, P\}$ is a set of unique port identifiers. Transfer functions take some point in the Network Space as input and yield, in turn, some regions of the Network Space. Intuitively, a switch forwards sets of packets arriving at some ingress port over various egress ports depending on the packets' headers. In combination, the network functions represent packet processing and their repeated application simulates the network's behaviour.

For this purpose, the author's define three kinds of transfer functions: the *switch* transfer functions $T : \mathcal{N} \to 2^{\mathcal{N}}$ which models packet processing by network devices, the *network* transfer function $\Psi : \mathcal{N} \to 2^{\mathcal{N}}$ that maps ports to devices and applies the respective switch transfer function, and the *topology* transfer function $\Gamma : \mathcal{N} \to 2^{\mathcal{N}}$ which connects ports and, hence, forwards packets on links.

For more details on the transfer functions and the verification with HSA, see Appendix A.2.1 resp. Appendix A.2.2.

2.2.2 Optimized HSA with NetPlumber

The tool NetPlumber by Kazemian et altera (2013) implements a highly optimized version of HSA. It offers simple yet effective modeling primitives and optimizations that enable fast (re-)verifications of network properties like reachability and loop detection. NetPlumber offers an incremental mode, i.e., after an initial verification, changes to the network can be modeled and reverified in short-circuit loops.

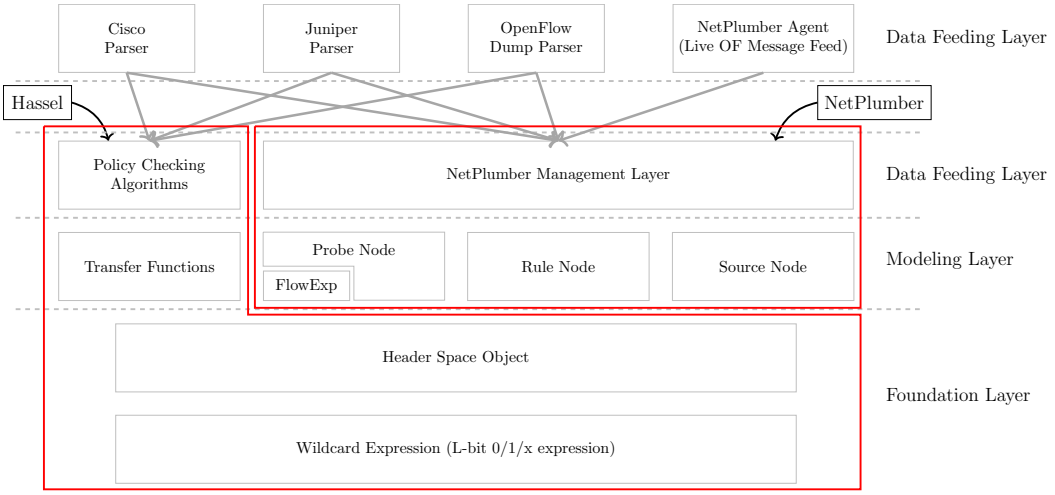


Fig. 2.3.: Block diagram of NetPlumber's architecture and its reuse of HSA components from the *Hassel* library (from [79]).

NetPlumber's implementation reuses components that have been realized for HSA in the *Hassel* library. Figure 2.3 gives an overview of NetPlumber's components and Hassel. Primarily, NetPlumber replaces HSA's transfer functions with a composition of modeling primitives that – in combination – enable a faster and more customizable analysis. For further notice, we provide more details on the similarities and differences between NetPlumber and HSA in Appendix A.2.3.

NetPlumber's approach is based on three layers of logically interconnected graphs: the *Network Model*, the *Plumbing Graph*, and the *Flow Graph*. The Network Model

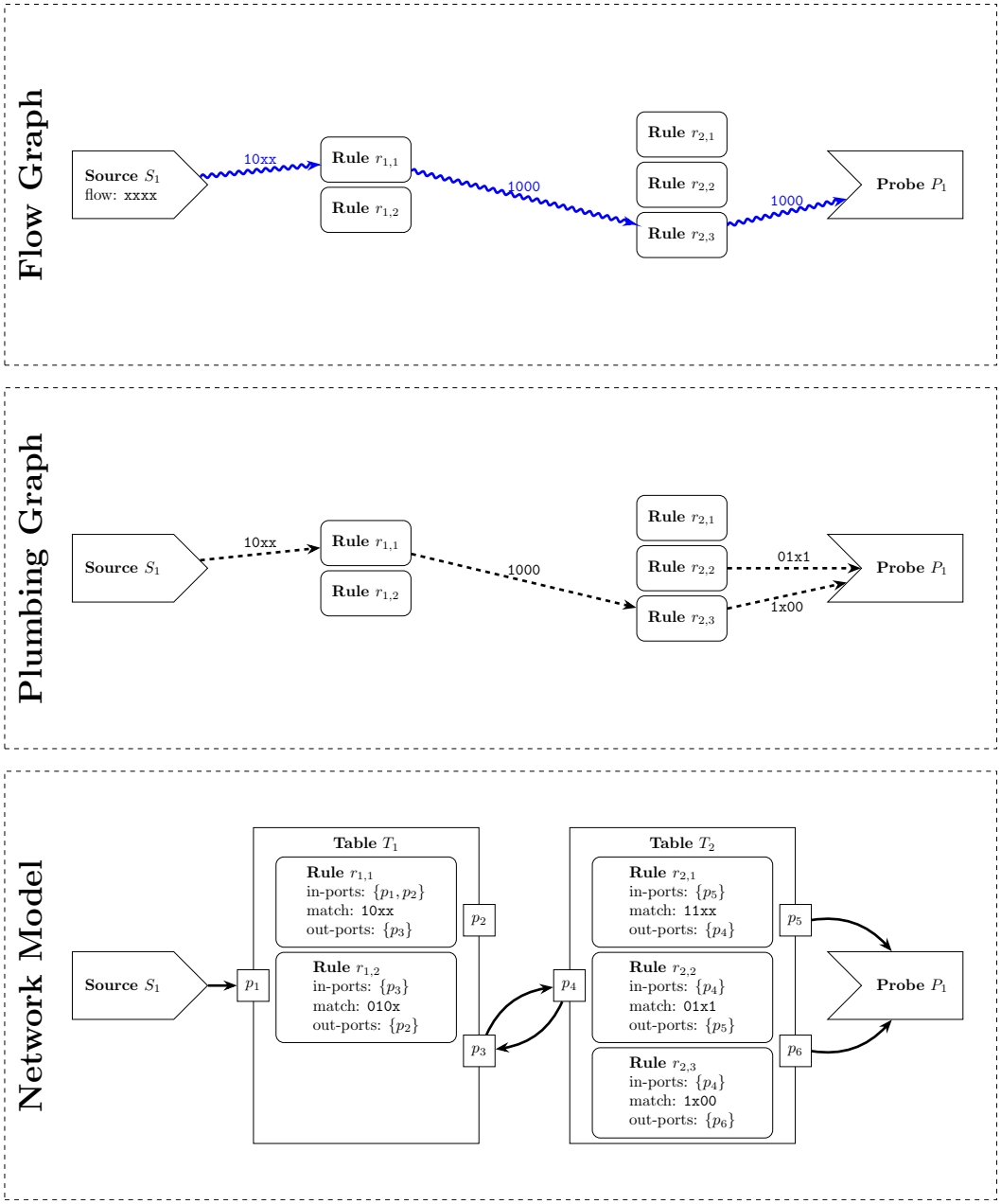


Fig. 2.4.: Layered graphs in NetPlumber with a descending degree of abstraction from the top to the bottom.

consists of *Tables* with attached *Ports* that are connected using unidirectional *Links*. A table holds a set of *Rules* that implement packet matching and perform actions on packet sets, i.e., rewriting and forwarding.

NetPlumber's optimizations conduct local dependency analysises between rules in order to construct the *Plumbing Graph*. This directed overlay graph stores rule dependencies within the tables and shortcuts to related rules in adjacent tables, i.e.,

tables that are directly connected via links. These shortcuts are called *Pipelines* or *Pipes*.

In order to verify network properties, NetPlumber calculates traffic propagation originating at special nodes – the *Source Nodes* – that are attached to ports and emit sets of packets. These propagations are called the *Flow Graph* and are calculated along the *Plumbing Graph's* pipeline structure. Throughout propagation, rule dependencies within the tables are considered, i.e., packets that are handled by rules with higher priorities are not propagated by rules with lower priorities. This way, the *Flow Graph* precisely covers all packet flows through the network. Finally, *Probe Nodes*, which are attached to ports, analyse incoming flows concerning their properties, e.g., where they originated from or if they traversed certain ports or tables.

Changes to the network model, e.g., adding or deleting a rule, or changing a link's status, are tried to be handled as local as possible in order to minimize reverification efforts. E.g., changing a rule only requires the recalculation of the rule's pipelines, the repropagation of flows traversing the rule or rules with dependencies to the rule, and reverification of these repropagated flows.

Figure 2.4 shows NetPlumber's layered graphs for an example network with two switches that have three ports and are connected with a cable. Further, the first switch holds two forwarding rules whereas the second holds three. The network model consists of two tables T_1 and T_2 with three ports each, a source node S_1, and a probe node P_1. T_1 holds two rules while T_2 has three rules. The source S_1 is attached to the port p_1 of table T_1 and the ports p_5 and p_6 of table T_2 are connected to the probe P_1. Also, the ports p_3 of T_1 and p_4 of T_2 are connected bidirectionally.

The Plumbing Graph's view is restricted to the source and probe as well as the rule nodes. It adds pipelines from P_1 to $r_{1,1}$, $r_{1,1}$ to $r_{2,3}$, $r_{2,2}$ to P_1, and $r_{2,3}$ to P_1. The pipelines have filter expressions that represent packet sets which were processed by the originating rule and that are legit for processing by the target rule. There are no further dependencies between rules within the tables in this example.

Finally, NetPlumber's Flow Graph shows the propagation of traffic that was emitted by S_1 through the network. The traffic flows along the pipelines and eventually reaches P_1 where the incoming packets and their paths may be analyzed, e.g., if they conform reachability policies. Loop detection is built-in the flow propagation but did not occur in our example.

We will detail NetPlumber's network modeling and the construction of the Plumbing resp. Flow Graphs throughout the remainder of this section.

2.2.2.1. Modeling Primitives

NetPlumber offers several primitives to model networks that can be added or removed at runtime. Each modeling action triggers an update of the Network Model, the Plumbing Graph, and the Flow Graph. E.g., adding a rule to a table triggers the calculation of dependencies and pipes, and forces (re-)propagations of flows running through the rule or dependent rules thereof. This may lead to a re-verification of network properties by affected probes.

Tables, Ports, Links *Tables* are the central building blocks of the Network Model. They hold *rules* (cf. the next paragraph) and *ports* are associated with them. Ports, in turn, are interconnected via unidirectional *links*. Together, tables, ports, and links form the network's topology. In NetPlumber, tables are referenced using a globally unique index.

As depicted in Figure 2.4, in our example Network Model, there are two tables T_A and T_B which hold three rules and have three ports each. The ports p_3 and p_4 are connected in both directions while S_1 is connected to p_1 and p_5 as well as p_6 are connected to P_1.

Rules NetPlumber's *rules* consist of a matching part, an optional rewriting part, and a forwarding part:

$$r = (\underbrace{in\_ports, match}_{\text{matching}}, \underbrace{mask, rewrite}_{\text{rewriting}}, \underbrace{out\_ports}_{\text{forwarding}})$$

The in_ports and out_ports are sets of ingress resp. egress ports which must belong to the table hosting the rule. Intuitively, the ingress ports match ports that the rule applies to whereas the egress ports are ports that forward matched and optionally rewritten packets. If a rule has an empty set of input ports, it processes packets from all of the table's ports while an empty set of output ports lets NetPlumber drop the processed packets. The $match$ is a wildcard expression that defines which packets this rule should be applied for. The $mask$ and $rewrite$ are wildcard expressions, too. The $mask$ marks bits that should be rewritten and the $rewrite$ specifies these bits new values. E.g., given an incoming set of packets `10xx`, a mask `0110`, and a rewriting expression `1111`, then, the resulting set of packets would be `111x`. Rewriting is optional due to the fact that by default, NetPlumber rules are initialized with an *all-zero* mask which needs to be overwritten in order to model rewriting

actions. In our example shown Figure 2.4, we have just pure forwarding but no rewriting rules.

A rule's priority within a table is given by its position indicated by an index. Specificantly, a lower index encodes a higher priority. Throughout this work, we denote a rule at position i as

$$r_i = (in\_ports_i, match_i, mask_i, rewrite_i, out\_ports_i)$$

If necessary, we also include a table's index t: $r_{t,i}$. Formally, a rule is defined as $\mathbb{N} \times 2^{\mathbb{N}} \times \mathcal{H} \times \mathcal{H} \times \mathcal{H} \times 2^{\mathbb{N}}$ (resp. $\mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathcal{H} \times \mathcal{H} \times \mathcal{H} \times 2^{\mathbb{N}}$ with tables).

Sources and Probes *Source* nodes emit arbitrary sets of packets which are implemented as header space objects. They are attached to one or more ports with links that always point away from the source node. Once attached, NetPlumber starts to calculate traffic propagation immediately. *Probe* nodes, on the other hand, absorb incoming flows and can perform analysis over the packet sets and the flow's paths, e.g., if they originate from certain sources or traversed some waypoints. One or more ports can be linked to a probe node and NetPlumber propagates existing flows running over these ports and starts analysis immediately.

In our example, we have one source node S_1 which emits all possible packets, i.e., `xxxx`. After propagating this flow along the model's pipelines and thereby precising the packet set, `1000` arrives at the probe node P_1 and may be further analyzed.

2.2.2.2. The Plumbing Graph

The Plumbing Graph offers shortcuts for efficient flow propagation calculations. It is computed statically, i.e., without considerations of actual packet flows as induced by source nodes, and connects rule nodes in two ways: *intra-table* dependencies and *inter-table* dependencies.

Intra-table dependencies occur within a table. I.e, if there are packets that are handled by rules with higher priorities. Formally, a rule

$$r_i = (in\_ports_i, match_i, mask_i, rewrite_i, out\_ports_i)$$

with priority i has an intra-table dependency to another rule

$$r_j = (in\_ports_j, match_j, mask_j, rewrite_j, out\_ports_j)$$

with a higher priority j if their matching parts overlap, i.e., if

$$i > j, in\_ports_i \cap in\_ports_j \neq \emptyset, match_i \cap match_j \neq \emptyset.$$

Note that a smaller index denotes a higher priority in NetPlumber.

Inter-table dependencies occur between adjacent tables that are connected via links. I.e., a rule

$$r_{a,i} = (in\_ports_{a,i}, match_{a,i}, mask_{a,i}, rewrite_{a,i}, out\_ports_{a,i})$$

from table a with index i has an inter-table dependeny with a rule

$$rule_{b,j} = (in\_ports_{b,j}, match_{b,j}, mask_{b,j}, rewrite_{b,j}, out\_ports_{b,j})$$

from another table b with index j if the packets that have been processed by rule $r_{a,i}$ are forwarded to the table with rule $r_{b,j}$ and are matchable by rule b. Formally,

$$\begin{aligned} \exists (p_1, p_2) \in L: \quad & p_1 \in out\_ports_{a,i}, \\ & p_2 \in in\_ports_{b,j}, \\ & rw(match_{a,i}, mask_{a,i}, rewrite_{a,i}) \ \cap \ match_{b,j} \neq \emptyset \end{aligned}$$

where $L \subseteq \mathbb{N} \times \mathbb{N}$ is the set of all links and $rw : \mathcal{H} \times \mathcal{H} \times \mathcal{H} \to \mathcal{H}$ is the rewriting function that masks header bits in wildcard expression and overwrites those with corresponding bits specified in another wildcard expression. For example, a router that performs DNAT rewrites the destination IP address of incoming packets from a public IP address to some private IP address.

To construct the Plumbing Graph, the intra-table dependencies and inter-table dependencies are calculated for every rule. The dependencies are stored as two kinds of edges between the nodes and the common packet sets are associated with these edges. In NetPlumber's terminology, the inter-table dependencies are called *pipeline* and the associated packet set is a *filter*. The intra-table dependencies are simply called *dependencies* and their associated packet set is referred to as *common headers*. Formally:

$$E_{PG} = \{(r_1, r_2, h)\}$$

where r_1 is the origin in case of a pipeline resp. the dependent rule in case of a dependency, r_2 is the target resp. effecting rule, and h is the *filter* resp. *common headers*.

Since NetPlumber implements rule matches, masks, and rewrites as wildcard expressions, the search for dependencies benefits from the highly optimized intersection

and rewrite operations thereof. Nevertheless, the complexity to find intra-table dependencies is quadratic per table while the inter-table dependency analysis is cubic[10]. In practice, these extensive searches need to be performed once throughout the initialization phase and updates require more localized efforts, e.g., adding a rule requires just one intra-table search and one inter-table analysis per adjacent table that is connected by at least one link originating from a port that the rule forwards processed packets to.

In addition to the pipelines between rules, in the Plumbing Graph, also source and probe nodes are connected to adjacent rules via pipelines. The analysis is performed similarly to the inter-table dependency analysis. Intuitively, source resp. probe nodes are treated as single rules in a virtual table with one outgoing resp. incoming port that is connected to the ports of regular tables. Both node types are handled as if they match all packets and do not perform any packet rewriting.

Concerning our example, there are no intra-table dependencies. E.g., the rules $r_{2,1}$ and $r_{2,3}$ do not overlap despite matching some common packet `1100` since their *in-ports* are distinct.

Table 2.1 gives an overview of the pipeline calculations and comments on the results. Ultimatively, the Plumbing Graphs edges are

$$E_{PG} = \{(S_1, r_{1,1}, \texttt{10xx}), (r_{1,1}, r_{2,3}, \texttt{1000}), (r_{2,2}, P_1, \texttt{01x1}), (r_{2,3}, P_1, \texttt{1x00})\}.$$

2.2.2.3. The Flow Graph

The Flow Graph represents traffic flows originating at the source nodes. It is conceptualized as an overlay graph of the Plumbing Graph, i.e., all nodes and edges of the Flow Graph map onto respective nodes and edges in the Plumbing Graph. Instead of pipeline filters the Flow Graph's edges are annotated with sets of packets traversing the network.

The Flow Graph is constructed by recursively applying the propagation algorithm along the Plumbing Graph in a per-flow depth-first manner starting at the source nodes. This search is perfomed until either 1) a probe node is reached, 2) a loop is

[10] For the intra-table analysis all rules are compared pairwise, i.e., there are $\frac{r \cdot (r-1)}{2}$ comparisons (where r is the amount of rules). Since a single comparison is considered to run in $\mathcal{O}(1)$ due to the fixed amount of header bits, the complexity is quadratic.
Further, for the inter-table analysis, comparing two tables is quadratic as well since their rules are compared pairwise, too. Nevertheless, a table may be neighboring multiple tables and, hence, the resulting complexity is cubic since the table needs to be compared to each neighbor.

a\b	$r_{1,1}$	$r_{1,2}$	$r_{2,1}$	$r_{2,2}$	$r_{2,3}$	P_1	**Comments**
S_1	10xx	$\emptyset$	-	-	-	-	Only table T_1 is adjacent to S_1 since it is connected to p_1. $r_{1,1}$ matches p_1 whereas $r_{1,2}$ does not.
$r_{1,1}$		-	$\emptyset$	$\emptyset$	1000	-	There is a pipeline from $r_{1,1}$ to $r_{2,3}$ since $r_{1,1}$ forwards packets over the link (p_3, p_4) which are also processed by $r_{2,3}$, i.e., $h =$ 1000.
$r_{1,2}$	-		$\emptyset$	$\emptyset$	$\emptyset$	-	$r_{1,2}$ forwards packets over p_2 which is not connected to T_2.
$r_{2,1}$	$\emptyset$	$\emptyset$		-	-	$\emptyset$	$r_{2,1}$ does not have a pipeline with P_1 since p_4 is not connected to P_1. Also, $r_{1,1}$ does not match packets transfered by $r_{2,1}$ over the link (p_4, p_3), i.e., 11xx$\cap$10xx = 1zxx = $\emptyset$.
$r_{2,2}$	-	-	-		-	01x1	$r_{2,2}$ has a pipeline to P_1 since it forwards packets via p_5 which is connected to P_1.
$r_{2,3}$	-	-	-	-		1x00	$r_{2,3}$ has a pipeline to P_1 since it forwards packets via p_6 which is connected to p_1.

Tab. 2.1.: Results of the pipeline calculations (from a to b).

detected, or 3) there is no suitable pipeline for further propagation. In NetPlumber, flows are implemented using header space objects. The algorithm is the following:

$propagate(f, hist, r_{i,t})$:

Parameters:

f – the flow to be propagated encoded as header space object
$hist$ – the set of previously visited tables
$r_{i,t}$ – the current rule i in table t

1. If $r_{i,t}$ is a probe node,

   stop propagation and start analysis as specified for the probe node, e.g., if the flow started from a certain source.

2. If the current table has been visited before, i.e., $t \in hist$,

   stop propagation and report the detected loop.

3. Remove packets handled by rules in the same table with higher priorities, i.e.,

   $f' := f - \bigcup_{(r_{t,i}, r_{t,j}, common) \in E_{PG}} \{f \cap common \mid j < i\}$.

4. Rewrite the flow, i.e.,

   $f'' := rw(f', mask_{t,i}, rewrite_{t,i})$.

5. Repeat propagation for all outgoing pipelines, i.e., $(r_{t,i}, r_{t',j}, filter) \in E_{PG}$ (with $t \neq t'$),

   If there are packets that are processable by the next rule, i.e., $f''' := f'' \cap filter$ and $f''' \neq \emptyset$:

   remember the current table and process these packets at the next rule, i.e., $propagate(f''', hist \cup \{t\}, r_{t',j})$.

Concerning our example, the algorithm starts at S_1 with an empty history and $f =$ `xxxx`. Since there is just one pipeline and since source nodes do not rewrite traffic, the flow is propagated to $r_{1,1}$ but just the portion that can be handled by the rule, i.e., $f'' =$ `10xx`. So, at $r_{1,1}$ the flow $f =$ `10xx` with the history $hist = \{S_1\}$ is processed.

Since $r_{1,1}$ is not a probe node and there is no loop, the algorithm, first, removes all packets handled by rules with higher priorities. There is no such rule and hence,

second, the algorithm continues by applying masking and rewriting which is not the case either for rule $r_{1,1}$. I.e., the processed flow is $f'' = f' = f$. Third, the algorithm propagates the processed flow over all outgoing pipelines which is just the one to rule $r_{2,3}$, i.e., $(r_{1,1}, r_{2,3}, \texttt{1000})$. After applying the pipeline filter, i.e., $\texttt{10xx} \cap \texttt{1000} = \texttt{1000}$ which is not empty, the history is updated, i.e., $hist = \{P_1\} \cup \{T_1\} = \{P_1, T_1\}$, and the filtered flow is propagated to be processed by $r_{2,3}$.

Again, $r_{2,3}$ is not a probe node and there is no loop. So, the algorithm removes packets handled by rules with higher priorities. This time, there are two rules with higher priorities but there are no dependencies with $r_{2,3}$ since $r_{2,1}$ matches packets arriving at another port and $r_{2,2}$ matches a distinct set of packets, i.e., $\texttt{01x1}$. So, the algorithm continues by rewriting the flow if that was specified. This is not the case for $r_{2,3}$. I.e., the processed flow is $f'' = f' = f$. Now, the flow should be propagated over all outgoing pipelines which is just the one that heads to P_1 and has $\texttt{1x00}$ as filter. After applying the pipeline filter, i.e., $\texttt{1000} \cap \texttt{1x00} = \texttt{1000}$, and updating the history, i.e., $hist = \{S_1, T_1\} \cup \{T_2\} = \{S_1, T_1, T_2\}$, the flow arrives at P_1.

Since P_1 is a probe node, propagation stops and the analysis that has been specified for P_1 (see next section for details) is performed immediately.

2.2.3 Network Verification with NetPlumber

NetPlumber analyzes incoming flows, i.e., packet sets and their paths through the network, when they arrive at probe nodes. A flow is denoted as f and consists of a packet set and a path, notated as $f.h$ resp. $f.p$. The particular analysis can be specified using the *FlowExp* language which resembles regular expressions over packet sets and paths. A FlowExp expression consists of three parts: a *quantor*, an optional *filter expression*, and a *test expression*:

Quantor ($\forall$ or $\exists$) Determines whether all ($\forall$) or at least one ($\exists$) incoming and (optionally) filtered flows need to satisfy the test expression in order to satisfy the overall analysis.

Filter expression Determines whether an incoming flow's packet set and path satisfy some specified criteria and is further analysed with the test expression.

Test expression Determines whether an incoming flow's packet set and path, that has not been filtered by the filter expression, satisfy some specified criteria.

Formally ($\sim$ denotes the satisfyability operator):

$$\forall f' \in \{f \mid f \sim filter\} : f' \sim test$$

resp.

$$\exists f' \in \{f \mid f \sim filter\} : f' \sim test.$$

Both, *filter* and *test* are defined in terms of *conditions* over packet sets or paths. A condition can be:

$\top \mid \bot$	A constant true or false.
$\neg condition$	A negated condition.
$condition \vee condition$	The disjunction of two conditions.
$condition \wedge condition$	The conjunction of two conditions.
$path$	A regular expression over the incoming flow's path.
$header$	A check if the incoming packet set is a subset of a packet set h, i.e, $f.h \subseteq h$,[11] where h can be denoted as a header space object or a wildcard expression.

A *path* is a list of *pathlets* which can be[12]:

$p \in P$	A port specifier where $P \subseteq \mathcal{P}$ is a set of ports.
$t \in T$	A table specifier where T is a set of tables (must be a subset of the Network Model's tables).
.	A skip operator that skips the previous hop.
.*	A Kleene skip operator that skips zero or more previous hops.
$	The end of the path (i.e., the beginning of the flow's path at some source node).
^	The start of the path (i.e., the end of the flow's path at the probe node).

Given the probe node P_1 in our example shall perform a reachability analysis that checks if all incoming flows come from allowed hosts. In our case, we define S_1 as the only allowed host. The expression looks as follows:

$$\forall f' \in \{f \mid f \sim \top\} : f' \sim \hat{}.^* t \in \{S_1\}\$$

Since we have only one flow with $f.flow = \texttt{1000}$ and $f.path = S_1 \rightarrow r_{1,1} \rightarrow r_{2.3} \rightarrow P_1$ arriving at P_1 and this flow's path satisfies the expression, i.e., it originates at S_1, the overall expression also holds and the analysis in P_1 succeeds. NetPlumber

[11] The original paper also specifies header checks for equality, i.e., $f.h = h$, and true subsets, i.e., $f.h \subset h$, but NetPlumber does not implement these. Instead, one could specify them with FlowExp using $f.h = h :\Leftrightarrow f.h \subseteq h \wedge h \subseteq f.h$ resp. $f.h \subset h :\Leftrightarrow f.h \subseteq h \wedge \neg(f.h = h)$ as a workaround.

[12] Note that paths are specified backwards, i.e., from the probe node back to the source node.

yields a log message signaling the result. Whenever a new flow arrives, the analysis is triggered and if the overall result changes, a log message stating the probe's state change is emitted. For instance, if we had a second source node S_2 that emits `xxxx` and is connected to the port p_4 of table T_2, the flow would be forwarded by rule $r_{2,2}$ to P_1 via port p_5, i.e., $f.h = \texttt{01x0}$ and $f.p = S_2 \rightarrow r_{2,2} \rightarrow P_1$. P_1's check would fail because this second flow does not originate from S_1 and, hence, the universal quantifier would not be satisfied anymore.

2.2.4 Discussion: Shortcomings of HSA and NetPlumber

HSA and NetPlumber fall short in five ways: 1) their technical approach on policy definition, 2) their inability to detect clear misconfigurations like firewall anomalies, 3) their inability to model stateful devices, 4) their unnatural data model, and 5) the static amount of header bits:

1) The term *policy* is defined in a very technical way in HSA as well as in NetPlumber. The HSA paper [78] does not use the term at all but offers the reachability between network ports and loops as checkable invariants. The NetPlumber paper [79] refers to FlowExp as NetPlumber's policy language. Concerning the verification of network security compliance neither offering is sufficient. HSA and NetPlumber need significant manual effort by security officials and administrators to formulate compliance requirements since these are stated on the organizational level whereas only technical means are offered by the tools. Hence, their technical scope limits their practical usability significantly.

2) Neither HSA nor NetPlumber offer capabilities to analyze configurations for clear misconfigurations like firewall anomalies.

3) HSA and NetPlumber focus on modeling stateless devices like switches or routers. They do not offer nor implement means to model and verify stateful devices like firewalls. This excludes a large portion of real-world networks and significantly limits their applicability for security compliance verification.

4) Header space objects in general and wildcard expressions in particular, are hard to handle in terms of specification and readability – even for experts of the network domain. The mapping of header bits onto positions in the wildcard expression and the encoding of header values needs to be performed outside of NetPlumber which is prone to error. Additionally, reading and interpreting

the results can be challenging since there is no built-in translation back into a readable format, e.g., IP prefixes or VLAN tags.

5) In HSA and NetPlumber, the wildcard expressions have a fixed size of l that is configured before runtime. This results in a limited flexibility when modeling dynamic protocols like IPv6 with extension header chains (cf. to the next Section 2.3 for details).

In this work, we overcome these shortcomings by introducing the user-friendly notion of *Domain-oriented HSA* (DHSA) in Chapter 5. Using DHSA, we are able to check organizational level policies specified with FPL (see Chapters 4 and 7) as well as firewall anomalies (cf. Chapter 8). Also, in Chapter 7, we introduce a technique called *state shell interweaving* to model stateful devices using DHSA. Finally, in Chapter 5, we implemented a mechanism for expanding wildcard expressions and header space objects at runtime.

2.3 IPv6 Specifics concerning Verification

A common challenge for verification techniques lays in dealing with exponentially large search spaces. Concerning network verification the search space's size is determined by the amount of header bits, i.e., the size of the Header Space. IPv4 packets contribute with 144 bits to the search space[13]. For IPv6, the base header alone consists of 320 bits [33, pp. 6f] and there can be an arbitrary amount of IPv6 extension headers resulting in an arbitrary amount of additional header bits. Thus, simple enumeration is infeasible – neither for IPv4 nor for IPv6. Furthermore, verifying IPv6 needs additional care due to its extension header chains.

2.3.1 IPv6 Extension Header Chains

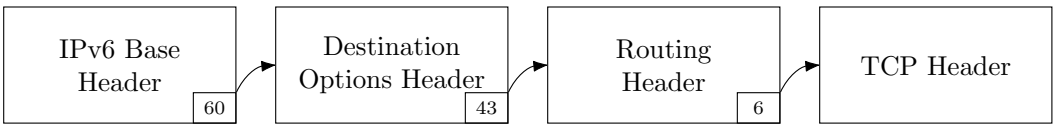


Fig. 2.5.: Example for an IPv6 extension header chain.

[13]The theoretical maximum size of an IPv4 header is 480 bits if IPv4 options are used. The IHL which is 4 bit wide indicates the IPv4 header's size and counts the amount of 32 bit words: $(2^4 - 1) \cdot 32 = 15 \cdot 32 = 480$.

The IPv6 standard [33] distincts between the static IPv6 *base header* and a collection of *extension headers* that can be used dynamically in a single packet using a chaining mechanism. Figure 2.5 shows an example of a header chain. Each IPv6 comprises a *next header* field which bears the IANA protocol number of the next header in the chain. The chain ends with the upper layer protocol, e.g., 6 for TCP, 17 for UDP, or 58 for ICMP [9], or the special *no next header* number, i.e., 59 [33, p. 24], if the packet is purely IPv6. In the example, following the IPv6 base header, there is a destination options header which specifies options intended to be processed by the packet's destination and, then, a routing header which comprises hops to be traversed by the packet. Finally, the TCP payload is referenced.

2.3.2 Practical Approximation of the Search Space for Extension Header Chains

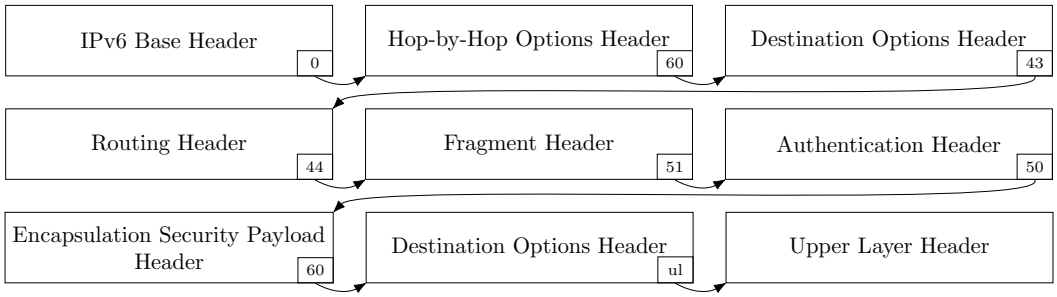


Fig. 2.6.: Maximum appearances and order of IPv6 extension headers as recommended by RFC 8200 [33, p. 10].

As seen in the previous Section 2.2, modeling the Header Space in tools like NetPlumber relies on an a-priori knowledge of the amount of header bits to encode the Header Space. We have shown that regarding IPv6 this is theoretically infeasible due to the dynamic nature of IPv6 extension header chains. Instead, we provide an approximation of a reasonable worst-case that covers most if not all cases of practical relevance.

First of all, the IPv6 standard does not forbid extension header chains of arbitrary length. A chain may be splitted and distributed over multiple packet fragments using IPv6 fragmentation headers and, hence, there is no formal upper limit for the amount of header bits.

Though, the standard offers recommendations which makes it unlikely that some real-world implementation would forward such a packet. First, it is recommended to use each header at most once [33, p. 10]. An exception is given by the *Destination*

Options header which may occur twice – before and after the *Routing Header*. In addition, the standard recommends an order in which the headers may occur. The resulting largest extension header chain is depicted in Figure 2.6. Assuming that the packet comprises the largest possible amount of options with maximum size each, its IPv6 part would consist of:

$$40 + 256 + 256 + 256 + 8 + 24 + 273 + 256 \text{ bytes} = 1,369 \text{ bytes} = 10,952 \text{ bits}.$$

Note that in practice, some devices' implementations or default configurations do not follow the standard's requirements for various reasons [52], e.g., hardware accelerators for packet classification may have a limited parsing and lookup capacity concerning the headers' size and may drop longer packets. Since these limitations are highly specific to the implementations and contrary to the standard's requirements, we did not put these into further consideration.

To conclude, even though the standard practically limits the amount of header bits, it is still huge and far beyond enumerability. We provide a technique to enable formal verifiability of dynamic IPv6 extension header chains in Section 6.3.

2.4 Summary

In this chapter, we laid the foundations to understand the remainder of this thesis. First, we gave an overview about the main terms in ISM, i.e., security policy, security specification, and security configuration, and their relations.

Then, we detailed *Header Space Analysis* (HSA) and its highly optimized implementation in *NetPlumber* which offers better performance and a higher expressiveness, e.g., reachability analysis including waypoints. So, HSA can form the basis for our *Domain-oriented HSA* (DHSA) and our improvements to NetPlumber, both of which, we will introduce in Chapter 5. For instance, our improvements enable optional optimizations of the memory footprint of header space objects and the extension of the Header Space at runtime. We also discussed several shortcomings of HSA and NetPlumber. I.e., 1) their technical policy definitions which is hardly usable for security officials and network administrators, 2) their inability to detect clear misconfigurations like firewall anomalies, 3) their limitation to stateless device models, 4) their unnatural data model that limits their usefulness even for domain experts, and 5) the fixed amount of header bits that limits applicability for dynamic protocols like IPv6.

Finally, we explained the specifics that make the verification of IPv6 networks a challenging venture. Particularly, the dynamic nature of IPv6 extension header chains needs special attention.

3 Related Work

In this chapter, we present and discuss the state of the art concerning the research area of *Network Verification*. We provide a comparative analysis, discuss the most important approaches in detail, and identify open questions. For the related work concerning high-level policies and firewall anomalies, refer to Section 4.2 resp. Section 8.2.

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* (Section 3.1) 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 analysis* (Section 3.2) 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, after describing our applied methodology, we give an overview of these verification approaches, highlight notable tools, and compare them to our own verification framework FaVe (cf. Chapter 6). Finally, since IPv6 is a rarely researched issue in this context, we provide an additional discussion in Section 3.3.

In Table 3.1 we give an overview of related tools that provide capabilities to support security analysis. The table is structured by the categories of *accessible compliance*, *policies*, *sanity*, modeling *features*, and *other* notable traits which are explained in the following:

Accessible Compliance This trait indicates whether the tool provides abstractions to describe security policies and compliance reporting suitable for non-academically skilled staff, e.g., through a reachability matrix – a type of visualization also used by commercial providers like Cisco [21].

Policies The *policies* section highlights invariants that are necessary for compliance verification.

> *Network Reachability:* The ability to determine the reachabilities between networked systems is a key feature in order to check for the eligibility of communication and, consequently, it enables compliance verfication.

	Tool		Policies			Sanity		Features					Other	
		Accessible Compliance	Network Reachability	Slice Isolation	Waypoints	Loop Detection	Black Holes	Stateless Packet Filtering	Stateful Packet Filtering	Multiple Devices	IPv6 Support	Shown Scalability	Verification Engine	Public Code
Firewall Verification	FPA [133]		✓					✓		✓			F	
	FIREMAN [153]		✓			✓		✓		✓			B	
	Jeffrey, Samak [73]		✓			✓		✓		✓		✓	S	
	ad6 [94]		✓			✓		✓		✓	✓	(✓)	S	✓
	fffuu6 [37]	(✓)	✓					✓	✓		(✓)	✓	T	✓
	FWS [15]		✓					✓	✓			✓	T	✓
	Yin et al. [150]	✓	✓					✓			(✓)	✓	T	
Data Plane Verification	ConfigChecker [5]		✓			✓		✓		✓		✓	B	
	FlowChecker [4]		✓	✓		✓		✓		✓		✓	B	
	Flover [136]		✓							✓			T	
	NetPlumber [79]		✓		✓	✓	✓			✓		✓	H	✓
	VeriFlow [81]		✓			✓	✓	✓		✓		✓	E	✓
	Libra [155]		✓		✓	✓	✓			✓		✓	E	
	SecGuru [14]	(✓)	✓					✓		✓		✓	T	
	NoD [90]		✓		✓	✓	✓	✓		✓		✓	T	✓
	SymNet [138]		✓	(✓)		✓	✓	✓	✓	✓			T	✓
	AP-Verifier [147]		✓	✓	✓	✓	✓			✓		✓	A	✓
	VMN [113]		✓	✓	✓	✓		✓	✓	✓			T	
	Delta-net [60]		✓			✓				✓		✓	A	
	APKeep [156]		✓		✓	✓	✓	✓		✓		✓	A	
	RCDC [72]	(✓)	✓					✓		✓		✓	T	
	NetSMC [154]		✓	(✓)	✓			✓	✓	✓		✓	?	
	Yousefi et al. [151]		✓					✓	✓	✓			B	
	FaVe + FPL [96]	✓	✓			✓		✓	✓	✓	✓	✓	H	✓

Tab. 3.1.: Overview of different formal network verification tools. The abbreviations in the column *Verification Engine* mean: A = Atoms, B = BDD, E = Equivalence Classes, F = Finite Automaton, H = HSA, S = SAT, T = Theorem Prover.

Slice Isolation: Networks can be logically divided resp. virtualized. Some tools provide special support for modeling and checking of the proper isolation of these network *slices*.

Waypoints: Some tools offer capabilities in order to check if traffic flows over certain point in the network, e.g., firewalls or IDSes.

Sanity This section describes a class of sanity invariants that are useful for security reasoning – independendly from specific network security policies.

Loop Detection: Packets that loop within a network consume extensive resources – particularly, if these packets are not dropped and loop forever, e.g., due to missing TTL decrementation in SDN-based layer-2 networks. This can be a serious security issue if an attacker is able to craft such packets and can saturate the network's ressources causing a Denial-of-Service.

Black Holes: Misconfigured network devices may cause certain packets to be dropped silently which may cause an outage of parts of the network and which is hard to debug. Some tools provide analysis to identify this unwanted behaviour.

Modeling Features Fourth, the *features* section clusters a set of modeling capabilities necessary for advanced security verification:

Stateless Packet Filtering: The tool is able to properly process the configuration of a stateless packet filter.

Stateful Packet Filtering: The tool is able to properly handle packet filters including stateful behaviour.

Multiple Devices: The tool can model networks with more than one device.

IPv6 Support: The tool can adequately model and process IPv6 configurations. We mark limited IPv6 support with round brackets, e.g., if the tool only supports IPv6 addresses or if it loses verification precision when modeling dynamic behaviour, e.g., extension header chaining.

Shown Scalability:[1] We define a tool as shown scalable if the authors *evaluated reachability* for workloads exceeding certain *sizes* while keeping a certain *timebox*. Particularly, we define this timebox to be a "coffee break" of *ten minutes*. For the verification of a single firewall, we define the mimimum workload to be a rule set of at least *1,000 firewall rules* and we require *10,000 networking rules* for whole networks, e.g., forwarding and filtering rules.

[1] It is quite challenging to conduct a fair performance comparison of publications ranging over two decades and different areas of research. The challenges occur due to hardware evolution, different workloads, and use case imposed requirements. Further, the size and complexity of common network configuration evolves and, hence, there is no commonly agreed definition for the term *scalability*.

Therefore, we chose our definition in order to support our use case of *continuous network security compliance* by, first, requiring runtimes that enable administrators to integrate the tool into daytime workflows instead of nightly runs. Second, we require configuration sizes of magnitudes that may occur in medium sized organizations, e.g., universities. By stating these two requirements, a tool's evaluation indicates scalability for compliance verification.

Other Finally, we highlight the tools' underlying verification engine and the public availability of an implementation.

In the following, we present the different approaches and tools in detail and, also, we discuss their similarities and differences with FaVe. First, we elaborate the *Firewall Verification Tools* and, second, we discuss the *Data Plane Verification Tools*.

3.1 Firewall Verification Tools

Tools to analyze firewalls follow a long tradition and leverage *Finite Automata* [133], *Binary Decision Diagrams* (BDD) [153], *Firewall Decision Diagrams* (FDD) [89], *Boolean Satisfiability* (SAT) [73, 94], *Theorem Prover* [37], *Constraint Solving* (CSP) [149], or *Satifiability Modulo Theories* (SMT) [15, 150] as verification techniques to discover anomalies and insecurities in packet filter rule sets and networks. Also, support for IPv6 addresses [148] and extension header chains [94] was shown. Later, analysis of mutable datapaths for stateful firewalls and other middleboxes has been presented featuring *Virtual State Tags* [37], *State Oracles combined with SMT solving* [113], and *Symbolic Model Checking* [154]. Next, we highlight approaches that support the analysis of stateful packet filters, i.e., fffuu6 and FWS, and discuss the remaining approaches subsequently.

fffuu6 Diekmann et altera [37] have presented an approach to verify stateful IPTables rule sets where their tool `fffuu` itself is fully proven by the theorem prover *Isabelle/HOL*. First, the tool safely transforms firewall rule sets into a more simple normal form and, then, 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 7.2.3). 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.

FWS The *FireWall Synthesizer* (FWS, [15]) is a tool to safely transcompile firewall rule sets from one firewall configuration language, e.g., `iptables`, to another, e.g., `ipfw` or `pf`. Given a firewall rule set, it is, first, decompiled into an *intermediate firewall configuration language* (called IFCL) which can be formally analyzed or translated into another firewall configuration language. The publicly available,

Z3-based tool offers reachability analysis for single firewalls and analyzes the TUM workload (cf. Section 7.2) in ten minutes.

In comparison, FaVe is not limited to single firewalls but supports a variety of network devices and topologies. Also, it has extensive support for dynamic protocols like IPv6. In addition, FaVe shows great scalability. For instance, it analyzes the TUM workload in about less than three seconds (cf. Section 7.2.3).

Yin et altera This work [150] introduces a formal tool that enables conformance verification of stateless firewall rule sets with high-level reachability policies. The authors let security officials and network adminstrators specify a set of tables where technical attributes, e.g., IPv6 prefixes or protocol and port, are given names that are then used in policy rules. A policy rule has the form:

$$s_i : \text{ if } S_i \text{ in } R_{i1} \text{ from } R_{i2} \text{ then } A_i$$

where:

- s_i is the rule's index in the rule set,
- S_i is a service name, i.e., protocol and port,
- R_{i1}, R_{i2} are region names, i.e., IPv6 prefixes, and
- A_i is an action, i.e., permit or deny.

In the paper, a *security policy* (SP) is a list of policy rules. A *firewall policy* (FP) has been defined in previous work [149] as a list of network quintuple rules. The tool has been evaluated using custom and synthetic workloads of up to 5,000 policy rules and up to 5,000 firewall rules generated using Classbench-ng [100]. Their evaluation shows run times of up to 2.36 seconds which indicates scalability.

In comparison, FaVe is able to check a wide variety of networks and network devices including stateful firewalls. Furthermore, it remains unclear how the SP rules have been created, e.g., derived from the FP or hand-crafted. Therefore, it is not possible to determine whether a) the SP/FP combination represents some trivial and easily solvable case, e.g., if the SP rules are a direct derivative of the FP rules where the technical attributes are pseudonymized to form SP rules, or b) if the SPs actually resemble real-world policies. This argument also holds for FPs generated by Classbench-ng. In contrast, in Section 7.2, we show FaVe's scalability using well defined synthetic as well as real-world workloads. Furthermore, we compared FaVe's firewall anomaly detection performance against a similar approach by Yin

et al. [148] in Section 8.5.1 and clearly outperformed their approach. We do not expect a significantly better performance of their compliance verification work due to the underlying technology, i.e., the Z3 theorem prover.

Note that this work has been published in parallel to ours on a similar topic, i.e., [96], and they do not ship an open source implementation. Therefore, we do not provide a comparatative study against their approach.

Other Approaches

Further approaches lack the ability to analyze stateful packet filters, lack IPv6 support [133, 153, 73, 149], or offer limited scalability [133, 153, 149]. For details on our previous approach ad6 [94] see Section 1.4.

3.2 Data Plane Verification Tools

These tools analyze snapshots of the data plane leveraging smart, very fast, and domain specialized data structures and algorithms. The approaches optimize verification in terms of several dimensions like packet equivalence, locality, header space limitations, network structure, expressiveness concerning state, or the ability for incremental updates. Some approaches scale to hundreds of millions of (very limited) network rules while others offer more balanced but still highly performant tradeoffs. In the following, we discuss the approaches with support for stateful packet filters and identify open questions.

SymNet A differing approach to verify network data planes is offered by SymNet [138, 39] which is based on symbolic execution. The models of packet processing devices are expressed with the imperative *Symbolic Execution Friendly Language* (SEFL) [138] and the author's network verification tool SymNet is shipped along with a variety of prebuild models for different network devices – including a stateful firewall.

In SymNet, network devices comprise ingress and egress ports that are connected unidirectionally in accordance with the network topology. Each ingress port is associated with a SEFL program that implements the device's packet processing and eventually drops or forwards packets via the egress ports. Forwarding and filtering rules are modeled as small sub programs that are included into the device's

program based on the device's configuration. In order to verify invariants, SymNet injects symbolic packets at ingress ports that are specified by the user, executes the symbolic programs, and propagates the packets through the network until a fix point is reached. SymNet's implementation is based on Z3 and it transforms SEFL programs, the devices models, the network topology, and the invariant to be verified into a Z3 instance that is solved for satisfiability.

SymNet checks pairwise reachability in the Stanford workload in about eight minutes while FaVe needs about two seconds for the same benchmark (cf. Section 7.2.2). Also, in comparison to FaVe, the tool does not scale well for large firewall rule sets as shown in the evaluation in Section 7.2.3.

Later, in [39], the authors introduce a technique that enables the verification of the equivalence of a network model with an abstract policy model. Nevertheless, the concrete features of this policy model are not elaborated and, hence, its stance against FPL and FaVe remains unclear.

VMN This tool (VMN [113]) aims at the verification of networks that contain complex middleboxes like stateful firewalls or intrusion prevention systems. The main idea is to introduce oracles to cope with complex functionality that would overcomplicate the models and analysis, e.g., stateful connections, related packet flows, etc. In this paper, an oracle is a variable that represents the outcome of some functionality. For instance, in a model of an Intrusion Prevention System (IPS), a boolean variable may represent the check if a packet is deemed malicious. Depending on the variable's value, the IPS model behaves differently, e.g., it drops or passes the packet. Throughout analysis, the verification engine only needs to vary the oracles' values instead of a likely much more complicated set of interdependent variables that model the actual functionality. Further, VMN tries to slice the network model into representative sub networks in order to verify invariants surrogatively on smaller workloads. This facilitates scalability for large networks. Their Z3 based prototype is evaluated using synthetic workloads and yields runtimes in the range of minutes for reachability analysis.

In contrast to FaVe, VMN strongly limits network and packet models, i.e., only the classical network quintuple plus interfaces are considered. More complex header spaces and dynamic protocols like IPv6 are not supported. Further, the synthetic workloads do not permit inference for VMN's behaviour in more realistic scenarios which have been evaluated for FaVe (cf. Section 7.2). In general, VMN seems to offer inferiour performance in comparison with other tools. Particularly, NetSMC, which

we will discuss next, provided a comparative evaluation and showed a significantly better performance.

NetSMC A notable approach to the verification of stateful network functions is NetSMC [154]. 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. Due to the absense of a more detailed description in their publication and the lack of an open source implementation, we could not determine whether NetSMC leverages some well-known verification engine, e.g., SAT or BDDs, or if they implemented some custom algorithms and data structures for low-level verification.

For their evaluation, they use a model for the *pfSense* firewall that was learned with their tool *Alembic* [103]. NetSMC supports an LTL (Linear Time Logic) dialect to specify policies. While being powerful, LTL can hardly be considered suitable for network administrators or security officials. For example, the following statement should ensure that traffic that has been flagged to be suspicious by a first IDS is redirected to a second IDS for deeper inspection:

$$\forall x \in Dept. \quad \mathsf{G}(\texttt{src} = x \wedge susp[x] > 10 \rightarrow \mathsf{G}(\texttt{src} = x \rightarrow \mathsf{f}(\texttt{loc} = H)))$$

In detail, the statement says that after a host x from department $Dept.$ has sent more than 10 suspicious packets (counted in the first IDS's internal state table $susp$), all of its packets should pass through the location H which is the second IDS.

Nevertheless, the example demonstrates the challenges of a rather literal LTL syntax. FPL, on the other hand, was designed to be understood by other people than computer scientists (see Section 4.1.2).

While clearly outperforming VMN, NetSMC's evaluation does not promise a performance that matches FaVe's. NetSMC analyzes a single stateful firewall with 800 rules in about 4 minutes whereas FaVe verifies compliance of a whole network with 3,396 rules in about 36 seconds (UP workload, cf. Section 7.2.1) or checks for firewall anomalies in a stateful firewall with 1,035 rules in 0.05 seconds (UP workload, cf. Section 8.5.1). Additionally, NetSMC lacks support for IPv6.

Yousefi et al. This work [151] takes a different approach to network verification and follows an idea with similarities to SymNet. The authors offer a top-down

approach where high-level specifications of network functions are formally analyzed and synthesized to actual network code, i.e., P4 [16].

In this work, a network function is specified in terms of a table of prioritized rules that match packet classes and perform a series of actions on packets, e.g, dropping, changing, or forwarding to another network function, and actions on the table, e.g., inserting, activating, deactivating, or removing rules. This collection of inter-related network functions is modeled as a state-transition system and can be verified for liveness properties, e.g., if a stateful firewall blocks uninitiated packet flows or if a set of hosts is eventually reached.

In contrast, FaVe focuses on the verification of real-world configurations. In general, FaVe supports both – green and brown field setups – whereas Yousefi et altera focus on the former. Their approach shows similarities to our CONTROL phase where we generate security configurations from FPL policies. Nevertheless, Yousefi et altera propose the synthesis of P4 programs for whole networks. Hence, their approach requires programmable network devices while we build on traditional network infrastructure, e.g., IPTables based packet filter firewalls. This way, we do not need to migrate existing infrastructure first.

Other Approaches

For the sake of brevity, we summarize the other dataplane verifications approaches in the following and provide more detailed descriptions in Appendix A.3.

Model Checkers Three approaches using classical model checking techniques have been explored – particularly with an emphasis on Software Defined Networks (SDN): *ConfigChecker* [5], *FlowChecker* [4], and *Flover* [136] where FlowChecker is an adaption of ConfigChecker to OpenFlow-SDN. In comparison, FPL and FaVe have a clear advantage in terms of accessibility and expressiveness. The *Computational Tree Logics* (CTL) [23] – which is used by ConfigChecker and FlowChecker – is hardly usable by non-academically skilled staff. Flover, on the other hand, does not provide any means for high-level policy specification at all. In addition and compared to FaVe, these tools provide fast but still not as rapid verification in the range of tens of seconds. *FlowChecker* is able to detect slicing violations and shadowing whereas *Flover* checks for non-bypass properties which include inconsistencies in the flow tables that forward unallowed traffic. Furthermore, they do not support to model complex devices with arbitrary header fields, stateful firewalls, and dynamic protocols like IPv6.

VeriFlow and Libra The tool *VeriFlow* [81] pioneered the idea to decompose the data plane into *equivalence classes* (EC) where any packet is handled by exactly one EC. This way, invariants can be verified on smaller and independent data sets which allows analysis to be conducted in parallel. Furthermore, configuration changes typically only affect a limited amount of ECs and, hence, reverification is only necessary for these ECs and invariants. VeriFlow shows a significant scalability for routed networks with ACLs where the evaluated synthetic FIB (BGP Forwarding Information Base) had five million entries. *Libra* [155] adopted the EC approach and, by limiting the header space to IPv4 destination addresses, the tool was able to analyze a workload of 316 switches with 150 million forwarding rules in about 93 seconds.

In comparison, FaVe offers a similar performance as VeriFlow but has been evaluated for much more complex networks including dynamic protocols like IPv6. FaVe is neither limited to routers with ACLs or L3-switches such as VeriFlow resp. Libra. Instead, FaVe offers a large variety of predefined and complex device models including stateful firewalls. Our evaluation with realistic workloads (cf. Section 7.2) reveal FaVe's scalability and applicability for large networks without being limited to be purely routed or datacenter-centric.

NoD, SecGuru, and RCDC Three works have been published by groups with close affiliations to Microsoft research: *NoD* [90], *SecGuru* [14], and *RCDC* [72]. They put an emphasis on the applicability to data center networks and explore approaches in two dimensions: generality of applicability and performance. In summary, NoD provides a more generic approach to network verification whereas SecGuru and RCDC achieve scalability to data center workloads through specialization to the network architecture. *Network-optimized Datalog* (NoD) adapts Datalog – historically a Prolog dialect for database operations [47] – to the domain of network verification. They extend Microsoft's Z3 theorem prover and their prototype shows good performance. *SecGuru* [14] takes a different approach by focusing on the verification of Azure cloud data center networks and by taking their high degree of structure into consideration. The main idea lies in the formulation of assumptions of a router's environment that allows a local and independent verification of the so-called *cloud contracts* in parallel. The authors do not provide a structured performance evaluation but state that the tool is used in Azure's operations on a daily basis with up to 40 thousand runs per month. The *Reality Checker for Data Centers* (RCDC, [72]) stems from SecGuru and overcomes some of the latter's limitations. Particularly, it still relies on the structure of data centers but enables the verification of end-to-end invariants and also flexibilizes contract definition. Similar to SecGuru, RCDC is

reportedly in active use in production and the authors state that its performance is on par.

In comparison, FaVe is not limited to cloud data centers, offers support for dynamic protocols like IPv6, and provides complex device models, e.g., stateful packet filters. Furthermore, SecGuru only offers hand-crafted cloud contracts and RCDC added only limited expressiveness. FPL, in contrast, allows more fine grained yet concise and auditable end-to-end policies.

AP-Verifier, Delta-net, and APKeep AP-Verifier [147] introduced the idea of splitting the header space into disjunct atoms (*atomic predicates* – AP) as basic modeling building blocks and as labels for highly performant verification. Network rules are modeled as sets of atoms and networks as graphs. Now, analysis can be performed using highly efficient graph and integer set algorithms. The approach shows great performance and scalability for workloads like the Stanford or Internet2 networks. Further, *Delta-net* [60] limits itself to IPv4 routed networks and incorporates the concepts of atoms and of locally bounded analysis for configuration updates through the so-called *delta-graphs*. This way, the tool achieves sub-millisecond reverification runtimes for updates of networks with hundreds of millions of forwarding rules. Finally, *APKeep* [156] combines several concepts from previous approaches, i.e., [147, 81, 60], and achieves very low reverification runtimes for network updates. In addition to atoms ([147]) which decompose the header space, they offer a flexible framework to model and compose network functions based on minimal and reusable modeling elements. Further, fast analysis is achieved through equivalence classes ([81]) and delta-graphs ([60]) resulting in outstanding performance where a network update is processed in sub-milliseconds if not few microseconds.

In comparison, FaVe offers splendid performance (cf. Section 7.2) while providing support for complex device models like stateful packet filters and dynamic protocols like IPv6. The other tools, on the other hand, were not evaluated for more complex header spaces than regular IPv4 network quintuples ([147, 156]) or even only IPv4 prefixes ([60]). Further, FPL offers accessible high-level policies whereas the other tools do not provide means ([147, 60]) for policy specification or offer only technical concepts like predicates and graph-traversing state machines to administrators who often lack academic training ([156]).

Control Plane Verification Tools

Tools like *Batfish* [46], *Minesweeper* [12], *Tiramisu* [2], or *Lightyear* [140] 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. Nevertheless, they are limited to few packet headers like flow quintuples (i.e., protocol, source and destination addresses and ports), VLAN tags, or MPLS labels as they specialize on routing analysis. Since they do not support firewalls or other complex devices, we do not consider these tools to be suitable for network security verification and we omit them in Table 3.1.

3.3 Discussion: Methods to verify IPv6 Networks

Network verification is a broadly researched area. Especially, the class of data plane checkers offers highly performant approaches. Domain specific but reasonably generic approaches like NetPlumber, VeriFlow, and APKeep offer excellent scalability, approaches that are specific to data centers like SecGuru or RCDC proof to be highly scalable and practicable, and approaches that are limited to IPv4 routed networks like Libra and DeltaNet show extreme scalability. In essence, there is a large variety of approaches to choose from. Nevertheless, the state of the art IPv6 protocol and its dynamic nature is a less frequently explored aspect of the field, i.e., only in [94], [37], and [150]. Next, we will discuss the specific challenges of verification regarding IPv6.

The different approaches from literature achieve verifiability by implementing one or more optimizations that can be classified into four dimensions – namely *Packet Encoding*, *Traffic Classes*, *Locally Bounded Analysis*, *Divide and Conquer* which – to some degree – might also be applicable for IPv6:

Packet Encoding Any approach on network verification needs to represent packets or packet sets in some way. Hence, an efficient encoding of packets in terms of memory consumption and processing runtime is key to overcome the inherent challenges of network verification. For instance, approaches that build upon classical model checking techniques encode packet bits as boolean variables for SAT solving or BDDs. E.g., the IPv6-ready tool *ad6* from our prelimary study in Section 1.4 used a SAT encoding. Also, HSA wildcard expressions

and header space objects aim to provide efficient packet set encodings and set operations.

In comparison to IPv4, this optimization should be also applicable to the verification of IPv6 networks if the amount of header bits is bounded. This has been shown in our preliminary study. Nevertheless, the growing amount of header bits in IPv6 degrades performance significantly.

Traffic Classes Some approaches try to reduce the search space by trying to identify sets of packets with identical behaviour in a preprocessing step first, e.g., network segments that can be described by a common prefix. Then, they perform analysis on these traffic classes instead of the individual packets. For example, VeriFlow [81] and AP-Verifier [147] apply this technique.

Compared to IPv4, the larger IPv6 addresses do not necessarily increase the number of traffic classes. But, they allow a more fine grained partitioning of networks by administrators and, therefore, over time an increase of traffic classes can be expected in real-world networks. This would also exacerbate formal verification. Hence, without further evaluation using real-world workloads, this technique's applicability to IPv6 networks has not been investigated to the best of our knowledge.

Locally Bounded Analysis This technique is based on the observation that changes to networks typically have local rather than global effects. The central idea is to identify dependencies and perform analysis as locally as possible in order to minimize effort when verifying upon configuration changes. For instance, NetPlumber [79] follows this approach by analyzing intra and inter table dependencies to calculate the Plumbing Graph and by re-propagating only affected traffic flows.

To some degree, this method works orthogonically to the challenges imposed by IPv6. Since it aims at rapid reverification upon configuration changes, its effect on the initial analysis is limited which is crucial for compliance verification. Nevertheless, throughout continuous reverification, also the verification of IPv6 networks should benefit from this technique.

Divide and Conquer The main idea behind this strategy is to divide networks into closed parts in a preprocessing step and, then, to apply analysis on these in parallel. Often, these parts are simpler than the global model which further simplifies and speeds up the indiviual analysises. For example, Libra [155] and SecGuru [14] implement this strategy. Also, we implemented and evaluated a parallelized version of FaVe (cf. Section 7.2.1).

3.4 Summary

Our literature review of the related work reveals three central findings. First, the verification of IPv6-enabled networks is rarely investigated and, if, the results are not convincing in terms of performance and scalability. Second, the naïve adoption of an IPv4 tool did not succeed in our case of ad6 [94] which follows the approach of Jeffrey and Samak [73]. Third, tools that are based on verifiers optimized for the network domain tend to scale much better than generic model checkers. Particularly, data plane checkers are highly performant and scalable. We will incorporate these findings in the remainder of this thesis.

4 High-Level Policy Specification

In order to verify a network configuration's compliance with organizational security requirements, security officials and administrators need to specify said requirements formally. As will be discussed in Section 4.2, current approaches for policy specification fall short for practical usage. In this chapter, we enable security officials to abstractly specify reachability policies in terms of a formal language: the declarative *FaVe Policy Language* (FPL, first described in [24]). FPL offers capabilities to concisely yet human-understandably express organizational entities, e.g., departments, roles, or services, in a hierarchical model and to specify their permitted communication relations. Networks can be checked for compliance with FPL policies using our verification framework *FaVe* which is presented in Chapter 6.

In literature, the term *high-level* policy is rarely defined resp. it is introduced only loosely, e.g., in [134]. Though, we need a proper definition in order to analyze existing approaches and guide FPL's language design. Therefore, we define the term as follows for this work[1]:

> A high-level policy language enables an abstract, concise, unambigious, and expressive specification of entities and their interactions.

From this definition, we derive the following requirements which guide FPL's design:

R1 *Abstraction* from the technical implementation of the policies.

R2 *Reachability* between entities as the most basic kind of policy.

R3 *State Specifications* that enable expressive policies.

R4 *Hierarchies* of entities in order to achieve conciseness.

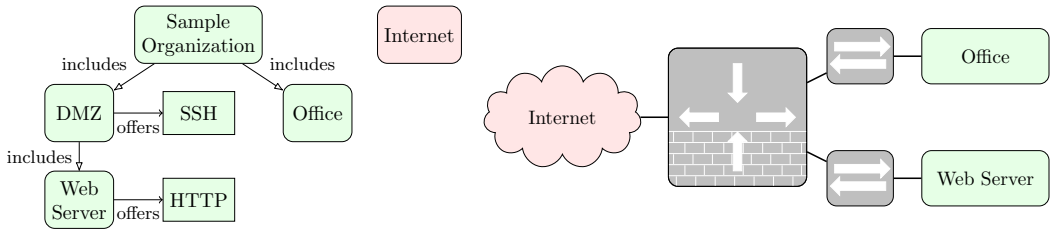
R5 *Conflictless Policies* which are achieved through unambiguity.

[1] Some work on network governance, e.g., [25], uses the term *intent* instead of *high-level policies*. In this work, we prefer the latter since it contrasts more clearly from *low-level* policies, e.g., firewall rule sets, which are often referred to as *policies* interchangeably in literature on network security.

The remainder of this chapter is structured as follows. First, we introduce FPL which consists of an organizational inventory and operator based rule specifications. Then, we present the state-of-the-art and discuss their differences with FPL.

4.1 The FaVe Policy Language

Previous approaches on network verification 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. FPL's main idea lies in the decoupling of organizational policy specification and technical implementation details, e.g., VLAN, IP addresses, ports, etc. This separation improves the understandability by auditors without a technical background. Also, security officials and administrators are enabled to discuss compliance matters in business rather than technical terms which reduces their semantical gap. Additionally, the separation of organizational roles and configuration details helps the stakeholders to focus on their respective strengths and enables an independent evolution of compliance rules and their actual implementation. I.e., it is not necessary to change the security policy when the network configuration is updated, e.g., when all web servers are migrated to another address space or from IPv4 to IPv6. In turn, if compliance mandates changes of the security policy, e.g., due to risk management, the verification report shows whether any updates to the network configuration are necessary to comply with the changed requirements. The benefit of FPL's approach is similar to the benefit of RBAC for user access – moreover, FPL may be integrated into RBAC frameworks that follow NIST standardization [123] as discussed in Appendix A.1.2.



(a) Inventory **(b)** Network

Fig. 4.1.: An example of an organization's inventory (left) and the organization's network topology (right).

In FPL, a *policy* consists of a set of *policy rules* that describe the allowed communication relations between organizational entities. E.g., departments, roles, or services.

We model the organization in terms of a hierarchically organized inventory that maps the entities' names, e.g., a research department, to their technical implementation in the network, e.g., the IPv4 address range 172.16.3.0/24.

For example, Figure 4.1a shows a simple inventory representing an organization consisting of an office and a DMZ for external services. In addition, the Internet as external entity is represented as well. The organization's network is shown in Figure 4.1b. It consists of a central firewall, two switches for the office network and the DMZ, and two hosts – a web server in the DMZ and a workstation in the office network. In our example, policy rules are prosaically expressed as follows:

(P1) "Public DMZ services may be accessed generally."

(P2) "The administration of DMZ servers may be performed through SSH access from office machines."

(P3) "Office machines may access the Internet."

(P4) "All other communication is forbidden."

We will use this example in the following sections to illustrate the inventory and policy specifications with FPL.

4.1.1 Inventory Description

As stated, the policy rules are defined over organizational entities while the verification by FaVe uses the network's topology and the device's configurations. To close this gap, we need to specify an inventory that maps the organizational entities to their technical implementation. Our approach is inspired by common network administration tools like Ansible [8] and Puppet [117]. This design decision aims at a high usability for administrators who are likely responsible for the inventories' creation and maintenance. In Appendix A.4 we provide a formal grammar for FPL.

FPL offers two language features to express and order organizational entities: *roles* and *services*. Services represent applications that are accessed by users in order to fulfil the organization's business processes. In the network, they map to transport layer protocols and ports, e.g., HTTPS on TCP port 443 or DNS on UDP port 53. Roles represent organizational units like departments, e.g., accounting or R&D, or user roles, e.g., clerks or developers. In the network, they map to entities like single machines, groups of hosts, or even complete subnets. Roles may offer services and

can hold attributes like IP addresses, VLANs, or domain names. Due to this design decision FPL fulfils the requirement for abstraction (R1).

Roles can be aggregated by abstract roles called *super roles*. These, in turn, behave like normal roles as they can hold technical attributes and offer services. When defining policies, one may refer to roles, super roles, and, optionally, their offered services. Due to this design decision FPL fulfils the requirement for hierarchical policies (R4).

For instance, the following listing defines the our example organization's inventory which was depicted in Figure 4.1a:

```
1  describe service HTTP
2      protocol = 'tcp'
3      port = 80
4  end
5
6  describe service SSH
7      protocol = 'tcp'
8      port = 22
9  end
10
11 describe role Office
12     description = 'Hosts of the office network.'
13     ipv6 = '2001:db8::200/120'
14 end
15
16 describe role WebServer
17     description = 'Public web servers.'
18     ipv6 = '2001:db8::110/124'
19     offers HTTP
20 end
21
22 describe role DMZ
23     description = 'Hosts of the DMZ network.'
24     ipv6 = '2001:db8::100/120'
25     offers SSH
26     includes WebServer
27 end
28
29 describe role SampleOrganization
30     description = 'All hosts of the organization.'
31     includes DMZ
32     includes Office
33 end
```

First, the inventory defines two TCP services – HTTP and SSH – which listen on their respective default ports. Then, the different roles are described:

Office The *Office* role that represents all office hosts and comprises an IPv6 address range.

WebServer The *WebServer* role represents all public web servers run by the organization. It comprises an IPv6 address range and offers the HTTP service.

DMZ The *DMZ* role groups all roles with services that are meant to be publicly accessible. It includes the *WebServer* role and comprises an IPv6 address range that also includes the web servers' addresses. In addition, it offers the SSH service for administration purposes.

SampleOrganization This super role groups all roles of the organization. I.e., it directly includes the *DMZ* and *Office* roles and, transitively, the *WebServer* role via the *DMZ*.

Internet To represent networks outside of the organization's administrative boundaries, FPL has a default Internet role with an unlimited address range for each attribute, e.g., 0::0/0 for IPv6. FPL users may specify an own Internet role in order to limit the attributes according to their needs. E.g., if their organization uses IPv6 internally but the ISP does not provide public IPv6 connectivity.

For our example, the *DMZ* does not group multiple roles and, hence, could have been omitted if the SSH service was directly attached to the *WebServer*. Though, we introduced the *DMZ* to be flexible in case of future changes, e.g., the introduction of mail servers. Moreover, the common term *DMZ* implies a better understanding of the fact that included roles offer public services. Policies can be defined based on this intent and may be more stable in the longer run.

As seen with the *DMZ* role, super roles offer the possibility to specify attributes and services that are common for all sub roles. This concept allows the definition of more concise sub roles as they do not need to list common attributes and services. In order to determine a role's factual attributes and offered services and, hence, to have a clear formal semantics of the model, we define a downstream resolution mechanism. The relation between super roles and sub roles is defined in terms of an acyclic directed graph where a super role points to an arbitrary amount of sub roles. Our implementation detects cycles in FPL inventories and refrains from further processing. Due to the model's loop-freeness, the downstream resolution mechanism can be defined unambigiously. This design decision supports FPL's goal to fulfil the requirement of conflictlessness (R5). In the following, we will describe the

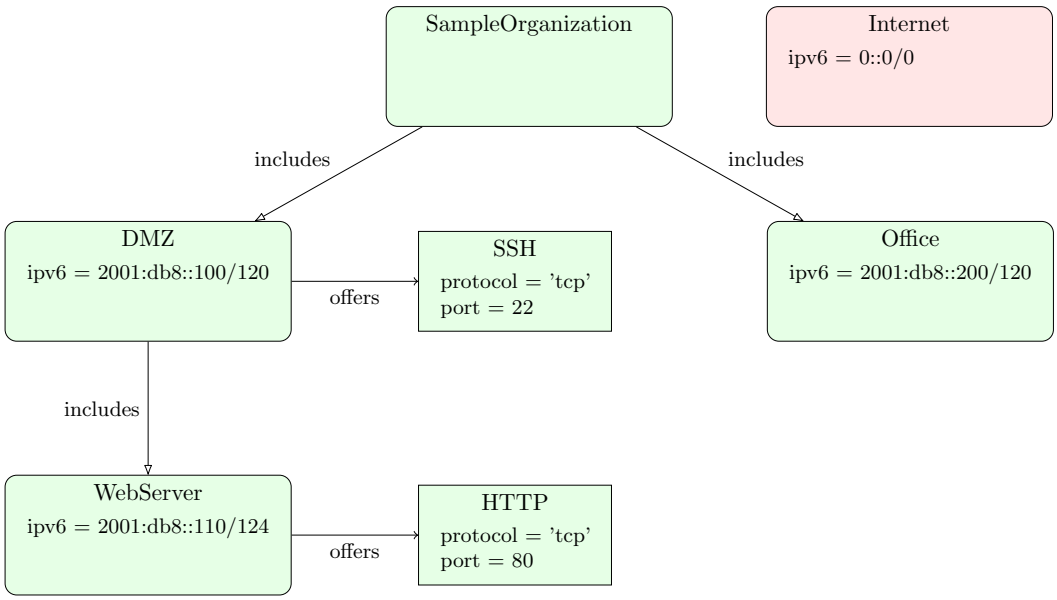
resolution mechanism prosaically and illustrated with our example. See Section 5.3 for a formal definition of the mechanism's semantics.

As a role may belong to multiple super roles there might be several root nodes, i.e., nodes without incoming edges. In turn, leaf nodes are characterized by their absence of outgoing edges. The attribute and, similarly, service resolution is performed by starting at the root nodes and by collecting all attributes and services along a path until a leaf is reached. In order to formally specify this procedure and, hence, the model's semantics, we define attributes as discrete sets. E.g., VLANs hold values from the range from 0 to 4,095 and IPv6 address ranges are subsets of 0::0/0. Thus, each role is characterized by a portion of the Header Space through the attributes they possess and inherit from super roles. If, throughout this propagation, an attribute supersedes another of the same kind, the more specific is propagated. Later, when verifying compliance (cf. Chapter 7), these leaf roles will serve as communication endpoints and their collected attributes will characterize traffic they emit. The services are propagated in a similar manner. I.e., leaf roles offer all services gathered over all incoming paths. Service traffic, then, is characterized by the role's attributes in combination with the service's attributes, i.e., protocol and ports. If we propagate a service that is accessed by a policy rule, said rule is replaced by rules accessing the propagated services instead.

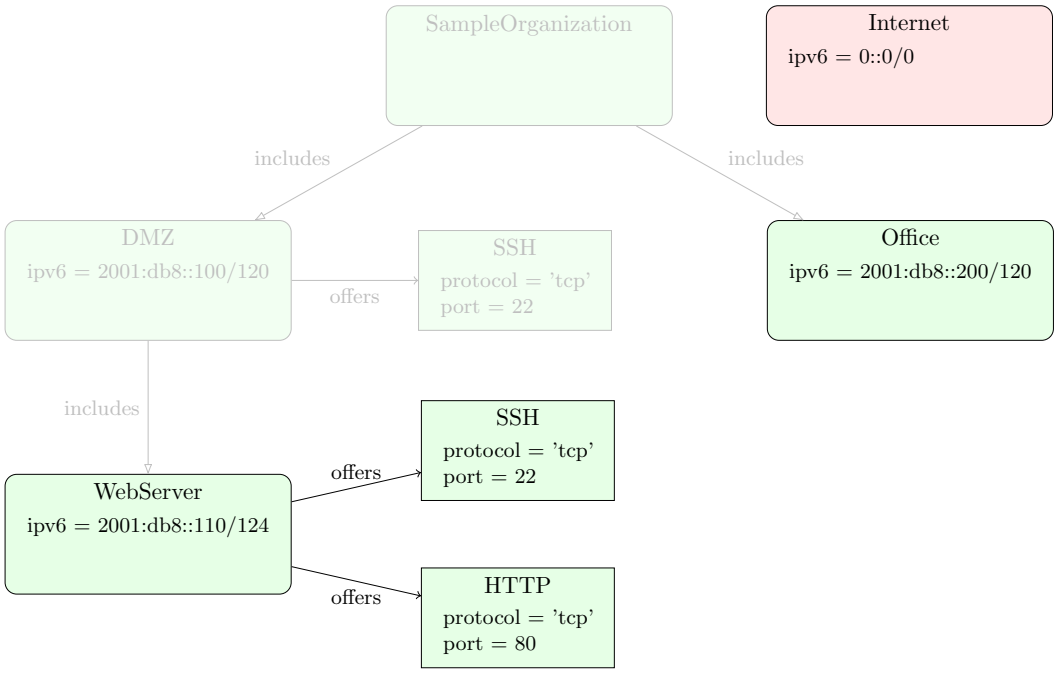
Figure 4.2a shows our example inventory before propagation while Figure 4.2b depicts its result. First, we start at the graph's roots which are the *Internet* and the *SampleOrganization* roles. Since the *Internet* is already a leaf, we can stop propagation here. The *SampleOrganization* neither comprises attributes nor offers services. Hence, there is nothing to propagate to its successors, i.e., the *Office* and *DMZ* roles. Again, the *Office* is a leaf and we can stop propagation here. The *DMZ* comprises the IPv6 address range 2001:db8::100/120 and offers the SSH service. Both are propagated to the role's successor which is the *WebServer* role. Since this role also comprises an IPv6 address range, we need to check which address range is more specific. Since the *WebServer*'s own IPv6 address range is a subset of the *DMZ*'s, it is kept. The propagated SSH service is simply added to the *WebServer*. As this role is a leaf node, we can stop propagation and since there are no further nodes to process, the propagation procedure ends as well.

4.1.2 Policy Specification

After modeling the organization through the inventory, we are now able to define policies based on the organizational terms without further knowledge of the technical



(a) Before propagation.



(b) After propagation.

Fig. 4.2.: FPL's attribute and service propagation.

implementation. In FPL, a policy is described as a set of reachability rules between roles and services. 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. Along with the unambigious

resolution of attributes and services, this design decision leads to the fulfilment of the requirement for conflictless policies (R5).

All policy rules are of the form:

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

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

`--->` unidirectional reachability

`<-->` bidirectional reachability

`<->>` stateful reachability

Unidirectional reachability A `--->` B means that traffic may flow from A to B but not from B to A. 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, i.e., the subject) reaches the recipient (right hand, i.e., the object) which in turn only sends packets reactively. The recipient is not allowed to 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. FPL's operators fulfil the requirement for reachability policies (R2) and, in addition, the stateful operator fulfils the requirement for state specifications (R3).

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., $B.S$ where a role B 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: $B.^*$. 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 some rule are reported as this indicates possible disruptions of business continuity.

For our example, an implementation of the policy rules (P1) to (P4) could look like this:

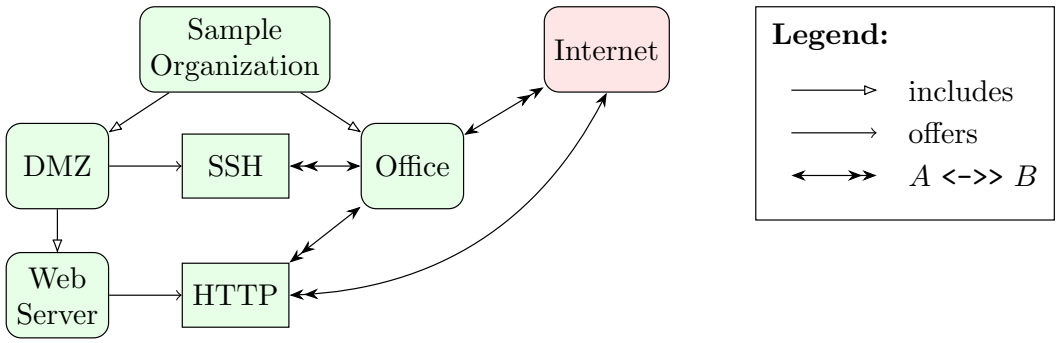


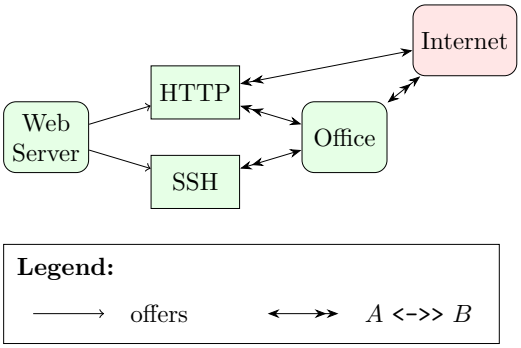
Fig. 4.3.: Graphical representation of the policy over the inventory in our example.

```
1 describe policies (default: deny)
2     Internet <->> WebServer.HTTP
3     Office <->> WebServer.HTTP
4     Office <->> DMZ.SSH
5     Office <->> Internet
6 end
```

The default policy realizes (P4) by denying all access except those allowed by other rules. (P1) allows access to public services which is HTTP and this is permitted by the second and third line. Rule (P2) allows SSH administration for office machines which is implemented by the fourth line. Finally, the rule (P3) permits Internet access for office machines. It is allowed by the fifth line.

This policy reflects the prosaic policy rules very well since each prosaic rule maps directly to a set of FPL rules while these sets are distinct. A downside shows in the need to split (P1) into two distinct FPL rules. While this seems to be acceptable for this small example, auditing the mapping of large prosaic policies to larger FPL policies becomes more challenging. Hence, we would, if possible, prefer a more concise FPL policy and a more direct, i.e., ideally one-on-one, mapping of prosaic rules to FPL rules.

In Figure 4.3, we show the permitted policy accesses concerning the inventory as seen in Figure 4.1a. Since the propagation procedure pushes attributes and services to the inventory graph's leafs, we can omit the *SampleOrganization* and *DMZ* roles in the representation as depicted in Figure 4.4a. An even more precise representation is given by the reachability matrix in Table 4.4b. For each subject-object relation, the matrix lists all access conditions, e.g., restrictions to certain services or stateful return traffic only. Notably, the *Office* role may only be reached by response traffic from the *Internet* or from the *WebServer*. Also, the Internet may only reach the web server via HTTP while SSH access is not allowed. The cell of the builtin Internet role is left blank as imposing policies on the whole Internet is infeasible.



(a) Graphical representation of the example policy over the inventory after attribute and service propagation. The *SampleOrganization* and *DMZ* super roles are omitted in this representation since they are not refered to by policy rule anymore.

Source\ Dest.	Internet	Office	WebServer
Internet	-	(✓)[3]	(✓)[1]
Office	✓	✗	(✓)[2]
WebServer	(✓)[3]	(✓)[3]	✗

(b) Reachability matrix of the policy with communication sources as rows and targets as columns. The symbols denote ✓ unconditional, (✓) restricted, and ✗ forbidden traffic flows. The types of restrictions marked by an upper index are the following: [1]HTTP traffic only, [2]HTTP or SSH traffic, [3]stateful return traffic only.

Fig. 4.4.: Graphical and tabular representations of the policy over the inventory after propagation in our example.

Both representations have their strengths. For example, the graphical representation gives a good overview of the general access relations between roles and services. The reachability matrix, on the other hand, explicitly lists inaccessabilities and in-depth conditions for communication flows. Both help when administrators debug mismatches between the policy specification and the results of compliance verification.

Since the SSH service has been propagated to the *WebServer*, we have two access rules to the *WebServer*'s services. Or, all services offered by the *WebServer* are accessed by the *Office*. This allows an optimization of the policy by merging the access rules to a wildcarded rule:

```
1 describe policies (default: deny)
2     Internet <–>> WebServer.HTTP
3     Office <–>> WebServer.*
4     Office <–>> Internet
5 end
```

While being more concise than the original policy, merging the *Office*'s SSH and HTTP accesses into a single rule does not improve the auditability and manageability of the policy. First, the intent of (P2) is not directly reflected since SSH access to the *DMZ* is not specified directly and only granted indirectly through the *WebServer* role. This level of indirection restrains auditability. Second, if the inventory changes,

this can have hidden impacts on the accesses permitted by the wildcard. E.g., if we add some service to the *DMZ* or the *SampleOrganization* roles – i.e., upstream roles – then the propagation in conjunction with the wildcard rule automatically grants access to this new service. Hence, manageability is impacted since the administrator needs to take possible propagations into account to prevent unwanted implied access permissions.

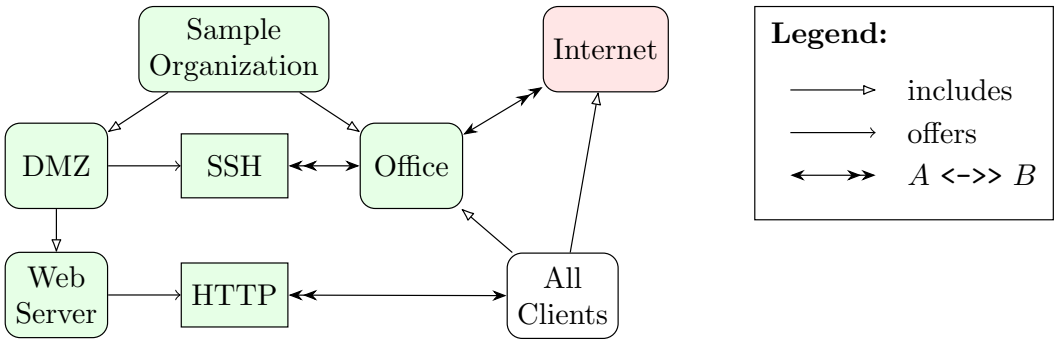


Fig. 4.5.: Graphical representation of the policy over the inventory extended by the *AllClients* role in our example.

Therefore, to improve auditability and policy maintanence, we propose another approach. If we introduce an artificial role that groups all clients, we could implement public HTTP access as stated by rule (P1) more precisely. We define the new role *AllClients* as follows:

```
1 describe role AllClients
2     description = 'All internal and external clients.'
3     includes Internet
4     includes Office
5 end
```

This new role includes the *Office* machines as well as the Internet and, as can be seen in Figure 4.5, it allows the specification of (P1) in a single rule:

```
1 describe policies (default: deny)
2     AllClients <–>> WebServer.HTTP
3     Office <–>> WebServer.SSH
4     Office <–>> Internet
5 end
```

AllClients directly reflects the intent of (P1) and helps to stabilize the policy regarding future changes of the inventory. Stating the intent explicitly through the new role's name improves auditability as we now have a one-on-one mapping of prosaic policy rules to FPL rules. In addition, manageability is improved. For example, if some role that comprises client machines is introduced, it can be simply incorporated into

the *AllClients* role. Hence, the policy does not require any further changes and an auditor just needs to check if the roles included by *AllClients* are legit.

We have seen that the benefits of grouping roles in terms of auditability and manageability already occur for this small example. The gain is even more evident for larger examples like the policy for the UP benchmark given in Appendix A.11 with only 33 FPL rules in comparison to 1,035 `iptables` rules. The policy rules are much easier to understand and analyze. The UP benchmark will be used in the evaluation in Section 7.2.1.

Default Self Reachability for Roles

A major challenge when securing networks is *lateral movement* within a network segment which occurs if an attacker surpassed the network's firewall, compromised a host, and seeks for further targets. A possible strategy is to establish micro segments which are not separated by a central firewall but a decentral architecture, e.g., by instrumenting host firewalls. Concerning the definition of compliance rules the question occurs whether hosts within a segment should be allowed to reach each other or if a more fine grained policy should be enforced. For this purpose FPL supports two modes of operation: *loose* or *strict*. With the loose mode for all roles A the reachability A `--->` A is allowed implicitly whereas the strict mode does not allow communications within a role. This allows to carefully craft exceptions if certain communications within a role should be allowed, e.g., for a web cache communicating with a web server within a DMZ. In our example we chose the strict mode as a more secure default which leads to ✗ on the rechability matrix' diagonale.

4.2 Related Work: Policy Specification Languages

The ability to express intent concerning the desired state of network security is fundamental for the understanding, assessment, and configuration of networks. High-level policy languages may fill the gap between prosaic, non-technical compliance descriptions and low-level security configuration. In Table 4.1, we compare different approaches and tools from literature with FPL based on the requirements for high-level specification languages stated in this chapter's introduction, their ability to generate security configuration, and their public availability:

Tool	Technical Abstraction	Reachability	State Specifications	Hierarchical Policies	Conflictless Policies	Configuration Generation	Public Code
Ponder [30]	✓	✓		✓			
Cuppens et al. [29]	✓	✓		✓		✓	
FML [59]		✓	✓	✓			
ForestFirewalls [119]	✓	✓			✓	✓	
TrustSec [21]	✓	✓				✓	
FirewallBuilder [106]	✓	✓		✓		✓	✓
Yin et al. [149]	✓	✓					
INTPOL [20]		✓				✓	
FPL [96]	✓	✓	✓	✓	✓	✓	✓

Tab. 4.1.: Overview of different tools for security policy specifications.

Technical Abstraction (R1): Policies are formulated based on proper abstractions instead of technical criteria like IP addresses or port numbers.

Reachability (R2): Policies are specified in terms of allowed or forbidden communication patterns which are based on the reachability between entities over networks.

State Specification (R3): The handling of stateful behaviour can be expressed explicitly, i.e., communication patterns can be stated. For instance, services may not respond without previous requests or communication must be strictly unidirectional.

Hierarchical Policies (R4): Entities can be grouped and ordered hierarchically.

Conflictless Policies (R5): The inability to specify policies with conflicting intents, e.g., different handling of a packet depending on the order of policy rules.

Configuration Generation: Policies are used to generate security configuration.

Public Code: There is a publicly available implementation.

Detailed Description and Discussion

In the following, we present the different approaches and tools in detail and, also, we discuss their similarities and differences with FPL.

Ponder This approach [30] aims at the specification of security policies that map onto different access control implementations – including firewalls. Nevertheless, its focus lies on operating system level entities like users, processes, or files rather than on networked entities like hosts or services.

Ponder offers an abstraction that groups entities in a hierarchical domain tree which is analoguous to Microsoft's Active Directory. These domains can be used in policy rules as subjects and objects. In Ponder policies, it is uncommon to use technical details in policy rules. Due to the language's focus on operating system entities, Ponder rules provide according actions, e.g., create, read, write, delete. This way, network reachability is not specified explicitly but rather implicitly since access to a remote object requires network connectivity in the first place.

Ponder's rules may be specified as deny and allow rules which can be mixed arbitrarily in a single rule set. Hence, conflicts between the rules are possible. Ponder has a built-in conflict detection and offers conflict resolution using so-called meta policies. Further, Ponder allows the specification of entity states in rules. For instance, the object may only be accessed if the subject is in a certain state, e.g., if it is an authenticated user. Concerning network security policies, the individual entities' states are of less importance since we are more interested in the state of their relation, e.g., responses are only allowed as answers to previous requests. With Ponder, this relation cannot be expressed in terms of the subject's and object's state attributes.

In contrast, FPL puts an emphasis on network security policies and networked entities. In FPL, reachability is expressed explicitly, state specifications are tied to the relation between entities and encoded in FPL's operators, and conflicts are prevented by design.

Cuppens et al. This approach follows the notion of *Organization-based Access Control* (OrBAC [75]) in order to specify security policies and generate firewall configurations, e.g., IPTables rule sets. In OrBAC, access permissions are expressed by tuples of *subjects*, i.e., hosts, that perform *actions* on *objects*, i.e., services. They are abstracted through *roles* that group subjects, activities that group actions, and views that group objects. With OrBAC, roles can be assigned to organizational entities,

e.g., departments, that can be structured in a tree-like hierarchy where a child node inherits all permissions from its parent node. Furthermore, OrBAC allows to flexibly specify additional prohibitions and obligations but these may introduce conflicts, i.e., forbidden obligations. The authors state that these conflicts need to be detected and resolved separately.

In contrast, FPL avoids conflicts by design, offers stateful security specification and is publicly available.

Flow Management Language (FML) The *Flow Management Language* (FML) [59] 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. Conflicts between rules of differing actions can be detected and resolved automatically. The following example from the paper illustrates FML's syntax and semantics:

$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req)$$
$$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow blacklist(U_s)$$
$$blacklist(eve)$$
$$blacklist(mallory)$$
$$\ldots$$

Rules are specified as predicates with a head which includes a list of variables that can be used in the rule's body. The example realizes a blacklisting of users, i.e., all flows are allowed unless sent by a user (U_s – the U resp. s denote the user resp. source) listed in a blacklist. In this case, the blacklist consists of eve and $mallory$ which are specified as facts. Other variables, for instance, include the source address A_s, the destination host H_t, the protocol $Prot$, or a flag Req that indicates if the flow is an initializing request. Since the predicates $allow$ and $deny$ may conflict due to their identical list of variables, FML's conflict resolution applies a secure-by-default precedence which values denying higher than allowing. Also, FML allows to specify a cascade of policies, e.g., $P_1 > P_2 > P_3 > ...$, that also introduce precedence. Decisions upon flows in policies that occur early in the cascade have a higher priority than those from later policies upon the same flows, e.g., P_1 over P_3. In fact, allow listing can only be expressed using two policies: a higher prioritized one for allowing flows and a lower prioritized one that denies everything. E.g., the following policies P_1 and P_2 with $P_1 > P_2$ implement an allow listing:

```
# P1 allows specific flows
```
$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = false$$
$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow H_t = 1.2.3.4 \wedge Prot = http \wedge Req = true$$
$$\ldots$$

```
# P_2 denies all flows by default
deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req)
```

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.

ForestFirewalls Another approach is given by *ForestFirewalls* [119] 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 [67] standards to group assets (*zones*) and specify their allowed reachabilities (*conduits*). Zones are required to be mutually exclusive, i.e., no asset may be assigned to more than one zone. Also, policies always follow an allow-listing approach, i.e., they have an implicit default deny rule and all other rules introduce exceptions by allowing specific traffic. This leads to the fact that the order of rules within a policy does not matter. The following example from the paper illustrates two simple policies:

```
Policy Company_policy {
    Z1 -> Z2 : https, dns;
    Z2 -> Z3 : http, ftp, dns;
}
```

The first policy allows HTTPS and DNS requests from the zone Z1 to Z2 while the second policy allows HTTP, FTP, and DNS requests from Z2 to Z3. Based on the assets that comprise the zones and the policies that specify service accesses, the authors generate security configuration for enforcement. The tool aims to check equivalence of high-level policies and conformity for industry best practices.

ForestFirewall comprises an operator similar to FPL's stateful reachability operator, but it does not cover explicit state specification, e.g., it is not possible to specify a strictly unidirectional communication pattern. Hence, policies are limited to specify service accesses instead of network coupling. In contrast, FPL's ability to group roles hierachially offers more flexibility and the formulation of more concise security policies than the zone concept.

Other Approaches Cisco's *TrustSec* [21] offers a matrix to describe reachability policies between fine grained segments for their *Software Defined Access* products. Assets can be grouped freely and access between these labeled segments can be used as sources or targets of traffic to be granted or denied. Based on the access matrix and associated assets, security configuration is generated and enforced. While helping administrators to keep an overview of their policies, access matrices grow large quickly for large and complex networks with diverse assets. Hence, 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. Further, since assets in TrustSec may be placed in multiple segments and access may be granted for one segment but denied for another, conflicting rules for individual assets may arise which needs to handled separately. Contrary, FPL avoids such conflicts by design.

Another approach is offered by the open source tool *FirewallBuilder* [106]. 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. Further, all policies are stateful by design and this behaviour cannot be turned off. Also, conflicts between rules are neither avoided nor detected comprehensively. FPL, on the other hand, avoids such conflicts by design (cf. Section 8.1).

Further, Yin et altera [149] introduced an accessible policy language based on tables that map technical attributes, e.g., IPv6 prefixes or protocol and port, to abstract names, e.g., DMZ or HTTP. A Policy is a list of prioritized rules that, in turn, have a fixed format, i.e.:

$$s_i : \text{ if } S_i \text{ in } R_{i1} \text{ from } R_{i2} \text{ then } A_i$$

where

- s_i is the rule's index in the rule set,
- S_i is a service name, i.e., protocol and port,
- R_{i1}, R_{i2} are region names, i.e., IPv6 prefixes, and
- A_i is an action, i.e., permit or deny.

FPL, in contrast, offers more concise and conflictless policies through its role hierarchy and propagation mechanisms for attributes and services. Since Yin et altera's policy language does not require regions to be distinct and actions can bei mixed freely, conflicts may occur – as opposed to FPL which has been designed to be

conflictless. Furthermore, FPL allows explicit state specification and is used for configuration generation.

The tool INTPOL [20] offers a policy language aimed at the specification of invariants for intent-based networking with SDN – particularly, for intents of the ONOS SDN controller. The language uses network quintuples and MAC addresses as primary specification blocks and rules may forward, drop, or modify packets. They use these policies in order to check conformance of a formal model of the SDN controller's low-level SDN rules and network topology with the LTL model checker NuSMV. They conduct pairwise rule comparisons and compare their performance with FIREMAN [153] (see Section 8.2 for details) – though, it remains unclear if they reimplemented the tool or extracted FIREMAN's numbers from the original paper. In contrast, FPL offers flexible role based abstractions from technical attributes, hierarchical roles for concise rule sets, and propagation mechanisms for conflictless policies.

Finally, an IETF draft for a *Security Policy Specification Language* (SPSL) [27] has not been standardized.

Secondary Features Often, the related approaches offer features beyond the set that we consider central for the specification of network security policies. In the following, we give an overview of these secondary features and their support in the other approaches:

- **Waypoints:** FML allows the specification of certain network nodes, e.g., IDS/IPS systems, that must be traversed during access. From a security policy point of view, it is not important which nodes enforce the specification but the fact that it is enforced somehow and at all times. Hence, modeling the network node's security functionality in a verifiable manner is sufficient which degrades the specification of waypoints to a mere placeholder until a proper modeling has been implemented. The specification of waypoints is more useful in terms of network operations, e.g., by requiring certain traffic classes to pass traffic optimization functionality.

- **Avoidance:** Opposing to waypoints, avoidance allows the specification of network nodes that must not be traversed and is supported by FML. Like waypoints, avoidance is more useful in network operations, e.g., by requiring certain low-priority traffic classes to avoid known bottlenecks and take less frequented but slower routes.

- **Rate Limiting:** For instance, only a certain amount of requests, e.g., SSH accesses, is allowed in order to save IT resources and potentially prevent password-guessing or denial-of-service. Saving resources is not a primary goal of security policy but rather a question of economic IT operations. Rate limiting may be specified in FML and FirewallBuilder.

- **QoS:** Quality of Service, i.e., a service's degree of availabity, is not a security but rather a usability need. For security policy and operations, the general availability, i.e., reachability, is central and temporary outages are a matter of IT operations and business continuity. For instance, if a service must be available, the security rules must not block access. An unreliable network or a service outage, on the other hand, is not a question of security policy. QoS specifications are offered by FML.

- **NAT:** As discussed in Section 1.8, we do not regard NAT as an important security feature in modern networks. Though, FML and FirewallBuilder include NAT in their policy specifications.

Conclusion As discussed, none of the previous approaches fulfils all requirements for high-level policy specifications. FML does not offer proper abstractions, e.g., security groups or roles, from technical details like IP addresses or ports. Further, TrustSec and ForestFirewalls lack hierarchical ordering of security groupings. In addition, Ponder, FirewallBuilder, ForestFirewalls, and TrustSec are unable to specify state handling explicitely and there are no public implementations available. Finally, Ponder, FML, TrustSec, and FirewallBuilder are unable to avoid conflicting policies by design.

4.3 Summary

The *FaVe Policy Language* (FPL) enables security officials and administrators to specify security policies which answers our first research question stated in Section 1.5. The main concept lies in the division of the specification of organizational entities in a hierarchical *inventory* and the definition of access *policy rules* using these entities instead of technical details like IP addresses. This separation of duties offers a good auditability, stable security policies, and – as shown in Appendix A.1.2 – full compatibility with NIST RBAC. The specification of security using FPL is essential in all three phases of our compliance management process, i.e., it enables the initial

verification in UNDERSTAND, the repeated reverification in ADJUST as well as CONTROL, and the generation of security configuration in CONTROL.

5 Domain-oriented Header Space Analysis

In order to verify compliance of a network's configuration with a security policy specification, it is necessary to support network administrators with accessible and flexible modeling. In Section 2.2 the concept of *Header Spaces* has been explained as an approach to describe configurations and to enable formal verification with efficient domain specific solvers, i.e., HSA and NetPlumber. As seen, previous approaches like HSA [79] focused on a high performance of the analysis rather than the usability of the modeling process and the understandability of the verification results. In addition, modeling is limited to static sets of header fields which excludes dynamic protocol elements like IPv6 extension header chains.

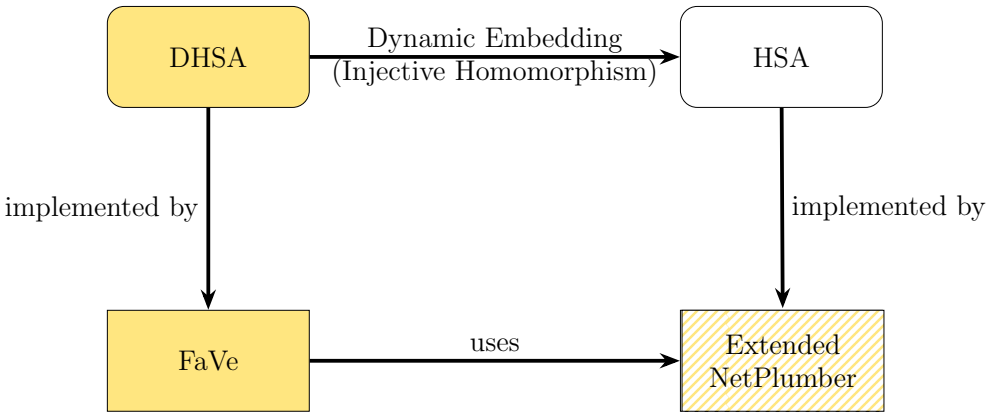


Fig. 5.1.: Overview of the general concepts, their implementations, and inter-relations.

To overcome these shortcomings, in Section 5.1, we introduce a *domain-oriented* notion of the *Header Space* called *Domain-oriented Header Space Analysis* (DHSA[1]). DHSA offers concise, flexible, and human-readable representations of sets of packets and common operations like intersections and unions. As depicted in Figure 5.1, DHSA dynamically embeds in HSA via an *injective homomorphism* and is imple-

[1]Domain-oriented Header Space Analysis and Domain-oriented Header Space Algebra share the same acronym. Since this has already been the case with HSA in the related work by Kazemian et altera, we also follow this convention in this work. We mean Domain-oriented Header Space *Analysis* when we elaborate on verification and we mean Domain-oriented Header Space *Algebra* when we elaborate on the mathematical model.

mented in our fast verification framework *FaVe* which uses an extended version of NetPlumber and will be detailed in Chapter 6.

Particulary, we introduce DHSA as follows. First, in Section 5.2, we show how to intuitively model network configurations using an IPTables rule set as example. Then, in Section 5.3, we use the concept to formally represent organizational entities as well as FPL inventories and, subsequentially, to formalize FPL's attribute and service propagation procedures. Afterwards, in Section 5.4, we prove that DHSA is a distributive lattice and finally, in Section 5.5 we show that our embedding of DHSA in HSA is an injective homomorphism.

5.1 Formalism

$\mathcal{H}^D$	Set of header field names.
h	Header field from $\mathcal{H}^D$.
$\mathcal{V}_h$	Value domain for a header field $h \in \mathcal{H}^D$.
v_h	Subset from $\mathcal{V}_h$.
$\mathcal{V}_{\mathcal{H}^D}$	Value domain for all header fields $h \in \mathcal{H}^D$.
t^h	Header field tuple from $\mathcal{H}^D \times \mathcal{V}_{\mathcal{H}^D}$.
M	Set of matches.
$\mathcal{M}$	Powerset over all matches.
R	Rule set.
r_i	Rule at position i.
m_i	Match for the rule at position i.

Tab. 5.1.: Symbols used in the formalization.

We formalize DHSA upon *tuples* of header fields and the respective value sets as basic building blocks. Different header field tuples form *matches* which can be collected in *match sets*. We define several *set operations* over header field tuples, matches, and match sets as well as some subset relations. Combined, these sets and operations form an *algebra* – the *Domain-oriented Header Space Algebra*. In Table 5.1, we give an overview of all symbols used in the formalization of DHSA.

Header Field Tuples

In general, we define DHSA on top of header field tuples t^h as basic units. This representation has two benefits. First, it allows simple conversions between machine

and human-understandable notations. Second, it offers a more flexible set notation compared to ternary bit vectors utilized by HSA (cf. Section 2.2).

Definition 1 (Header Field Tuple)**.** *A header field tuple consists of a header field $h \in \mathcal{H}^D$ from the set of header fields and a value set $v_h \subseteq \mathcal{V}_h$ from its respective value domain, i.e.:*

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

For instance, the header field tuple $(\texttt{dport}, \{80\})$ can be used to describe the subset of the Header Space comprising all packets with destination port 80. Another example would be the tuple $(\texttt{dip}, \{\texttt{2001:db0::100/120}\})$ which represents all packets with IPv6 source addresses in the range from `2001:db0::100` to `2001:db0::1ff`.

Using our tuple representation, we divide the *Header Space* into dimensions where each dimension is covered by one distinct header, i.e., $\mathcal{H}^D := \{h_1, h_2, ..., h_n\}$ with $\forall i, j \in \{1..n\}, i \neq j : h_i \neq h_j$. Corresponding, the Header Space's value domain is defined as $\mathcal{V}_{\mathcal{H}^D} := \{\mathcal{V}_{h_i} \mid h_i \in \mathcal{H}^D\}$.

In the following, we will use a Header Space consisting of the three headers *destination IP* (`dip`), *protocol* (`proto`, from the IP layer), and *destination port* (`dport`):

$$\begin{aligned}\mathcal{H}^D = \{ & \\ & h_1 = \texttt{dip}, \\ & h_2 = \texttt{proto}, \\ & h_3 = \texttt{dport} \\ \}&\end{aligned}$$

The corresponding value domains are

$$\begin{aligned}\mathcal{V}_{\texttt{dip}} &= \{\texttt{0::0/0}\}, \\ \mathcal{V}_{\texttt{proto}} &= \{\texttt{0..255}\}, \text{ and} \\ \mathcal{V}_{\texttt{dport}} &= \{\texttt{0..65535}\}.\end{aligned}$$

And, thus, the Header Space's value domain is

$$\mathcal{V}_{\mathcal{H}^D} = \{\mathcal{V}_{\texttt{dip}}, \mathcal{V}_{\texttt{proto}}, \mathcal{V}_{\texttt{dport}}\} = \{\{\texttt{0::0/0}\}, \{\texttt{0..255}\}, \{\texttt{0..65535}\}\}.$$

Matches and Sets of Matches

In order to model network rules and characterize endpoints, e.g., firewall rules or FPL roles, we define *matches* and *sets of matches* as regions in the Header Space.

Definition 2 (Match)**.** *A match m consists of a set of header field tuples*

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

with unique header field identifiers.

For any match, we require that all header fields are initialized by default. This conforms to real-world rule specifications as well as endpoint capabilities. Rules express specific matching criteria for packets while leaving all header fields that are not relevant for that particular rule unspecified. Endpoints, on the other hand, are able to emit and accept any traffic. We adapt this semantics by initializing unspecified fields with their respective value domain, i.e., $v_h = \mathcal{V}_h$.

For example, the match

$$m = \{(\texttt{dip}, \{\texttt{0::0/0}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{22\})\}$$

describes all SSH packets.

Definition 3 (Empty Match)**.** *A match m is empty if at least one of its header field tuples is empty, i.e., $\exists(h, v_h) \in m : v_h = \emptyset$. We denote empty matches as $\emptyset$.*

Definition 4 (Match Set)**.** *We denote a set of matches as $M := \{m_1, m_2, ...\}$ where each $m_i \in M$ is a match with $i = 1..|M|$.*

Semantically, a match set unions the packet sets represented by the contained matches.

For instance, the match set

$$\begin{aligned} M = \{ & \\ & \{(\texttt{dip}, \{\texttt{2001:db8::1}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{80\})\}, \\ & \{(\texttt{dip}, \{\texttt{2001:db8::4/126}\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \{53\})\} \\ \} & \end{aligned}$$

comprises HTTP packets destined to the host at `2001:db8::1` and DNS packets destined to the IP address range `2001:db8::4/126`.

Further, we need set operations over matches, i.e., intersection and union. Also, a subset relation is necessary. These are to be used when we propagate attributes in FPL inventories, verify compliance, or find firewall anomalies.

Definition 5 (Powerset over all Matches)**.** *Finally, we denote the powerset over all matches as:*

$$\mathcal{M} := 2^M.$$

Intersections

Definition 6 (Tuple Intersection)**.** *For tuples of the same header field h, the intersection operation is defined as follows:*

$$\cap : (\mathcal{H}^D \times \mathcal{V}_{\mathcal{H}}^D) \times (\mathcal{H}^D \times \mathcal{V}_{\mathcal{H}}^D) \to (\mathcal{H}^D \times \mathcal{V}_{\mathcal{H}}^D)$$
$$t_1^h \cap t_2^h = (h, v_1) \cap (h, v_2) := (h, v_1 \cap v_2)$$

For example intersecting the tuples

$$(\texttt{dip}, \{\texttt{2001:db8::100/120}\}) \cap (\texttt{dip}, \{\texttt{2001:db8::110/124}\})$$

results in

$$(\texttt{dip}, \{\texttt{2001:db8::110/124}\}),$$

whereas,

$$(\texttt{dip}, \{\texttt{2001:db8::100/120}\}) \cap (\texttt{dip}, \{\texttt{2001:db8::200/120}\}) = (\texttt{dip}, \emptyset)$$

which is an empty tuple.

Definition 7 (Match Intersection)**.** *The intersection of two matches m_1 and m_2 is defined as*

$$\cap : M \times M \to M$$
$$m_1 \cap m_2 := \{t_1^{h_i} \cap t_2^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_1^{h_i} \in m_1, t_2^{h_i} \in m_2\}$$

This operation is complete since all headers in a match are initialized either explicitly or implicitly by default.

For instance, intersecting two matches

$$m_a = \{(\texttt{dip}, \{\texttt{2001:db8::100/120}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{22\})\}$$

and

$$m_b = \{(\texttt{dip}, \{\texttt{2001:db8::110/124}\}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\}$$

results in

$$m_a \cap m_b = \{(\texttt{dip}, \{\texttt{2001:db8::110/124}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{22\})\}$$

since

$$\begin{aligned} & (\texttt{dip}, \{\texttt{2001:db8::100/120}\}) \cap \\ & (\texttt{dip}, \{\texttt{2001:db8::110/124}\}) \\ = \; & (\texttt{dip}, \{\texttt{2001:db8::110/124}\}), \\ & (\texttt{proto}, \{6\}) \cap (\texttt{proto}, \mathcal{V}_{\texttt{proto}}) = (\texttt{proto}, \{6\}), \text{ and} \\ & (\texttt{dport}, \{22\}) \cap (\texttt{dport}, \mathcal{V}_{\texttt{dport}}) = (\texttt{dport}, \{22\}). \end{aligned}$$

Definition 8 (Match Set Intersection)**.** *For match sets, the intersection is defined as*

$$\begin{aligned} & \cap : \mathcal{M} \times \mathcal{M} \to \mathcal{M} \\ & M_1 \cap M_2 := \{m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\}. \end{aligned}$$

All matches m_1 from M_1 are intersected with all matches m_2 from M_2.

For instance, intersecting

$$\begin{aligned} M_a = \{ & \\ & m_1 = \{(\texttt{dip}, \{\texttt{2001:db8::1}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{80\})\}, \\ & m_2 = \{(\texttt{dip}, \{\texttt{2001:db8::4/126}\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \{53\})\} \\ \} & \end{aligned}$$

with

$$\begin{aligned} M_b = \{ & \\ & m_3 = \{(\texttt{dip}, \{\texttt{2001:db8::1}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{80\})\}, \\ & m_4 = \{(\texttt{dip}, \{\texttt{2001:db8::2/127}\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \{53\})\} \\ \}, & \end{aligned}$$

results in

$$\begin{aligned} M_a \cap M_b \; & = \{m_1 \cap m_3, \overbrace{m_1 \cap m_4}^{\emptyset}, \overbrace{m_2 \cap m_3}^{\emptyset}, \overbrace{m_2 \cap m_4}^{\emptyset}\} \\ & = \{\{(\texttt{dip}, \{\texttt{2001:db8::1}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{80\})\}\}. \end{aligned}$$

Subset Relations

Definition 9 (Match Subset Relation)**.** *The subset relation of two matches m_1 and m_2 is defined as*

$$\subseteq: M \times M$$
$$m_1 \subseteq m_2 :\Leftrightarrow m_1 \cap m_2 = m_1.$$

For example,

$$m_a = \{(\texttt{dip}, \{\texttt{2001:db8::110/124}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{22\})\}$$

is a subset of

$$m_b = \{(\texttt{dip}, \{\texttt{2001:db8::100/120}\}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\}$$

since

$$\begin{aligned} m_a \cap m_b = \; & \{(\texttt{dip}, \{\texttt{2001:db8::110/124}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{22\})\} \cap \\ & \{(\texttt{dip}, \{\texttt{2001:db8::100/120}\}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\} \\ = \; & \{(\texttt{dip}, \{\texttt{2001:db8::110/124}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{22\})\} \qquad = m_a. \end{aligned}$$

Definition 10 (Match Set Subset Relation)**.** *The subset relation of two match sets M_1 and M_2 is defined as*

$$\subseteq: \mathcal{M} \times \mathcal{M}$$
$$M_1 \subseteq M_2 :\Leftrightarrow \forall m_1 \in M_1 : \exists m_2 \in M_2 : m_1 \subseteq m_2.$$

Given the match sets M_a and M_b from the previous intersection example, we see that M_a is not a subset of M_b since:

$$\underbrace{\underbrace{(\underbrace{m_1 \subseteq m_3}_{\text{true}} \vee \underbrace{m_1 \subseteq m_4}_{\text{false}})}_{\text{true}} \wedge \underbrace{\underbrace{(\underbrace{m_2 \subseteq m_3}_{\text{false}} \vee \underbrace{m_2 \subseteq m_4}_{\text{false}})}_{\text{false}}}_{\text{false}}}_{\text{false}} \Rightarrow M_a \not\subseteq M_b.$$

Unions

Definition 11 (Match Set Union)**.** *The union of arbitrary match sets M_1 and M_2 uses the regular set union, i.e.:*

$$\cup : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$$
$$M_1 \cup M_2 := \{m_1, m_2 \mid m_1 \in M_1, m_2 \in M_2\}$$

For example, given the following match sets:

$$
\begin{aligned}
M_a = \ \{ & \\
& \{(\texttt{dip}, \{\texttt{2001:db8::4/127}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{53\})\}, \\
& \{(\texttt{dip}, \{\texttt{2001:db8::6/127}\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \{53\})\} \\
\} &
\end{aligned}
$$

and

$$
\begin{aligned}
M_b = \ \{ & \\
& \{(\texttt{dip}, \{\texttt{2001:db8::4/127}\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \{53\})\} \\
\}. &
\end{aligned}
$$

Then, their union is:

$$
\begin{aligned}
M_a \cup M_b = \ \{ & \\
& \{(\texttt{dip}, \{\texttt{2001:db8::4/127}\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \{53\})\}, \\
& \{(\texttt{dip}, \{\texttt{2001:db8::6/127}\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \{53\})\}, \\
& \{(\texttt{dip}, \{\texttt{2001:db8::4/127}\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \{53\})\} \\
\}. &
\end{aligned}
$$

Note that, while containing all packets that were also contained in the operands, the resulting match set is not necessarily optimal. E.g., packets may be represented through multiple matches and matches could be further merged for more concise representations. For instance, in our example, the second and third match could be merged into a single match. All tuples are equal except for the IP address ranges which could be unioned and represented by the IP prefix `2001:db8::4/126`.

Algebra

Finally, we define DHSA:

Definition 12 (DHSA)**.** *The Domain-oriented Header Space Algebra (DHSA) is defined as*

$$\mathcal{DHSA} := (\mathcal{M}; \cap, \cup, M_\Omega, \emptyset).$$

The domain $\mathcal{M}$ is defined over the powerset of all possible packet matches and the intersection and union operate on sets of matches. In addition, the full set M_Ω and the empty set $\emptyset$ serve as neutral elements for the intersection resp. union. For details about DHSA's algebraic properties, refer to Section 5.4.

Discussion: Correctness, Completeness, and Complexity

The intersection as well as the union operations are correct and complete as discussed in the following. For more details of DHSA's algebraic properties refer to Section 5.4.

Correctness Intersections of matches resp. match sets do not contain packets that were not present in both operands. Meanwhile, the union of matches resp. match sets does not contain packets that were not present in either operand. Hence, both operations work correctly.

Completeness Intersections of matches resp. match sets consider all packets that were present in both operands. Meanwhile, the union of matches resp. match sets contains all packets that were present in either operand. Hence, both operations' results are complete.

Complexity The match intersection $m_1 \cap m_2$ runs linearly in $\mathcal{O}(|\mathcal{H}^D|)$. Since we translate DHSA to HSA, we can use NetPlumber with its efficient data structures. NetPlumber's run time is constant for the intersection for fixed $\mathcal{H}^D$ but this constant factor depends on the amount of bits necessary to store the wildcard expressions. Due to our enhancement of NetPlumber in Section 6.3 that enables dynamic expansion of $\mathcal{H}^D$ at runtime, we treat NetPlumber's intersection as linear-time operation.

The match set intersection $M_1 \cap M_2$ runs cubicly in $\mathcal{O}(max(|M_1|, |M_2|)^2 \cdot |\mathcal{H}^D|)$ since the match intersection runs linearly in $\mathcal{O}(|\mathcal{H}^D|)$ and the matches from M_1 and M_2 are intersected pairwise, i.e., $\mathcal{O}(max(|M_1|, |M_2|)^2)$.

The match set union $M_1 \cup M_2$ runs quadratically in $\mathcal{O}(max(|M_1|, |M_2|)^2)$ since match sets use the regular set union. Lazy implementations, that may contain duplicates, run in $\mathcal{O}(1)$ as these sets could be implemented as lists that are simply concatenated.

5.2 Example 1: Modeling a Firewall Rule Set

As first example, we want to model an IPTables rule set as sketched in Algorithm 5.1. The following rule set can be used to implement the FPL security specification which was introduced in Section 4.1.2 for the network depicted in Figure 4.1b:

Input: Firewall Rule Set (IPTables)
Output: A list of Headers $\mathcal{H}^D$ and a list of rules with DHSA matches

Procedure:

1. Initialize an empty rule list R and an index $i = 1$.
2. Parse all rules, i.e., per rule:
   - Go over all fields to create a list of header field tuples.
3. Go over all tuples to gather a set of header fields used in the rule set (results in $\mathcal{H}^D$).
4. Move the default rule (the first rule in an IPTables rule set) to the end of the rule list.
5. For each rule:
   - First, create a match set m_i that encodes the tuples from the rule's tuple list and sets all unstated headers to their respective default.
   - Then, append $r_i : m_i \rightarrow a_i$ to R (where a_i is the rule's action, e.g., `ACCEPT` or `DROP`).
   - Finally, increment i.
6. Return $\mathcal{H}^D$ and R

Algorithm 5.1.: IPTables Rule Set Modeling

```
(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::110/124 -p tcp --dport 80 \
      -j ACCEPT
(5) ip6tables -A FORWARD -s 2001:db8::200/120 -d 2001:db8::100/120 \
      -p tcp --dport 22 -j ACCEPT
```

Note that rule (1) is a commonly used sanity check against IP spoofing and was not derived from the security specification directly. Before modeling the rule set, we need to determine the rule set's Header Space. The header fields are the *IPv6 source* and *destination addresses*, the *protocol*, the *source* and *destination port*, the

virtual *state* field, and the virtual *ingress* and *egress interface* fields[2]. Therefore, the corresponding Header Space is given by:

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

First, we define that a rule r at index i in a rule set R is composed of a matching part m and an action a:

$$r_i : m \rightarrow a$$

For instance, the rule (4) from the example rule set is formalized as follows:

$$
\begin{aligned}
r_4 : \ \{ & \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), \\
& (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\}), \\
& (\texttt{proto}, \{6\}), \\
& (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\})) \\
\} & \rightarrow \mathit{accept}
\end{aligned}
$$

In order to model a complete firewall rule set R, we define it as an ordered set of rules:

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

[2]The state as well as the interface fields only exist in the packet filter and are not part of the packet's headers. Therefore, we refer to them as *virtual* header fields.

Hence, the formalization of the example rule set is as follows (note, that the default rule (0) becomes the last rule r_6, and also, we omit the empty input and output filtering tables R_{input} resp. R_{output}):

$$
\begin{aligned}
R_{forward} = \{ & \\
& r_1 : \{ \\
& \quad (\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& \quad (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\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{state}, \mathcal{V}_{\texttt{state}}) \\
& \} \rightarrow \mathit{drop}, \\
& r_2 : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& \quad (\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{state}, \{\texttt{ESTABLISHED}\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_3 : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}), \\
& \quad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\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{state}, \mathcal{V}_{\texttt{state}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_4 : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& \quad (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\}), \\
& \quad (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}), \\
& \quad (\texttt{state}, \mathcal{V}_{\texttt{state}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_5 : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& \quad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \{\texttt{2001:db8::100/120}\}), \\
& \quad (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}), \\
& \quad (\texttt{state}, \mathcal{V}_{\texttt{state}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_6 : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),
\end{aligned}
$$

$$
\begin{aligned}
&(\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&(\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&(\texttt{state}, \mathcal{V}_{\texttt{state}})\\
\} &\rightarrow \mathit{drop}\\
\}
\end{aligned}
$$

5.3 Example 2: Formalizing FPL's Inventories and Propagation

As a second example, we want to examine how to formally describe FPL inventories and formalize the attribute as well as the service propagation. As discussed, we model FPL inventories as directed acyclic graphs (DAGs). For our example from Chapter 4 (depicted in Figure 4.2a), the vertices are

$$V = \{\text{SampleOrganization}, \text{DMZ}, \text{WebServer}, \text{Office}, \text{Internet}\}$$

and the edges are

$$
\begin{aligned}
E = \{ &\\
&(\text{SampleOrganization}, \text{DMZ}),\\
&(\text{SampleOrganization}, \text{Office}),\\
&(\text{DMZ}, \text{WebServer})\\
\}.&
\end{aligned}
$$

In addition, we need to link the technical attributes as well as the offered services to a role. In terms of the DAG model, we introduce the annotation relations $AR : V \times A$ and $SR : V \times S$ where A and S are match sets that hold the roles' attributes resp. their offered services. If an attribute is specified explicitly in the inventory, the match's respective entry is set to that value. Otherwise, the entry defaults to the full

domain $\mathcal{V}_h$ of that header field h which is also required by our definition of matches. For our example, AR is specified as

$$
\begin{aligned}
AR = \{ & \\
& (\text{SampleOrganization}, \{(\texttt{ipv6}, \{\texttt{0::0/0}\})\}), \\
& (\text{Office}, \{(\texttt{ipv6}, \{\texttt{2001:db8::200/120}\})\}), \\
& (\text{DMZ}, \{(\texttt{ipv6}, \{\texttt{2001:db8::100/120}\})\}), \\
& (\text{WebServer}, \{(\texttt{ipv6}, \{\texttt{2001:db8::110/124}\})\}), \\
& (\text{Internet}, \{(\texttt{ipv6}, \{\texttt{0::0/0}\})\}) \\
\} &
\end{aligned}
$$

whereas SR is the following:

$$
\begin{aligned}
SR = \{ & \\
& (\text{SampleOrganization}, \emptyset), \\
& (\text{Office}, \emptyset), \\
& (\text{DMZ}, \{(\texttt{proto}, \{\texttt{6}\}), (\texttt{port}, \{\texttt{22}\})\}), \\
& (\text{WebServer}, \{(\texttt{proto}, \{\texttt{6}\}), (\texttt{port}, \{\texttt{80}\})\}), \\
& (\text{Internet}, \emptyset) \\
\} &
\end{aligned}
$$

Attribute Propagation

Intuitively, the algorithm for attribute propagation works in a two step manner. To determine all attributes for some node $v \in V$, first, we collect the attribute sets for all incoming paths and, second, consolidate these with the node's attributes. For this purpose, we define two functions that recursively traverse the DAG backwards until the root nodes are reached. The functions are $ap : V \to \mathcal{M}$ for the *attribute path propagation* and $ac : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ for the *attribute consolidation*. The path propagation is defined as:

$$
ap(v) = \begin{cases} \{a\} \text{ with } (v, a) \in AR, \text{ if } \nexists u \in V : (u, v) \in E \\ ac(a, \bigcup_{(u,v)\in E} ap(u)) \text{ with } (v, a) \in AR, \text{ else} \end{cases}
$$

The first case simply returns the node's attributes if it is a root node. Otherwise, the node recursively calls ap for all nodes that originate from an incoming edge and, then, consolidates the resulting set of attributes with the node's own attributes. The attribute consolidation is defined as:

$$
ac(\{m\}, M) = \{m\} \cup \{n \mid \forall n \in M : m \not\subseteq n\}
$$

The consolidation of some attribute m is performed by, first, filtering out all attributes that are supersets of m and, afterwards, adding m to this filtered set. This way, only the most specific attributes are propagated which fulfils a central property of FPL's attribute propagation (cf. Section 4.1).

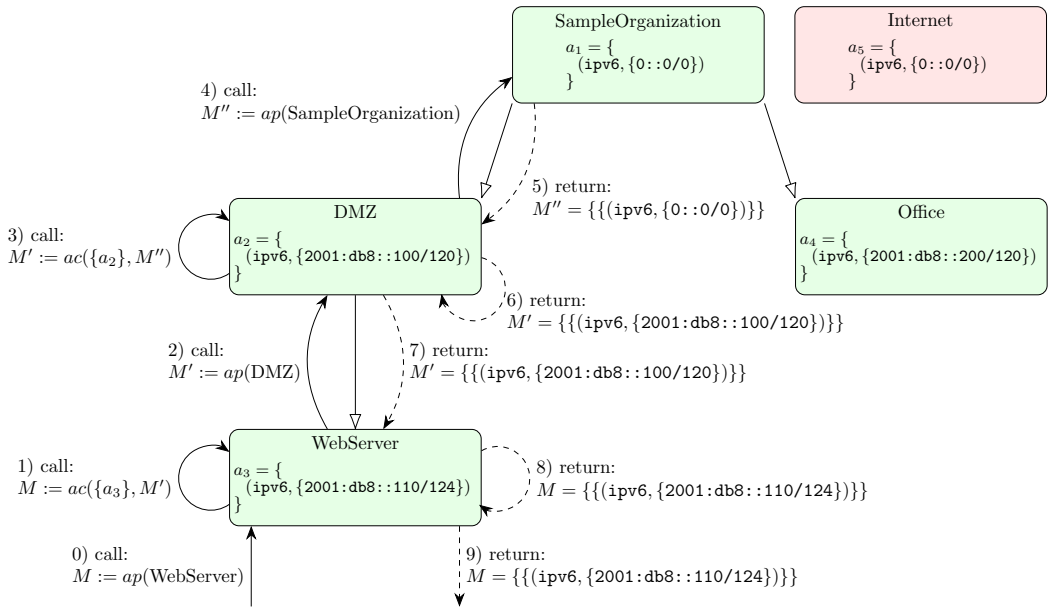


Fig. 5.2.: Example execution of the attribute resolution algorithm to derive the characterizing attributes of the *WebServer* role.

Figure 5.2 illustrates the algorithm's execution for the node *WebServer* step by step:

0) The algorithm starts by calling ap on the *WebServer* node in order to determine its set of characterizing attributes.

1) Since the node is not a root node, it is necessary to collect and consolidate all attributes propagated along all incoming paths. The set of incoming paths' attributes is denoted by M' and consolidated with the node's attributes through ac, i.e., $M = ac(\{a_3\}, M')$.

2) There is just one incoming edge $(\text{DMZ}, \text{WebServer}) \in E$ and, hence, the path union equals $M' = ap(\text{DMZ})$.

3) Again, the node *DMZ* is not a root node which requires collecting attributes from incoming paths in M'' and consolidating these with the node's attributes, i.e., $ac(\{a_2\}, M'')$.

4) There is just one incoming edge $(\text{SampleOrganization}, \text{DMZ}) \in E$ and, hence, the path union equals $M'' = ap(\text{SampleOrganization})$.

5) This time the node *SampleOrganization* is a root node and, therefore, ap returns the node's attributes, i.e., $M'' = \{a_1 = \{(\texttt{ipv6}, \{\texttt{0::0/0}\})\}\}$.

6) Consolidating the *DMZ*'s attributes with M'' results in the propagation of $a_2 = \{(\texttt{ipv6}, \{\texttt{2001:db8::100/120}\})\}$ because $\{\texttt{2001:db8::100/120}\}$ is more specific than $\{\texttt{0::0/0}\}$.
I.e., $M' = ac(\{a_2\}, M'') = \{a_2\} \cup \emptyset = \{(\texttt{ipv6}, \{\texttt{2001:db8::100/120}\})\}$ since $\{n \mid \forall n \in M'' : a_2 \not\subseteq n\} = \emptyset$.

7) As stated in 2) M' is also the result for the node *DMZ*, i.e., $ap(\text{DMZ}) = M'$.

8) The consolidation in the node *WebServer* works similar to 6):
$M = ac(\{a_3\}, M') = \{a_3\} \cup \emptyset = \{(\texttt{ipv6}, \{\texttt{2001:db8::110/124}\})\}$ because $\{\texttt{2001:db8::110/124}\} \subseteq \{\texttt{2001:db8::100/120}\}$.

9) M is also ap's result for the node *WebServer* which terminates the attribute propagation algorithm.

Input: The DAG model including the list of nodes V, the list of edges E, and the list of node attributes AR.
Output: The DAG model with propagated attributes.

Procedure:

1. Recursively derive the attributes for each node using ac.
2. Store the results along with the node using an attribute relation $AR' : V \to \mathcal{M}$.
3. Return V, E, and AR'

Algorithm 5.2.: Attribute Propagation

In Algorithm 5.2, we sketch the attribute propagation for the complete DAG model.

Termination Properties and Complexity The algorithm always terminates since the recursive backward traversion in DAGs eventually reaches some root node as there are no cycles in DAGs by construction. In addition, since ap is a total function (i.e., it is defined for any DAG along any related attribute relation AR), it is possible to cache and reuse the ap's result for any node. This way, for each node, ap is only calculated once and, hence, the complexity to the attribute propagation runs with linear complexity for any DAG, i.e., $\mathcal{O}(|\mathcal{V}|)$.

Service Propagation

Input: The DAG model including the list of nodes V, the list of edges E, and the list of offered services SR.
Output: The DAG model with propagated services.

Procedure:

1. Recursively derive the services for each node using ps.
2. Store the results along with the node using a service relation $SR' : V \to \mathcal{M}$.
3. Return V, E, and SR'.

Algorithm 5.3.: Service Propagation

Services are propagated in a similar way as attributes but, due to the fact that services are globally unique, the algorithm is simpler. For this purpose, we define a service propagation function $ps : V \to \mathcal{M}$ that calculates the set of services for some node $v \in \mathcal{V}$:

$$ps(v) = \begin{cases} s \text{ with } (v,s) \in SR, \text{ if } \nexists u \in V : (u,v) \in E \\ s \cup \bigcup_{(u,v)\in E} ps(u)) \text{ with } (v,s) \in SR, \text{ else} \end{cases}$$

If the node is a root node, its services are returned. Otherwise, all services from all incoming paths are collected by recursively traversing the DAG backwards and unioning each node's set of services. In Algorithm 5.3, we sketch the service propagation for the complete DAG model. The algorithm's termination properties and complexity are the same as for the attribute propagation for the same reasons. For instance, calculating the set of services for the WebServer yields

$$\begin{aligned} ps(\text{WebServer}) = \{ & \\ & \{(\texttt{proto}, \{6\}), (\texttt{port}, \{22\})\}, \\ & \{(\texttt{proto}, \{6\}), (\texttt{port}, \{80\})\} \\ \} & \end{aligned}$$

since it offers HTTP and inherits the SSH service from the DMZ.

5.4 Algebraic Properties of DHSA

In order to provide mathematical soundness to our embedding of DHSA into HSA using a homomorpism in Section 5.5, we first show that DHSA offers the algebraic properties of a *distributive lattice* with *neutral elements*. For reference, in Definition 12, we defined DHSA as follows:

$$\mathcal{DHSA} := (\mathcal{M}; \cap, \cup, M_\Omega, \emptyset).$$

Specificantly, for distributive lattices the following algebraic laws apply [34]:

$$\begin{array}{lrcll}
\forall x, y \in \mathcal{M}: & x \cap y & = & y \cap x & \text{(commutativity)} \\
\forall x, y \in \mathcal{M}: & x \cup y & = & y \cup x & \\
\forall x, y, z \in \mathcal{M}: & x \cap (y \cap z) & = & (x \cap y) \cap z & \text{(associativity)} \\
\forall x, y, z \in \mathcal{M}: & x \cup (y \cup z) & = & (x \cup y) \cup z & \\
\forall x, y, z \in \mathcal{M}: & x \cap (y \cup z) & = & (x \cap y) \cup (x \cap z) & \text{(distributivity)} \\
\forall x, y, z \in \mathcal{M}: & x \cup (y \cap z) & = & (x \cup y) \cap (x \cup z) & \\
\forall x \in \mathcal{M}: & x \cap x & = & x & \text{(idempotency)} \\
\forall x \in \mathcal{M}: & x \cup x & = & x & \\
\forall x, y \in \mathcal{M}: & x \cap (x \cup y) & = & x & \text{(absortion)} \\
\forall x, y \in \mathcal{M}: & x \cup (x \cap y) & = & x &
\end{array}$$

In the following, we examplarily proof the *commutativity* of the *intersection* operator. The proofs for *union commutativity*, *associativity*, *distributivity*, *idempotency*, and *absortion* can be found in Appendix A.5.

Theorem 1. *DHSA's intersection operation is commutative, i.e.:*

$$\forall x, y \in \mathcal{M} : x \cap y = y \cap x.$$

We prove the commutativity of DHSA's intersection operation by proving that intersecting tuples, matches, and match sets is commutative.

Lemma 1. *The tuple intersection is commutative, i.e.:*

$$\forall t_1^h, t_2^h \in \mathcal{H}^D \times \mathcal{V}_{\mathcal{H}^D} : t_1^h \cap t_2^h = t_2^h \cap t_1^h.$$

Proof.

$$
\begin{aligned}
t_1^h \cap t_2^h &= (h, v_{t_1}) \cap (h, v_{t_2}) \\
&= (h, v_{t_1} \cap v_{t_2}) && \text{| cf. Def. 6} \\
&= (h, v_{t_2} \cap v_{t_1}) && \text{| since the value sets are regular sets} \\
& && \text{| (Def. 1) and the regular set intersection} \\
& && \text{| is commutative (Def. 6)} \\
&= (h, v_{t_2}) \cap (h, v_{t_1}) \\
&= t_2^h \cap t_1^h
\end{aligned}
$$

□

Lemma 2. *The intersection of single matches is commutative, i.e.:*

$$\forall \{m_1\}, \{m_2\} \in \mathcal{M} : m_1 \cap m_2 = m_2 \cap m_1.$$

Proof.

$$
\begin{aligned}
m_1 \cap m_2 &= \{t_1^{h_i} \cap t_2^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_1^{h_i} \in m_1, t_2^{h_i} \in m_2\} && \text{| cf. Def. 7} \\
&= \{t_2^{h_i} \cap t_1^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_1^{h_i} \in m_1, t_2^{h_i} \in m_2\} && \text{| since the tuple} \\
& && \text{| intersection is} \\
& && \text{| commutative} \\
& && \text{| (Lemma 1)} \\
&= m_2 \cap m_1
\end{aligned}
$$

□

Lemma 3. *The intersection of match sets is commutative, i.e.:*

$$\forall M_1, M_2 \in \mathcal{M} : M_1 \cap M_2 = M_2 \cap M_1.$$

Proof.

$$
\begin{aligned}
M_1 \cap M_2 &= \{m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\} && \text{| cf. Def. 8} \\
&= \{m_2 \cap m_1 \mid m_2 \in M_2, m_1 \in M_1\} && \text{| since intersecting} \\
& && \text{| single matches is} \\
& && \text{| commutative} \\
& && \text{| (Lemma 2)} \\
&= M_2 \cap M_1
\end{aligned}
$$

□

Neutral Elements

In addition to its properties as distributive lattice, DHSA bears neutral elements for its intersection and union operations. I.e., the full match set $M_\Omega := \{m_\Omega\}$, where $m_\Omega := \{t_\Omega^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D\}$ is the match where all tuples' values equal their full value domain (i.e., $t_\Omega^{h_i} := (h_i, \mathcal{V}_{h_i})$), behaves neutrally when used in the intersection operation. Respectively, the empty set $\emptyset$ serves as neutral element for the union operation. For proofs of the neutrality properties, refer to Appendix A.6.

Comparison with HSA

We have shown that DHSA is a distributive lattice which is true for HSA as well. In addition, HSA defines a complementation operation and suffices the complementation laws. Hence, HSA is a boolean lattice (cf. Section 2.2.1.4 for details).

Since we did not define complementation for DHSA, the complementation laws do not hold. Therefore, DHSA is a distributive lattice but not a boolean lattice. The neutral elements for the intersection and union operations can be found in DHSA and HSA alike. For HSA being a boolean lattice, this is mandatory whereas for DHSA, it is an addition.

Concerning the embeddability, distributive lattices can be embedded into boolean lattices using injective homomorphisms. I.e., all elements as well as all operations of the domain DHSA can be mapped onto respective elements and operations of the codomain HSA. Therefore, DHSA can be embedded into HSA using this injective homomorphism.

5.5 Homomorphism

In order to embed DHSA in HSA, we use an injective homomorphism. After giving a short excursion into homomorphisms, we define the homomorphism function, proof the homomorphism properties, and show injectivity.

Note: Since we use the $\cap$, $\cup$, and $\emptyset$ symbols in DHSA and HSA alike, we improve clarity by denoting the symbols with the superscripts ^{D} for DHSA resp. ^{H} for HSA, e.g., $\cap^D$, $\cup^H$, or $\emptyset^D$.

Excursion: Homomorphisms

In universal algebra, a homomorphism is a *structure-preserving* mapping between two algebraic structures of the same type [34]. An algebra's type is determined by the arity of the algebras' operations. Let $\mathcal{A} = (A; (f_i)_{i \in I})$ and $\mathcal{B} = (B; (g_i)_{i \in I})$ be algebras where $(f_i)_{i \in I}$ and $(g_i)_{i \in I}$ are functions f_i resp. g_i indiced over I. Each operation has an arity n_i and the algebra's type τ is defined over the sequence of these arities, i.e., $\tau = (n_i)_{i \in I}$. For instance, DHSA is defined as $\mathcal{DHSA} := (\mathcal{M}; \cap^D, \cup^D, M_\Omega, \emptyset^D)$ where both, the intersection and the union operations, have an arity of 2 while the neutral elements are constants with arity 0. Hence, the type is $\tau^{\mathcal{DHSA}} = (n_1 = 2, n_2 = 2, n_3 = 0, n_4 = 0)$. HSA, on the other hand, is defined as $\mathcal{HSA} := (2^{\mathcal{H}}; \cap^H, \cup^H, ^-, H_\Omega, \emptyset^H)$ with $\tau^{\mathcal{HSA}} = (2, 2, 1, 0, 0)$.

A mapping $\sigma : A \to B$ is a *homomorphism* from $\mathcal{A}$ to $\mathcal{B}$ if they have the same type, i.e., $\tau^{\mathcal{A}} = \tau^{\mathcal{B}}$, and if for each $i \in I$ and the operation f_i with its parameters $a_1, ...a_{n_i} \in A$ and the corresponding operation g_i the following holds:

$$\sigma(f_i(a_1, ..., a_{n_i})) = g_i(\sigma(a_1), ..., \sigma(a_{n_i})).$$

Therefore, using a homomorphism allows to either compute in $\mathcal{A}$ and then map the result to $\mathcal{B}$ or to map the parameters to $\mathcal{B}$ first and then compute the result in $\mathcal{B}$.

In our case, we can see that the types mismatch since HSA comprises the complementation as an additional operation, i.e., $\tau^{\mathcal{DHSA}} = (2, 2, 0, 0) \neq (2, 2, 1, 0, 0) = \tau^{\mathcal{HSA}}$ Therefore, we limit our reflections to a sub-algebra $\mathcal{HSA}^-$ which is a distributive lattice plus neutral elements for the intersection resp. union operations. Formally:

$$\mathcal{HSA}^- := (2^{\mathcal{H}}; \cap^H, \cup^H, H_\Omega, \emptyset^H).$$

For $\mathcal{HSA}^-$ the type is $\tau^{\mathcal{HSA}^-} = (2, 2, 0, 0)$ which equals $\tau^{\mathcal{DHSA}}$.

Further, we need to show that the operations are closed under the homomorphism σ, i.e.:

$$\begin{aligned}
\sigma(\emptyset^D) &= \emptyset^H, \\
\sigma(M_\Omega) &= H_\Omega, \\
\sigma(M_1 \cap^D M_2) &= \sigma(M_1) \cap^H \sigma(M_2), \text{ and} \\
\sigma(M_1 \cup^D M_2) &= \sigma(M_1) \cup^H \sigma(M_2).
\end{aligned}$$

Injective Homomorphisms Like other relations, homomorphisms can be surjective, *injective*, or bijective. A homomorphism is injective if for every two distinct preimages

from the domain the resulting images in the codomain are distinct, too. Though, it is not necessary that every element from the codomain is mapped to, i.e., there may exist elements in the codomain without a preimage under the homomorphism. Injective homomorphisms are also called Monomorphisms or *Embeddings*.

Formally, a function $f : A \to B$ is injective if

$$\forall x, y \in A : (f(x) = f(y)) \to x = y.$$

A homomorphism $\sigma : A \to B$ for some algebras $\mathcal{A} = (A; (f_i)_{i \in I})$ and $\mathcal{B} = (B; (g_i)_{i \in I})$ is injective if all operations behave injective under the homomorphism, i.e., for an operation f_i with an arity n_i:

$$\forall a_1, b_1, ..., a_{n_i}, b_{n_i} \in A : \sigma(f_i(a_1, ..., a_{n_i})) = \sigma(f_i(b_1, ..., b_{n_i})) \to a_1 = b_1, ..., a_{n_i} = b_{n_i}.$$

Particularly, to show the injectivity of DHSA's homomorphism σ,

$$\forall M_1, M_2 \in \mathcal{M}, \forall h \in 2^{\mathcal{H}} : (\sigma(M_1) = h \wedge \sigma(M_2) = h) \to M_1 = M_2$$

must hold.

Definitions

Our homomorphism consists of a top-level function that maps match sets onto header space objects and helper functions that expand DHSA matches to sets of *wildcardable* matches and transform such matches into wildcard expressions.

For example, given a Header Space consisting of the IPv4 destination address field, the protocol field and the destination port, i.e., $\mathcal{H}^D = \{h_1 = \texttt{dip}, h_2 = \texttt{proto}, h_3 = \texttt{dport}\}$, a match set

$$M = \{\{(\texttt{dip}, \{1.2.3.4\}), (\texttt{proto}, \{6, 17\}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\}\}$$

would result in the header space object

$$H = \{$$
`00000001,00000010,00000011,00000100,00000110,xxxxxxxx,xxxxxxxx;`
`00000001,00000010,00000011,00000100,00010001,xxxxxxxx,xxxxxxxx`
$$\}$$

consisting of two wildcard expressions since the value set $\{6, 17\}$ cannot be represented using a single wildcard expression. Hence, the match in M needed to be expanded into two matches

$$\{(\texttt{dip}, \{1.2.3.4\}), (\texttt{proto}, \{6\}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\}$$

and

$$\{(\texttt{dip}, \{1.2.3.4\}), (\texttt{proto}, \{17\}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\}$$

before M could have been translated to HSA.

In particular, we define a homomorphism σ that transforms a DHSA match set into a header space object by, first, enumerating all *packets* matched by the match set and, then, transforming these packets into wildcard expressions. The resulting set of wildcard expressions is a proper header space object. Formally:

Definition 13 (Homomorphism)**.**

$$\sigma : \mathcal{M} \rightarrow 2^{\mathcal{H}}$$
$$\sigma(M) := \{WC(p) \mid p \in packets(m), m \in M\}$$

where $packets$ and WC are helper functions with the following semantics.

The helper function $packets : M \rightarrow \mathcal{M}$ enumerates all packets matched by a match. A *packet* is a match where for each header field tuple, the value set contains only a single element. E.g., the match $\{(\texttt{dip}, \{\texttt{1.2.3.4}\}), (\texttt{proto}, \{\texttt{6}\}), (\texttt{dport}, \{80\})\}$ is a packet while $\{(\texttt{dip}, \{\texttt{1.2.3.4}\}), (\texttt{proto}, \{\texttt{6,17}\}), (\texttt{dport}, \{53\})\}$ is not. Consequently, the matches that represent single packets result in wildcard expressions that solely consist of `0` or `1` but not `x`. Formally, the packet enumeration is defined as:

Definition 14 (Packet Enumeration)**.**

$$packets : M \rightarrow \mathcal{M}$$
$$packets(m) := \{$$
$$\{(h_i, \{v\}) \mid \forall i = 1..|\mathcal{H}^{\mathcal{D}}| : h_i \in \mathcal{H}^{\mathcal{D}}\} \mid v \in v_{h_i}, (h_i, v_{h_i}) \in m, \forall i = 1..|\mathcal{H}^{\mathcal{D}}| : h_i \in \mathcal{H}^{\mathcal{D}}$$
$$\}.$$

Further, the helper function $WC : M \rightarrow \mathcal{H}$ transforms a *wildcardable* DHSA match into a wildcard expression. A match is wildcardable if it can be represented with a single wildcard expression. E.g., the match

$$\{(\texttt{dip}, \{\texttt{1.2.3.4}\}), (\texttt{proto}, \{\texttt{6}\}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\}$$

is wildcardable since the protocol's value 6 is encoded as 00000110 while the destination address's and port's values are encoded as

00000001,00000010,00000011,00000100 resp. xxxxxxxx,xxxxxxxx

resulting in the wildcard expression

00000001,00000010,00000011,00000100,00000110,xxxxxxxx,xxxxxxxx.

The match $\{(\texttt{dip}, \{\texttt{1.2.3.4}\}), (\texttt{proto}, \{6, 17\}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}})\}$, on the other hand, is not wildcardable since there is no single wildcard expression that encodes both protocol values. Note that packets are always wildcardable.

Homomorphic Properties of σ

We show that σ is an injective homomorphism from $\mathcal{DHSA}$ to $\mathcal{HSA}^-$ by proving the homomorphism properties for the intersection and union operations as well as for the neutral elements. Finally, we discuss σ's injectitivity.

Theorem 2. *σ is an injective homomorphism from $\mathcal{DHSA}$ to $\mathcal{HSA}^-$.*

Lemma 4. *σ maps DHSA's empty element to HSA's empty element.*

Proof.

$$\sigma(\emptyset^D) = \{WC(p) \mid p \in packets(m), m \in \emptyset^D\} = \{\} = \emptyset^H$$

□

Lemma 5. *σ maps DHSA's full set to HSA's full set: $\sigma(M_\Omega) = H_\Omega$.*

Proof. (Where the helper function $max : \mathcal{V}_{\mathcal{H}} \to \mathcal{V}_{\mathcal{H}}$ returns a set containing only the largest element of a value set.)

$$
\begin{aligned}
\sigma(M_\Omega) &= \{WC(p) \mid p \in packets(m), m \in \{m_\Omega\}\} \\
&= \{WC(p) \mid p \in packets(m_\Omega)\} \\
&= \{ \\
&\quad WC(\{(h_1, \{0\}), ..., (h_n, \{0\})\}), \\
&\quad WC(\{(h_1, \{0\}), ..., (h_n, \{1\})\}), \\
&\quad \dots \\
&\quad WC(\{(h_1, max(\mathcal{V}_{h_1})), ..., (h_n, max(\mathcal{V}_{h_n}))\}) \\
&\} \\
&= \{\texttt{000..000}, \texttt{000..001}, ..., \texttt{111..111}\} \\
&= \{\texttt{xx..xx}\} = H_\Omega
\end{aligned}
$$

□

Lemma 6. *σ fulfils the homomorphism properties for the intersection operation.*

Proof.

$$
\begin{aligned}
\sigma(M_1 \cap^D M_2) &= \{WC(p) \mid p \in packets(m), m \in M_1 \cap^D M_2\} && \text{| cf. Def. 13} \\
&= \{WC(p) \mid && \text{| cf. Def. 8} \\
&\quad p \in packets(m_1 \cap^D m_2), \\
&\quad m_1 \in M_1, \\
&\quad m_2 \in M_2 \\
&\} \\
&= \{WC(p_1) \mid && \text{| since taking all} \\
&\quad p_1 = p_2, && \text{| packets from a} \\
&\quad p_1 \in packets(m_1), m_1 \in M_1, && \text{| match intersection} \\
&\quad p_2 \in packets(m_2), m_2 \in M_2 && \text{| is the same as} \\
&\} && \text{| taking the same} \\
& && \text{| packets from the} \\
& && \text{| individual matches} \\
&= \{WC(p_1) \mid \\
&\quad p_1 = p_2, \\
&\quad p_1 \in packets(m_1), m_1 \in M_1, \\
&\quad p_2 \in packets(m_2), m_2 \in M_2 \\
&\} \cap^H \{WC(p_2) \mid \\
&\quad p_1 = p_2, \\
&\quad p_1 \in packets(m_1), m_1 \in M_1, \\
&\quad p_2 \in packets(m_2), m_2 \in M_2 \\
&\} \\
&= \sigma(M_1) \cap^H \sigma(M_2)
\end{aligned}
$$

□

Lemma 7. *σ fulfils the homomorphism properties for the union operation.*

Proof.

$$\begin{aligned}\sigma(M_1 \cup^D M_2) &= \{WC(p_1), WC(p_2) \mid \\ &\quad p_1 \in packets(m_1), m_1 \in M_1, \\ &\quad p_2 \in packets(m_2), m_2 \in M_2 \\ &\quad \} \\ &= \{WC(p_1) \mid p_1 \in packets(m_1), m_1 \in M_1\} \cup^H \\ &\quad \{WC(p_2) \mid p_2 \in packets(m_2), m_2 \in M_2\} \\ &= \sigma(M_1) \cup^H \sigma(M_2)\end{aligned}$$

□

Lemma 8. *$\sigma : \mathcal{M} \to \mathcal{H}$ is injective, i.e.:*

$$\forall M_1, M_2 \in \mathcal{M} : \forall h \in \mathcal{H} : (\sigma(M_1) = h \wedge \sigma(M_2) = h) \to M_1 = M_2$$

Proof. Before proving injectivity, we define that two match sets are equal iff they contain the same packets. Formally:

Definition 15 (Match Set Equality)**.**

$$M_1 = M_2 :\Leftrightarrow \bigcup_{m_1 \in M_1} packets(m_1) = \bigcup_{m_2 \in M_2} packets(m_2)$$

We prove the lemma by contradiction. Assume that $M_1 \neq M_2$, then there exists a packet p

- that is contained in M_1 but not in M_2 (*Case 1*)

or

- that is not contained in M_1 but in M_2 (*Case 2*).

Formally:

$$p \in \textstyle\bigcup_{m_1 \in M_1} packets(m_1), p \notin \bigcup_{m_2 \in M_2} packets(m_2) \quad (1)$$
$$p \notin \textstyle\bigcup_{m_1 \in M_1} packets(m_1), p \in \bigcup_{m_2 \in M_2} packets(m_2) \quad (2)$$

To prove the first case, we apply the homomorphism to both match sets and compare the results:

$$\begin{aligned}\sigma(M_1) = \quad & \{WC(p), WC(p_1) \mid \\ & \quad p \in packets(m'), m' \in M_1, p_1 \in packets(m_1), m_1 \in M_1 \\ & \} =: h_1 \\ \sigma(M_2) = \quad & \{WC(p_2) \mid p_2 \in packets(m_2), m_2 \in M_2\} =: h_2\end{aligned}$$

Since p is not in M_2, $WC(p)$ cannot be in h_2. Therefore, $h_1 \neq h_2$ and the premise is false.

The second case can be proven analogously. Hence, if $\sigma(M_1) = h = \sigma(M_2)$ should hold, then M_1 must be equal to M_2 which is a contradiction to our assumption. Thus, the lemma holds and σ is injective.

□

5.6 Summary

In this chapter, we presented the *Domain-oriented Header Space Analysis* (DHSA) which enables an accessible modeling of network devices along with their configuration. We demonstrate modeling with DHSA regarding IPTables rule sets and by formalizing the attribute and service propagations sketched in Chapter 4.

Also, we have proven that DHSA is a distributive lattice and we have shown that our embedding in HSA is an injective homomorphism. These formal assurances imply high confidence in the functional correctness of the DHSA-based models and algorithms in our verification framework FaVe concerning the solving in the HSA-based NetPlumber

6 The Fast Verification Framework FaVe

In order to realize the management workflows stated in Section 1.2 – namely the compliance verification workflow as well as the firewall anomaly detection workflow – we implemented *FaVe* – a framework for fast formal network verification. As shown in Figure 6.1, FaVe is based on DHSA which was introduced in Chapter 5 and offers accessible means and automation for administrators to model networks, verify compliance with FPL security specifications, and detect firewall anomalies. In addition, FaVe automates most tasks to minimize operational costs as well as human error and, therefore, enables continuous repetitions as part of the security management workflows.

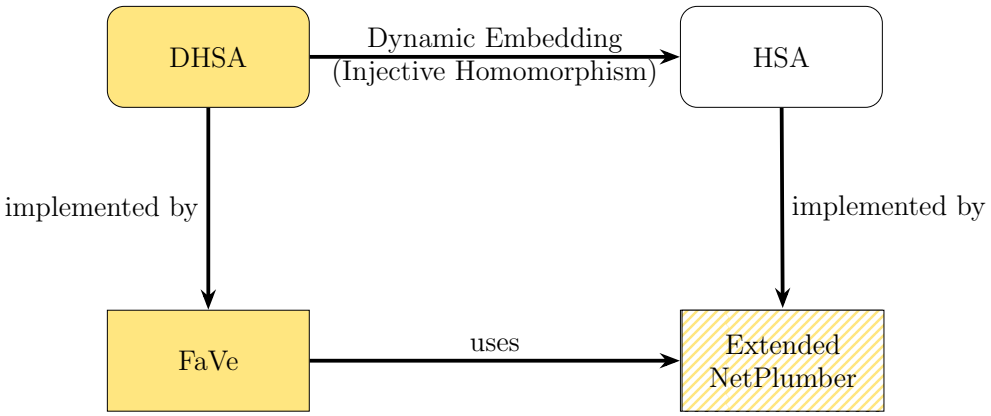


Fig. 6.1.: Overview of the general concepts, their implementations, and inter-relations.

This chapter is structured as follows. After detailing the main security workflows, we provide details about FaVe's architecture in Section 6.1 as well as for all major components. Finally, in Section 6.3, we show how to model dynamic protocols like IPv6 extension header chains and how we support these procedures in FaVe.

Compliance Verification Workflow The compliance verification workflow is realized as shown in Figure 6.2. The 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

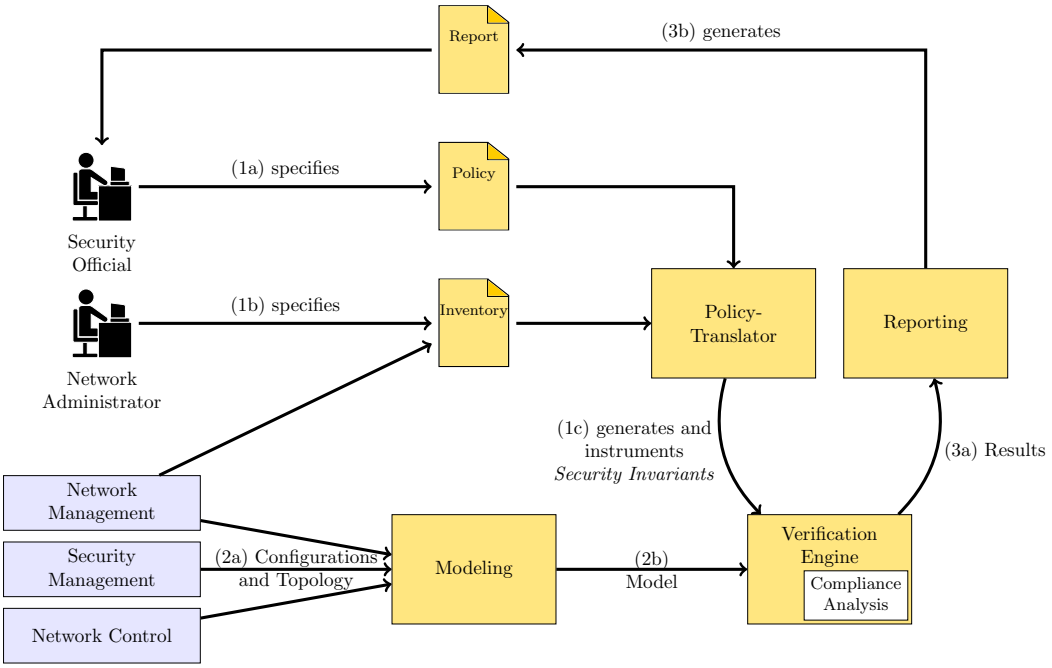


Fig. 6.2.: The management workflow to check network security compliance (yellow) and its integration with existing management systems (blue).

terms of the network, e.g., the reachability of services or network segments. The network administrator specifies an *inventory* (1b) which maps these abstract terms to their technical implementation in the network. This security specification is used to generate *security invariants* (1c) that are instrumented in a *verification engine*.

In the second part, the *network topology* and the *configurations* of network devices like routers, firewalls, or switches need to be modeled using the originial device configurations as inputs (2a), 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 (2b).

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

Firewall Anomaly Detection Workflow As shown in Figure 6.3 the detection of firewall anomalies is realized as a two step workflow. First, the firewall rule set is modeled. Second, the model is checked for anomalies (2a) by the *verification engine* and, then, the results are *reported* (2b) to the network administrator.

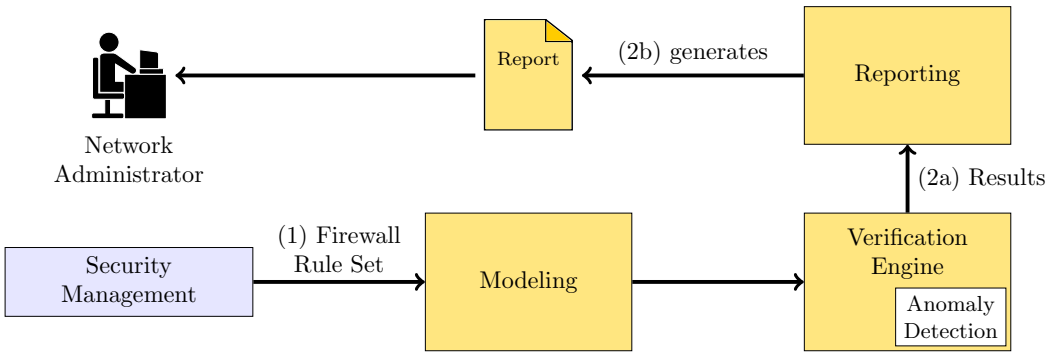


Fig. 6.3.: The management workflow to detect firewall anomalies.

6.1 Architecture

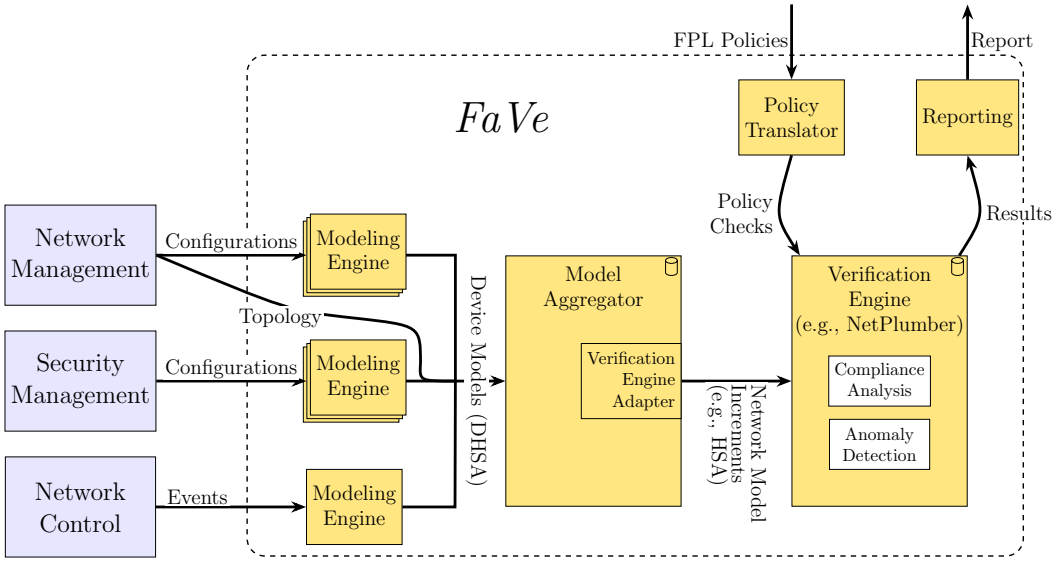


Fig. 6.4.: Overview of FaVe's incremental modeling and verification pipeline.

The workflows implemented with FaVe follow a *pipeline* pattern: Device configurations are modeled individually, aggregated to an overall network model, and analyzed by a verification engine concerning some invariants. E.g., if some firewall contains firewall anomalies or if the network model complies with some FPL security specification. Figure 6.4 gives an overview of FaVe's pipelined architecture. FaVe consists of a set of components: the *Modeling Engines* and the *Model Aggregator* implement the modeling of networks, the *Policy Translator* implements FPL instrumentations, and the *Verification Engine Adapter* provides integration with the verification engine. Finally, the *Reporting* engine generates reports. These components and their interplay are described in the following.

6.1.1 Modeling Engines

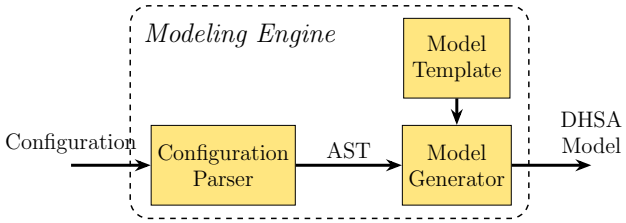
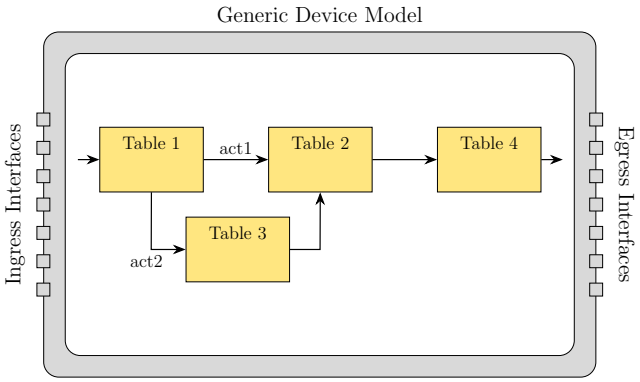
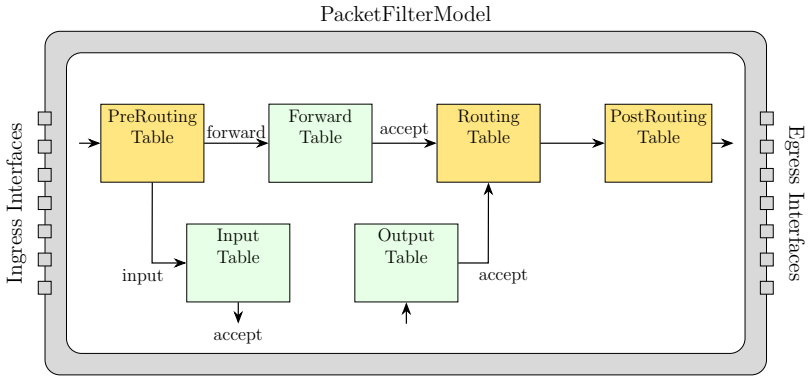


Fig. 6.5.: Internals of a modeling engine.

Modeling Engines transform a device configuration into a DHSA-based device model for further processing by FaVe's modeling pipeline. Figure 6.5 shows the internals of a Modeling Engine. First, the configuration is parsed into an *Abstract Syntax Tree* (AST) which is, then, used to instantiate a *Model Template* for that particular device type, e.g., a packet filter.



(a) Generic device model



(b) Packet filter model

Fig. 6.6.: The generic device model (top) and an example of a packet filter model (bottom).

Model templates consist of an inner and an outer part. Figure 6.6a shows the generic device model while Figure 6.6b depicts a packet filter model template. 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 set of rules based on the dependencies between the rules' match parts. Then, these sub flows are processed based on the respective set of actions, e.g., forwarded, rewritten, or dropped. Branching is modeled through the application of forwarding actions, i.e., depending on the action, different tables can be processed next.

For instance, in the case of the packet filter's *PreRouting* table, packets are distinguished based on their target, i.e., if they are destined for the packet filter itself (the *Input* table) or to be further forwarded in the network (the *Forward* table). All in all, the internal pipeline of the packet filter model comprises six tables. Details on the packet filter model as well as for other device models will be given in Chapter 7.

6.1.2 Model Aggregator

The *Model Aggregator* combines the device models according to the network topology. I.e., it connects the devices' egress ports with the respective ingress ports. In addition, the model aggregator stores the network model and is able to calculate increments between subsequent models. If the verification engine supports incremental solving, these increments can speed up reverification.

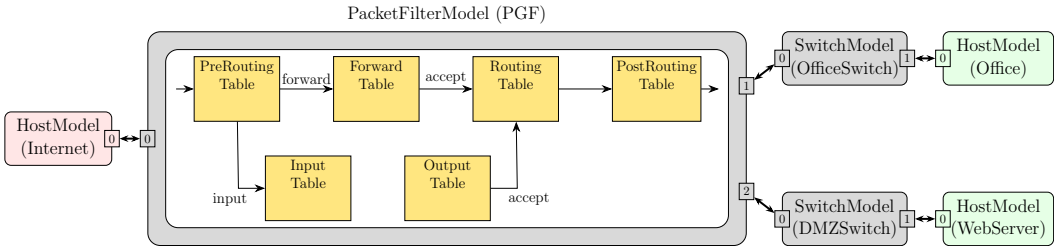


Fig. 6.7.: Example network model.

Figure 6.7 shows how our example network from Section 4.1 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 as seen before. Intuitively, the topology is modeled by the following table which connects the devices's egress ports with the respective ingress ports (*PGF* denotes the packet filter):

Egress Port	Ingress Port
Internet.egress.0	PGF.ingress.0
PGF.egress.0	Internet.ingress.0
PGF.egress.1	OfficeSwitch.ingress.0
OfficeSwitch.egress.0	PGF.ingress.1
PGF.egress.2	DMZSwitch.ingress.0
DMZSwitch.egress.0	PGF.ingress.2
OfficeSwitch.egress.1	Office.ingress.0
Office.egress.0	OfficeSwitch.ingress.1
DMZSwitch.egress.1	WebServer.ingress.0
WebServer.egress.0	DMZSwitch.ingress.1

6.1.3 Verification Engine Adapter

Verification Engines need to offer capabilities to verify compliance of network models with FPL security specifications as well as the detection of firewall anomalies. Since existing verification engines cannot process DHSA models directly, we specify *Verification Engine Adapters* that perform the necessary steps in order to use a verification engine. Particularly, FaVe reuses and extends the well-known NetPlumber [79] engine which models networks with HSA (see Section 2.2.2 for details).

Being a part of the *Model Aggregator*, our *NetPlumber adapter* efficiently implements the embedding of DHSA models in HSA models. A detailed introduction into the embedding's implementation will be given in Section 6.2. Further, details of the compliance verification and the firewall anomaly detection are given in Chapter 7 resp. Chapter 8.

FaVe supports an incremental mode where subsequent reverifications can be performed on model increments instead of the full model. This feature requires incremental solving by the verification engine, which, for instance, is offered by NetPlumber.

6.1.4 Policy Translator

The *Policy Translator* transforms FPL security specifications into traffic generators and security invariants to be instrumented and executed in the verification engine. These endpoints need to be attached to the network model based on the topology and need to take the inventory into consideration. There are two attachment modes that are based on the endpoint's model complexity. I.e., complex endpoints are

modeled by a packet filter model, e.g., in order to enable configuring host firewall rules, whereas simple endpoints are not modeled explicitly by some device model. FaVe offers pairs of *traffic emitters* and *traffic consumers* that comprise a port that can be directly connected to the network or to internal pipelines of complex models. This way, simple endpoints can be represented directly. With complex endpoints, they are attached to the internal pipeline. E.g., an emitter could be attached to the packet filter's output pipeline whereas a consumer could be attached to the input pipeline. The emitted traffic aligns with the modeled FPL role, i.e., the destination fields are set according to the role, e.g., the destination IP range or VLAN tag.

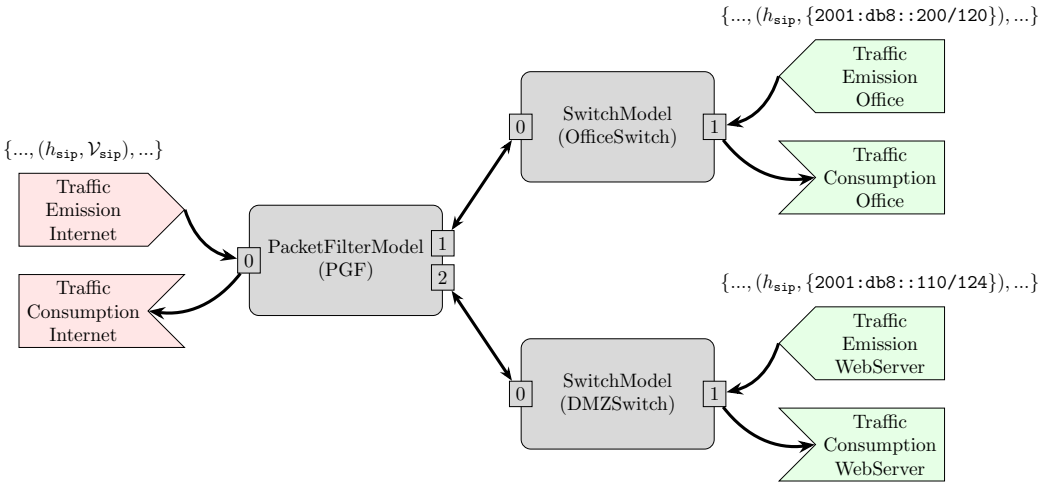


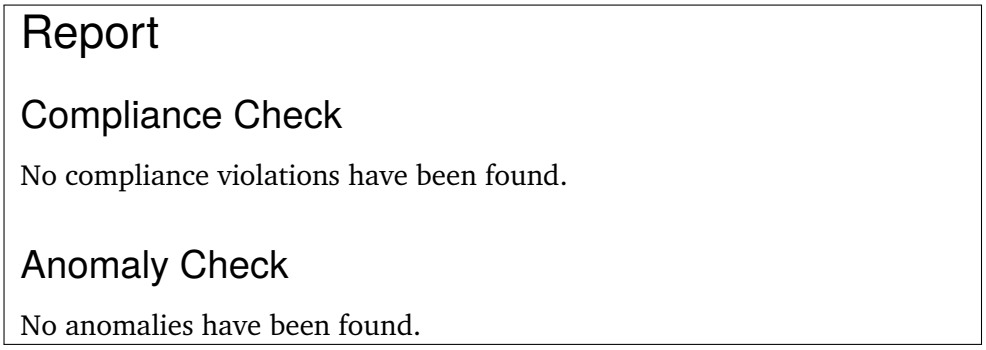
Fig. 6.8.: Example with simple host models directly represented by pairs of emitters and consumers.

Figure 6.8 shows the attachments for traffic emission and consumption in our example network. In this example, we model each endpoint with simple host models, i.e., by a pair of traffic emitters and consumers directly attached to the network. The emitters generate traffic with the source IP set to the respective endpoint's address range, e.g., `2001:db8::110/124` for the web server. Each attachment's port is unidirectionally connected to the port originally connecting the host model. E.g., the Internet's emitter's port is connected to the firewall's ingress port facing the Internet while the latter's respective egress port is connected to the Internet consumer's port.

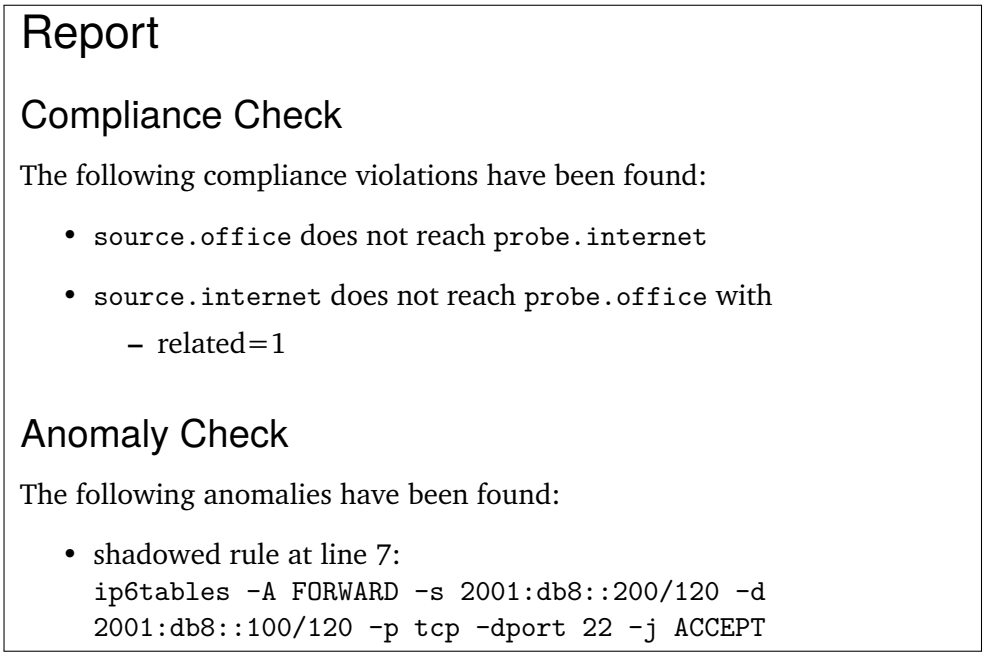
6.1.5 Reporting

FaVe offers a reporting engine that generates a report helping the security officials and administrators to understand the verification results. The report consists of two

sections where the first one details compliance violations and the second one lists firewall anomalies. For more details on compliance and firewall anomaly checking refer to Chapter 7 resp. Chapter 8.



(a) Report for a firewall rule set without any compliance violations or firewall anomalies.



(b) Report for a firewall rule set that violates compliance and includes a firewall anomaly.

Fig. 6.9.: Example reports as provided by FaVe.

For instance, the report depicted in Figure 6.9a shows the results when checking our example network with the firewall rule set from Section 5.2 against the compliance specification from Section 4.1.2. Since everything is compliant and there are no anomalies in the firewall rule set, the report is blank.

Assume we change the firewall rule set by removing rule (3) and duplicate rule (5). Removing rule (3) results in a compliance violation of the rule `Office <->> Internet`

and the duplication of rule (5) introduces a shadowing anomaly. The resulting rule set looks as follows:

```
(1) ip6tables -P FORWARD DROP
(2) ip6tables -A FORWARD --in-interface eth0 \
      -s 2001:db8::0/32 -j DROP
(3) ip6tables -A FORWARD -m conntrack --ctstate \
      ESTABLISHED -j ACCEPT
(4) #ip6tables -A FORWARD --out-interface eth0 \
      -s 2001:db8::200/120 -j ACCEPT
(5) ip6tables -A FORWARD -d 2001:db8::110/124 -p tcp \
      --dport 80 -j ACCEPT
(6) ip6tables -A FORWARD -s 2001:db8::200/120 \
      -d 2001:db8::100/120 -p tcp --dport 22 -j ACCEPT
(7) ip6tables -A FORWARD -s 2001:db8::200/120 \
      -d 2001:db8::100/120 -p tcp --dport 22 -j ACCEPT
```

When checking for compliance and anomalies, FaVe yields a report as depicted in Figure 6.9b. The compliance section shows that the Office role cannot reach the Internet and return traffic is not permitted either. This violates the compliance rule `Office <->> Internet`. The anomaly section reveals that the duplicated firewall rule (7) is shadowed by previous rules.

6.1.6 Extensibility

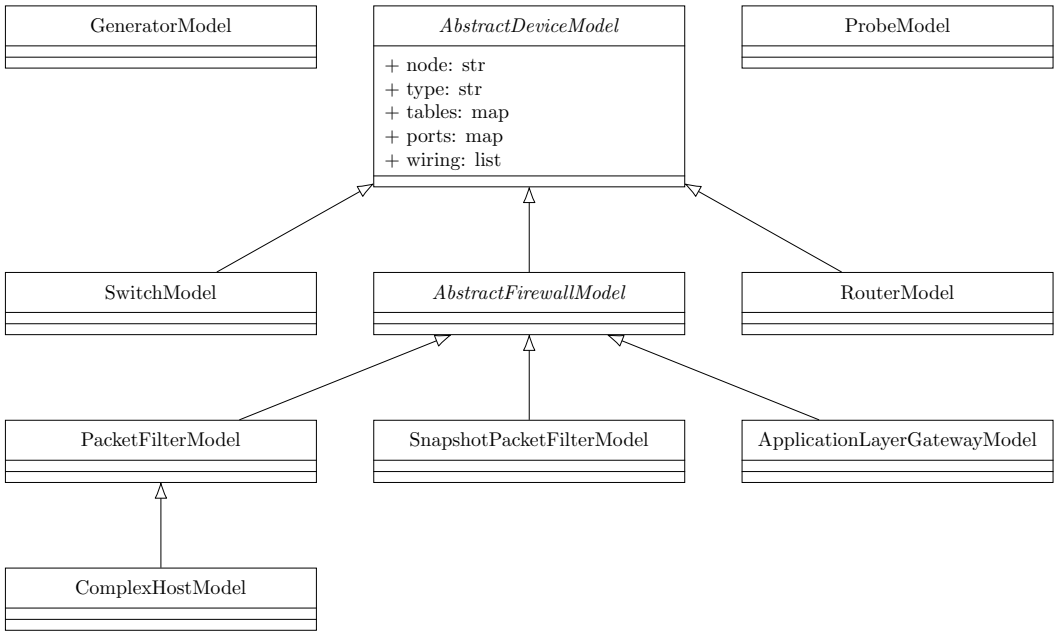


Fig. 6.10.: Overview of FaVe's device model template classes.

The FaVe framework can be extended in two aspects. First, new device models can be added and, second, alternative verification engines may be introduced.

New device models The addition of a new device model can be achieved by implementing a new modeling engine. As depicted in Figure 6.5, there are three components: a *configuration parser*, a *model template*, and a *model generator*. Model templates are realized through polymorphism. Figure 6.10 gives an overview of the device model template classes currently shipped with FaVe. New device models can inherit from the abstract device model or any of the specific models. In addition, one needs to implement the parser and the model generator. First, the parser reads the configuration and yields an *Abstract Syntax Tree* (AST). Then, the model generator instantiates the model template with the configuration data from the AST. Finally, the new device model needs to be registered with the model aggregator by their name and class.

For instance, as depicted in Figure 6.10, we used this mechanism to realize device models for routers that extend the abstract device model and snapshot-based packet filters that extend the abstract firewall model. For details on these device models, refer to the Sections 7.1.2 resp. 7.1.5.

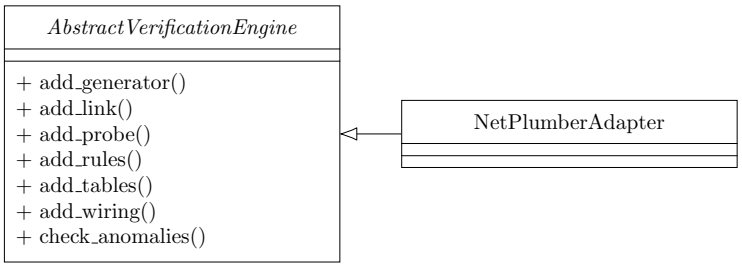


Fig. 6.11.: Overview of FaVe's verification engine classes.

Alternative verification engines To replace the verification engine, one needs to implement an adapter class that transforms a DHSA model into the model that is specific to the new engine. In particular, the modeling building blocks, i.e., tables, links, ports, and rules, need to be represented as well as a mapping from DHSA onto the engine's formalism. As depicted in Figure 6.11 the adapter class can be derived from FaVe's `AbstractVerificationEngine` class. Finally, FaVe's model aggregator needs to be configured to use the alternative verification engine.

6.1.7 Implementation of the Prototype

FaVe's prototype is implemented in Python and reuses the publicly available NetPlumber verification backend [76] which is implemented in C/C++.[1] The modeling engines are realized as small standalone tools which communicate over UNIX domain sockets with the Model Aggregator component and include models for routers with Cisco ACLs, switches, and stateful `ip6tables` packet filters. The Model 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.

We implemented some extensions for NetPlumber which add necessary features or improve performance:

Header Space Enlargement We enable the enlargement of the Header Space at runtime, i.e., the amount of bits that encode header fields in NetPlumber. We need this feature to model dynamic protocol elements like IPv6 extension header chains (cf. Section 6.3). Our implementation accesses every wildcard expression and header space object and enlarges memory to fit the new amount of header bits. By using `realloc`[2] on Linux, often memory reallocation can be performed in-place which helps to prevent expensive copying operations.

Header Space Object Compression We implemented a simple compression for header space objects that reduces memory consumption and that needs to be performed explicitly. The compression performs a pairwise comparison of the objects' wildcard expressions and removes duplicates and expressions that encode subsets.

Propagation Tree Analysis NetPlumber permanently tracks the propagation of network traffic and analyzes the state of invariant satisfaction while the invariants are attached to probe nodes, i.e., the leaves of the propagation trees. This approach has two drawbacks: First, regarding larger configuration changes, NetPlumber performs a lot of analysis efforts and might yield several analysis notifications about transient states while only the final result is of interest. And, second, reconstructing the final result from these transient notifications puts additional effort on NetPlumber's user, i.e., FaVe.

Therefore, we added a new operational mode to NetPlumber to perform analysis on demand. In order to check reachabilities, we analyze the whole

[1] We added several bug fixes and code enhancements to NetPlumber which we open-sourced along with FaVe: `https://github.com/cllorenz/fave-project`.

[2] `https://linux.die.net/man/3/realloc`

propagation tree in one run. Beginning at some source node, the analysis traverses the flow tree, collects all leaves, and checks if the packet sets that stem from the source may reach some destination, i.e., the host, network, or service. We rely on this feature when we verify network security compliance in Chapter 7.

To suppress NetPlumber's permanent analysis, we simply instrument all probe nodes with a FlowExp expression that always returns true while not requiring NetPlumber to perform further analysis of incoming flows.

Firewall Anomaly Detection We implemented an analysis that searches individual NetPlumber tables for firewall anomalies like shadowing or generalization. For instance, shadowing anomalies are detected by traversing the table from the highest to the lowest priority, collect the set of packets that were handled already, and compare that set with the next rule in order to determine if there are packets left for that rule. We rely on this feature when we search for firewall anomalies in Chapter 8 where we provide details on the supported anomalies and the respective detection algorithms. Throughout this analysis, we also utilize the compression of header space objects.

6.2 Embedding of DHSA in HSA

The embedding of DHSA in HSA, which is implemented in FaVe's Verification Engine Adapter for NetPlumber, is based on a *header mapping* Map_H and performed through a *rule embedding* function $embed\_r$. The header mapping ensures that values from DHSA's header field tuples are always stored in the same areas of the HSA wildcard expressions that form the rules in NetPlumber. The rule embedding function concisely encodes DHSA rules as sets of NetPlumber rules[3] while conserving the homomorphic properties of σ (cf. Section 5.5). The homomorphim is defined over the enumeration of single packets and a naïve implementation would not only be inefficient but run for an infeasibly long time. For instance, enumerating all 2^{256} IPv6 packets would take longer than the universe exists until this point in time[4]. In

[3] We use the terms *HSA rule* and *NetPlumber rule* interchangeably. We speak of HSA rules whenever we want to highlight the more conceptual character while we refer to implementation characteristics when we speak of NetPlumber rules.

[4] Assuming a single step took one picosecond, enumerating the IPv6 address space would require $2^{256} \approx 1.16 \times 10^{77}$ ps. Since the universe is about 13.7 billion years old (cf. `\cite{https://esahubble.org/science/age_size/}`), there have been 13.7bn years $\approx 4.32 \times 10^{17}$ s $\approx 4.32 \times 10^{29}$ ps since the big bang which is much shorter.

the following, we will describe an efficient solution for this matter and discuss the formal implications in greater detail in Section 6.2.3.

6.2.1 Header Mapping

The header mapping Map_H stores a header field's position on the HSA wildcard expressions which comprise the internal data structures of NetPlumber. Formally, the header mapping is defined as $Map_H \subseteq \mathcal{H}^D \times \mathbb{N} \times \mathbb{N}$. An entry $(h, l, u) \in Map_H$ for a header field $h \in \mathcal{H}^D$ consists of the embedding boundaries $l, u \in \mathbb{N}$ on the HSA wildcard expression where $l < u < \sum_{h \in \mathcal{H}^D} bits(h)$. The helper function $bits : \mathcal{H}^D \to \mathbb{N}$ returns the bitwidth of some header field, e.g., 128 bits for the IPv6 destination address. Given our example rule set, the header mapping is the following (the headers were added in the order of their appearance in the firewall rule set):

$$
\begin{aligned}
Map_H = \{ & \\
& (\texttt{iif}, 0, 31), \\
& (\texttt{sip}, 32, 159), \\
& (\texttt{state}, 160, 167), \\
& (\texttt{oif}, 168, 199), \\
& (\texttt{dip}, 200, 327), \\
& (\texttt{proto}, 328, 335), \\
& (\texttt{dport}, 336, 351) \\
\} &
\end{aligned}
$$

For instance, the IPv6 source address starts at bit 32 and ends at bit 159 since it is 128 bits wide and $32 + 128 - 1 = 159$.

6.2.2 Embedding Rules and Rule Sets

As sketched in Algorithm 6.1, we embed a set of DHSA rules in a NetPlumber table by expanding the individual DHSA rules into sets of DHSA rules that, in turn, can be directly translated to distinct NetPlumber rules. Meanwhile, we preserve the matching semantics of the original DHSA rule set.

Input:

- DHSA rule set R
- Header Mapping Map_H
- NetPlumber table

Output: A set of NetPlumber rules that can be placed in the NetPlumber table.

Procedure:

1. Partition the NetPlumber table in order to fit the NetPlumber rules constructed from R.
2. For each DHSA rule r from R, create a set of distinct NetPlumber rules in the following manner:
   (1) Map the rule's action to the according NetPlumber forwarding ports.
   (2) Expand the rule's DHSA match to a set of wildcardable DHSA matches that, together, match the same packets.
   (3) Translate the wildcardable DHSA matches into HSA wildcard expressions according to the header mapping Map_H.
   (4) Create a distinct index for each wildcard expression that points into the table space of the NetPlumber table which has been reserved for this DHSA rule.
   (5) Construct NetPlumber rules using these indices, wildcard expressions, and forwarding ports.
3. Return all NetPlumber rules that were created in this way.

Algorithm 6.1.: Top level algorithm that transforms a DHSA rule set into a set of NetPlumber rules that can be installed in a NetPlumber table.

In the following, we illustrate the algorithm's application for the fifth DHSA rule r_5 from our example rule set (cf. Figure 6.13), i.e.:

$$
\begin{aligned}
r_5 : \{ & \\
& (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\}), \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}) \\
\} \quad & \rightarrow \mathit{accept}
\end{aligned}
$$

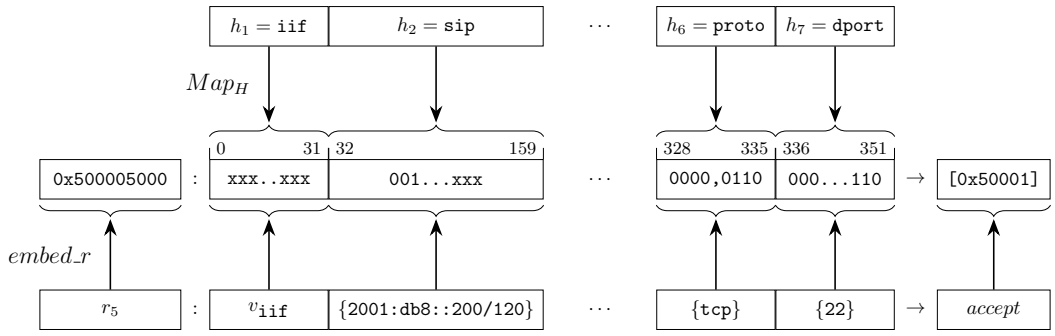


Fig. 6.12.: Map_H mapping and $embed\_r$ embedding from DHSA in HSA for the rule r_5 from the example rule set.

The mapping Map_H and embedding through $embed\_r$ is depicted in Figure 6.12. First, the *accept* action is mapped to port `0x50001`. Then, it is checked whether the DHSA match is wildcardable. Since this is the case, the wildcard expression is constructed by translating all header field values to wildcard expression snippets and positioning these in the resulting wildcard expression according to Map_H. E.g., the protocol field's value {`tcp`} as `0000,0110` from bit 328 to bit 335. Afterwards, the rule index is calculated as `0x500005000` and, finally, the NetPlumber rule is compiled.

Rule Embedding

Formally, the rule embedding function is defined as follows:

$$
\begin{aligned}
&embed\_r : \mathbb{N} \times R^D \rightarrow 2^{R^H} \\
&embed\_r(t, r_i : m_i \rightarrow a_i) := \{ \\
&\quad index(t, i, j-1) : wc\_m(m_j, sort_l(Map_H)) \rightarrow action(t, a_i) \mid \\
&\qquad M' := expand\_m(m_i), \\
&\qquad j \in 1..|M'|, \\
&\qquad m_j \in M' \\
&\}
\end{aligned}
$$

where:

$R^D := \mathbb{N} \times M \times A$ denotes DHSA rules where M is the set of DHSA matches and $A := \{accept, drop\}$ are actions (DHSA rules have been described previously in Section 5.5).

$R^H := \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathcal{H} \times \mathcal{H} \times \mathcal{H} \times 2^{\mathbb{N}}$ denotes NetPlumber rules (NetPlumber rules have been described previously in Section 2.2.2.1).

$t \in \mathbb{N}$ is the NetPlumber table's identifier.

$r_i \in \mathbb{N}$ is the DHSA rule's index.

$m_i \in M$ is the DHSA rule's match.

$a_i \in A$ is the DHSA rule's action.

$index()$ calculates the index for the resulting NetPlumber rule.

$wc\_m()$ encodes a wildcardable DHSA match as a HSA wildcard expression.

$action()$ calculates NetPlumber forwarding ports for DHSA actions.

$sort_l()$ sorts a header mapping ascending by the lower bound.

$expand\_m()$ expands a DHSA match to a set of wildcardable matches.

In the following, we describe these functionalities in greater detail.

Action Resolution In particular, the action resolution is defined as:

$$action : \mathbb{N} \times A \to 2^{\mathbb{N}}$$
$$action(t, a) := \begin{cases} \{t \cdot 2^{16} + 1\}, \text{ if } a = accept \\ \emptyset, \text{ else} \end{cases}$$

We model the acceptance of packets by forwarding through a NetPlumber port which is calculable using the table's index. Dropping packets, on the other hand, is performed by forwarding through no port at all which causes NetPlumber to stop propagating these packets. Concerning our example, resolving the action results in the port set $\{$`0x50001`$\}$ since the target table has the index `0x5` and the rule's action is *accept*.

Match Translation Before constructing wildcard expressions for the NetPlumber rules, the DHSA rule's match is expanded using the function $expand\_m$ that takes a

DHSA match and creates a DHSA match set which matches the same packets but where each match is wildcardable. In particular, $expand\_m$ is defined as

$$
\begin{aligned}
&expand\_m : M \to \mathcal{M} \\
&expand\_m(m) := \{ \\
&\quad \{(h_i, v'_{h_i}) \mid \forall i = 1..|\mathcal{H}^D|, h_i \in \mathcal{H}^D\} \mid \\
&\qquad (h_i, v_{h_i}) \in m, \\
&\qquad (h_i, l, u) \in Map_H, \\
&\qquad v'_{h_i} \in split\_v(v_{h_i}, u + 1 - l), \\
&\qquad \forall i = 1..|\mathcal{H}^D|, h_i \in \mathcal{H}^D\}
\end{aligned}
$$

where the helper function $split\_v : \mathcal{V}_{\mathcal{H}} \times \mathbb{N} \to 2^{\mathcal{V}_{\mathcal{H}}}$ splits an arbitrary value set into a set of wildcardable value sets. Matches from $expand\_m$ are required to be embeddable using a single HSA wildcard expression. Due to $split\_v$, this is the case since the value set of each header field can be represented using a single wildcard expression. E.g., the destination IP address range `2001:db8::100/120` from our example rule r_5 is wildcardable since it can be represented by

`00100000,00000001,00001101,10111000,000...000,00000001,xxxxxxxx`.

and, hence, $split\_v(v_{\texttt{dip}}) = \{v_{\texttt{dip}}\}$. Also, all other header fields of r_5 are wildcardable and $expand\_m$ yields the match set $\{m_5\}$

Now, each match from the match set produced by $expand\_m$ is converted to a wildcard expression using the function wc_m which is formally defined as follows ($\circ$ denotes the concatenation of two wildcard expressions):

$$
\begin{aligned}
&wc\_m : M \times 2^{\mathcal{H}^D \times \mathbb{N} \times \mathbb{N}} \to \mathcal{H} \\
&wc\_m(m, \{(h, l, u)\} \cup Map_H) := \begin{cases} wc\_v(v, u + 1 - l), \text{ if } Map_H = \emptyset \\ \qquad (\text{where } (h, v) \in m) \\ wc\_v(v, u + 1 - l) \circ wc\_m(m, Map_H), \text{ else} \\ \qquad (\text{where } (h, v) \in m) \end{cases}
\end{aligned}
$$

The function traverses the header field mapping Map_H, creates a wildcard expression for the respective header field tuple's value set, and concatenates these snippets to a wildcard expression that encodes the whole match. For this purpose, $wc\_v : \mathcal{V}_{\mathcal{H}^D} \times \mathbb{N} \to \mathcal{H}$ translates a wildcardable value set to a wildcard expression with a certain width.

NetPlumber assumes that a bit at a certain position has the same meaning over all wildcard expressions but externalizes the specification of this meaning to the tool's

user. Hence, in order to use NetPlumber as a verfication engine, we need to make sure that the DHSA header fields always map onto the same bits in NetPlumber. For this purpose, we need to provide a stabily sorted version of Map_H that can be traversed from the most to the least significant bits. Therefore, we use the helper function $sort_l : 2^{\mathcal{H}^D \times \mathbb{N} \times \mathbb{N}} \to 2^{\mathcal{H}^D \times \mathbb{N} \times \mathbb{N}}$ which sorts Map_H in ascending order based on the entries' lower bounds.

Concerning our example, $sort_l(Map_H)$ yields Map_H since the header mapping is already sorted by the lower bounds. Then, $wc\_m(m_5, Map_H)$ traverses Map_H from `iif` to `dport` and transforms each header field tuple (h, v_h) from m_5 to a wildcard expression. E.g., for the ingress interface `iif` the expression is `xxx..xxx` with a length of $u+1-l = 31+1-0 = 32$ bits. These snippets are concatenated throughout the traversal of the header mapping. The whole wildcard expression for m_5 is shown in Figure 6.12.

Rule Placement Finally, the NetPlumber rules need to be constructed and placed in the NetPlumber table. For this purpose, $embed\_r$ calculates a globally unique index which places the rules within discrete intervals of the NetPlumber table while preserving the relative order, and hence matching semantics, of the original rule set. Based on the target table's index t, the DHSA rule's index i, and an offset n, the indices are calculated by the helper function

$$index : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$index(t, i, n) := (t \cdot 2^{32}) + (i \cdot 2^{12}) + n.$$

Concerning our example, $index(\texttt{0x5}, 5, 0)$ yields `0x500005000` which is in the interval $[\texttt{0x500005000}, \texttt{0x500005fff}]$ that has been reserved for the NetPlumber rules stemming from this DHSA rule. Tables in NetPlumber have a capacity of $2^{32} = 4,294,967,296$ entries and we reserve $2^{12} = 4096$ entries per DHSA rule. Hence, FaVe can support rule sets of up to $2^{32-12} = 16,777,216$ DHSA rules per NetPlumber table which is far more than we encountered in any of our studied real-world network configurations.

Rule Set Embedding

Whole DHSA rule sets are embedded by, first, embedding each rule individually and, then, unioning the resulting sets of NetPlumber rules. Formally:

$$embed\_t : \mathbb{N} \times 2^{R^D} \rightarrow 2^{R^H}$$
$$embed\_t(t, R) := \bigcup_{r_i : m_i \rightarrow a_i \in R} embed\_r(t, r_i : m_i \rightarrow a_i).$$

Since the indices in the DHSA rule set are unique, also the indices for the NetPlumber rules are distinct by construction. Further, the relative order between DHSA rules and the resulting NetPlumber rules is preserved by the embedding algorithm. Hence, also the matching semantics is abided throughout the transformation.

Given the example IPTables rule set from Section 5.2, the transformation from the configuration via DHSA to HSA is depicted in Figure 6.13. First, the IPTables modeling engine parses the rule set and creates a packet filter model with the forward filtering table holding the DHSA rules. After aggregating the packet filter model into the network model, the Model Aggregator, then, efficiently embeds the DHSA model in a HSA model to be placed in NetPlumber.

For this step, each rule is processed by $embed\_r$ where, first, the rule's action is mapped to a set of ports, i.e., $\emptyset$ for $drop$ (denoted as [] for NetPlumber ports) and $\{$`0x50001`$\}$ for $accept$ (denoted as [`0x50001`]). Then, second, the rule's match is expanded to a set of wildcardable matches. Since all matches from our example are already wildcardable, for each match m_i, $M' := expand\_m(m_i) = \{m_i\}$. Third, each wildcardable match is converted to a wildcard expression by traversing the header field mapping Map_H from the lowest to the highest entry, translating the respective header field tuple's value set to a wildcard expression, and concatenating these snippets to a wildcard expression for the whole match. These wildcard expressions are shown at the bottom in Figure 6.13. Fourth and finally, each rule $m_j \in M'$ gets a globally unique, semantics preserving index and a NetPlumber rule is constructed using this index, the wildcard expression, and the action's port set.

6.2.3 Discussion: Preservation of the homomorphic Properties of σ

We implemented the embedding of DHSA in HSA using wildcardable DHSA matches while we defined our homomorphism σ using packet enumeration. Despite the fact, that literal enumeration is practically infeasible due to the Header Space's size which comprises 2^l packets where l is the amount of bits that encodes all header fields.

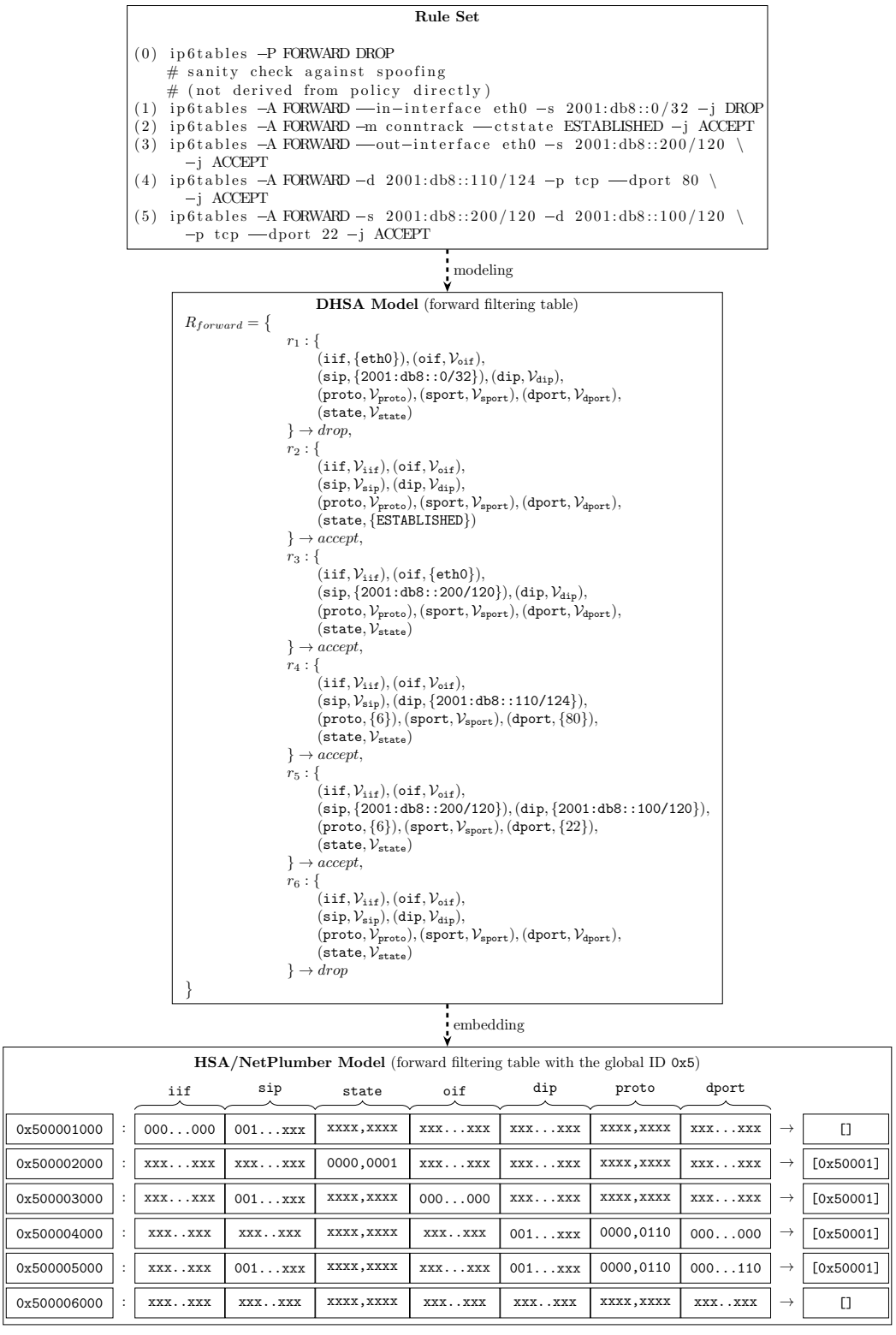


Fig. 6.13.: Example modeling of a packet filter rule set in terms of DHSA and its subsequent embedding in HSA.

This leaves the question whether the embedding's implementation in FaVe actually yields equivalent results?

We argue that splitting DHSA matches into sets of wildcardable matches is an efficient and effective technique to handle large amounts of packets simultaneously without

(a) losing any packet nor

(b) handling too many packets.

The match expansion $expand\_m$ builds upon the split into wildcardable value sets by $split\_v$ and the enumeration of all combinations of wildcardable value sets.

$split\_v$ splits a value set into a set of wildcardable value sets, i.e., $V := split\_v(v) = \{v_1, v_2, ..., v_n\}$ where $w := wc\_v(v')$ with $v' \in V$ yields a proper wildcard expression that precisely encodes v'. By construction, we know that $\bigcup_{v' \in V} v' = v$ and, therefore, we know that all values are encoded in some w and that also the encodings $\{wc\_v(v') \mid v' \in V\}$ contain all values from v. This satisfies the requirement (a) that no value is lost.

Further, there exists no value p in the set of wildcardable value sets that was not present in the original value set, i.e., $\nexists p \in \mathcal{V}_h : p \in v', v' \in split\_v(v)$. This is implied by the fact that by construction $\bigcup_{v' \in split\_v(v)} v' = v$ which leaves no room for any p as additional value. This satisfies the requirement (b) that no value is added.

Since $split\_v$ abides (a) and (b), the combinatorial enumeration over all header fields while creating the matches neither removes nor adds any packets, either. Hence, $expand\_m$ also abides (a) and (b) while preserving the semantics of enumerating single packets.

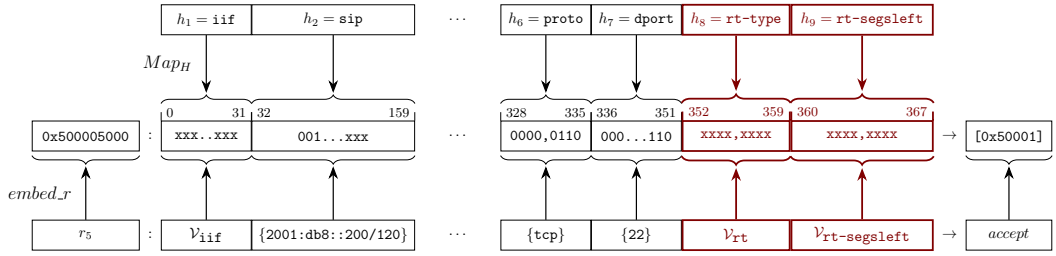
Note that there is no guarantee that the value sets created by $split\_v$ are mutually distinct, i.e., a value may be present in multiple value sets. Hence, also the matches resulting from the combinations of header fields may contain shared packets but there exists no packet that was not present in the original match. Therefore, this issue is a matter of a possibly inefficient implementation rather than formal assurance.

6.3 Handling of Dynamic Protocols

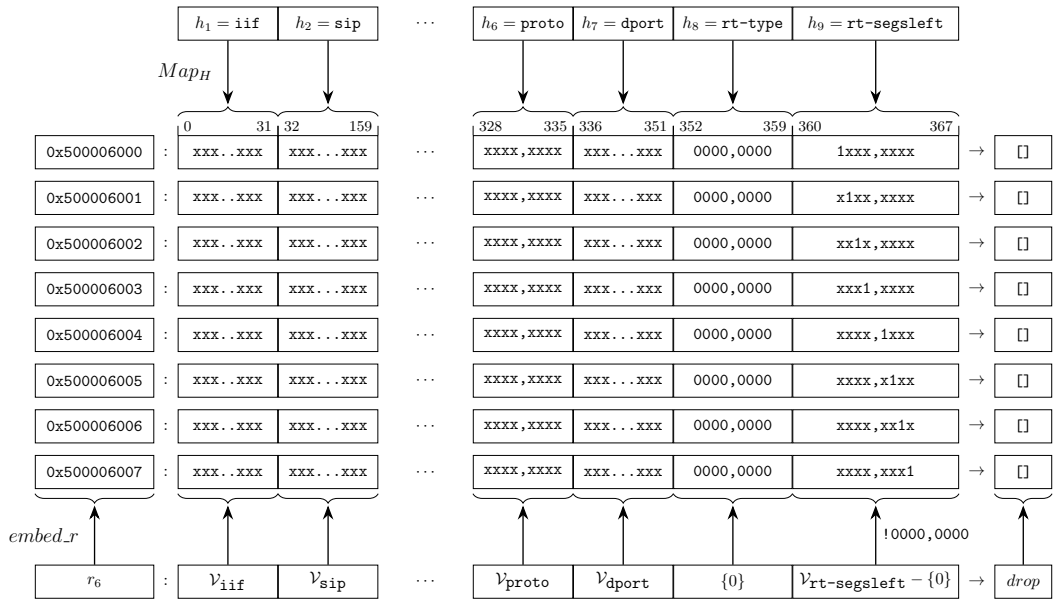
Dynamic protocol elements like the IPv6 extension header chains pose a challenge (cf. Section 2.3) for many network verification frameworks as discussed in the related

work in Chapter 3. We tackle this problem by allowing an expansion of the DHSA Header Space at runtime, i.e., users may introduce new header fields on the fly. Whenever a configuration is to be changed, we analyze its usage of header fields and, if necessary, expand the Header Space. Then, the field can be used when modeling the new configuration. This way, we are able to dynamically extends our model if needed by dynamic protocol elements like IPv6 extension header chains. On the other hand, if some dynamic element is never used by any device configuration, then it is not relevant for the network at all and does not need to be modeled. This fact also keeps our models concise in practice.

Runtime Expansion



(a) Expansion of an existing rule at runtime (expansion in red).



(b) Introduction of the new rule at runtime.

Fig. 6.14.: Expansion of the DHSA and HSA models at runtime for an example configuration after adding a new rule with additional header fields.

Given, that some change of a network device's configuration introduces the need to process some new header field h. Then, two steps need to be performed before the configuration can be modeled and instrumented by FaVe. First, the header mapping Map_H needs to be extended with the new header field. I.e., in the case of NetPlumber as verification backend, the new entry maps to the interval on the wildcard expression which is located just right of the rightmost entry $(h_r, l_r, u_r) \in Map_H$. Formally, some new entry (h, l_h, u_h) is introduced to Map_H where $l_h := u_r+1$ and $u_h := l_h + bits(h)$. Hence, (h, l_h, u_h) is the new rightmost entry in Map_H, i.e., $\nexists(h', l, r) \in Map_H : l > u_h$. Second, for all DHSA rules that already reside in FaVe and all HSA rules in NetPlumber that were created to encode the DHSA rules, the matches resp. wildcard expressions need to be expanded with the header field's default value set $\mathcal{V}_h$ resp. `xx..xx`.

For example, the firewall rule set from Section 6.2 (resp. Section 5.2) should be extended by a rule that filters IPv6 routing headers of type 0 with active segments[5]. The resulting rule set is the following:

```
(0) ip6tables -P FORWARD DROP
(6) ip6tables -A FORWARD -m rt --rt-type 0 \
      ! --rt-segsleft 0 -j 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::110/124 -p tcp \
      --dport 80 -j ACCEPT
(5) ip6tables -A FORWARD -s 2001:db8::200/120 \
      -d 2001:db8::100/120 -p tcp --dport 22 -j ACCEPT
```

[5]Filtering such packets is recommended after the deprecation of these headers in RFC 5095 [3].

Before modeling the new rule (6), first, the header mapping Map_H needs to be expanded with the header fields `rt-type` and `rt-segsleft` which have a width of eight bits each. The new mapping is as follows (new entries highlighted in red):

$$
\begin{aligned}
Map_H : \{ & \\
& (\texttt{iif}, 0, 31), \\
& (\texttt{sip}, 32, 159), \\
& (\texttt{state}, 160, 167), \\
& (\texttt{oif}, 168, 199), \\
& (\texttt{dip}, 200, 327), \\
& (\texttt{proto}, 328, 335), \\
& (\texttt{dport}, 336, 351), \\
& \textbf{(rt-type, 352, 359),} \\
& \textbf{(rt-segsleft, 360, 367)} \\
\} &
\end{aligned}
$$

Second, the existing rules in the DHSA model and the corresponding HSA rules in NetPlumber need to be expanded with the default value sets. Figure 6.14a shows this expansion for the existing rule (5). The DHSA rule (bottom) is expanded by adding the header field tuples $(\texttt{rt-type}, \mathcal{V}_{\texttt{rt-type}})$ resp. $(\texttt{rt-segsleft}, \mathcal{V}_{\texttt{rt-segsleft}})$ and the corresponding HSA wildcard expression residing in NetPlumber is expanded by $8 + 8 = 16$ bits that are all set as don't cares, i.e., all x. This expansion at runtime can be a costly operation as we will see later. Finally, we can model the new rule as depicted Figure 6.14b and add it to the model. Since the rule comprises the negation of the 8-bit `rt-segsleft` field, the DHSA rule expands to eight individual HSA rules. Particularly,

$$\mathcal{V}_{\texttt{rt-segslef}} - \{0\} = \{1..max(\mathcal{V}_{\texttt{rt-segslef}})\} = \{1..255\}$$

and

$$
\begin{aligned}
split\_v(\{1..255\}) = \{ & \\
& \{128..255\}, \\
& \{64..127, 192..255\}, \\
& \{32..63, 96..127, ..., 255\}, \\
& \{16..31, 48..63, ..., 255\}, \\
& \{8..15, 24..31, ..., 255\}, \\
& \{4..7, 12..15, ..., 255\}, \\
& \{2, 3, 6, 7, ..., 255\}, \\
& \{1, 3, 5, ..., 255\} \\
\} &
\end{aligned}
$$

since

$$\begin{aligned}
wc_v(\{128..255\}) &= \texttt{1xxxxxxx},\\
wc_v(\{64..127, 192..255\}) &= \texttt{x1xxxxxx},\\
wc_v(\{32..63, 96..127, ..., 255\}) &= \texttt{xx1xxxxx},\\
wc_v(\{16..31, 48..63, ..., 255\}) &= \texttt{xxx1xxxx},\\
wc_v(\{8..15, 24..31, ..., 255\}) &= \texttt{xxxx1xxx},\\
wc_v(\{4..7, 12..15, ..., 255\}) &= \texttt{xxxxx1xx},\\
wc_v(\{2, 3, 6, 7, ..., 255\}) &= \texttt{xxxxxx1x}, \text{ and}\\
wc_v(\{1, 3, 5, ..., 255\}) &= \texttt{xxxxxxx1}
\end{aligned}$$

(We apply the complementation algorithm for wildcard expressions from Section 2.2.1.2 to 00000000).

Evaluation: Microbenchmark

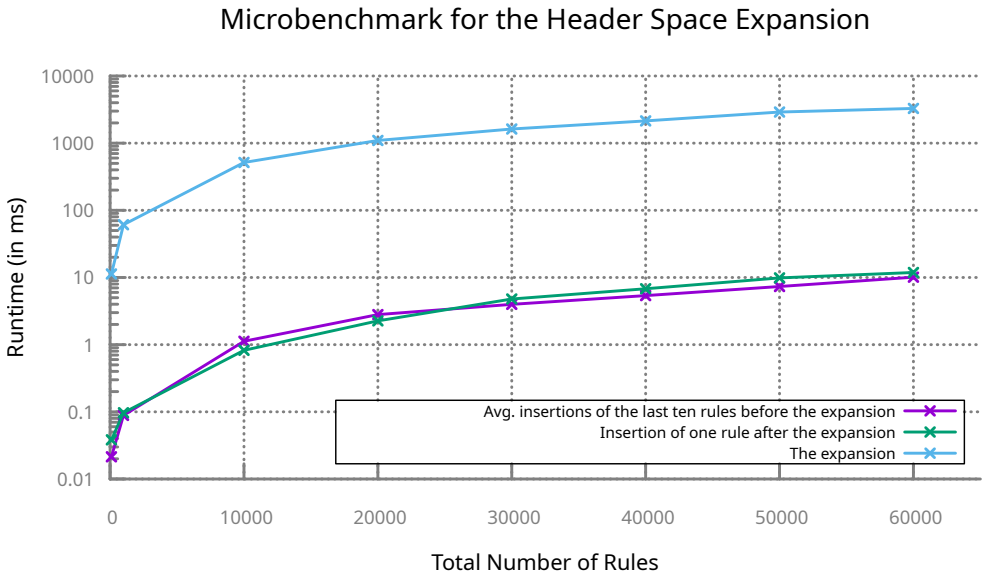


Fig. 6.15.: Measurement results of the runtime expansion in FaVe (logarithmic scale).

To quantify the impact of Header Space expansions at runtime, we microbenchmarked this issue[6]. For this purpose, we used the measurement environment as detailed in Section 1.4 and conducted a series of measurements. Each experiment was repeated ten times and consisted of the following series of steps:

[6]We published these experiments in [92] but, in order to improve comparability, we reran the measurements using the same implementation and environment as for all other measurements in this work.

1. FaVe is initialized with a model comprising a single table that holds $n - 10$ random rules with

   $$n \in \{10; 100; 1,000; 10,000; 20,000; 30,000; 40,000; 50,000; 60,000\}.$$

   The initial Header Space comprises the IPv6 destination address, the protocol number, and the destination port.

2. Then, additional 10 random rules are added to the table and their insertion time is measured and averaged.

3. Now, the Header Space is extended with the IPv6 source address and the source port header fields in one expansion operation whose runtime is measured, too.

4. Finally, another random rule – now with the new Header Space's size – is added and its insertion time is measured.

Figure 6.15 shows the measurement results of these experiments. The final single rule insertion's runtime lies slightly above the previous ten rule insertions before the expansion for most measurements. E.g., ∼12 ms versus ∼11 ms for $n = 60,000$. The Header Space expansion's runtime lies between 2 and 3 orders of magnitudes above the insertion of the final single rule, e.g., with a factor of about 295 for $n = 50,000$ (9.86 ms vs. 2,906.37 ms on average).

The measurements show that the Header Space's expansion at runtime is a costly operation and, if possible, it should only be used with care. Instead of running the operation multiple times, it might be better to aggregate configuration updates in order to introduce multiple additional header fields at once. We apply this strategy throughout the initialization phase where we can analyze several configurations beforehand and *"preheat"* FaVe's Header Space before adding the actual models. This way, the most relevant header fields are already present when configurations change and introduce new header fields over time.

Discussion: Header Space Convergence

In fact, we observed a convergence of the used header fields and, hence, the comparably high costs for expanding the Header Space is bearable since it enables modeling dynamic protocol elements with FaVe. This observation is not too surprising since FaVe only enables a monotone modeling of the Header Space at runtime, i.e., its expansion. This way, it gets less likely that further header fields are required over time, even though, configuration changes frequently. In essence, the Header

Space's complexity converges which is an important aspect when we evaluate FaVe's scalability.

6.4 Summary

In this chapter, we presented our fast verification framework *FaVe* which implements the modeling, solving, and reporting needed to realize the security management workflows as introduced in Chapter 1.

For this purpose, we detail FaVe's architecture, its modeling capabilities, and its extensibility. FaVe implements DHSA (cf. Chapter 5) for user-friendly modeling, the derivation of packet flow specifications from FPL roles to simulate traffic, and encodings of checks representing FPL rules. The homomorphism that embeds DHSA in HSA is realized efficiently by FaVe, too. Also, we extended NetPlumber with capabilities to compress header space objects, to expand the Header Space at runtime, to analyze propagation trees on demand, and to find firewall anomalies in NetPlumber tables (cf. Chapter 8 for details).

Finally, we explain how FaVe supports the handling of dynamic protocol elements through its ability to expand the Header Space at runtime and we give insights into the performance of this costly operation.

7 Use Case: Network Security Compliance

In this chapter, we enable security officials and network administrators to continuously verify the compliance of security policies specified with FPL in FaVe. We finalize network modeling, show feasibility as well as performance, and provide means to generate security configuration from FPL policies. For these purposes, we use DHSA (cf. Chapter 5) and the modeling primitives offered by FaVe (cf. Chapter 6) to model a broad variety of common network devices – especially stateful firewalls. Our evaluation shows that FaVe is able to verify common and complex scenarios, scales to large networks, and outperforms the state-of-the-art. These findings support the initial verification in the UNDERSTAND phase as well as the rapid reverifications necessary for the ADJUST phase. Further, we support the continuous uphold of compliance throughout the CONTROL phase by generating security configuration from FPL policies.

The remainder of this chapter is structured as follows: First, in Section 7.1 we introduce several models for network devices and firewalls – particularly switches in Section 7.1.1, routers in Section 7.1.2, stateless packet filters in Section 7.1.4, stateful packet filters with *state snapshots* in Section 7.1.5, and stateful packet filters with *state deduction* in Section 7.1.7 as an alternative and more scalable approach. For models based on *state snapshots*, state information, e.g., connections, is gathered at runtime which requires reverification whenever a state snapshot is updated. We perform an evaluation to quantify the rate of changes reverifiable by FaVe in Section 7.1.6 and reveal clear scalability limitations of this naïve approach. Hence, in Section 7.1.7, we offer an alternative approach that *deducts* stateful behavior by analyzing firewall rule sets statically using a technique called *state shell interweaving*. Further, in Section 7.1.3, for the first time in this work, we show that our approach towards continuous compliance is feasible for a real-world example. In Section 7.2 we then conduct thorough evaluations that show FaVe's ability to verify compliance, its scalability, and its competitiveness. Finally, in Section 7.3, we introduce means to generate firewall configurations from FPL policies.

7.1 Modeling Device Configurations

We ship FaVe with a series of predefined device models which will be introduced in the following. All device models use the modeling primitives as offered by our verification framework FaVe (cf. Section 6.1.6).

7.1.1 Switches

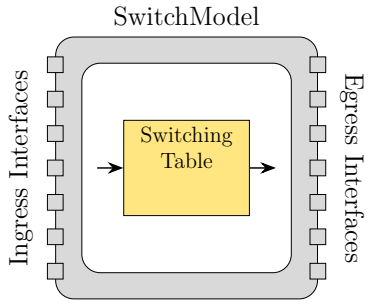


Fig. 7.1.: Model of a simple network switch.

As depicted in Figure 7.1, we model simple switches using a single table and an arbitrary amount of network interfaces. The model supports L2 as well as L3 switching using VLANs or MAC addresses resp. IP addresses. We implemented a parser for static Cisco-like configurations and snapshots of switching tables.

```
# Interfaces to VLANs
interface ge1/0/0
  description Upstream
  switchport trunk encapsulation dot1q
  switchport trunk allowed vlan 2
  switchport trunk allowed vlan 4095
  switchport mode trunk

interface ge/1/0/1
  description DMZ
  switchport access vlan 1
  switchport mode access

# static routes
vlan route 1 2
vlan route 2 1
vlan route 4095 1
```

Listing 7.1.: Example switch configuration with a Cisco-like syntax.

For instance, we model the DMZ switch from our example network (cf. Figure 4.1b) with two network interfaces `ge1/0/0` and `ge1/0/1`. Its static configuration is shown

in Listing 7.1 and modeling results in the following DHSA rule set for the switching table:

$$
\begin{aligned}
R_{switching} = \{ & \\
& r_1 : \{(\texttt{iif}, \{\texttt{ge1/0/1}\}), (\texttt{vlan}, \{1\})\} \rightarrow \texttt{ge1/0/0}, \\
& r_2 : \{(\texttt{iif}, \{\texttt{ge1/0/0}\}), (\texttt{vlan}, \{2\})\} \rightarrow \texttt{ge1/0/1}, \\
& r_3 : \{(\texttt{iif}, \{\texttt{ge1/0/0}\}), (\texttt{vlan}, \emptyset)\} \rightarrow \texttt{ge1/0/1} \\
\}. &
\end{aligned}
$$

Note that we encode the forwarding ports as actions of the DHSA rules as explained in Section 6.2.2. I.e., throughout analysis, packets that match such a rule are propagated through the denoted interface.

7.1.2 Routers

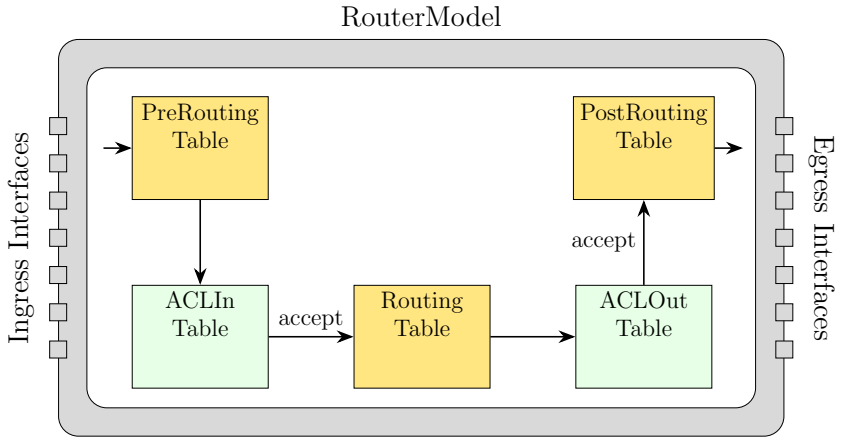


Fig. 7.2.: Model of a network router. The green tables contain rules derived from ACL configurations whereas the yellow tables originate from static device configuration and routing information.

We model routers using an arbitrary amount of network interfaces and five tables as shown in Figure 7.2. In the following, we illustrate these tables' purposes using the example configurations for a Cisco router from Listing 7.2. The example mimics the example network from Section 4.1 but uses a router instead of a stateful packet filter. Note that our router model is stateless and, hence, its ACLs cannot realize the stateful filtering used in the original IPTables rule set.

PreRouting This table tags packets with their ingress interface and assures that only packets with suitable VLAN tags associated with that interface are processed

```
# Interfaces to VLANs
interface ge1/0/0
  description Internet IF
  switchport access vlan 4095
  switchport mode access


interface ge1/0/1
  description DMZ IF
  switchport trunk encapsulation dot1q
  switchport trunk allowed vlan 1
  switchport mode trunk

interface ge1/0/2
  description Office IF
  switchport trunk encapsulation dot1q
  switchport trunk allowed vlan 2
  switchport mode trunk
```

(a) Interfaces.

```
# VLANs to IPv6-Ranges and ACLs
interface vlan 4095
  description Internet VLAN
  ipv6 address 0::0/0
  ipv6 access-group 100 in
  ipv6 access-group 101 out

interface vlan 1
  description DMZ VLAN
  ipv6 address 2001:db8::100/120
  ipv6 access-group 102 in
  ipv6 access-group 103 out

interface vlan 2
  description Office VLAN
  ipv6 address 2001:db8::200/120
  ipv6 access-group 104 in
  ipv6 access-group 105 out
```

(b) VLANs.

```
# Internet ingress
access-list 100 deny ipv6 2001:db8::0/32 any
access-list 100 permit ipv6 any any

# Internet egress
access-list 101 permit 2001:db8::100/120 any
access-list 101 permit 2001:db8::200/120 any
access-list 101 deny ipv6 any any

# DMZ ingress
access-list 102 permit ipv6 2001:db8::100/120 any
access-list 102 deny ipv6 any any

# DMZ egress
access-list 103 permit ipv6 any 2001:db8::110/124
access-list 103 permit ipv6 2001:db8::200/120 2001:db8::100/120
access-list 103 deny ipv6 any any

# Office ingress
access-list 104 permit ipv6 2001:db8::200/120 any
access-list 104 deny ipv6 any any

# Office egress
access-list 105 permit ipv6 any 2001:db8::200/120
access-list 105 deny ipv6 any any
```

(c) Access control lists.

```
# static routing
ipv6 route 2001:db8::100/120 vlan 1
ipv6 route 2001:db8::200/120 vlan 2
ipv6 route any vlan 4095
```

(d) Static routing.

Listing 7.2.: Example router configurations.

by the router. I.e., the tables drops packets with mismatching VLAN tags for trunk interfaces or sets VLAN tags for access interfaces.

Concerning the example, the PreRouting table is populated based on the configurations for interfaces and VLANs, i.e., Listing 7.2a resp. Listing 7.2b, resulting in the following DHSA rule set:

$$
\begin{aligned}
&R_{pre} = \{ \\
&\quad r_1 : \{(\texttt{iif}, \{\texttt{ge1/0/0}\}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{accept}, \\
&\quad r_2 : \{(\texttt{iif}, \{\texttt{ge1/0/1}\}), (\texttt{vlan}, \{1\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{accept}, \\
&\quad r_3 : \{(\texttt{iif}, \{\texttt{ge1/0/2}\}), (\texttt{vlan}, \{2\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{accept}, \\
&\quad r_4 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \mathcal{V}_{\texttt{vlan}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{drop} \\
&\}.
\end{aligned}
$$

ACLIn Incoming packets are filtered in this table according to some filtering rules specified by the administrator. In addition, in the case of IPv4, the ACL filters packets that come from the Internet and have source addresses from IP ranges that are reserved for internal networks, i.e., `10.0.0.0/8`, `172.16.0.0/12`, and `192.168.0.0/16` [122].

The ACLIn table is derived from our example's VLAN and ACL configurations, i.e., Listing 7.2b resp. Listing 7.2c. For each VLAN, we copy the configured access list, e.g., access list 100 for VLAN 4095, and set the VLAN's tag in these rules. The resulting DHSA rule set is the following:

$$
\begin{aligned}
&R_{acl\_in} = \{ \\
&\quad r_1 : \{ \\
&\qquad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{4095\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
&\qquad (\texttt{sip}, \{ \\
&\qquad\quad \texttt{64:ff9b::10.0.0.0/104}, \\
&\qquad\quad \texttt{64:ff9b::172.16.0.0/108}, \\
&\qquad\quad \texttt{64:ff9b::192.168.0.0/112} \\
&\qquad \}) \\
&\quad \} \rightarrow \mathit{drop}, \\
&\quad r_2 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{drop}, \\
&\quad r_3 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{accept} \\
&\quad r_4 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{1\}), (\texttt{sip}, \{\texttt{2001:db8::100/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{accept}, \\
&\quad r_5 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{1\}), (\texttt{sip}, \{\mathcal{V}_{\texttt{sip}}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{drop}, \\
&\quad r_6 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{2\}), (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{accept}, \\
&\quad r_7 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{2\}), (\texttt{sip}, \{\mathcal{V}_{\texttt{sip}}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{drop} \\
&\}
\end{aligned}
$$

Routing In this table, packets are routed according to their destination IP addresses. Particularly, they are retagged with a VLAN that routes out of an interface facing the next hop. We support static configurations and snapshots of routing tables.

Modeling our example's static configuration, i.e., from Listing 7.2d, results in:

$$
\begin{aligned}
&R_{routing} = \{ \\
&\quad r_1 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \mathcal{V}_{\texttt{vlan}}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), (\texttt{dip}, \{\texttt{2001:db8::100/120}\})\} \rightarrow vlan\_1, \\
&\quad r_2 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \mathcal{V}_{\texttt{vlan}}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), (\texttt{dip}, \{\texttt{2001:db8::200/120}\})\} \rightarrow vlan\_2, \\
&\quad r_3 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \mathcal{V}_{\texttt{vlan}}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow vlan\_4095 \\
&\}
\end{aligned}
$$

Note that the rules' actions encode the rewriting of the VLAN field of matched packets as explained in Section 6.2.2. E.g., according to the first rule, packets towards `2001:db8::100/120` are tagged with VLAN 1.

ACLOut Outgoing packets can be filtered as well using filtering rules provided by the administrator.

Analogously to the ACLIn table, modeling our example's ACLOut table uses the VLAN and ACL configurations, i.e., Listing 7.2b resp. Listing 7.2c, resulting in the following DHSA rule set:

$$
\begin{aligned}
&R_{acl\_out} = \{ \\
&\quad r_1 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \{\texttt{2001:db8::100/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow accept, \\
&\quad r_2 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow accept, \\
&\quad r_3 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow drop, \\
&\quad r_4 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{1\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\})\} \rightarrow accept, \\
&\quad r_5 : \{ \\
&\qquad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{1\}), \\
&\qquad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \{\texttt{2001:db8::100/120}\}) \\
&\quad \} \rightarrow accept, \\
&\quad r_6 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{1\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow drop, \\
&\quad r_7 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{2\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::200/120}\})\} \rightarrow accept, \\
&\quad r_8 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{2\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow drop \\
&\}
\end{aligned}
$$

PostRouting This last table prevents simple routing loops and forwards packets through the egress interfaces. For these purposes it, first, statically filters packets eligable for routing through their respective ingress interfaces. Then, it clears the ingress port field and optionally strips VLAN tags from packets to be send out of ports which are in access mode.

Concerning our example, the PostRouting table is modeled using the interface and VLAN configurations, i.e., Listing 7.2a resp. Listing 7.2b, and the resulting DHSA rule set is the following:

$$
\begin{aligned}
R_{post} = \{ \\
&r_1 : \{(\texttt{iif}, \{\texttt{ge1/0/0}\}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{drop}, \\
&r_2 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{4095\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \texttt{ge1/0/0}, \\
&r_3 : \{(\texttt{iif}, \{\texttt{ge1/0/1}\}), (\texttt{vlan}, \{1\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{drop}, \\
&r_4 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{1\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \texttt{ge1/0/1}, \\
&r_5 : \{(\texttt{iif}, \{\texttt{ge1/0/2}\}), (\texttt{vlan}, \{2\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \mathit{drop}, \\
&r_6 : \{(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{vlan}, \{2\}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}})\} \rightarrow \texttt{ge1/0/2} \\
\}
\end{aligned}
$$

7.1.3 Microbenchmark: The IFI Network

At this point, we are able to demonstrate FaVe's ability to verify the compliance of a real-world network. We use FaVe to model a real-world network of our computer science institute consisting of a router and multiple switches. We verify compliance against a FPL policy which was formulated by the security official who has been responsible for our institute. The *IFI* benchmark uses the real router ACL configuration and the topology of our institute's network. The FPL inventory and policy (cf. Appendix A.8) were created by our security official. The network, inventory, and policy are of reasonable size to be classified as a non-trivial example but they can still be inspected manually by a human being. Therefore, we gain high confidence in the correctness of our prototype's implementation. The ACL configuration can be found in Appendix A.9.

Devices	1 router, 16 switches, 1 dummy
Subnets	16
ACL Rules	75
Routing	IPv4
Header Bits	144
Policy Checks	288
Stateful Rules	no (resp. treated as bidirectional stateless)
Header Fields	`iif`, `oif`, `vlan`, `sip`, `dip`

Tab. 7.1.: Overview of the IFI benchmark.

As shown in Table 7.1 the network of the IFI benchmark consists of a central router, 16 switched subnets, and a dummy representing the Internet. The router is configured with 75 Cisco ACL rules and routing uses IPv4. The total amount of

header bits included in this benchmark is 144, e.g., for IPv4 addresses or VLAN tags. The FPL policy comprises of four stateful rules but since router ACLs offer only stateless rules, we treat these policy rules as if they used the bidirectional stateless operator instead. The individual policy checks for pairwise reachability sum up to 288[1].

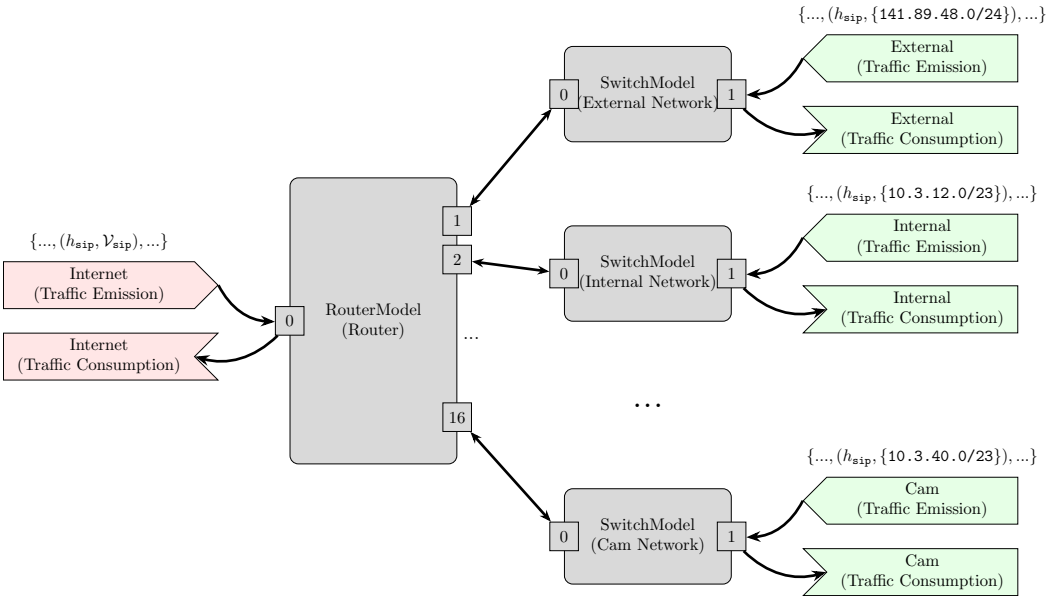


Fig. 7.3.: Model of the IFI network.

As shown in Figure 7.3, we modeled the network using the previously introduced router and switch models. The Internet is represented by a traffic emitter as well as a traffic consumer. The emitter generates traffic from any IPv4 address, i.e., the source IP range is 0.0.0.0/0. Each network segment is modeled as a switch and an emitter-consumer-pair where the emitter generates packets from the network's IPv4 address range as stated in the FPL inventory, e.g., 10.3.40.0/23 for the *Cam* network.

In particular, we measure the runtime of three phases:

1. The *initialization* phase comprises all tasks needed for setting up the network, e.g., including the instrumentation of tables, their connection with links,

[1]The amount of checks is quadratic since each leaf role (resp. non-super role) plus the built-in Internet role is checked against any other leaf role and the Internet. The only exception is given by the Internet role which is not checked against itself since the Internet cannot be governed by the organization that specifies the policy. Hence, the exact formula is

$$(\#leaf\_roles + 1)^2 - 1 := (16 + 1)^2 - 1 = 17^2 - 1 = 289 - 1 = 288.$$

and their population with rules. FaVe logs the runtime for each task while NetPlumber logs the total runtime for this phase.

2. During the *reachability* phase, the propagation of packets through the model is measured. For this purpose, FaVe and NetPlumber perform the tasks of setting up traffic emitters and connecting these with the network model. FaVe logs the runtime for each task while NetPlumber logs the total duration of this phase.

3. The *compliance* phase checks if the reachability trees conform with the policy which is a single task in FaVe as well as NetPlumber. Hence, both provide this phase's runtime with a single log message.

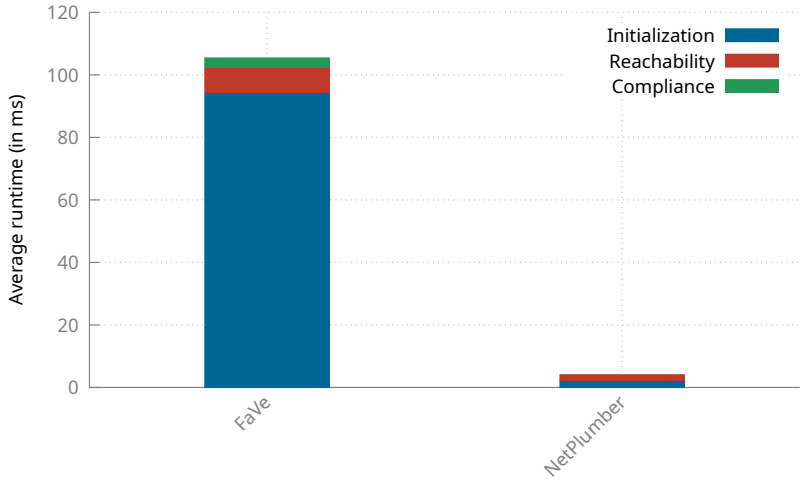


Fig. 7.4.: Average runtimes after ten repetitions of the IFI benchmark (in ms). For each experiment, the coefficient of variation of the total runtime is about 6 % for FaVe and 3 % for NetPlumber.

Figure 7.4 shows the average runtime after ten repetitions of the benchmark. With about 105 ms the overall runtime of this functionally oriented benchmark is negligable. In comparison with NetPlumber, FaVe shows a significant overhead as its total runtime is about 26 times longer (105.41 ms vs. 4.06 ms). The major part is due to the initialization which is roughly 41 times slower (94.37 ms vs. 2.29 ms). The reachability analysis has a more limited impact as it takes about 4.5 times longer (8.03 ms vs. 1.77 ms) but with a much shorter phase runtime.

Nevertheless, the real-world IFI benchmark shows that FaVe's compliance verification workflow works in an end-to-end and a functionally correct manner. We have shown that the network's configuration – including access control – conforms with the security requirements stated as FPL policies. The total runtime of well under one second is negligable but since the workload's size is also limited, further investiga-

tions regarding FaVe's performance are necessary. We provide such evaluation in Section 7.2.

7.1.4 Stateless Packet Filters

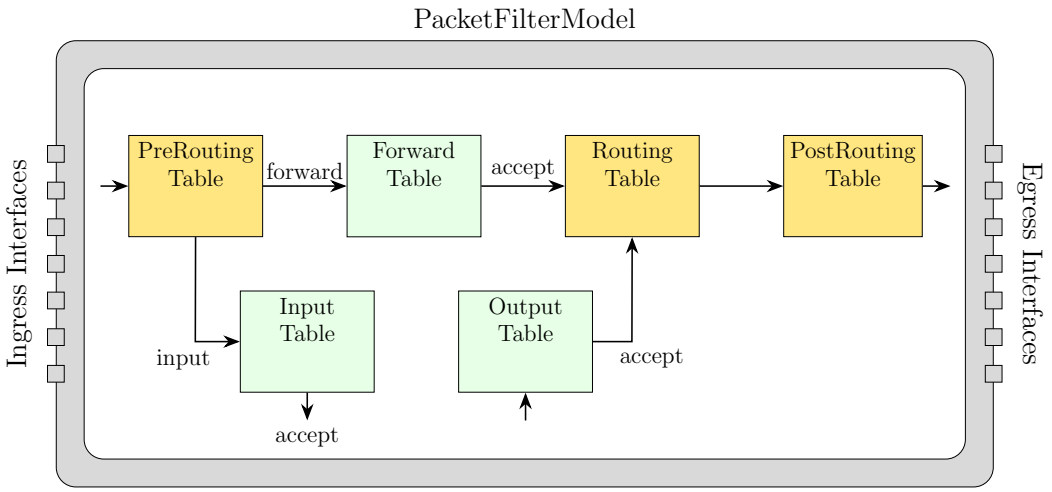


Fig. 7.5.: Model of a stateless packet filter. The green tables are instrumented using the security configuration while the yellow tables are either static, i.e., PreRouting and PostRouting, or derived from device configuration or snapshots, i.e., Routing.

Stateless packet filters represent the most basic variant of a firewall. They filter packets based on lists of static header fields and typically operate on the OSI layers 3 and 4, e.g., based on IP address ranges, the protocol field, and service ports. FaVe's model is shown in Figure 7.5 and consists of six tables.

PreRouting Incoming packets arrive at the PreRouting table and can be forwarded in two ways. First, if they are addressed at the packet filter's IP address, they are forwarded to the *Input* table. Otherwise, they are passed to the *Forward* table. For this purpose, the PreRouting table comprises two ports that encode the *input* and *forward* actions. Further, the table contains two static rules (assume the packet filter's IP address, for instance, is 1.2.3.4):

$$
\begin{aligned}
&R_{pre} = \{ \\
&\quad r_1 : \{..., (\texttt{dip}, \{1.2.3.4\}), ...\} \rightarrow input \\
&\quad r_2 : \{..., (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), ...\} \rightarrow forward \\
&\}
\end{aligned}
$$

Input Packets destined at the packet filter are further inspected in the Input table. After being accepted, the packets are processed by the firewall's internals, e.g., passed to application sockets. We do not model these internals but in FaVe we can attach a consumer that represents the firewall in FPL policies. In the case of IPTables, the rules from the `INPUT` chain are installed in this table.

Forward If a packet should be forwarded by the firewall, it is inspected in the Forward table and, if accepted, passed on for routing. For instance, for IPTables, this table is instrumented with the rules from the `FORWARD` chain.

Routing In this table, packets are prepared for emission towards the network. Based on destination criteria like IP addresses, the packet is tagged with an egress interface, i.e., it sets the `oif` fields accordingly, and passed to post routing. In FaVe, this table can be configured using static configuration or snapshots of a packet filter's routing table.

PostRouting Based on the routing decision, this table forwards packets through the respective egress interface. But, before that, it clears the packet's `oif` field. Also, the PostRouting table prevents simple routing loops by dropping packets where the ingress and egress interfaces, i.e., `iif` and `oif`, are the same.

Output Further, packets that originate from the packet filter are filtered by the Output table in a similar way as in the Input and Forward tables. For instance, for IPTables, rules from the `OUTPUT` chain are installed in this table. Accepted packets are, then, routed and emitted through the respective egress interfaces.

Today, pure stateless packet filters are seen rather rarely in practice. On the one hand, modern routers support stateless filtering of the most common header fields from the OSI layers 3 and 4, i.e., IP addresses, the protocol field, and service ports. On the other hand, state tracking in software packet filters, e.g., Linux' `iptables` or OpenBSD's `pf`, is computationally inexpensive and, therefore, typically enabled by default. This is driven by the fact that the most simple and generic implementation of a packet filter's decision process on a packet is to linearly traverse a rule set until a matching entry is found and the respective action can be applied. The diversity of header fields and arbitrary combinability in rules often prevent to use classification algorithms that are more efficient than a linear search [56]. In contrast, state tracking is based on a well known set of header fields, typically the quintuple

comprising of the source and destination address, the protocol field, and the source and destination ports. This allows to use hash maps with constant access complexity as seen in `iptables` or binary search trees with logarithmic complexity as used in `pf`.

7.1.5 Stateful Packet Filters with State Snapshots

Our model for stateful packet filters extends the stateless packet filter model (cf. Section 7.1.4) by introducing state tables that are instrumented at runtime – a technique we refer to as *state snapshotting*. The idea is that we enable the gathering of state information from the firewalls' state tables and instrumenting the corresponding firewall models with this information.

But why is it necessary to cover stateful behaviour at all? The main reason lies in the fact that, in order to verify compliance of FPL rules using the stateful operator, it is necessary to verify packet flows in both directions between two roles. Especially, the packets in backwards direction need to be strictly stateful in the sense that they must not pass if no previous packet in forth direction initiated a connection.

Therefore, in [92], we introduced a stateful packet filter model which is depicted in Figure 7.6. While the PreRouting, Routing, and PostRouting tables remain the same, the Input, Output, and Forward tables are now denotes with *Filter*, e.g., *Input Filter*, and accompanied by a *State* table, e.g., *Input State*. In the model's pipeline, the state tables are processed before their respective filtering tables. If a state table contains a matching state entry, a packet is accepted right away and processed accordingly, e.g., if it is accepted by the Forward State Table, it is routed without further inspection by the Forward Filter Table. If a state table does not contain a matching state entry, the table misses and the packet is passed to the respective filtering table.

Entries in the state tables have a restricted form, i.e., they necessarily have the source resp. destination address fields, the protocol field, and the source resp. destination port fields set to single values whereas all other fields have the full value domain specified. Also, per connection, there are two entries – one in forth and one in backwards direction. I.e., the backwards entry has the source and destination fields

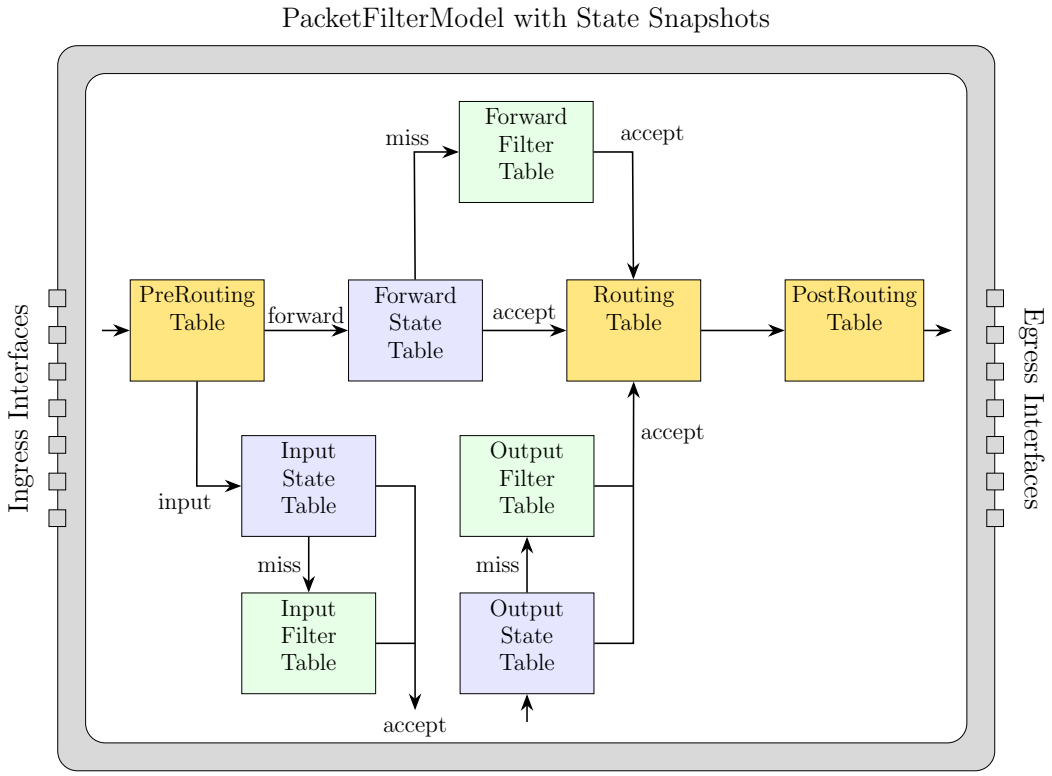


Fig. 7.6.: Model of a stateful packet filter with state snapshots. The green tables are instrumented using the security configuration while the yellow tables are either static, i.e., PreRouting and PostRouting, or derived from device configuration or snapshots, i.e., Routing. The blue state tables are determined at runtime based on snapshots of the observed firewall's corresponding state tables.

swapped. For instance, a state table with one entry, i.e., represented by two rules r_1, r_2, and the default miss-rule may look like this:

$$
\begin{aligned}
& r_1 : \{ \\
& \quad (\texttt{sip}, \{1.2.3.4\}), (\texttt{dip}, \{4.3.2.1\}), \\
& \quad (\texttt{proto}, \{\texttt{tcp}\}), (\texttt{sport}, \{13579\}), (\texttt{dport}, \{80\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_2 : \{ \\
& \quad (\texttt{sip}, \{4.3.2.1\}), (\texttt{dip}, \{1.2.3.4\}), \\
& \quad (\texttt{proto}, \{\texttt{tcp}\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \{13579\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{\texttt{0xffffffff}} : \{ \\
& \quad (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& \quad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}) \\
& \} \rightarrow \mathit{miss} \\
& \}.
\end{aligned}
$$

Our model uses a simplified state machine that comprises of only two states: *established* and *closed*. Only established connections have entries in the state tables while all other states, e.g., from the TCP state machine, are implicitly handled as closed. From a security point of view this is not a problem, since the real state table is maintained by a real firewall that guarantees the correct initialization of the connection. If the connection setup does not succeed, then there would have been no information flow that was not covered by the stateless rules in the rule table before. Thus, it is safe to have an abstract view on the state table in the model.

7.1.6 Microbenchmark: Request Throughput Limitations

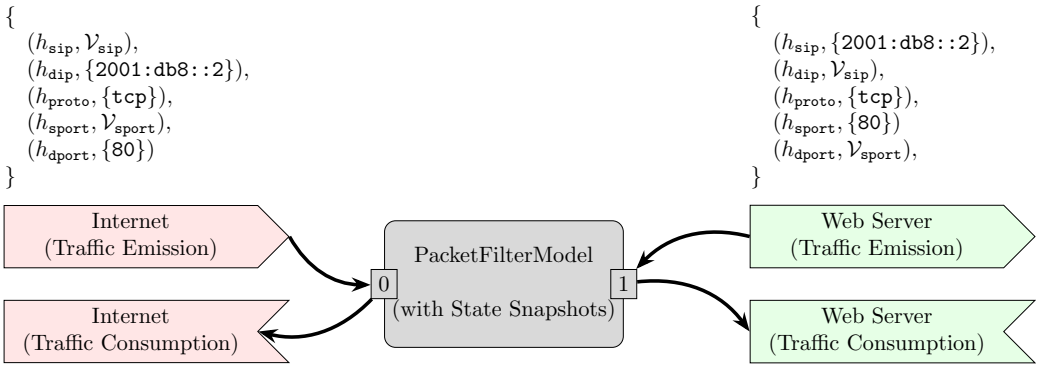


Fig. 7.7.: Network model of the benchmark.

Since state snapshots are dependent on runtime data, we conducted measurements in order to determine possible scalability limitations. We approach this question by modeling a minimal network, which includes some state snapshotting, and check for limitations. If the latter occur in this small example, state snapshotting as a modeling technique will, most likely, also fall short for larger networks.

As depicted in Figure 7.7, our example network consists of the Internet, a stateful packet filter based on state snapshots, and a single web server. The firewall is instrumented using the following rule set:

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

The second rule (2) allows packets destined to the web server and the first rule (1) accepts packets belonging to the connections that were initially permitted by the second rule. The rule set's default (0) is to drop packets. The network's setup and

configuration force entries of the forwarding state table to have a certain form, i.e., incoming packets have `2001:db8::2` as destination IP, TCP as protocol, and 80 as destination port whereas outgoing packets have the reverse form. We can simulate requests to the web server by introducing connection entries to the state tables where the destination resp. source addresses and ports are chosen randomly. Therefore, we use the *request rate* (in Hertz) as scalability metrics.

In our benchmark, an experiment runs for 10 seconds and is repeated ten times. We conduct two kinds of experiments: one with active traffic emitters and one without. The latter allow to further determine scalability limitations since only the very basic insertion of rules into tables is measured without further traffic propagation in NetPlumber. Per experiment, we discretely adapt the request rate:

$$i \cdot \frac{Req.}{s}$$

where i is the number of requests and

$$i \in \{1, 10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}.$$

Also, we let the requests arrive with an equal distribution. In particular, we pregenerate all requests and, then, send them periodically in a request-by-request manner until the experiment's runtime is over. E.g., for a request rate of 10 Hertz, we generate $10 \cdot 10 = 100$ requests and send 10 requests per second, i.e., we wait for 100 milliseconds before sending the next request.

FaVe decouples API requests, e.g., the instrumentation of state table entries, from the actual instrumentation and analysis in NetPlumber, we measure the overtime of the whole experiment. Particularly, after sending all requests, we initiate FaVe's shutdown and measure the time between the last request and the actual shutdown. Since the shutdown takes constant time for a given network, i.e., it is independent from the amount of rules in the model's tables, we can determine if the request rate exceeds FaVe's processing rate when the shutdown time increases.

Figure 7.8 shows the results of our experiments. In the case of inactive traffic emitters, the request rate exceeds FaVe's processing rate between 400 and 500 Req/s whereas for active traffic emitters this already occurs between 10 and 50 Req/s. The results clearly show the limitations of state snapshots as a modeling technique for stateful firewalls. Modern web servers can serve tens of thousands of concurrent connection setups, e.g., cf. to [120], per second and, often, several parallel server instances can be found in real-world installations. Therefore, we need to address state handling with a more scalable modeling approach.

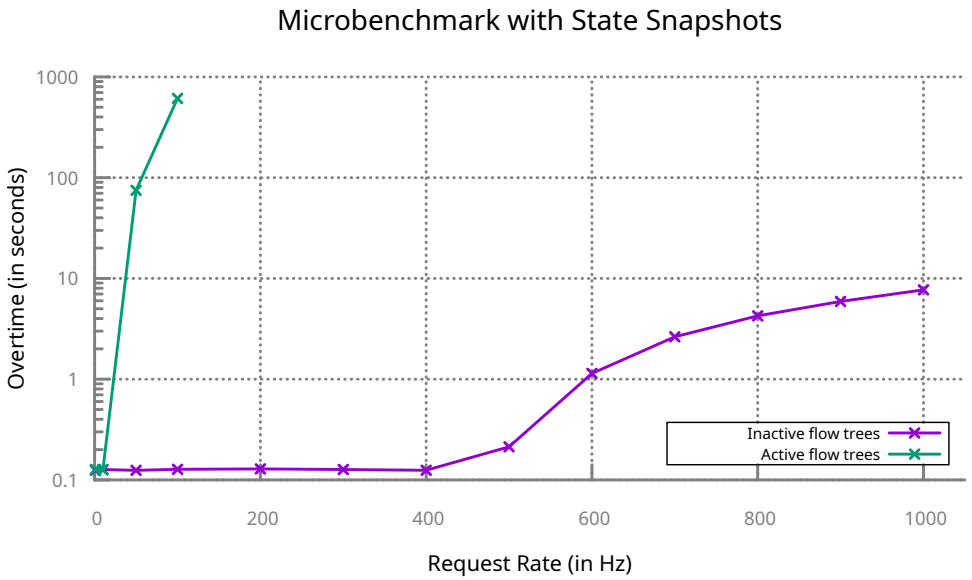


Fig. 7.8.: Measurement results to determine acceptable request rates for state snapshots.

7.1.7 Stateful Packet Filters with State Deduction

As seen in the previous Section 7.1.6, our naïve approach to model stateful behaviour with state snapshots does not scale well enough in order to support meaningful networks. Also, as described in Chapter 3, the most verification tools from literature are limited to the modeling of stateless behaviour. Particularly, the fast data plane approaches like NetPlumber or AP-Verifier cannot verify stateful policies due to their limited expressiveness.

Nevertheless, it is prevalent that modeling stateful firewalls in a scalable manner is crucial in order to support real-world setups. In FaVe, we overcome these shortcomings by performing a static analysis of a given firewall rule set, deduce all possible states, and concisely encode these in our model. Our basic idea to model stateful behaviour is the derivation of a virtual rule set – called the *state shell* – which only comprises 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 the overall filter semantics. By using only stateless rules, this approach allows FaVe to re-use fast data plane engines for verification while also covering stateful behaviour as well. We call this technique *State Shell Interweaving*.

Before we provide details on the formalization of stateful packet filter behaviour, we start with a description of the internals of Linux `iptables/netfilter` [107][2]. Providing our formalization and prototype implementation for `iptables` bears two benefits. First, it is one of the most common packet filter implementations found in practice. Second, transcompilation to other firewall configurations, e.g., for OpenBSD's `pf` or FreeBSD's `ipfw`, to an `iptables` configuration is possible as shown by Bodei et altera in [15]. Hence, our formalization for `iptables` provides a *generic* packet filter model suitable for a large variety of real-world scenarios.

7.1.7.1. Excursion: 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 chains, 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.

```
(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).

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

Discussion: Statelessness in `iptables` Turning off state tracking for individual services is possible by introducing a `-j CT --notrack` rule in `netfilter`'s `raw` table. For example, the rule set:

```
(0) ip6tables -t raw -p tcp --dport 22 -A FORWARD -j CT --notrack
(1) ip6tables -t filter -P FORWARD DROP
(2) ip6tables -t filter -A FORWARD -m conntrack --cstate ESTABLISHED -j ACCEPT
(3) ip6tables -t filter -A FORWARD -p tcp --dport 22 -j ACCEPT
```

does not track state for SSH packets. An incoming SSH packet is marked for stateless operations by rule (0) and, hence, not marked with any state by the conntrack table. The packet is then accepted by rule (3). SSH packets in the opposite direction, i.e., where the source port is 22, are not marked by rule (0) but since there is no established connection in the `conntrack` table, these packets are not matched by rule (2) and, hence, dropped by the default rule (1).

Though, stateless packet filtering with `iptables` is not very common. For instance, we found only 1 rule set out of 39 in the real-world collection from [36] where this rule applied statelessness to traffic on the completely internal `lo` interface.[3]

Modeling State with Virtual Rules `netfilter`'s concept to tag packets with their connection's 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 but naïve 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 (a) with reversed direction for rule (1) in front, i.e.:

```
(0) ip6tables -P FORWARD DROP
(a) ip6tables -A FORWARD -p tcp --sport 22  -j ACCEPT
(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
(b) ip6tables -A FORWARD -s 2001:db8::1 -p tcp --sport 80 -j ACCEPT
(4) ip6tables -A FORWARD -d 2001:db8::1 -p tcp --dport 80 -j ACCEPT
```

[3] On the other hand, state tracking is implemented very efficiently and disabling stateful packet filtering most likely does not improve performance in practice.

this virtual rule would accept SSH packets originating from `2001:db8::2` whereas in reality these packets would have been dropped by rule (2) before reaching the state checking rule (3). Hence, a more sophisticated method to model stateful behaviour of `iptables` is needed and it must take all inter-rule dependencies into account.

7.1.7.2. Modeling State using State Shell Interweaving

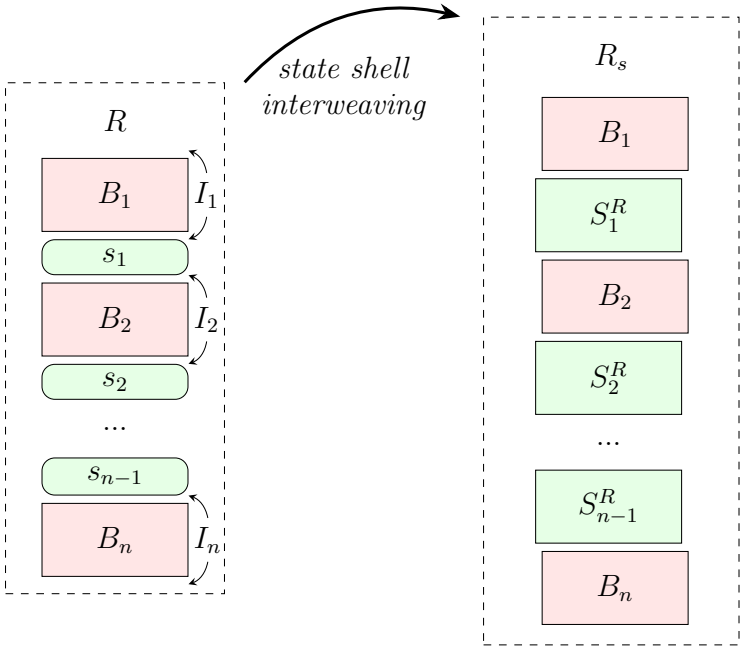


Fig. 7.9.: Principle of interweaving the state shell with the original rule set.

As seen in Section 7.1.7.1, `iptables` permits rule set configurations with complex state introduction and checking dependencies. The original rule set R, as depicted in Figure 7.9, 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 [36].

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

1. First, 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 portion of the Header Space as before.

In the following, we apply this procedure to our example `iptables` rule set from Section 4.1.2 (for all details refer to Appendix A.10), i.e.:

```
(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::110/124 -p tcp --dport 80 \
      -j ACCEPT
(5) ip6tables -A FORWARD -s 2001:db8::200/120 -d 2001:db8::100/120 \
      -p tcp --dport 22 -j ACCEPT
```

Particularly, we need to model this rule set in DHSA terms using the Algorithm 5.1 from Section 5.2:

$$
\begin{aligned}
R_{forward} = \{ & \\
& r_1 : \{ \\
& \quad (\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& \quad (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& \quad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& \quad (\texttt{state}, \mathcal{V}_{\texttt{state}}) \\
& \} \rightarrow \mathit{drop}, \\
& r_2 : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& \quad (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& \quad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& \quad (\texttt{state}, \{\texttt{ESTABLISHED}\}) \\
& \} \rightarrow \mathit{accept}, \\
& r_3 : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}),
\end{aligned}
$$

$$
\begin{aligned}
&(\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&(\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&(\texttt{state}, \mathcal{V}_{\texttt{state}})\\
\} &\rightarrow \mathit{accept},\\
r_4 : \{ &\\
&(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&(\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\}),\\
&(\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}),\\
&(\texttt{state}, \mathcal{V}_{\texttt{state}})\\
\} &\rightarrow \mathit{accept},\\
r_5 : \{ &\\
&(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&(\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \{\texttt{2001:db8::100/120}\}),\\
&(\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}),\\
&(\texttt{state}, \mathcal{V}_{\texttt{state}})\\
\} &\rightarrow \mathit{accept},\\
r_6 : \{ &\\
&(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&(\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&(\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&(\texttt{state}, \mathcal{V}_{\texttt{state}})\\
\} &\rightarrow \mathit{drop}\\
\}&
\end{aligned}
$$

where the Header Space is:

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

Then, we introduce a virtual header field to the Header Space that indicates if packets flow in backward direction: $h_9 := \texttt{back}$ with $\mathcal{V}_{\texttt{back}} := \{0, 1\}$ where 0 encodes the forth and 1 represents the backward direction.

Afterwards but 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:

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

Concerning our example, rule (2) is the only state checking rule and, hence:

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

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:

$$\rho : \mathcal{H}^D \times \mathcal{V}_{\mathcal{H}} \to \mathcal{H}^D \times \mathcal{V}_{\mathcal{H}}$$

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

For instance, $\rho((\texttt{dport}, \{22\}))$ yields $(\texttt{sport}, \{22\})$ whereas $\rho((\texttt{proto}, \{6\}))$ results in $(\texttt{proto}, \{6\})$.

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

$$m_{back} := \{..., (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\})\}$$

which has only the backwards flag set to 1. Therefore, S^R is defined as:

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

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.

For instance, concerning our example, rule (4) results in the following r_4 in the general reverse state shell:

$$
\begin{aligned}
r_4 : \{ & \\
& (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}) \\
& (\texttt{sip}, \{\texttt{2001:db8::110/124}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}) \\
& (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\}) \\
\} \quad & \rightarrow \mathit{accept}
\end{aligned}
$$

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 7.9). 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 $s_l : m_l \rightarrow a_l$ 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 7.9. Formally:

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

Each state shell entry is specialized concerning the state checking rules' matches 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. So, rule actions may be in conflict and we need to chose the stricter one. For this purpose, we define a total order for rule actions, i.e., $accept < drop$, and a strictness helper function to determine the right action for each rule in the conditional reverse state shell, i.e.:

$$stricter : A \times A \to A$$
$$stricter(a_1, a_2) := \begin{cases} accept, \text{ if } a_1 = a_2 = accept \\ drop, \text{ else.} \end{cases}$$

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[4]:

$$\begin{aligned} S_1^R = \{ & \\ & r_{(2\cdot1+1)\cdot6+1=19} : m_1^{S^R} \cap m_2 \to drop, \\ & r_{(2\cdot1+1)\cdot6+3=21} : m_3^{S^R} \cap m_2 \to accept, \\ & r_{(2\cdot1+1)\cdot6+4=22} : m_4^{S^R} \cap m_2 \to accept, \\ & r_{(2\cdot1+1)\cdot6+5=23} : m_5^{S^R} \cap m_2 \to accept, \\ & r_{(2\cdot1+1)\cdot6+6=24} : m_6^{S^R} \cap m_2 \to drop \\ \}. & \end{aligned}$$

3. State Shell Interweaving Before we may weave the conditional reverse state shells into the original rule set, we need to re-index the rules within the blocks B_i to make space for the new reverse rules. In addition to re-indexing, the rules that apply explicitly to unknown connections are set to act in forth direction only. This is done by setting the *backwards* flag to 0 explicitly. For this purpose, we introduce m_{forth} (analogously to m_{back}) which has only the backwards flag set to 0, i.e, $m_{forth} := \{..., (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{0\})\}$.

[4]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. For details, refer to Appendix A.10.

Formally, we re-index the blocks B_i for each $(i, k, l) \in I$ as follows:

$$
\begin{aligned}
B_i := \{ & \\
& r_{2i \cdot |R| + j} : m \to a_j \mid \; r_j : m_j \to a_j \in R, \\
& \quad k < j < l, \\
& \quad m := \begin{cases} m_j \cap m_{forth}, \text{ if } (\texttt{state}, \{\texttt{NEW}\}) \in m_j \\ m_j, \text{ else} \end{cases} \\
\} &
\end{aligned}
$$

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

$$
\begin{aligned}
B_1 = \{ & \\
& r_{2 \cdot 1 \cdot 6 + 1 = 13} : m_1 \to \mathit{drop} \\
\} & \\
B_2 = \{ & \\
& r_{2 \cdot 2 \cdot 6 + 3 = 27} : m_3 \to \mathit{accept}, \\
& r_{2 \cdot 2 \cdot 6 + 4 = 28} : m_4 \to \mathit{accept}, \\
& r_{2 \cdot 2 \cdot 6 + 5 = 29} : m_5 \to \mathit{accept}, \\
& r_{2 \cdot 2 \cdot 6 + 6 = 30} : m_6 \to \mathit{drop} \\
\} &
\end{aligned}
$$

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

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

As we already adapted all rules' indices, we simply need to collect all rules by unioning 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[5]:

$$
\begin{aligned}
R_S = B_1 \cup S_1^R \cup B_2 = \{ & \\
& r_{13} : m_1 \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow drop, \\
& r_{19} : (m_1^{S^R} \cap m_2) \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow drop, \\
& r_{21} : (m_3^{S^R} \cap m_3) \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow accept, \\
& r_{22} : (m_4^{S^R} \cap m_4) \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow accept, \\
& r_{23} : (m_5^{S^R} \cap m_5) \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow accept, \\
& r_{24} : (m_6^{S^R} \cap m_6) \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow drop, \\
& r_{27} : m_3 \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow accept, \\
& r_{28} : m_4 \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow accept, \\
& r_{29} : m_5 \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow accept, \\
& r_{30} : m_6 \backslash \{(\texttt{state}, \mathcal{V}_{\texttt{state}})\} \rightarrow drop \\
\} &
\end{aligned}
$$

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

Limitations

Our approach to handle state has some limititations concerning the complexity of the initial rule set. Currently, there is no direct support for custom chains which are part of `iptables` and frequently used by administrators. In [37], a semantics preserving algorithm is presented that transforms a rule set with custom chains into a simple enough rule set. Therefore, rule sets would need to be preprocessed before being analyzed by FaVe.

Further, 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` [108] 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.

[5] For more details on the particular header fields refer to Appendix A.10.

7.2 Evaluation

Benchmark	UP	Stanford	Internet2	TUM/TUMv6
Network	1 firewall, 23 switches, 130 hosts, 1 dummy	16 routers	9 routers	1 firewall
Rules	3,396 (of which 1,035 are in the main firewall)	8,792	77,841	3,795
IP Version	**IPv6**	IPv4	IPv4	IPv4/IPv6
Roles	71	16	9	-
Policy Checks	11,902 (of which 4,953 are 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`

Tab. 7.2.: Overview of the benchmarks.

For the evaluation, we ran five benchmarks that show the approach's real-world usefulness and scalability, especially in comparison with state-of-the-art tools, and its IPv6 performance.[6] Table 7.2 provides an overview of the benchmarks' characteristics. We ran all experiments on the machine specified in Table 1.1 from Section 1.4.

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. It was first presented and used in the evaluation of ad6 [94] and further extended in [96] to its current size. Second, we show the scalability concerning large networks by running the well known *Internet2* and *Stanford* workloads [76][7]. Third, we compare FaVe against the available open source firewall

[6] The benchmarks including policies and configurations have been published along with our prototype here: `https://github.com/cllorenz/fave-project`.

[7] For the Stanford and I2 workloads, the original paper speaks of more than 757,000 forwarding and 1,500 ACL rules [79] resp. 126,000 forwarding rules. We reproduced their resuls which included a preprocessing step that compressed these rules down to 8,792 resp. 77,841 rules. For the Stanford workload, compressing the rules needed about 35 seconds while the benchmark ran in ca. 1 second. Without compression the Stanford benchmark ran in 28,280 seconds. The scripted reproduction can be found here: `https://github.com/cllorenz/hassel-reproduction`

verification tools `fffuu` [37] and `SymNet` [39]. For this comparison, we use the TUM workload from [36] 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 IPv4 rules and is the largest out of a collection of 41 real-world rule sets [36]. Fourth, we compare FaVe's IPv6 performance against its IPv4 performance. For this purpose, we use the TUMv6 workload – an IPv6 version of the TUM workload. The TUMv6 is derived from the TUM workload by embedding the IPv4 prefixes into the IPv6 address space that has been reserved for the IPv4 Internet [10]. Particularly, we prefixed each IPv4 prefix with `64:ff9b::/96`, e.g., `1.2.3.4/31` becomes `64:ff9b::102:0304/127`.

Benchmark Methodology The methodology of all benchmarks follow the same principles. For the experiments, there are three measurement phases: *Initialization*, *Reachability*, and *Compliance*:

Initialization We measure the time necessary to initially instrument the tool under evaluation with the model, i.e., from modeling the first element until the tool signals completion of all modeling tasks.
For instance, FaVe's initialization phase covers all steps conducted to build the model. During this phase, FaVe aggregates and transforms models, which includes the interweaving of the state shells, increment calculation, and the instrumentation of NetPlumber as its verification engine.

Reachability We measure the time necessary to calculate reachabilities in the tool under evaluation, i.e., from linking the first traffic emitter to the model until the tool signals completion of the last traffic propagation.
For instance, FaVe performs all steps to calculate reachabilities, i.e., for each role a source node is connected to the network model which causes FaVe's verification engine NetPlumber to calculate the propagation of packet flows.

Compliance We measure the time necessary to calculate conformance of the reachabilities with the specified compliance rules, i.e., from starting the first compliance analysis until the tool signals the completion of the last one.
For instance, FaVe reads the FPL rules and instruments NetPlumber as its verification engine to check the reachabilities for compliance with these rules.

Note that not all benchmarks use all phases. For instance, since there are no agreed compliance rules for the Internet2 and Stanford workloads, the compliance phase is not performed for these benchmarks. We indicate such circumstances when they occur.

We repeat each experiment 10 times.

Transforming FPL rules to Stateful Compliance Checks in FaVe In the *Compliance* phase, compliance analysis is performed by checking each role's reachibility tree for conformance with the FPL policies. For this purpose, we translate all FPL rules plus all pairs of roles lacking an explicit rule as follows: A rule $A\ op\ B$ translates to a set of reachability constraints that need to hold for all paths between A and B:

`--->`	At least one path from A to B must exist but no path from B to A, e.g., the FPL rule `WebServer ---> LogStash` results in two checks: one checking reachability from `WebServer` to `LogStash` and one checking non-reachability from `LogStash` to `WebServer`.
`<-->`	At least one path from A to B must exist and also at least one path from B to A 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. For instance, the rule `Internet <->> WebServer` results in three checks: one for the unrestricted access of the `WebServer` from the `Internet`, one for answers from the `WebServer` to the `Internet` marked with `back` $= 1$, and one for forbidden initializations from the `WebServer` to the `Internet` marked with `back` $= 0$.
`default`	There must not exist any path from A to B due to FPL's default which is to deny any other packet flows.

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

7.2.1 Complex Policies: The UP Campus Network

The *UP* benchmark is a synthetic representation of an university campus network with a large perimeter firewall and several network segments containing different hosts and services.[8] The firewall rule sets follow best practices (cf. [66]) and the

[8]The policy specification of the UP benchmark is given in Appendix A.11.

network topology resembles real-world setups. This benchmark shows FaVe's ability to verify compliance for complex networks.

As shown in Table 7.2, the UP benchmark consists of a central perimeter firewall and 23 switched network segments. These include a DMZ with 8 hosts, a WIFI domain, and 21 generic network segments with 6 hosts each. Each host comprises a small host 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. These rule sets do not include the state shells yet as these are calculated within FaVe at runtime and, therefore, they are not part of the configuration provided by the administrator. The firewall rule sets consist of a large variety of header fields as they include several rules that realize IPv6 specific requirements, e.g., filtering ICMPv6 traffic [31]. The policy checks sum up to 11,902 including 4,953 state checks.

In the experiments, we measure the *Initialization* as well as the *Reachability* phases of the verification process with FaVe and NetPlumber respectively. Also, we measure the *Compliance* phase with FaVe but not with NetPlumber[9].

To determine the runtime overhead of FaVe compared to NetPlumber we implemented a dumping function. Since NetPlumber is not capable to directly 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 format which is suitable to be processed by NetPlumber. For the measurements with NetPlumber, we load the aggregated model directly into NetPlumber without involving FaVe[10]. 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.

As shown in Figure 7.10, the overall runtime (*FaVe1*) 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 is suitable for 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 7.10 shows that the runtime for

[9] The measurement results in this chapter have been conducted with a previous version of NetPlumber which did not offer built-in compliance checks. Particularly, the used version is the same as the one used for [96] and, hence, the results stated here are the same as published. Our subsequent improvements to NetPlumber include said built-in compliance checks which run much faster than the external analysis that we conducted in the paper. Hence, the stated numbers are worse than the expected numbers with the current NetPlumber version.

[10] Since the used version of NetPlumber did not support built-in compliance checks, we omit this phase for the comparison experiments.

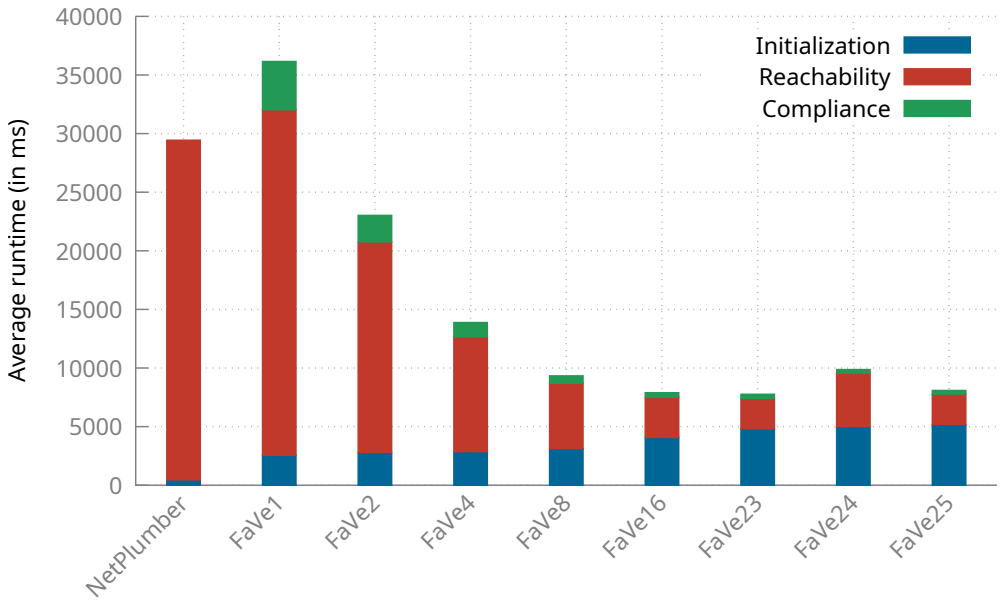


Fig. 7.10.: Average runtimes after ten repetitions of the UP benchmark (in ms). The indices for FaVe indicate the number of backend instances as well as the number of threads used to check for compliance. For each experiment the coefficient of variation of the total runtime is below 3 %.

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. Our measurements with multiple backend instances reveal 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.

The last measurement with 24 instances shows a performance degregation with 9.87 s versus 7.89 s for 16 instances. Additional measurements with 23 and 25 instances (7.76 s resp. 8.09 s) revealed that this specific configuration triggered a collision where one instance was assigned a large amount of long running tasks. This behaviour can be avoided by changing the current distribution via round-robin with fixed buckets to a dynamic approach. This is beyond this work's scope. Performance improvements are further limited by the growing initialization time which nearly

doubles throughout the measurements. Nevertheless, all in all, parallelization allows large performance gains compared to NetPlumber alone and further supports periodic reverification.

7.2.2 Scaling to large Networks

Next, we show FaVe's scalability for large networks by measuring the well known real-world *Stanford* and *Internet2* workloads [76]. As listed in Table 7.2, these IPv4 routed networks each 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 [79] 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[11].

We augmented the benchmarks by specifying a synthetic 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, e.g., ASes, connected to that router.

Tool	Init	Reach	Compl.	Total
Stanford				
FaVe	2.08	0.11	0.08	2.28
NetPlumber	0.50	0.11	-	0.61
Vanilla-NetPlumber	0.90	0.12	-	1.01
Internet2				
FaVe	47.85	65.00	0.55	113.39
NetPlumber	31.86	56.44	-	88,30
Vanilla-NetPlumber	77.95	71.92	-	147.87

Tab. 7.3.: Mean runtimes after ten repetitions for the Stanford and Internet2 Benchmarks (in seconds). The coeffients of variation are below 3.3 % resp. 1 %.

Table 7.3 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.

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

For the even larger Internet2 benchmark, FaVe's runtime is less than 2 minutes. Since there are only few policy 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. Since, the runtimes of FaVe and NetPlumber range in the same order of magnitude, 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.

Further, we included measurements using the original NetPlumber, i.e., without our improvements to NetPlumber (cf. Section 6.1.7), for comparison. It shows that, in the case of the Internet2 workload, FaVe runs about 23 % faster than the Vanilla-NetPlumber (113.39 s versus 147.87 s). In direct comparison, our improved NetPlumber is about 40 % faster for the Stanford workload (0.61 s versus 1.01 s). We investigated on this issue and profiling revealed that a change in the book-keeping of rules in NetPlumber's rule tables had a major impact on the runtime of rule insertions in tables with many rules – a very common theme in both workloads. Particularly, we replaced a linear list with a hash map that performed better regarding the large amounts of rules.

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.

7.2.3 Comparison with State-of-the-Art

Now, we compare FaVe against the public available tools `fffuu` [37] and `SymNet` [39], since both tools are able to verify stateful packet filters (cf. Chapter 3). For the evaluation of `fffuu`, Diekmann et al. used the so-called TUM benchmark. This is a real-world firewall rule set from [36] 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.

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

Tool	Mean	Median	StdDev.
FaVe	2.42	2.42	0.05
NetPlumber	1.35	1.35	0.01
fffuu	100.48	100.49	0.08
SymNet	oom	oom	oom

Tab. 7.4.: TUM Benchmark results after ten repetitions (in seconds).

trees. FaVe is instrumented with a packet filter model and the same rule set. To establish a comparable measurement, a traffic generator is attached directly to the forward filter table and a probe as target directly behind. This way we measure the filtering behaviour of the forward filtering table in isolation. The measurements for FaVe include the initialization phase and the calculation of a reachability tree. As shown in Table 7.4, 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[12]. `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.

Third, analogously to the benchmarks in Section 7.2.2, we conduct a measurement with NetPlumber by directly loading a dump provided by FaVe. With 9.73 s, the direct usage of NetPlumber is faster but lacks the comfortable modeling offered by FaVe and its ability to check compliance with high-level policies specified with FPL.

7.2.4 IPv6 Performance

Finally, we measure FaVe's performance when verifying IPv6 compared to IPv4. Figure 7.11 shows the runtimes of the IPv4 and IPv6 versions of the TUM workload for FaVe and NetPlumber. We can see that FaVe's total runtime for IPv6 lies in the same order of magnitude as for IPv4. Particularly, verifying IPv6 is 3.3 times slower than IPv4 (8.15 s vs. 2.42 s) The same observation holds for NetPlumber which is 5.3 times slower (7.21 s vs. 1.35 s). While the initialization phase has very similar runtimes per tool, the reachability phase differs more significantly. Here, FaVe's

[12]Our modifications can be found here: `https://github.com/cllorenz/iptables-sefl`

runtime for IPv4 is ca. 68.5 times faster than for IPv6 (0.09 s vs. 5.88 s). This is similar for NetPlumber with a factor of ca. 66.6 (0.09 s vs. 5.87 s).

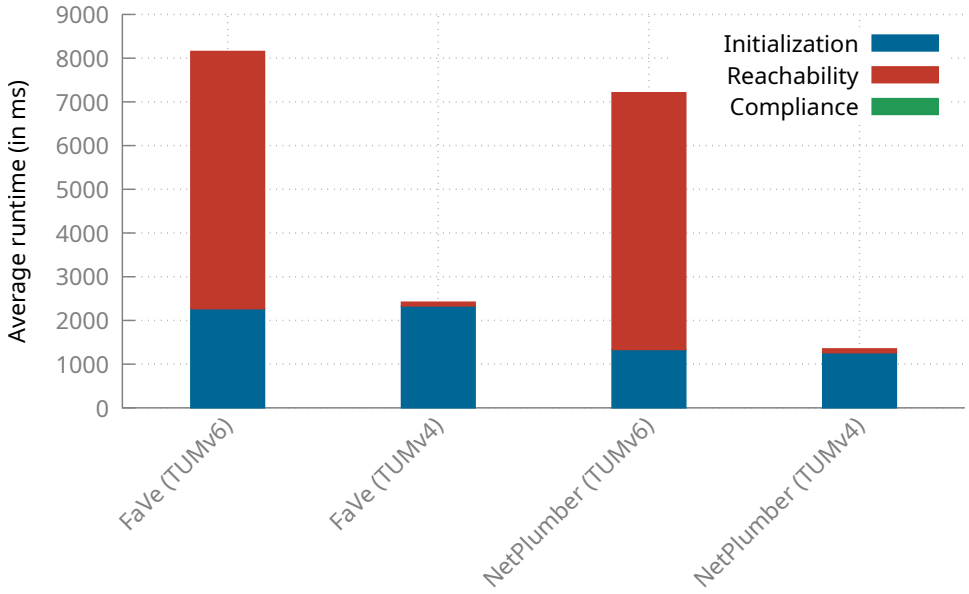


Fig. 7.11.: Average runtimes after ten repetitions of the TUM and TUMv6 benchmarks (in ms). For each experiment the coefficient of variation of the total runtime is below 2.2 %.

We can conclude that all in all FaVe performs well for IPv6. The overhead of a factor of 3.3 over IPv4 is meaningful but still in the same order of magnitude. Also, the total runtime of about 8 seconds is well acceptable. Further, the overhead stems from the underlying NetPlumber engine and, hence, it is not introduced by FaVe's more high-level modeling.

7.3 Firewall Configuration Generation

After initially determining the state of network security compliance in the UNDERSTAND phase and subsequent iterative transformation in the ADJUST phase, once the desired state of network security compliance has been reached, it needs to be maintained in the CONTROL phase. For this purpose, we enable administrators to generate security configuration – particularly firewall rule sets – from the compliance specification, i.e., from FPL policies. These rule sets are compliant by design and can be deployed automatically. The following work has been explored in the Master's thesis of Benjamin Plewka under my supervision [115].

For the rule set generation, we follow some design principles and good practices, e.g., [66] for IPv6 handling. Particularly, we apply the following design decisions:

General Structure Rule sets can be organized as flat lists of rules or as DAGs with conditional jumps between lists of rules. For instance, the latter semantics is supported by hierarchical matchers like Linux' IPTables, FreeBSD's ipfw, or, to some degree, OpenBSD's pf. The former semantics, in turn, can be handled by simple matchers, e.g., OpenFlow switches. Despite the fact, that FPL is organized hierarchically and our primary packet filter IPTables offers hierarchical matching, we opted to implement a nearly flat list of rules. Only some IPv6 specific rules are organized in custom chains and, optionally, these can be flattened as well.

Anti Spoofing Known internal address ranges can be blocked on external interfaces. We generate appropriate rules for this purpose.

IPv6 Readyness IPv6 requires certain ICMPv6 messages for core functionality like neighbor discovery or router solicitation to work properly. We add a set of rules required for proper IPv6 operations.

IPv6 Hardening Security best practices for IPv6 [66] and the most recent standard [33] require to block resp. allow only certain headers and fields in IPv6 packets. We add a set of sanity rules that enforce these rules.

Early State Checking Matching packets in linear lists is comparably expensive and, therefore, an early check if a packet belongs to a known connection helps to improve overall performance. In the case of IPTables, `conntrack` marks a packet's state before the packet enters the rule tables and we place the state checking rule as early as possible but without security compromises.

Hence, the resulting rule set abides good practices for rule set designs and has the following structure:

1. Rules that mark packets for stateless processing.
2. Default rules, i.e, DROP.
3. Rules that prevent spoofing of internal address ranges.
4. Rules that enable proper IPv6 operations, i.e., ICMPv6.
5. Rules that harden IPv6 operations, e.g., dropping of forbidden headers.
6. Rules for state checking.

7. Rules for access grants.

Input:

- Propagated FPL `Inventory`
- FPL `Policy`

Output: An IPTables rule set for a logically centralized firewall.

Procedure:

1. Initialize empty lists:
   - `Suppress` ← []
   - `Main` ← []
2. Add default rule to `Main`:
   `Main` ← `Main` + [`DEFAULT_RULE`]
3. Add anti spoofing rules:
   For each `Role` in `Inventory` do:
   1) Fetch `Source` role from `Role`.
   2) Fetch address range `Range` from `Source` (`ipv4` resp. `ipv6`).
   3) Apply template for anti spoofing rules and append to `Main`:
      `Main` ← `Main` + [$template\_anti\_spoofing$(`Range`)]
4. Add static rules for proper IPv6 operations:
   `Main` ← `Main` + `STATIC_ICMPV6_RULES`
5. Add custom chain and static jump rule for the IPv6 hardening:
   `Main` ← `Main` + `IPV6_HARDENING_CHAIN`
   `Main` ← `Main` + [`STATIC_HARDENING_JUMP_RULE`]
6. Add state checking rule:
   `Main` ← `Main` + [`STATE_CHECKING_RULE`]
7. Add access rules:
   For each `Rule` in `Policy` do:
   1) Fetch `Source` role, `Destination` role, and `Conditions` from the `Rule`.
   2) If the `Conditions` indicate a stateless rule, then:
      1) Apply rule template to suppress packets for stateless processing:
         `Suppress` ← `Suppress` + [$template\_suppress$(`Source`, `Dest`)]
      2) Apply rule template to statelessly handle packets:
         `Main` ← `Main` + [$template\_stateless$(`Source`, `Dest`)]
   3) Otherwise, add a regular state creating access rule:
      `Main` ← `Main` + [$template\_stateful$(`Source`, `Dest`)]
8. Concatenate and return rule lists: `Suppress` + `Main`

Algorithm 7.1.: Generation algorithm of IPTables rule sets from FPL security specifications.

Algorithm 7.1 shows our procedure that generates an IPTables rule set from a FPL security specification. In the following, we detail the IPTables rules and rule

templates used. Further, Appendix A.13 shows the resulting IPTables rule set for our example FPL inventory and policy.

Static Default Rule The default rule is:

```
ip6tables -P FORWARD DROP
```

Rule Template for Anti-Spoofing The $template\_anti\_spoofing()$ is applied to some `Role` and has the following form:

```
ip6tables -A FORWARD \
    -i <external-interface> \
    -s <source-ips> \
    -j DROP
```

The `<source-ips>` are IP address ranges fetched from the `Role` whereas interfaces of the firewall that face external networks like the Internet are refered to by `<external-interface>`. Since the latter is specific to the firewall where the rule set should be deployed to, it is passed as a global parameter to the generation algorithm.

For instance, the DMZ role from our example yields the following anti-spoofing rule:

```
ip6tables -A FORWARD -i eth1 -s 2001:db8::100/120 -j DROP
```

Static Rules for IPv6 Readyness The `STATIC_ICMPV6_RULES` are the following:

```
ip6tables -A FORWARD \
    -p icmpv6 --icmpv6-type destination-unreachable \
    -j ACCEPT
ip6tables -A FORWARD \
    -p icmpv6 --icmpv6-type packet-too-big \
    -j ACCEPT
ip6tables -A FORWARD \
    -p icmpv6 --icmpv6-type echo-request \
    -m limit --limit 900/min \
    -j ACCEPT
ip6tables -A FORWARD \
    -p icmpv6 --icmpv6-type echo-reply \
    -m limit --limit 900/min \
    -j ACCEPT
ip6tables -A FORWARD \
    -p icmpv6 --icmpv6-type ttl-zero-during-transit \
```

```
    -j ACCEPT
ip6tables -A FORWARD \
    -p icmpv6 --icmpv6-type unknown-header-type \
    -j ACCEPT
ip6tables -A FORWARD \
    -p icmpv6 --icmpv6-type unknown-option \
    -j ACCEPT
```

They pass ICMPv6 packets that enable proper operations of IPv6 protocols like neighbor discovery or router solicitation as specified in the IPv6 standard [33]. Further, a reasonable rate limit for pings is introduced.

Custom Chain and Static Rule for IPv6 Hardening These rules are introduced as a custom chain `IPV6_HARDENING_CHAIN` which contains the following rules:

```
ip6tables -N routinghdr
ip6tables -A routinghdr -m rt --rt-type 0 ! --rt-segsleft 0 -j DROP
ip6tables -A routinghdr -m rt --rt-type 2 ! --rt-segsleft 1 -j DROP
ip6tables -A routinghdr -m rt --rt-type 0 --rt-segsleft 0 -j RETURN
ip6tables -A routinghdr -m rt --rt-type 2 --rt-segsleft 1 -j RETURN
ip6tables -A routinghdr -m rt ! --rt-segsleft 0 --j DROP
```

First, the custom chain `routinghdr` is established. Then, rules are added that block packets with a routing header of type 0 with further segments as well as packets with a routing header of type 2 and not exactly one segment. Further, packets with a routing header of type 0 and without further segments resp. packets with a routing header of type 2 and with one segment are considered for subsequent processing by the packet filter. Any other packet with a routing header and more segments is denied. Finally, the custom chain is accessed via the `STATIC_HARDENING_JUMP_RULE`:

```
ip6tables -A FORWARD -m ipv6header --header ipv6-route -j routinghdr
```

Alternatively, to avoid using a custom chain, the following set of rules works equivalently:

```
(1) ip6tables -A FORWARD \
        -m ipv6header --header ipv6-route \
        -m rt --rt-type 0 ! --rt-segsleft 0 \
        -j DROP
(2) ip6tables -A FORWARD \
        -m ipv6header --header ipv6-route \
        -m rt --rt-type 2 ! --rt-segsleft 1 \
        -j DROP
(3) ip6tables -A FORWARD \
        -m ipv6header --header ipv6-route \
```

```
        -m rt --rt-type 0 --rt-segsleft 0 \
        -j MARK --xmark 0x1/0x1
(4) ip6tables -A FORWARD \
        -m ipv6header --header ipv6-route \
        -m rt --rt-type 2 --rt-segsleft 1 \
        -j MARK --xmark 0x1/0x1
(5) ip6tables -A FORWARD \
        -m ipv6header --header ipv6-route \
        -m rt ! --rt-segsleft 0 \
        -m mark ! --xmark 0x1/0x1 \
        --j DROP
```

The action `-j MARK -xmark 0x1/0x1` sets a bit in a matching packet's meta data which can be matched in subsequent rules. In our case, packets that are marked this way, i.e., by the rules (3) and (4), are spared from being dropped by rule (5). We opted to use a custom chain by default since custom chains are a common practice and they are more convenient than marking and matching packets in IPTables.

Static Rule for State Checking The following static rule provides regular state checking:

```
ip6tables -A FORWARD -m conntrack --ctstate ESTABLISHED -j ACCEPT
```

Templates for Stateless Rules By default, IPTables does not provide stateless handling since all packets are handled by `conntrack` first and marked with a state which can be matched. In order to enforce truly stateless filtering, packets must be prepocessed and marked for stateless handling. The $template\_suppress()$ can be applied to pairs of `Source` and `Dest`ination roles and has the following form:

```
ip6tables -t raw -A PREROUTING \
    <in-interface/vlan-matching> <source-matching> \
    <out-interface/vlan-matching> <dest-matching> \
    <proto-match> \
    <comment> \
    -j NOTRACK
```

The `<in-interface/vlan-matching>` and `<source-matching>` placeholders contain source matching fields, e.g., `-i`, `-s` or `--sport`, and are taken from the `Source` role. On the other hand, the placeholders `<out-interface/vlan-matching>` and `<dest-matching>` are translated to destination matching fields, e.g., `-o`, `-d`, or `--dport`. They are taken from the `Dest` role. If necessary, the `<proto-match>` placeholder, i.e., if the FPL rule specifies some service access, the protocol field `-p` is set

as well. Finally, the <comment> placeholder provides a short comment that states the access between the involved roles. The action NOTRACK marks the ctstate field in the packet's metadata as UNTRACKED which causes conntrack to pass on the packet without state inspection.

For instance, assume our example policy would contain a rule Office ---> DMZ. Applying the template would result in the following state suppression rule:

```
ip6tables -t raw -A PREROUTING \
    -s 2001:db8::200/120 \
    -d 2001:db8::100/120 \
    -m comment --comment "Office to DMZ" \
    -j NOTRACK
```

Packets that have been marked for stateless filtering are, then, matched in the main rule set by rules created by applying the $template\_stateless()$ to the same Source resp. Destination roles:

```
ip6tables -A FORWARD \
    <in-interface/vlan-matching> <source-matching> \
    <out-interface/vlan-matching> <dest-matching> \
    <proto-match> \
    -m conntrack --ctstate UNTRACKED \
    <comment> \
    -j ACCEPT
```

The conntrack state UNTRACKED which needs to be set by our preprocessing rule is used as match criterion in order to distinguish stateless from regular filtering. The other matching fields are the same as for the respective preprocessing rule.

Consequently, our example rule Office ---> DMZ would result in the following rule which is added to the main rule set:

```
ip6tables -A FORWARD \
    -s 2001:db8::200/120 \
    -d 2001:db8::100/120 \
    -m conntrack --ctstate UNTRACKED \
    -m comment --comment "Office to DMZ" \
    -j ACCEPT
```

Template for Regular Stateful Rules Finally, regular stateful rules are created by applying the $template\_stateful()$ to the Source and Destination roles in the same manner as stateless rules:

```
ip6tables -A FORWARD \
    <in-interface/vlan-matching> <source-matching> \
    <out-interface/vlan-matching> <dest-matching> \
    <proto-match> \
    -m conntrack --ctstate NEW \
    <comment> \
    -j ACCEPT
```

Matching the `conntrack` state `NEW` limits this rule to match only packets that belong to new connections. If a packet is accepted by these rules, `conntrack` updates these new connections as `ESTABLISHED` and subsequent packets that belong to these connections are accepted by the state checking rule which was introduced before. Also, matching `NEW` clearly distinguishes stateful from stateless rules which specificantly match `UNTRACKED` packets.

For instance, the rule `Office <->> WebServer.SSH` from our example results in the following stateful rule:

```
ip6tables -A FORWARD --protocol tcp --dport 22 \
    -s 2001:db8::200/120 -d 2001:db8::110/124 \
    -m conntrack --ctstate NEW \
    -m comment --comment "Office to WebServer" -j ACCEPT
```

7.4 Summary

In this chapter, we have shown that FaVe's automatic verification process of security policies stated in a high-level and semi-natural language is feasible and scalable.

Particularly, several device models have been introduced in order to allow administrators to accessibly model common and advanced network scenarios using real-world device configurations. FaVe ships with device models for switches, routers, and stateless as well as stateful packet filter firewalls. We have shown, that network models consisting of these devices can be verified automatically for conformance with policies specified in FPL.

For the fast verfication of common stateful packet filters like `iptables`, we introduced the *state shell interweaving* – a deductive 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.

Our extensive 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.

FaVe shows convincing performance to serve as compliance verification tool throughout all three phases of the proposed security management process from Section 1.2 – namely UNDERSTAND, ADJUST, and CONTROL. Particularly, initial verification in UNDERSTAND and continuous verfication in ADJUST and CONTROL are enabled through FaVe's low runtimes. This, answers our third research question positively.

In addition, we have shown that compliant security configuration – IPTables rule sets in our case – can be synthesized directly from FPL policies. In the CONTROL phase, this feature relieves administrators from manual adjustments which further increases the economic scalability of security management.

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

8 Use Case: Firewall Anomaly Detection

In this chapter, we support network administrators with automatic tooling which fosters the reduction of complexity of firewall configurations. Often, these complexities occur due to the configurations' growth over time and the administrator's struggle to inspect and maintain large rule sets manually [132, 144]. In the context of firewalls, *anomalies* are misconfigurations which affect the system's performance and usability. In the case of networks with multiple firewalls, anomalies like *cross-paths* or *cycles* are possible which even pose security threats [94]. In this work, we focus on single firewall anomalies.

For instance, the most important firewall anomaly is *shadowing*. A shadowed rule does not contribute to the filtering behaviour of a rule set as all packets that match the rule have already been handled by rules with higher priorities. Nethertheless, all packets which reach the rule will be checked against it but can never be matched. Therefore, removing shadowed rules would not change the filtering semantics but it would improve the filtering performance and, hence, this would increase packet throughput and would reduce overall energy consumption.

Besides performance, the detection of anomalies gains importance as a core metric in the systematic assessment of firewall user experience [144], since a simplified rule set increases the firewall's maintainability. Further, the ongoing transition towards the state-of-the-art networking with IPv6 introduces an additional challenge for anomaly detection. Control protocols like *neighbor discovery* or *router advertisement* and dynamic extension header chains lead to an increased complexity of firewall rule sets and the state space.

It is evident that automatic support for firewall management is necessary. Therefore, detecting anomalies in such rule sets is a long lasting research topic [132, 153, 48, 73, 62, 80, 11, 126, 94, 127, 148]. Nevertheless, none of these approaches has yet gained broader adoption. A major obstacle is the runtime of the validation tools. For instance, Cisco's Security Manager offers the rule-wise detection of shadowing and

redundancy anomalies [22]. However, best practices[1] recommend the analysis of only small rule sets.

Therefore, fast and scalable approaches are desirable in this context. As introduced in Section 1.3, our *Automatic Anomaly Detection* (AAD) workflow consists of three steps:

1. Modeling: First, the administrator provides the rule set, e.g., an IPTables configuration, which is modeled formally using FaVe.
2. Analysis: Then, the model is analyzed automatically for anomalies, e.g., shadowed or generalized rules.
3. Reporting: Finally, a report is generated that lists all rules that are affected by an anomaly. This information can be used to safely adjust the firewall rule set.

This chapter is structured as follows: First, in Section 8.1, we provide detailed descriptions of firewall anomalies. After reviewing the related work in Section 8.2, we formalize the anomalies using DHSA in Section 8.3 and give insights into our implementation in Section 8.4. Finally, in Section 8.5, we evaluate our prototype's performance in comparison with the state-of-the-art as well as for its ability to scale to very large rule sets.

8.1 Firewall Anomalies

Firewall anomalies occur if two or more filtering rules match the same packet [97]. Typically, they emerge if rule sets are large and have been changed over time repeatedly. This may lead to the situation that certain dependencies between rules yield a filtering behaviour that might not be intended by the administrator. In essence, the rule set is no longer maintainable efficiently.

Before we classifiy different anomalies, we give an examplary IPTables rule set for further illustrations:

```
(0) ip6tables -P FORWARD DROP
(1) ip6tables -A FORWARD -d 2001:db8::1/127 -j ACCEPT
(2) ip6tables -A FORWARD -d 2001:db8::0 -j ACCEPT
(3) ip6tables -A FORWARD -d 2001:db8::2/127 -j ACCEPT
(4) ip6tables -A FORWARD -d 2001:db8::0/126 -j ACCEPT
(5) ip6tables -A FORWARD ! -d 2001:db8::0/126 -j ACCEPT
(6) ip6tables -A FORWARD -d 2001:db8::102 -j ACCEPT
```

[1] https://community.cisco.com/t5/network-security/duplicate-firewall-rules-shadow-redundant/td-p/2973394

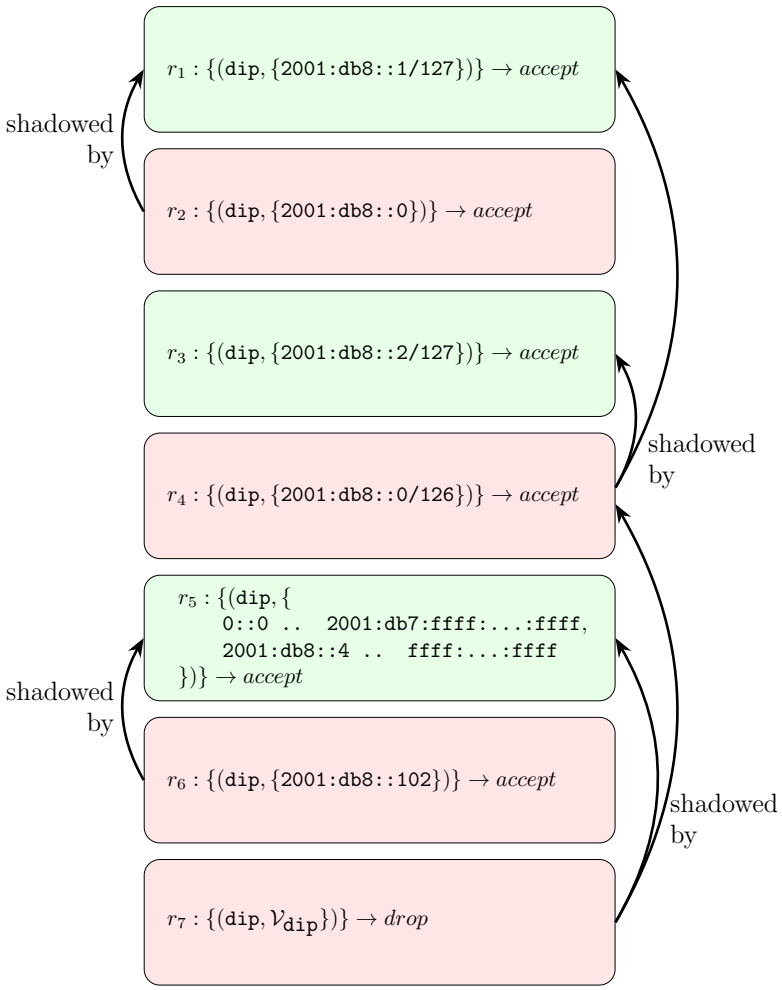


Fig. 8.1.: Shadowed rules in the DHSA model of the example rule set. The red rules are shadowed by the rules indicated by the arrows.

Subsequently, we explain and assess the most relevant types of anomalies.

Shadowing A rule is shadowed if all packets that are relevant for this rule, i.e., that the rule matches, are processed by one rule or a combination of rules with higher priorities. As shown in Figure 8.1, the rule (2) from the example (resp. r_2 in the Figure) is directly shadowed by rule (1) resp. r_1 while the rule r_4 is shadowed by a combination of the rules r_1 and r_3. Shadowed rules have a negative impact on the firewall's performance as they need to be checked for packets without having an impact on the filtering semantics of the rule set. Therefore, removing a shadowed rule improves the rule set's performance and, in addition, improves its maintainability for administrators.

Note that some approaches in literature also require that the shadowed rule has a different action than the shadowing rule, e.g., [148, 11]. Though, in this

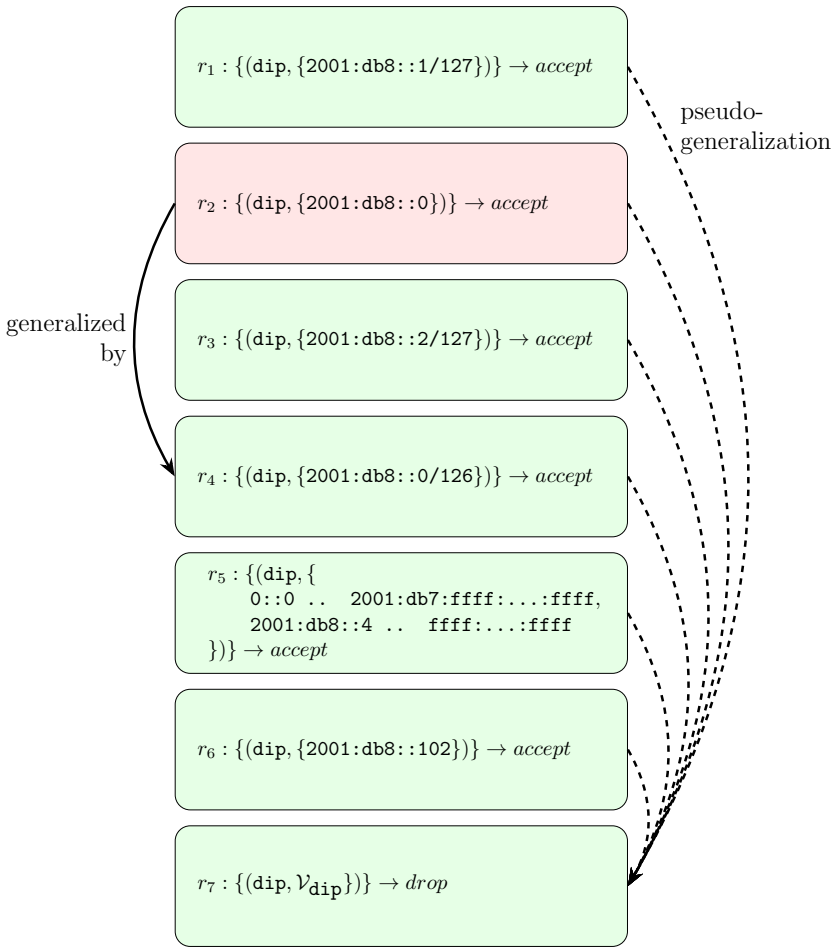


Fig. 8.2.: Generalized and pseudo-generalized rules in the DHSA model of the example rule set. The red rule r_2 is generalized by the rule r_4 as indicated by the solid arrow.

case, the administrator's intent remains unclear. If the shadowed rule denies traffic that is allowed by the upper rules, it is not clear whether this behaviour is rooted in the security policy. As this is not decidable without knowledge of the actual policy, automatic resolution is not possible and the conflict has to be reported and handled manually.

For resolving these conflicts, we suggest a two step approach: First, specify a clean high-level description of such intents and use a policy checker, e.g. FPL and FaVe. Second, as this guarantees a compliant rule set, any rule without an impact on the filtering semantics, i.e., which is shadowed by our definition, can be reported for safe removal.

Generalization A rule is generalized by one or more rules with lower priorities if these handle the same packets. In our example and depicted in Figure 8.2, rule r_2 is generalized by rule r_4. Also, the rules r_1 to r_6 are generalized by

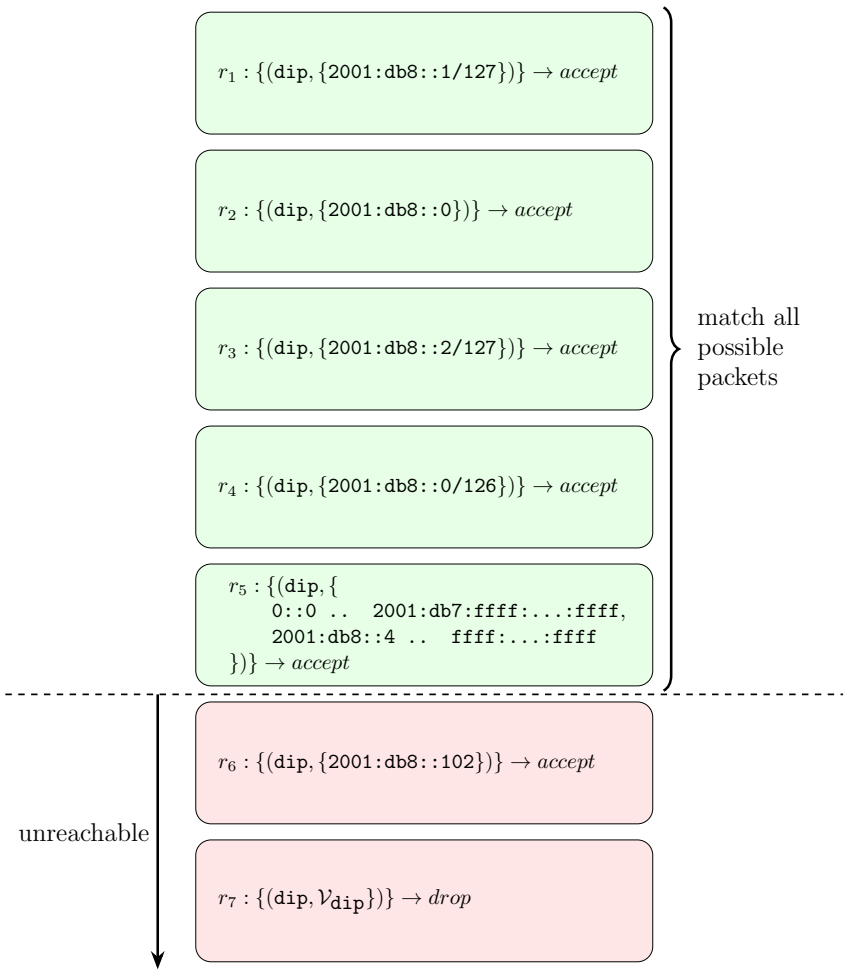


Fig. 8.3.: Unreachable rules in the DHSA model of the example rule set. The rules up to the dashed line match all possible packets and, hence, the red rules below the line are not reachable.

the default rule r_7. The example shows that the generalization by the default rule is pretty common and intended. Hence, the *pseudo-generalization* by the default rule should not be reported.

All in all, generalization can be regarded rather an information for the administrator than an anomaly. In order to safely remove a generalized rule it is necessary to analyse its relation with all overlapping rules with a lower priority until all relevant packets have been handled. If any of these dependent rules has another action, the generalized rule cannot be removed. In this case, reporting this generalized rule can help administrators to reevaluate their intents.

Unreachability A rule is unreachable if all packets have been processed by rules with higher priorities. Or stated differently: A rule is reachable if at least one

packet is checked against this rule. Note that it is not necessarily a matching packet. Unreachable rules are always shadowed and, hence, they can be removed safely. As shown in Figure 8.3, the rules r_6 and r_0 are unreachable as the rules r_1, r_3, and r_5 process all possible packets. Further, the rules r_4 and r_5 also match all packets.

Redundancy Two rules are redundant if they handle the same packets and have the same action. Hence, the rule with the lower priority can be removed safely. Our definition of shadowing covers redundant rules, too, as we do not distinct by action. Hence, we also detect redundant rules when we check for shadowed rules and we can safely remove them.

Correlation Two rules correlate if they overlap, i.e., they match a common set of packets, have different actions, and the higher prioritized rule does not shadow the other. Since this behaviour may be intended by the administrator, this anomaly may be only reported.

8.2 Related Work

As shown in Table 8.1, the field of firewall anomaly detection has been under research for decades. Though, none of these approaches has yet gained broader application – particularly, due to their impractically long runtimes. For instance, Cisco's Security Manager offers the rulewise detection of shadowing and redundancy anomalies [22]. However, best practices recommend the analysis of only small rule sets of "sections of 100 rules"[2]. Also, the open source tool *FirewallBuilder* ([106], cf. Section 4.2) implements a limited shadowing detection. Nevertheless, the tool seems to be discontinued and there is no successor.

In the following, after describing the applied methodology, we give an overview of these verification approaches, highlight notable tools, and compare them to our own verification framework FaVe (cf. Chapter 6). In Table 8.1, we give an overview of related tools that offer firewall anomaly detection. The table is structured by the categories of single and multi firewall anomalies, as well as other notable traits which are explained in the following:

[2] `https://community.cisco.com/t5/network-security/duplicate-firewall-rules-shadow-redundant/td-p/2973394`

Tool	Single Firewall Anomalies						Multi Firewall Anomalies				Other			
	Shadowing	Unreachability	Correlation	Generalization	Redundancy	Shown Scalability	Shadowing	Unreachability	Loop Detection	Cross Path	Aggregated Anomalies	Verification Engine	IPv6 Support	Public Code
FPA [132]	✓		✓	✓	✓		✓					F		
FIREMAN [153]	✓		✓	✓	✓		✓			✓	✓	B		
Abedin et al. [1]	✓		✓		✓							I		
MIRAGE [48]	✓				✓		✓				✓	I		✓
Jeffrey, Samak [73]		✓				✓		✓	✓		✓	S		
Prometheus [111]	✓		✓	✓						✓	✓	B		
Kotenko, Polubelova [83]	✓		✓	✓	✓							F		
FAME [62]	✓		✓	✓	✓						✓	B		
Khorchani et al. [80]	✓		✓	✓	✓	✓						C		
Basumatari, Hazarika [11]	✓		✓	✓	✓	✓						C		
Krombi et al. [84]	✓			✓								F		
Saâdaoui et al. [126]	✓		✓	✓	✓	✓					✓	S		
Hanamsagar et al. [58]	✓		✓				✓					?		
ad6 [94]	✓	✓				✓	✓	✓	✓	✓	✓	S	✓	✓
FARE [127]	✓					✓				✓	✓	C		
Yin et al. [148]	✓		✓	✓	✓							T	(✓)	
HSViz [86]	✓		✓	✓	✓		✓					?		
Lin et al. [88]												C		
Bringhenti et al. [17]	✓		✓	✓								T		
Hakani, Mann [57]	✓		✓	✓	✓							C		
FaVe + NetPlumber	✓	✓		✓		✓			✓		✓	H	✓	✓

Tab. 8.1.: Overview of different formal firewall anomaly detection tools. The abbreviations in the column *Verification Engine* mean: B = BDD, C = Custom Engine, F = Finite Automaton, H = HSA, I = Intervals, S = SAT, T = Theorem Prover.

Single Firewall Anomalies Also often referred to as *intra-firewall*, these anomalies occur within a single firewall rule set whenever there is a packet that could be handled by multiple rules. Hence, the packet's fate depends on these rules' order resp. priorities. For a more detailed description of the shadowing, unreachability, generalization, redundancy, and correlation anomalies, see Section 8.1. In addition and similar to our methodology in Chapter 3, we acknowledge an approach's *shown scalability* claim if their evaluation reveals the tool's ability to perform a shadowing detection of a packet filter rule set with at least 1,000 rules throughout a coffee break of 10 minutes.

Multi Firewall Anomalies In more complex environments which use a network of firewalls, further anomalies like cycles and cross-paths are possible. Also, the concept

of single firewall anomalies, e.g., shadowing or unreachability, can be applied if we consider paths with multiple firewalls. On these paths, earlier, i.e., *upstream*, firewalls behave as higher prioritized in comparison to later, i.e., *downstream*, firewalls. Hence, a rule in an upstream firewall could shadow a rule in a downstream firewall. Often, multi firewall anomalies are referred to as *inter-firewall* anomalies.

Shadowing: A rule in a downstream firewall is shadowed if all relevant packets have been dropped by rules in upstream firewalls or handled by rules with higher priorities in the same firewall. We limit our investigation on shadowing and the closely related unreachability anomalies since especially the former is the most important and, by far, most commonly analyzed anomaly in literature.

Unreachability: Rules are unreachable if all packets have already been dropped by upstream firewalls or handled by rules with higher priorities in the same firewall.

Loop Detection: Packets that loop within a network consume extensive resources. Even though, firewalls typically operate on layer-3 or higher and, hence, packets do not loop forever, they unnecessarily consume network ressources until their TTL reaches 0. This can be a serious security issue if an attacker is able to craft large amounts of such packets and can cause a Denial-of-Service of the network in the worst case.

Cross Path: This anomaly occurs in situations where a packet is handled differently on different paths through the network. This is not possible in clearly separated networks which are based on a DMZ, but may occur e.g., if a firewall cluster is used for load balancing and two firewalls are configured differently.

Other Further, notable features are the tool's ability to detect aggregated anomalies and analyze IPv6 rule sets:

Aggregated Anomalies: A notable distinction is given by the expressiveness of the analysis. Most approaches check the relation between two rules at a time [19], i.e., they perform a *pairwise* detection. But, some approaches offer a more general notion of anomalies as they analyze the relations between multiple rules at once [153, 73, 62, 126, 94], i.e., they perform an *aggregated* detection. Hence, these approaches find more and more subtle anomalies than those which only support pairwise detection.

Verfication Engine: We highlight which formalism resp. verification engine is utilized by the approach.

IPv6-Support: The tool can adequately model and process IPv6 configurations. We mark limited IPv6 support with round brackets, e.g., if the tool only supports IPv6 addresses or if it loses verification precision when modeling dynamic behaviour, e.g., extension header chaining.

Public Code: The authors also published their prototype.

Scope: In this work we focus on single-firewall anomalies for three reasons. First, the underlying verification engine NetPlumber is already capable to detect cycles in network configurations modeled with FaVe. Second, cross-path anomalies only occur in rather uncommon scenarios where packets may take arbitrary routes passing differently configured firewalls. E.g., in a load balanced firewall cluster where the individual firewalls are configured manually. Third, therefore, a more clean approach to detect unwanted forwarding and filtering behaviour would be the specification of intent using high-level policies and, then, checking the network configuration for compliance, e.g., as offered by FPL and FaVe.

Detailed Description

Since we aim to support state-of-the-art IPv6 networks, we put an emphasis on the only two approaches with at least rudimentary IPv6 support [94, 148]. Though, since we already described our previous approach ad6 in the preliminary study in Section 1.4, we omit further details here. Furthermore, only few approaches have shown scalability [73, 80, 11, 94, 126, 127] among which the approach by Basumatari and Hazarika [11] did perform best and will be described in further detail. We chose these three approaches as state-of-the-art for comparison with FaVe in Section 8.5.1. Since there is no available source code for STL [11] and the work from Yin et altera [148], we reimplemented their approaches.

As seen in Table 8.1, all approaches are able to detect shadowing or, at least, unreachability anomalies for single firewalls. Notably, the tools put an emphasis on the analysis of single firewalls rather than networks of firewalls. Or, from a different perspective, there is no tool that detects only multi-firewall anomalies like loops but not single firewall anomalies. Also, no tool has shown scalability for multi firewall networks and we refrained from including this as a column. We argued that multi-firewall anomalies are less important except for loop detection which can be

performed very efficiently using data plane verification tools (cf. Section 3.2), e.g., NetPlumber.

Basumatari and Hazarika This approach follows the formalism of *Spatial Temporal Logics* (STL, [146]) but instead of using a generic model checker the authors implemented a custom one. In STL's terminology, they model firewall rules as areas of a topological space $\mathfrak{T}$, i.e., Header Space, and a rule set as a temporal sequence $\mathfrak{F}$. As a result, the model is a multidimensional spatio-temporal structure. In order to compare two rules, they define five *rule relations* (RR, similar to [133]): disjoint DR, partially overlapping PO, equal EQ, proper part PP, inverse proper part PPi. For instance, the *proper part* between two rules R_1 and R_2 is defined as:

$$PP(R_1, R_2) := (\forall x \in \mathfrak{T} : x \in R_1 \rightarrow x \in R_2) \wedge (\exists x \in \mathfrak{T} : x \notin R_1 \wedge x \in R_2)$$

The inverse proper part is defined as $PPi(R_1, R_2) := PP(R_2, R_1)$.

Using these relations, the authors describe firewall anomalies as temporal expressions. For instance, given a model $\mathfrak{M}$, a rule R, and an index n, they define the modal *sometimes* operator $\Diamond$ as ($\mathfrak{T}$ is the topological space, $\mathcal{R}(\mathfrak{T})$ is the rule set):

$$\mathfrak{M}, R, n \models \Diamond[RR(R, R')] \text{ iff } \exists R' \in \mathcal{R}(\mathfrak{T}) : n' \in \mathbb{N}, n < n' \text{ and } RR(R, R') \text{ holds}$$

Read: The rule set satisfies the expression if for a specified rule there is a rule in a specific relation with a lower priority. Now the shadowing anomaly can be described as:

$$\mathfrak{M}, R, n \models \Diamond[EQ(R, R') \vee PPi(R, R')] \wedge \mathfrak{A}(R, n) \neq \mathfrak{A}(R', n')$$

where $\mathfrak{A}(R, n)$ determines a rule's action.

Read: A rule R shadows a subsequent rule R' if their matches are equal or inverse proper parts and if the rules perform different actions. Note that the latter requirement differs from our definition (cf. Section 8.1).

Since all supported anomalies, i.e., shadowing, generalization, redundancy, and correlation, are defined using only the $\Diamond$-operator, it is sufficient to implement and optimize only this one. The authors implement their model checker using three algorithms. For two rule matches, they perform a field-wise comparison and construct a bit vector that holds flags indicating if for any field the value sets are distinct, equal, a subset, or a superset. Based on these bit patterns, a second algorithm determines the relation of two rules, i.e., if they are disjoint, equal, partially overlapping, proper parts, or inverse proper parts. Finally, for a given rule,

a third algorithm traverses the rule set, determines rule relations, and checks if conditions for an anomaly holds. The authors evaluate their prototype with synthetic workloads with up to 20,000 quintuple rules. The stated run time for the largest workload is about 4.5 seconds which clearly indicates scalability. We reimplemented the approach[3] and performed comparative measurements in Section 8.5.1 which show that FaVe clearly outperforms STL by a factor of 115 for a large real-world workload. Furthermore, FaVe shows its scalability to 15,000 rules for multiple synthetic workloads in Section 8.5.2.

Yin et al. This approach [148] performs firewall anomaly analysis by pairwise rule comparison and offers limited IPv6 support, i.e., IPv6 prefixes only. It models rule matches as predicates over intervals and constructs Z3 formulas in order to determine two rules' relations. Based on rule priority, relation, and action, anomalies can be identified.

Particularly, header fields are modeled as sequences of intervals, i.e., IPv6 addresses comprise eight intervals where each interval covers 16 bits in the address. For instance, the IPv6 destination address prefix `2001:db8::0/32` is modeled as:

$$
\begin{bmatrix}
(\texttt{0x2001}, \texttt{0x2001}),\\
(\texttt{0x0db8}, \texttt{0x0db8}),\\
(\texttt{0x0}, \texttt{0xffff}),\\
(\texttt{0x0}, \texttt{0xffff}),\\
(\texttt{0x0}, \texttt{0xffff}),\\
(\texttt{0x0}, \texttt{0xffff}),\\
(\texttt{0x0}, \texttt{0xffff}),\\
(\texttt{0x0}, \texttt{0xffff})
\end{bmatrix}.
$$

[3] The code is available at `https://github.com/cllorenz/fave-project/tree/master/stl-anomalies`.

The resulting Z3 formula is the following:

$$
\begin{aligned}
&\texttt{AND}(\\
&\qquad (\texttt{==}(\texttt{0x2001}, \text{dip}_1)),\\
&\qquad (\texttt{==}(\texttt{0x0db8}, \text{dip}_2)),\\
&\qquad (\texttt{<=}(\texttt{0x0}, \text{dip}_3)), (\texttt{<=}(\text{dip}_3, \texttt{0xffff})),\\
&\qquad (\texttt{<=}(\texttt{0x0}, \text{dip}_4)), (\texttt{<=}(\text{dip}_4, \texttt{0xffff})),\\
&\qquad (\texttt{<=}(\texttt{0x0}, \text{dip}_5)), (\texttt{<=}(\text{dip}_5, \texttt{0xffff})),\\
&\qquad (\texttt{<=}(\texttt{0x0}, \text{dip}_6)), (\texttt{<=}(\text{dip}_6, \texttt{0xffff})),\\
&\qquad (\texttt{<=}(\texttt{0x0}, \text{dip}_7)), (\texttt{<=}(\text{dip}_7, \texttt{0xffff})),\\
&\qquad (\texttt{<=}(\texttt{0x0}, \text{dip}_8)), (\texttt{<=}(\text{dip}_8, \texttt{0xffff}))\\
&).
\end{aligned}
$$

The protocol and port fields are modeled in a similar manner using one interval each. A match is a conjunction of the formulas representing the IPv6 source and destination addresses, the source and destination ports, and the protocol field. In order to determine the relation of two rule matches R_1 and R_2, up to three formulas are created and solved with Z3. First, if the formula $\texttt{AND}(R_1, R_2)$ is not satisfiable, the rules match no common packet and, hence, they are disjoint and there is no anomaly. If the rules are not disjoint two further formulas are constructed and solved – one for the complete inclusion of R_1 in R_2, i.e., $\texttt{AND}(\texttt{NOT}\ R_1, R_2)$, and one for the complete inclusion of R_2 in R_1, i.e., $\texttt{AND}(R_1, \texttt{NOT}\ R_2)$. If both are not satisfiable, i.e., there is no packet matched by one rule but not the other, then the rules are equal. If the one formula holds but not the other, then the one rule is included in the other. Finally, if both formulas are satisfiable, then both rules match packets that are not matched by the other and, hence, they simply overlap. Now, the authors perform pairwise analysis and depending on the rules' relation, anomalies can be determined, e.g., shadowing is present if the lower prioritized rule is equal to or included in the higher prioritized rule and they have different actions. Further the tool detects generalization, redundancy, and correlation anomalies.

The authors evaluate their approach with synthetic rule sets that were generated with ClassBench-ng [100] and the largest workload consisted of 2,000 IPv6 quintuple rules. The analysis took about 30 minutes for this workload which does not indicate scalability. In Section 8.5.1, we compare our reimplementation of this approach[4] with Fave using a large real-world rule set where FaVe outperforms the tool by a factor of nearly 5,000.

[4]The code is available at `https://github.com/cllorenz/fave-project/tree/master/z3-anomalies`.

Other Approaches

Several formalisms and generic verification engines have been explored, i.e., BDDs [153, 111, 62], finite automata [132, 83, 84], intervals [1, 48], SAT [73, 126, 94], and theorem prover [148]. Also, proposals of custom approaches have been evaluated, i.e., FDDs [127], Visibility Logics [80], STL [11], Double Decision Trees [88], and Firewall Trees [57]. These custom approaches to formal verification tend to scale better with the exception of some SAT based approaches [73, 126, 94]. Particularly, ad6 [94] which is an IPv6 extension to the approach of Jeffrey and Samak [73] did not scale well to larger networks. The approach of Saâdaoui et altera [126] analyzed 1,000 synthetic quintuple rules in less than a second but did not provide any means for multi-firewall networks. Further, the custom approach FARE [127] analyzed a real-world rule set with 3,768 rules in about 18 seconds. Nevertheless, these were only a list of IPv4 source addresses for deny listing and scalability was not shown for a larger network – despite support for the detection of cross-path anomalies.

Finally, the detection of the more expressive aggregated anomalies were shown to be detected in [153, 28, 73, 62, 126, 94, 127]. Along with [73, 126, 94, 127], FaVe is able to perform these analysises in a scalable manner.

8.3 Characterizing Firewall Anomalies with DHSA

As seen previously in Section 8.2, most of the existing anomaly detection approaches are limited to IPv4 and rely on general model checking techniques which yield limited performance results. We overcome these shortcomings by, first, modeling firewall rule sets and describing anomalies in terms of DHSA. Then, second, we enhanced FaVe and NetPlumber to perform rapid analysis thereof.

As stated with the related work (cf. Section 8.2), anomalies can be defined *pairwise* or *aggregated*. I.e., per rule, a single rule is checked for negative influence versus multiple rules in combination. In this work, we support the more general notion of aggregated anomalies.

Shadowing To detect shadowed rules, we traverse the rule set R linearily step by step and for each rule $r_i : m_i \rightarrow a_i$ we apply the following check:

$$\{m_i\} \subseteq \bigcup_{j<i} \{m_j\}$$

If the i-th match is a subset, then all relevant packets have been handled by rules with higher priorities. Hence, rule r_i is shadowed.

For instance, to check rule r_2 from our example (cf. Figure 8.1), we collect the matches of the previous rules which is just the match of rule r_1:

$$\bigcup_{j<2} \{m_j\} = \{\{(\texttt{dip}, \{\texttt{2001:db8::1/127})\}\}$$

Then, we check if $\{m_2\}$ is a subset of the collected matches:

$$\{m_2\} \subseteq \{m_1\} \Leftrightarrow \{\{(\texttt{dip}, \{\texttt{2001:db8::0}\})\}\} \subseteq \{\{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\}\}$$

Since

$$\{(\texttt{dip}, \{\texttt{2001:db8::0}\})\} \cap \{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\} = \{(\texttt{dip}, \{\texttt{2001:db8::0}\})\}$$

the check yields true, and we correctly identify r_2 to be shadowed by r_1.

A second and more complex example is the shadowing check of r_4. First, we collect the matches of the rules with higher priorities, i.e., the rules r_1 to r_3:

$$\begin{aligned} \textstyle\bigcup_{j<4}\{m_j\} = \; & \{ \\ & \quad \{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\}, \\ & \quad \{(\texttt{dip}, \{\texttt{2001:db8::0}\})\}, \\ & \quad \{(\texttt{dip}, \{\texttt{2001:db8::2/127}\})\} \\ & \} \\ \hat{=} \; & \{\{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}\} \end{aligned}$$

We can simplify the resulting match set by, first, removing the match

$$\{(\texttt{dip}, \{\texttt{2001:db8::0}\})\}$$

which is a subset of

$$\{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\}$$

and, second, by merging

$$\{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\} \text{ with } \{(\texttt{dip}, \{\texttt{2001:db8::2/127}\})\}$$

to

$$\{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}.$$

Now, we can check if $\{m_4\}$ is a subset of $\bigcup_{j<4}\{m_j\}$. I.e.,

$$\{\{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}\} \subseteq \{\{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}\}$$

which is true and, thus, r_4 is correctly detected as shadowed. Note that it is a combination of the rules r_1 and r_3 that shadows r_4.

Unreachability To detect if a rule $r_i : m_i \to a_i$ is unreachable, we collect all matches of rules with a higher priority. Then, we check if this match set equals the full header space M_Ω:

$$\bigcup_{j<i}\{m_j\} = M_\Omega$$

For instance, we check the unreachability of rule r_6 from our example (cf. Figure 8.3). First, we collect the matches of the rules r_1 to r_5:

$$\begin{aligned}
\textstyle\bigcup_{j<6}\{m_j\} = \; & \{ \\
& \quad \{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\}, \\
& \quad \{(\texttt{dip}, \{\texttt{2001:db8::0}\})\}, \\
& \quad \{(\texttt{dip}, \{\texttt{2001:db8::2/127}\})\}, \\
& \quad \{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}, \\
& \quad \{(\texttt{dip}, \{ \\
& \qquad \texttt{0::0..2001:db7:ffff:...:ffff}, \\
& \qquad \texttt{2001:db8::4..ffff:...:ffff} \\
& \quad \})\} \\
& \} \\
\hat{=} \; & \{ \\
& \quad \{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}, \\
& \quad \{(\texttt{dip}, \{\texttt{0::0/0} \setminus \texttt{2001:db8::0/126}\})\} \\
& \} \\
\hat{=} \; & \{\{(\texttt{dip}, \{\texttt{0::0/0}\})\}\}
\end{aligned}$$

We check if this match set equals the full header space M_Ω which is true. Hence, r_6 is not reachable and neither are all subsequent rules.

Generalization To detect if a rule $r_i : m_i \to a_i$ is generalized by rules with lower priorities, we traverse the rule set in reverse order and skip the default rule which always generalizes all other rules.

$$\{m_i\} \subseteq \bigcup_{i<j<|R|} \{m_j\}$$

For instance, we check if rule r_2 from our example is generalized (cf. Figure 8.2). First we collect the matches of the rules r_3 to r_6 (but not r_7):

$$
\begin{aligned}
\bigcup_{j<6}\{m_j\} = \; & \{ \\
& \{(\texttt{dip}, \{\texttt{2001:db8::2/127}\})\}, \\
& \{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}, \\
& \{(\texttt{dip}, \{ \\
& \quad \texttt{0::0..2001:db7:ffff:...:ffff}, \\
& \quad \texttt{2001:db8::4..ffff:...:ffff} \\
& \})\}, \\
& \{(\texttt{dip}, \{\texttt{2001:db8::102}\})\} \\
& \} \\
\hat{=} \; & \{ \\
& \{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\}, \\
& \{(\texttt{dip}, \{\texttt{0::0/0} \setminus \texttt{2001:db8::0/126}\})\}, \\
& \{(\texttt{dip}, \{\texttt{2001:db8::102}\})\} \\
& \} \\
\hat{=} \; & \{ \\
& \{(\texttt{dip}, \{\texttt{0::0/0}\})\}, \\
& \{(\texttt{dip}, \{\texttt{2001:db8::102}\})\} \\
& \} \\
\hat{=} \; & \{\{(\texttt{dip}, \{\texttt{0::0/0}\})\}\} = \{m_\Omega\} = M_\Omega
\end{aligned}
$$

Then, we check if $\{m_2\} = \{\{(\texttt{dip}, \{\texttt{2001:db8::0}\})\}\}$ is a subset of M_Ω which is true. Hence, we correctly identify r_6 as generalized.

8.4 Implementation

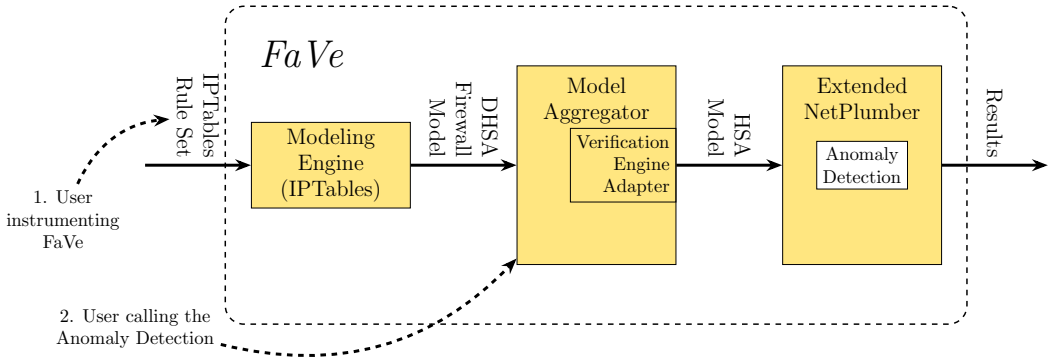


Fig. 8.4.: User interactions with and extensions to FaVe and NetPlumber.

Our prototype reuses and extends FaVe. Figure 8.4 depicts the user interactions and the extended components used in our AAD workflow. In a first step, the user instruments FaVe's *IPTables Modeling Engine* with a rule set. It is *preprocessed* and modeled as DHSA packet filter model. This model is transformed by the *Model Aggregator* to a HSA model. Finally, the extended NetPlumber is instrumented with the model. In a second step, after instrumentation, the user calls the anomaly check and the results can be used to generate a report.

We implemented the anomaly detection directly in NetPlumber to benefit from its efficient data structures. Changes to the rest of FaVe were minimal, i.e., we only needed to make the anomaly detection available for users. In detail, we implemented the algorithms to detect anomalies as specified in Section 8.3. For this purpose, we extended NetPlumber with a merging function for wildcard expressions which is, then, used in a memory-preserving union operation that inserts wildcard expressions in header space objects.

Match Merging We introduce the term *mergeability* for two DHSA matches if they can be represented with only one HSA match that, in turn, results in a single wildcard expression when translated to HSA. We define the function $merge : M \times M \to \mathcal{M}$ to perform such merges of two matches into one. The result is a match set containing either the merged match or the original matches.

In NetPlumber, two matches are mergeable if they differ only in one position. For instance, in our example, the matches of the rule r_1 and r_3 can be merged, i.e.,

$$merge \left(\begin{array}{l} \{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\}, \\ \{(\texttt{dip}, \{\texttt{2001:db8::2/127}\})\} \end{array} \right) = \{(\texttt{dip}, \{\texttt{2001:db8::0/126}\})\},$$

because their representations as wildcard expressions, i.e.,

`00100000,00000001,00001101,10111000,00000000,...,00000000,0000000x`

resp.

`00100000,00000001,00001101,10111000,00000000,...,00000000,0000001x`,

differ in the second to last bit and, hence, can be merged into one expression:

`00100000,00000001,00001101,10111000,00000000,...,00000000,000000xx`.

Memory-preserving Match Insertion Now, we can define a function that inserts single matches into a match set in a memory saving manner. It builds upon mergeability and subset relations between matches resp. match sets. Formally, the memory-preserving match insertion function is defined as:

$$merge\_insert : M \times \mathcal{M} \to \mathcal{M}$$

$$merge\_insert(m, M) := \begin{cases} M, \text{ if } \exists m' \in M : m \subseteq m' \\ (M \backslash \{m'\}) \cup \{m\}, \text{ if } \exists m' \in M : m' \subseteq m \\ (M \backslash \{m'\}) \cup \{m''\}, \text{ if } \exists m' \in M : merge(m, m') = \{m''\} \\ M \cup \{m\}, \text{ else} \end{cases}$$

The first and second case try to find an element m' from the match set that is a superset resp. subset of m. If found, the superset remains in resp. is introduced into the resulting match set. The third case tries to find an element m' that can be merged with m to a single match m'' which then replaces m'. The last case applies if m cannot be introduced in a memory-saving manner. It is, then, simply included in the match set.

For instance, when we introduce the match m_2 from rule r_2 into a match set comprising of the match m_1 from rule r_1, we get the following match set:

$$merge\_insert \left(\begin{array}{l} \{(\texttt{dip}, \{\texttt{2001:db8::0}\})\}, \\ \{ \\ \quad \{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\} \\ \} \end{array} \right) = \{\{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\}\}.$$

Since m_2 is a subset of m_1, the first case applies and m_2 can be ignored, i.e., the result is $\{m_1\}$.

As second example, we introduce the match m_3 from rule r_3 into an already compressed match set created from the matches m_1 and m_2 from the rules r_1 and r_2, i.e., $merge\_insert(m_2, \{m_1\}) = \{m_1\}$:

$$merge\_insert \left(\begin{array}{l} \{(\texttt{dip}, \{\texttt{2001:db8::2/127}\})\}, \\ \{ \\ \quad \{(\texttt{dip}, \{\texttt{2001:db8::1/127}\})\} \\ \} \end{array} \right) = \{\{(\texttt{dip}, \{\texttt{2001:db8::1/126}\})\}\}.$$

The match m_3 does not have a subset relation with m_1 but can be merged with m_1. Hence, in the result, m_1 is replaced with the match from the merge, i.e., $m'' = \{(\texttt{dip}, \{\texttt{2001:db8::1/126}\})\}$.

Memory-preserving Match Set Union Now, we can use the compressing insertion function to implement a memory-preserving variant of the union of a list of wildcard expressions to a header space object, e.g., in order to implement the shadowing check: $\{m_i\} \subseteq \bigcup_{j<i}\{m_j\}$.

Originally, this union was performed through a element-wise appending which is simple and fast but not memory-efficient. Starting with an empty header space object $\emptyset$, our implementation traverses the list of wildcard expressions and inserts each into the header space object.

For instance, the pseudo-code algorithm for the shadowing check is the following:

```
# Input: List of rules R
# Output: List of shadowed rules

def shadowed_rules(R):
    res ← ∅
    acc ← ∅

    for r_i : m_i → a_i in R:
        if m_i ⊆ acc:
            res ← res ∪ {r_i : m_i → a_i}

        acc ← merge_insert(m, acc)

    return res
```

The list of rules is traversed sequentially and each rule is checked against the set of packets that has been accumulated for the rules with higher priorities. If it is a subset, the rule is marked as shadowed by being added to the result rule set.

Rule Set Preprocessing NetPlumber lacks the native support of port ranges, since its data structure is based on ternary bit vectors which are not suited to represent arbitrary value ranges. This fact led to our decision to implement a preprocessing of the rule set in FaVe's modeling engine, i.e., before we build the DHSA model. This preprocessing transforms rules with port ranges into a set of rules that use prefixes and, combined, offer the same filtering semantics.

E.g., the rule

```
iptables -A FORWARD -p tcp -m multiports --dports 1:3 -j ACCEPT
```

which covers the ports 1, 2, and 3 is transformed into the rules:

```
iptables -A FORWARD -p tcp -m multiports --dports 2:3 -j ACCEPT
iptables -A FORWARD -p tcp --dport 1 -j ACCEPT
```

Hence, the DHSA match for the first rule is $\{(\texttt{dport}, \{2, 3\})\}$ resp. $\{(\texttt{dport}, \{1\})\}$ for the second. The resulting wildcard expressions are `00000000,0000001x` resp. `00000000,00000001`. To construct the prefixes, we use the algorithm from [18].

If a rule uses port ranges for source and destination, we convert both ranges into sets of prefixes and construct the rules combinationally. I.e., each source port prefix is combined with each destination port prefix. The worst case expansion of a range of width W is $2 \cdot (W - 1)$ [18]. This is $2 \cdot (16 - 1) = 30$ for our 16-bit port ranges. Hence, in the worst case, a single rule which uses source and destination port ranges can expand to $\frac{30 \cdot (30-1)}{2} = 435$ rules.

8.5 Evaluation

Benchmark	UP cf. Section 7.2	TUM/TUMv6 cf. Section 7.2	ClassBench-ng generated using [100]
Type	synthetic	real	synthetic
Rules	1,035	3,795	500, 1,000, 2,000, 5,000, 10,000, 15,000
IP Version	IPv6	IPv4/IPv6	IPv6
Header Fields	`iif, oif, proto, sip6, dip6, sport, dport, limit, nxt_hdr, rtsegs, rttype, icmp6type`	`iif, oif, vlan, sip, dip, proto, sport, dport, tcp flags`	`sip, dip, proto, sport, dport`

Tab. 8.2.: Overview of the benchmarks.

We ran four benchmarks to show our approach's scalability and real-world usefulness as well as its IPv6 performance. In addition to the measurements with FaVe, we run the benchmarks with our previous tool ad6 [94] and the approach from [148]. These were the only tools from literature with native IPv6 support. For this purpose, we reimplemented the Z3-based algorithm from Yin et altera[5]. Also, we reimplemented and measured the STL approach [11] since it was the best scaling approach in literature using a custom solver.

[5]The benchmarks along with tools' source codes are available here: `https://github.com/cllorenz/fave-project`

Table 8.2 gives an overview of the benchmarks used in this work. Each workload consists of a single firewall rule set. We measured the run time of the shadowing detection which is the only anomaly detectable by all four tools. We conducted ten runs for each workload and all measurements were performed in the evaluation environment which has been used throughout this thesis (cf. Section 1.4).

8.5.1 Comparison with the State-of-the-Art

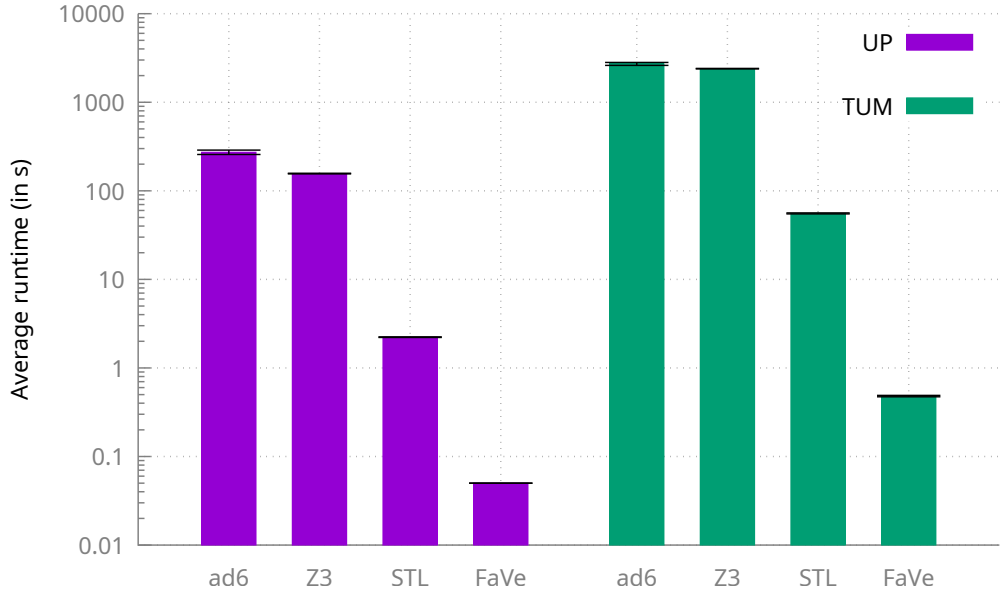


Fig. 8.5.: Run time comparison of the shadowing detection for the UP and TUM workloads with different tools (in s). We plot the mean run times with the (neglectable) standard deviations as confidence intervals.

Our experiments aim to compare the different approaches in three dimensions:

1. *Complexity Readiness:* Since the firewall anomaly detection is about the overcoming of complexity, we use two reasonably complex rule sets with a multitude of different header fields, i.e., 12 fields for the UP resp. 10 fields for the TUM workloads, and rule set size, i.e., 1,035 rules for the UP resp. 3,795 rules for the TUM workloads.

2. *Real-world Usefulness:* Both benchmarks are of realistic size to determine real-world applicability. Especially, as a comparably large real-world rule set, the TUM workload is particularly representative.

3. *Future Prospectivity:* We also investigate the tools' ability to detect firewall anomalies in state-of-the-art networks with IPv6 which is used in the UP benchmark.

Throughout all experiments, all four tools detect the same shadowing anomalies for the UP and TUM rule sets which is 0 for UP resp. 263 for TUM. This gives a high confidence in the correctness of the tools' mutually independent implementations.

As shown in Figure 8.5, ad6 and the Z3-based approach have comparable performance: 273 vs. 157 seconds for the UP and 2,712 vs. 2,395 seconds for the TUM benchmark. These similarities likely occur due to the fact that both tools rely on generic SAT solvers [104]. In contrast, FaVe runs more than 3 resp. 4 orders of magnitude faster and performs the shadowing detection in below one second, i.e., 0.05 s for the UP resp. 0.48 s for the TUM workload. FaVe outperforms ad6 and the Z3 approach – for instance, by a factor of nearly 5,000 on the TUM workload in comparison with Z3 (2,395 s vs. 0.48 s). Further, FaVe also outperforms the domain-specific STL approach by a factor of 115 on the TUM workload (55.58 s vs. 0.48 s). We assume that FaVe's performance superiority is rooted algorithmically. Our shadowing implementation in NetPlumber requires only one linear traversal of the rule set and our memory preserving accumulation minimizes inclusion checks used in a rule's shadowing analysis. Although the latter implies a quadratic worst case complexity, the TUM benchmark seems benign in this regard. STL's pairwise rule comparison, on the other hand, always results in a quadratic amount of inclusion checks, i.e., $\frac{|\#rules|^2}{2}$. Further, a single rule comparison in STL which requires $2 \cdot |\#header\_fields|$ integer comparisons might be slower than NetPlumber's highly optimized operations on wildcard expressions if a lot of header fields are involved.

All in all, the results clearly show that FaVe is capable to perform exceptionally in complex, real-world, and future-proof networks. The fact that STL performs second best and significantly better than ad6 and Z3 further underlines the strengths of domain-specific approaches to network security verification over general model checking.

8.5.2 Scaling to very large Rule Sets

By evaluating FaVe's performance concerning very large rule sets, we aim to identify principle limitations even though the sizes of the used rule sets clearly exceed real-world setups. For this purpose, we use the ClassBench-ng [100][6] generator for

[6]Available from `https://classbench-ng.github.io/`

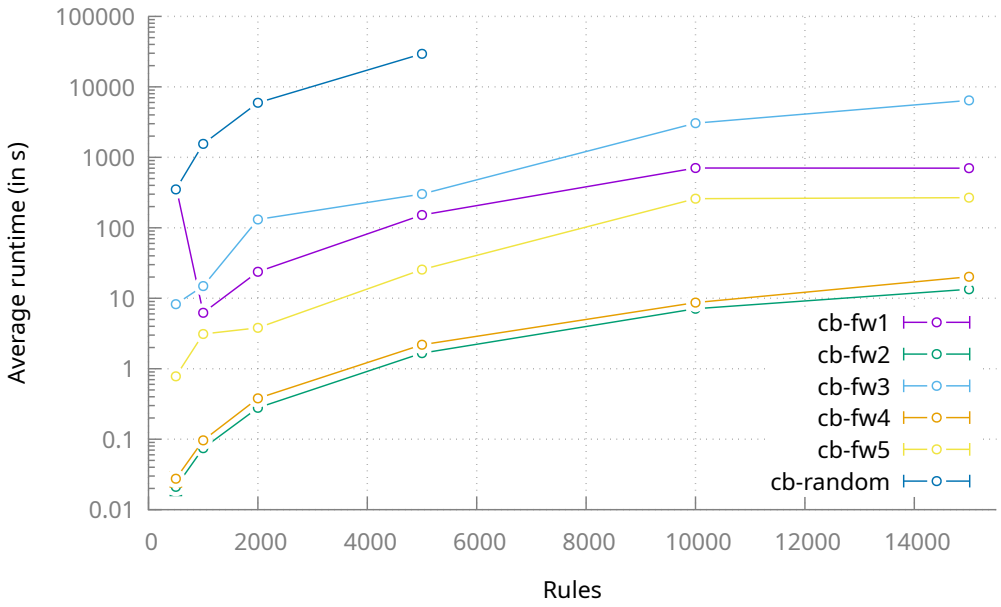


Fig. 8.6.: Scalability measurements using the ClassBench-ng workloads for the shadowing detection with FaVe (in s). We plot the mean run times with the (neglectable) standard deviations as confidence intervals.

synthetic workloads which was also used by Yin et al. [148]. The rules generated by this tool consist of IPv6 quintuples, i.e., source and destination prefixes, the protocol, as well as source and destination *port ranges*. The generation is based on seeds that were derived from some real-world rule sets and ought to preserve the filtering characteristica in generated rule sets [100]. While Yin et al. used only two different seeds and generated rule sets with up to 2,000 rules, we use all available firewall seeds (cb-fw1 to cb-fw5) as well as the cb-random which generates random rule sets. For these seeds, rule-sets with up to 15,000 rules were generated.

Since Yin et al. do not report which seeds they used, a direct comparison is not possible. But as shown in Figure 8.6, the scaling of FaVe is also very good for the extremly large rule sets based on the real-world seeds cb-fw1 – cb-fw5. The best cases, i.e., cb-fw2 and cb-fw4, need up to about 20 seconds for 15,000 rules. Even the cb-fw3 rule set, which performed the worst, had a runtime of about 107 minutes which is still acceptable for such large and very rare rule sets. Two exceptions are given by the cb-random rule set and by the cb-fw1 rule set with 500 rules.

Discussion We investigated on the reasons for the exceptionel long run times for the cb-fw1 workload with 500 rules and the cb-random rule sets. It turns out that they use a very large portion of complex port ranges in their rules.

Rulesets	Mean (in %)	Max (in %)
net-network	9.45	37.50
UP	0.00	0.00
TUM	4.69	4.69
cb-fw1	49.81	100.00
cb-fw2	0.00	0.00
cb-fw3	53.00	62.00
cb-fw4	1.32	1.50
cb-fw5	34.73	45.40
cb-random	100.00	100.00

Tab. 8.3.: Occurence of rules with port ranges.

For comparison, we analyzed the set of real-world rule sets published by [36] (titled *net-network*). Out of the 41 rule sets, none used port ranges for their source and destination ports simultaniously. In general, source port ranges are used rarely in just 10 out of 6,488 rules in total. On the contrary, all rules of the cb-fw1 workload with 500 rules use port ranges for their source and destination. This results in about 93,000 rules after preprocessing. Meanwhile, the theoretical maximum would have been $435 \cdot 500 = 217,500$.

Table 8.3 shows an analysis of the portions of port range rules for different rule sets, i.e., rules where the source part or destination part or both specify a port range. It shows that the use of port ranges in real-world rule sets is not very high, i.e., below 10 % on average (9.45 %). Also, the ClassBench-ng workloads which scale well correspond to a low portion of port range rules and vice versa. Thus, it is evident that a large portion of complex port ranges corresponds to long run times. Also Table 8.3 shows that complex port ranges are not a very common pattern in real-world rule sets (see for example the TUM benchmark).

8.5.3 IPv6 Performance

Finally, we measure FaVe's performance when verifying IPv6 compared to IPv4. Figure 8.7 shows the run times of the IPv4 and IPv6 versions of the TUM workload for FaVe. We can see that FaVe's total run time for IPv6 lies in the same order of magnitude as for IPv4. Particularly, verifying IPv6 is about 38 % slower than IPv4 (0.77 s vs. 0.48 s).

We can conclude that all in all FaVe performs well for IPv6. The overhead over IPv4 is meaningful but still in the same order of magnitude. Also, the total run time of less than a second is well acceptable.

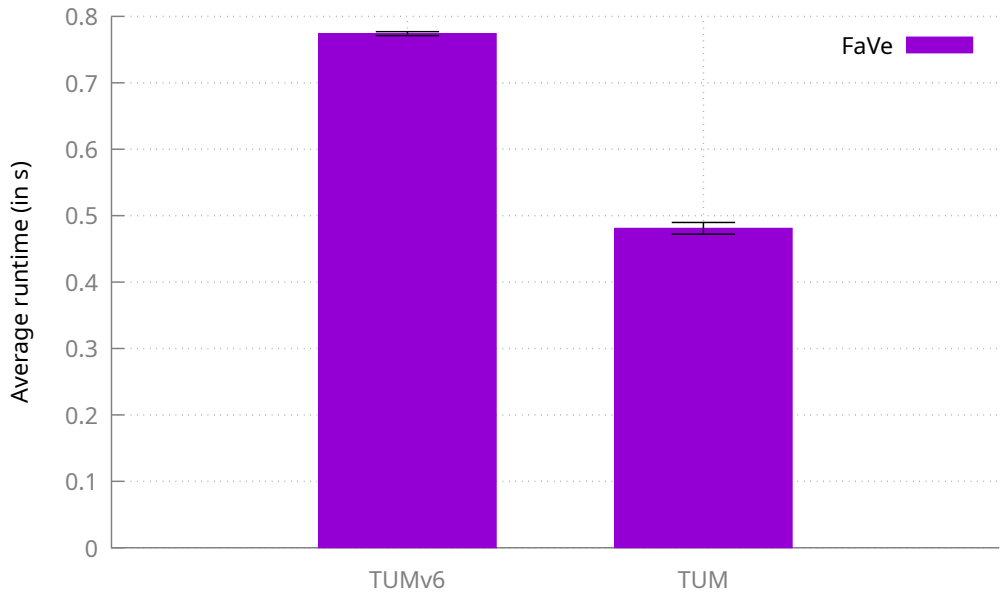


Fig. 8.7.: Measurements using the TUM and TUMv6 workloads for the shadowing detection with FaVe (in s). We plot the mean run times with the (neglectable) standard deviations as confidence intervals.

8.6 Summary

In this chapter, we leveraged DHSA and FaVe for the detection of firewall anomalies as part of the *Automatic Anomaly Detection* (AAD) workflow (cf. Section 1.3). Particularly, our approach detects aggregated shadowing, unreachability, and generalization anomalies, and also offers IPv6 support. Detecting these anomalies helps administrators to reduce the complexity of firewall rule sets significantly and, hence, improves usability and long term maintainability. Our evaluation revealed FaVe's very short run times which enable fast and often repetitions of the AAD workflow.

The presented benchmark results show a broad scalability for very large IPv6 workloads. In comparison with state-of-the-art approaches from literature and, with a run time of 0.48 seconds for the large real-world TUM rule set, our approach outperforms the next best by a factor of nearly 115. Hence, rapid anomaly detection even of large firewall rule sets is possible. In addition, our nearly 5,000-fold performance gain over model checking approaches indicates that a domain specific modeling and solving approach is suited much better for firewall anomaly detection.

Finally, our evaluation revealed that a large amount of port ranges in rules – though not very common – increases run time significantly. A possible approach to address this challenge, is to add native support to FaVe and NetPlumber to model ranges. We

expect that this feature would bring down run times to a few seconds as measured for most of the benchmarks.

To conclude, FaVe's results enable administrators to use the AAD workflow often and repeatedly to gain continuous feedback on configuration changes throughout the UNDERSTAND, ADJUST, and CONTROL phases. After initial determination of rule set complexity in the UNDERSTAND phase, subsequent runs in the ADJUST and CONTROL phases help to achieve and maintain a high degree of configuration quality. Though, once firewall rule sets are generated automatically in the CONTROL phase, the AAD loses relevance since these rule sets do not contain anomalies by design. Nevertheless, our analysis with DHSA and FaVe could be extended to other kinds of configuration, e.g., large routing tables or router configurations, which can be complex but are not natural candidates for synthesis from FPL due to their non-security related nature.

9 Conclusion and Outlook

In this work, we have conceptualized and demonstrated how to achieve and maintain continuous network security compliance. We have successfully applied formal methods to the challenge of verifying compliance of network configurations with organizational security requirements. In particular, we addressed three main research questions 1) on how to describe the desired state of network security, 2) on how to model network configurations, and 3) on how to achieve continuity-enabling performance. Our approach aims at the reduction of complexity through high-level security policy specifications, automated network modeling, and compliance verification. Subsequently, we offer the possibility to generate security configurations from policies. In addition, the reduction of complexity can be further advanced through the detection and elimination of firewall anomalies.

First, we enable security officials and network administrators to specify high-level security policies – particularly, access control through reachability – with our user friendly *FaVe Policy Language* (FPL). FPL allows the formal description of the desired state of network security and, thereby, we answer the first research question.

Further, we provide the fast verification framework *FaVe* which offers accessible means to model networks and security configuration for formal compliance verification. FaVe ships with numerous device models – including stateful packet filters – and modeling is based on our *Domain-oriented Header Space Algebra* (DHSA) which allows human-readable models and embeds into classical Header Space Algebra (HSA) through an injective homomorphism. FaVe efficiently implements the latter along with support for dynamic protocols – particularly IPv6. Finally, we reused and extended the well-known NetPlumber as FaVe's verification backend for fast HSA solving. By designing and implementing FaVe prototypically, we answer the second research question.

Our evaluation has shown that FaVe offers splendid performance for compliance verification. It models and verifies a complex university campus network in less than 36 seconds and outperforms the state-of-the-art by a factor of 41. This also indicates the superiority of a domain-specific approach to network security verification over the generic approaches from literature. In total, FaVe's performance allows for fast

initial verification as well as for rapid reverification and, hence, enables continuous compliance analysis. This solves the challenges of the third research question.

In addition, the reduction in rule set complexity with the help of static firewall anomaly detection, has been evaluated thoroughly. Our prototypical implementation using DHSA and FaVe conducts analysis in less than half a second for a large real-world rule set and outperforms the state-of-the-art by a factor of 115. Further, we have shown our approach's scalability for enormously large rule sets. The evaluation clearly revealed that the domain-specific verification performs much better for the detection of firewall anomalies than generic approaches as used by the state-of-the-art. FaVe's exceptional performance when detecting firewall anomalies further strengthens the case that FaVe is able to continuously verify meaningful, complexity-driving properties and, hence, underlines FaVe's stance on the third research question.

FPL, DHSA, and FaVe support security officials and network administrators to transform their networks towards a compliance-enhanced PDCA cycle. They can specify the desired state of network security with FPL, model their network and security configurations with FaVe's DHSA-based devices models, and continuously verify compliance with FaVe. Further, they can determine whether their firewall configuration contains unnecessary complexities in the shape of firewall anomalies. Once the desired state of network security has been reached, anomaly-free security configuration can be generated from FPL policies which keeps up compliance automatically throughout all phases of the PDCA cycle and, hence, poses a solid cornerstone of a thorough ISM.

Outlook

Despite the fact that FaVe broadly supports security officials and network adminstrators to achieve a compliance-enhanced PDCA cycle within their ISM, there are still open questions worthwhile further research. Possible areas of interest include Zero Trust Networking, Application Level Policies, further performance improvements, and AI support for managing FPL.

Crypto Overlays for Zero Trust Networking During the recent years, Zero Trust Networking (ZTN) [124] has emerged as an alternative approach in network security. Instead of network segmentation using a centralized network architecture with perimeter firewalls, ZTN aims at achieving security through more fine grained and,

ideally, cryptographically secured means. One approach in ZTN is the establishment of crypto overlays over untrusted networks that serve in a group or end-to-end-manner. In order to verify compliance in these kinds of networks, we would need to enhance FPL with more advanced specification capabilities (also cf. Section 1.8), e.g., for confidentiality, authenticity, or integrity. For instance, we could introduce new operators that require such traits, e.g., mutual authentication and encryption for confidentiality. Further, we would need to model encryption and authentication configurations, e.g., TLS or IPsec, and propagate encryption related meta information in NetPlumber, e.g., through virtual header fields.

Application Level Policies In this work, policies are limited to network level access control that is enforced through the restriction of network reachability. This kind of compliance verification is especially suitable for network segmentation approaches but leaves out access control measures on the application layer. Examples for network devices that enforce policies on the application layer include *Application Layer Gateways* (ALGs) or *Web Application Firewalls* (WAFs). In this regard, open challenges include the specification of meaningsful policies on the application level with FPL, e.g., concerning authentication or encryption with TLS, and efficient device modeling with DHSA. We have conceptualized and implemented an ALG model (cf. Appendix A.14) which is based on generic snapshots but does not offer deeper policies. We have shown that the snapshot-based approach has significant performance and scalability limitations in Section 7.1.6 and, hence, further research is necessary, e.g., by adapting the state shell interweaving from Section 7.1.7 to support generic models like finite automata.

Performance Our evaluation revealed further potential for even better performance. First, rules including value ranges, e.g., port ranges, require preprocessing that may result in large amounts of normalized rules and, hence, performance degregations (cf. Section 8.5.2). An idea to cope with this challenge would be the introduction of a data structure in NetPlumber that allows a concise and efficient processing of ranges and that accompanies the regular wildcard expressions in header space objects. For instance, ranges can be represented by intervals that can be concisely implemented by pairs of integers. Besides the correct implementation of set operations for these extended data structures in NetPlumber, it is necessary to decide when to use ranges in FaVe and also conduct proper bookkeeping.

Second and despite NetPlumber's great performance, it might be worthwhile to try another verification backend. For instance, AP-Verifier [147] or APKeep [156] (cf.

Chapter 3) have shown notable performance results in literature and are promising candidates to improve FaVe's overall performance and scalability even further.

AI Support for managing FPL One of FaVe's virtues is its ability to directly generate configuration from FPL policies that codify the security official's intent. Despite FPL's suitability for non-academically skilled users, it would be appealing if one could specify policies in natural language or, even more appealing, if ISM documents could be processed directly. Large Language Models (LLMs) with their abilities to process natural language are promising candidates for such supporting systems. The idea is to use FPL as a high-level intermediate representation. The workflow starts with an AI supported creation of FPL inventories and policies which are then used to generate security configurations directly.

A first step would be an AI assistant that helps security officials with their handling of FPL. We explored this idea in a Master's thesis [61] that yielded promising results. The approach was to specialize resp. fine-tune a coding-oriented LLM, i.e., CodeLlama [125], for the processing of FPL and to set up a chat system that enables security officials to create and manage FPL using natural language. In general, the study showed this approach's feasibility but the LLM's answer quality remained behind our expectations – most likely due to the low amount of training data which included only about 250 samples. Yet, the approach is promising and should be pursued further and with more training data in the future.

Bibliography

[1] Muhammad Abedin et al. “Detection and Resolution of Anomalies in Firewall Policy Rules”. In: *20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*. Sophia Antipolis, France, 2006, pp. 15–29. DOI: 10.1007/11805588_2.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. “Tiramisu: Fast Multilayer Network Verification”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Santa Clara, CA, USA, 2020, pp. 201–220. URL: http://www.usenix.org/conference/nsdi20/presentation/abhashkumar.

[3] Joe Abley, Pekka Savola, and George Neville-Neil. *Deprecation of Type 0 Routing Headers in IPv6*. RFC 5095. IETF, Dec. 2007.

[4] Ehab Al-Shaer and Saeed Al-Haj. “FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures”. In: *3rd ACM Workshop on Assurable and Usable Security Configuration (SafeConfig)*. Chicago, IL, USA, 2010, pp. 37–44. DOI: 10.1145/1866898.1866905.

[5] Ehab Al-Shaer et al. “Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security”. In: *17th IEEE International Conference on Network Protocols (ICNP)*. Plainsboro, NJ, USA, 2009, pp. 123–132. DOI: 10.1109/ICNP.2009.5339690.

[6] Xiaoliang Wu et al. *VeriFlow - Real-Time Network Property Verifier*. Last access on November 8th, 2024. URL: https://github.com/xwu64/Network-Monitor-Middleware/tree/master.

[7] *Allgemeine Verwaltungsvorschrift zum materiellen Geheimschutz (Verschlusssachenanweisung – VSA)*. German Law. 2018. URL: https://www.verwaltungsvorschriften-im-internet.de/bsvwvbund_13032023_SII554001405.htm.

[8] *Ansible*. Last access on April 23rd, 2025. 2020. URL: https://docs.ansible.com/.

[9] *Assigned Internet Protocol Numbers*. Last access on May 1st, 2023. Internet Assigned Numbers Authority (IANA). URL: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[10] Congxiao Bao et al. *IPv6 Addressing of IPv4/IPv6 Translators*. RFC 6052. IETF, Oct. 2010.

[11] Nayan Basumatary and Shyamanta M. Hazarika. “Model checking a firewall for anomalies”. In: *1st International Conference on Emerging Trends and Applications in Computer Science (ICETACS)*. Shillong, India, 2013, pp. 92–96. DOI: 10.1109/ICETACS.2013.6691402.

[12] Ryan Beckett et al. “A General Approach to Network Configuration Verification”. In: *Conference of the ACM Special Interest Group on Data Communication, (SIGCOMM)*. Los Angeles, CA, USA, 2017, pp. 155–168. DOI: 10.1145/3098822.3098834.

[13] David Elliott Bell and Leonard J. LaPadula. “Secure Computer Systems: A Mathematical Model, Volume II”. In: *Journal on Computer Security* 4.2/3 (1996). The original techreport got lost. This is a reconstruction by Lenoard LaPadula., pp. 229–263.

[14] Nikolaj Bjørner and Karthick Jayaraman. “Checking Cloud Contracts in Microsoft Azure”. In: *11th International Conference on Distributed Computing and Internet Technology (ICDCIT)*. Vol. 8956. Bhubaneswar, India, 2015, pp. 21–32. DOI: 10.1007/978-3-319-14977-6_2.

[15] Chiara Bodei et al. “FWS: Analyzing, maintaining and transcompiling firewalls”. In: *Journal of Computer Security* 29.1 (2021), pp. 77–134. DOI: 10.3233/JCS-200017.

[16] Pat Bosshart et al. “P4: Programming Protocol-independent Packet Processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95. DOI: 10.1145/2656877.2656890.

[17] Daniele Bringhenti, Lucia Seno, and Fulvio Valenza. “An Optimized Approach for Assisted Firewall Anomaly Resolution”. In: *IEEE Access* 11 (2023), pp. 119693–119710. DOI: 10.1109/ACCESS.2023.3328194.

[18] Yeim-Kuan Chang. “A 2-Level TCAM Architecture for Ranges”. In: *IEEE Transactions on Computers (ToC)* 55.12 (2006), pp. 1614–1629. DOI: 10.1109/TC.2006.189.

[19] Rupali Chaure and Shishir K. Shandilya. “Firewall anamolies detection and removal techniques – a survey”. In: *International Journal on Emerging Technologies* 1.1 (2010), pp. 71–74.

[20] Ankur Chowdhary et al. “Intent-Driven Security Policy Management for Software-Defined Systems”. In: *IEEE Transactions on Network and Service Management (TNSM)* 19.4 (2022), pp. 5208–5223. DOI: 10.1109/TNSM.2022.3183591.

[21] Cisco. *SD-Access Segmentation Design Guide*. Last Access: March 20, 2025. 2018. URL: https://community.cisco.com/t5/networking-knowledge-base/sd-access-segmentation-design-guide/ta-p/4935734.

[22] Cisco. *User Guide for Cisco Security Manager 4.24*. Last access on April 23rd, 2025. 2021. URL: https://www.cisco.com/c/en/us/td/docs/security/security_management/cisco_security_manager/security_manager/424/User/csm-user-guide-424/chapter16-managing-firewall-access-rules.html.

[23] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logics of Programs*. Yorktown Heights, NY, USA, 1981, pp. 52–71. DOI: 10.1007/BFb0025774.

[24] Vera Clemens. “Sprachen zur Spezifikation von Netzwerksicherheitspolicies”. Bachelor’s Thesis. University of Potsdam, Germany, 2018.

[25] Alexander Clemm et al. *Intent-Based Networking – Concepts and Definitions*. RFC 9315. IETF, Oct. 2022.

[26] *Common Criteria for Information Technology Security Evaluation – Part 2: Security functional components*. CCRA Standard. Common Criteria Recognition Agreement (CCRA), Sept. 2012. URL: https://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R4.pdf.

[27] Matthew Condell, Charles Lynn, and John Zao. *Security Policy Specification Language*. Last access on March 7th, 2025. URL: https://datatracker.ietf.org/doc/html/draft-ietf-ipsec-spsl-01.

[28] Frédéric Cuppens, Nora Cuppens-Boulahia, and Joaquín García-Alfaro. "Detection of network security component misconfiguration by rewriting and correlation". In: *5th Conference on Security and Network Architectures (SAR)*. Seignose, France, 2006.

[29] Frédéric Cuppens et al. "A Formal Approach to Specify and Deploy a Network Security Policy". In: *2nd IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST)*. Vol. 173. Toulouse, France, 2004, pp. 203–218. DOI: 10.1007/0-387-24098-5_15.

[30] Nicodemos Damianou et al. "The Ponder Policy Specification Language". In: *International Workshop on Policies for Distributed Systems and Networks (POLICY)*. Bristol, UK, 2001, pp. 18–38.

[31] Elwyn B. Davies and Janos Mohacsi. *Recommendations for Filtering ICMPv6 Messages in Firewalls*. RFC 4890. IETF, May 2007.

[32] Ginger Davis, Alfredo Garcia, and Weide Zhang. "Empirical Analysis of the Effects of Cyber Security Incidents". In: *Risk Analysis* 29.9 (2009), pp. 1304–1316. DOI: 10.1111/j.1539-6924.2009.01245.x.

[33] Stephen E. Deering and Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. IETF, July 2017.

[34] Klaus Denecke. *Algebra und Diskrete Mathematik für Informatiker*. 2003. DOI: 10.1007/978-3-322-80109-8.

[35] *Departement of Defense Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD. US Department of Defense (DoD), 1985.

[36] Cornelius Diekmann. *net-network*. Last access on April 23rd, 2025. 2016. URL: https://github.com/diekmann/net-network.

[37] Cornelius Diekmann et al. "Verified iptables firewall analysis". In: *2016 IFIP Networking Conference and Workshops (IFIP Networking)*. Vienna, Austria, 2016, pp. 252–260. DOI: 10.1109/IFIPNetworking.2016.7497196.

[38] Katharina Dietz et al. "Moving Down the Stack: Performance Evaluation of Packet Processing Technologies for Stateful Firewalls". In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. Miami, FL, USA, 2023, pp. 1–7. DOI: 10.1109/NOMS56928.2023.10154224.

[39] Dragos Dumitrescu et al. "Dataplane Equivalence and Its Applications". In: *16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Boston, MA, USA, 2019, pp. 683–697. URL: https://www.usenix.org/conference/nsdi19/presentation/dumitrescu.

[40] Raphael Durner et al. “Detecting and mitigating denial of service attacks against the data plane in software defined networks”. In: *IEEE Conference on Network Softwarization (NetSoft)*. Bologna, Italy, 2017, pp. 1–6. DOI: 10.1109/NETSOFT.2017.8004229.

[41] *eXtensible Access Control Markup Language (XACML)*. OASIS Standard. Last access on May 5th, 2023. OASIS, 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[42] *Federal Information Security Management Act (FISMA)*. US Law. 2002.

[43] Niels Ferguson and Bruce Schneier. *A Cryptographic Evaluation of IPsec*. Tech. rep. Counterpane Internet Security, Inc., 2003.

[44] Andreas Fiessler et al. “FireFlow - High Performance Hybrid SDN-Firewalls with OpenFlow”. In: *43rd IEEE Conference on Local Computer Networks (LCN)*. Chicago, IL, USA, 2018, pp. 267–270. DOI: 10.1109/LCN.2018.8638090.

[45] Andreas Fiessler et al. “HyPaFilter+: Enhanced Hybrid Packet Filtering Using Hardware Assisted Classification and Header Space Analysis”. In: *IEEE/ACM Transactions on Networking (ToN)* 25.6 (2017), pp. 3655–3669. DOI: 10.1109/TNET.2017.2749699.

[46] Ari Fogel et al. “A General Approach to Network Configuration Analysis”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, CA, USA, 2015, pp. 469–483. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel.

[47] Hervé Gallaire and Jack Minker, eds. *Symposium on Logic and Data Bases*. Toulouse, France, 1978. DOI: 10.1007/978-1-4684-3384-5.

[48] Joaquín García-Alfaro, Nora Cuppens-Boulahia, and Frédéric Cuppens. “Complete analysis of configuration rules to guarantee reliable network security policies”. In: *Internation Journal of Information Security* 7 (2008), pp. 103–122. DOI: 10.1007/s10207-007-0045-7.

[49] Steffen Gebert et al. “Demonstrating a Personalized Secure-by-Default Bring Your Own Device Solution Based on Software Defined Networking”. In: *28th International Teletraffic Congress (ITC)*. Würzburg, Germany, 2016, pp. 197–200. DOI: 10.1109/ITC-28.2016.133.

[50] *Gesetz über das Bundesamt für Sicherheit in der Informationstechnik (BSI-Gesetz – BSIG)*. German Law. 2009. URL: https://www.gesetze-im-internet.de/bsig_2009/BJNR282110009.html.

[51] *Gesetz zur Erhöhung der Sicherheit informationstechnischer Systeme (IT-Sicherheitsgesetz)*. German Law. 2015.

[52] Fernando Gont et al. *Operational Implications of IPv6 Packets with Extension Headers*. RFC 9098. IETF, Sept. 2021.

[53] Fernando Gont et al. *Temporary Address Extensions for Stateless Address Autoconfiguration in IPv6*. RFC 8981. IETF, Feb. 2021.

[54] Nicholas Gray et al. “A priori state synchronization for fast failover of stateful firewall VNFs”. In: *International Conference on Networked Systems (NetSys)*. Göttingen, Germany, 2017, pp. 1–6. DOI: 10.1109/NETSYS.2017.7903964.

[55] *Grundschutz NET.1.1: Netzarchitektur und -design*. BSI Standard. Bundesamt für Sicherheit in der Informationstechnologie (BSI), Feb. 2021.

[56] Sven Hager. “System-Specialized and Hybrid Approaches to Network Packet Classification”. PhD Thesis. Humboldt-Universität zu Berlin, 2020. DOI: 10.18452/21780.

[57] Dhwani Hakani and Palvinder Singh Mann. “Intra Firewall Anomaly Policies Detection in Cloud Environment Using Firewall Tree”. In: *Transactions of the Indian National Academy of Engineering* 10 (2024), pp. 63–72. DOI: 10.1007/s41403-024-00504-4.

[58] Ameya Hanamsagar et al. “Detection of Firewall Policy Anomalies in Real-time Distributed Network Security Appliances”. In: *International Journal of Computer Applications* 116 (2015), pp. 7–13. DOI: 10.5120/20497-2769.

[59] Timothy L. Hinrichs et al. “Practical Declarative Network Management”. In: *1st ACM Workshop on Research on Enterprise Networking (WREN)*. Barcelona, Spain, 2009, pp. 1–10. DOI: 10.1145/1592681.1592683.

[60] Alex Horn, Ali Kheradmand, and Mukul Prasad. “Delta-net: Real-time Network Verification Using Atoms”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, USA, 2017, pp. 735–749. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex.

[61] Ferdinand Hoske. “Serverless Architectures for Large Language Model Fine-Tuning and Inference”. Master’s Thesis. University of Potsdam, Germany, 2024.

[62] Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. “Detecting and Resolving Firewall Policy Anomalies”. In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* 9.3 (2012), pp. 318–331. DOI: 10.1109/TDSC.2012.20.

[63] Vincent C. Hu et al. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. Special Publication SP800-162. US National Institute of Standards and Technology (NIST), 2014.

[64] Matteo Iaiani et al. “Analysis of Cybersecurity-related Incidents in the Process Industry”. In: *Reliability Engineering & System Safety* 209 (2021), pp. 1–20. DOI: 10.1016/j.ress.2021.107485.

[65] IBM. *Discretionary access control*. Last access on May 17th, 2023. URL: https://www.ibm.com/docs/en/aix/7.2?topic=security-discretionary-access-control.

[66] IDSv6-Project. *Exemplary ip6tables init script*. Last access on April 23rd, 2025. 2013. URL: http://www.idsv6.de/Downloads/iptables_ruleset.sh.

[67] *IEC 62443 – Industrial Communication Networks – Network and System Security*. IEC Standard. International Electrotechnical Commission (IEC), July 2009.

[68] *ISO/IEC 27001:2013 – Information Technology, Security Techniques, Information Security Management Systems, Requirements*. ISO Standard. International Organization for Standardization (ISO), Oct. 2013.

[69] *IT-Grundschutz*. BSI Standard. Bundesamt für Sicherheit in der Informationstechnologie (BSI). URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/IT-Grundschutz/it-grundschutz_node.html.

[70] *IT-Grundschutz-Kataloge*. BSI Standard. Bundesamt für Sicherheit in der Informationstechnik (BSI), Jan. 2016. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Downloadserver/IT-Grundschutz-Kataloge/IT-Grundschutz-Kataloge-15-EL.html.

[71] Rishi Iyengar. *Massive SolarWinds hack has big businesses on high alert*. Last access on April 23rd, 2025. 2021. URL: https://edition.cnn.com/2020/12/19/tech/solarwinds-hack-companies/index.html.

[72] Karthick Jayaraman et al. "Validating Datacenters at Scale". In: *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Beijing, China, 2019, pp. 200–213. DOI: 10.1145/3341302.3342094.

[73] Alan Jeffrey and Taghrid Samak. "Model Checking Firewall Policy Configurations". In: *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*. London, UK, 2009, pp. 60–67. DOI: 10.1109/POLICY.2009.32.

[74] Xin Jin, Ram Krishnan, and Ravi S. Sandhu. "A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC". In: *26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec)*. Vol. 7371. Paris, France, 2012, pp. 41–55. DOI: 10.1007/978-3-642-31540-4_4.

[75] Anas Abou El Kalam et al. "Organization based Access Control". In: *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*. Lake Como, Italy, 2003, pp. 120–131. DOI: 10.1109/POLICY.2003.1206966.

[76] Peyman Kazemian. *Hassel*. Last access on April 22th, 2025. 2012. URL: https://bitbucket.org/peymank/hassel-public/wiki/Home.

[77] Peyman Kazemian. "Header Space Analysis". PhD Thesis. Stanford University, 2013.

[78] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Jose, CA, USA, 2012, pp. 113–126. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[79] Peyman Kazemian et al. "Real Time Network Policy Checking Using Header Space Analysis". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, USA, 2013, pp. 99–111. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian.

[80] Bassam Khorchani, Sylvain Hallé, and Roger Villemaire. “Firewall anomaly detection with a model checker for visibility logic”. In: *IEEE Network Operations and Management Symposium (NOMS)*. Maui, HI, USA, 2012, pp. 466–469. DOI: 10.1109/NOMS.2012.6211932.

[81] Ahmed Khurshid et al. “VeriFlow: Verifying Network-Wide Invariants in Real Time”. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, USA, 2013, pp. 15–27. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid.

[82] Christian Klein. “Verifikation von Netzsicherheitseigenschaften mit AP-Verifier”. Master’s Thesis. University of Potsdam, Germany, 2020.

[83] Igor Kotenko and Olga Polubelova. “Verification of security policy filtering rules by Model Checking”. In: *IEEE 6th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Prague, Czech Republic, 2011, pp. 706–710. DOI: 10.1109/IDAACS.2011.6072862.

[84] Wadie Krombi, Mohammed Erradi, and Ahmed Khoumsi. “Automata-based approach to design and analyze security policies”. In: *12th Annual International Conference on Privacy, Security and Trust (PST)*. Toronto, ON, Canada, 2014, pp. 306–313. DOI: 10.1109/PST.2014.6890953.

[85] Marshall A. Kuypers, Thomas Maillart, and Elisabeth Paté-Cornell. *An Empirical Analysis of Cyber Security Incidents at a Large Organization*. Tech. rep. Center for International Security and Cooperation, Stanford University, 2016. URL: https://fsi-live.s3.us-west-1.amazonaws.com/s3fs-public/kuypersweis_v7.pdf.

[86] Hyunjung Lee et al. “HSViz: Hierarchy Simplified Visualizations for Firewall Policy Analysis”. In: *IEEE Access* 9 (2021), pp. 71737–71753. DOI: 10.1109/ACCESS.2021.3077146.

[87] Yuchong Li and Qinghui Liu. “A comprehensive review study of cyber-attacks and cyber security; Emerging trends and recent developments”. In: *Energy Reports* 7 (2021), pp. 8176–8186. DOI: 10.1016/j.egyr.2021.08.126.

[88] Zhiming Lin and Zhiqiang Yao. “Firewall Anomaly Detection Based on Double Decision Tree”. In: *Symmetry* 14.12 (2022). DOI: 10.3390/sym14122668.

[89] Alex X. Liu, Eric Torng, and Chad R. Meiners. “Firewall Compressor: An Algorithm for Minimizing Firewall Policies”. In: *The 27th Conference on Computer Communications (INFOCOM)*. Phoenix, AZ, USA, 2008, pp. 176–180. DOI: 10.1109/INFOCOM.2008.44.

[90] Nuno P. Lopes et al. “Checking Beliefs in Dynamic Networks”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, CA, USA, 2015, pp. 499–512. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes.

[91] Claas Lorenz. “Anomaly Detection for Distributed IPv6 Firewalls”. Master’s Thesis. University of Potsdam, Germany, 2014.

[92] Claas Lorenz, Sebastian Kiekheben, and Bettina Schnor. “FaVe: Modeling IPv6 Firewalls for Fast Formal Verification”. In: *International Conference on Networked Systems (NetSys)*. Göttingen, Germany, 2017, pp. 1–8. DOI: 10.1109/NETSYS.2017.7903956.

[93] Claas Lorenz and Bettina Schnor. “Firewall Management: Rapid Anomaly Detection”. In: *IEEE 24th International Conference on High Performance Computing & Communications (HPCC)*. Chengdu, China, 2022, pp. 1465–1472. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00226.

[94] Claas Lorenz and Bettina Schnor. “Policy Anomaly Detection for Distributed IPv6 Firewalls”. In: *12th International Conference on Security and Cryptography (SECRYPT)*. Colmar, France, 2015, pp. 210–219. DOI: 10.5220/0005517402100219.

[95] Claas Lorenz et al. “An SDN/NFV-Enabled Enterprise Network Architecture Offering Fine-Grained Security Policy Enforcement”. In: *IEEE Communications Magazine* 55.3 (2017), pp. 217–223. DOI: 10.1109/MCOM.2017.1600414CM.

[96] Claas Lorenz et al. “Continuous Verification of Network Security Compliance”. In: *IEEE Transactions on Network and Service Management (TNSM)* 19.2 (2022), pp. 1729–1745. DOI: 10.1109/TNSM.2021.3130290.

[97] Mandavilli Madhuri and Knvssk Rajesh. “Systematic Detection and Resolution of Firewall Policy Anomalies”. In: *International Journal of Research in Computer and Communication Technology (IJRCCT)* 2.12 (2013), pp. 1387–1392.

[98] *Managementsysteme für Informationssicherheit (ISMS)*. BSI Standard. Bundesamt für Sicherheit in der Informationstechnik (BSI), Oct. 2017. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/IT-Grundschutz/BSI-Standards/BSI-Standard-200-1-Managementsysteme-fuer-Informationssicherheit/bsi-standard-200-1-managementsysteme-fuer-informationssicherheit_node.html.

[99] Rahim Masoudi and Ali Ghaffari. “Software Defined Networks: A survey”. In: *Journal of Network and Computer Applications* 67 (2016), pp. 1–25. DOI: 10.1016/j.jnca.2016.03.016.

[100] Jirí Matousek et al. “ClassBench-ng: Recasting ClassBench after a Decade of Network Evolution”. In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. Beijing, China, 2017, pp. 204–216. DOI: 10.1109/ANCS.2017.33.

[101] Per Håkon Meland et al. “A Retrospective Analysis of Maritime Cyber Security Incidents”. In: *International Journal on Marine Navigation and Safety of Sea Transportation (TransNav)* 15 (2021), pp. 519–530. DOI: 10.12716/1001.15.03.04.

[102] Microsoft. *Overview of role-based access control in Azure Active Directory*. Last access on May 17th, 2023. URL: https://learn.microsoft.com/en-us/azure/active-directory/roles/custom-overview.

[103] Soo-Jin Moon et al. “Alembic: Automated Model Inference for Stateful Network Functions”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, USA, 2019, pp. 699–718. URL: https://www.usenix.org/conference/nsdi19/presentation/moon.

[104] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 4963. Budapest, Hungary, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[105] Hannah Murphy and David Sheppard. *Saudi Aramco confirms data leak after $50 million cyber ransom demand*. Last access on April 23rd, 2025. 2021. URL: https://arstechnica.com/information-technology/2021/07/saudi-aramco-confirms-data-leak-after-50-million-cyber-ransom-demand/.

[106] NetCitadel. *FirewallBuilder*. Last access on April 23rd, 2025. 2012. URL: https://github.com/fwbuilder/fwbuilder.

[107] Netfilter. *Manual page for iptables*. Last access on April 23rd, 2025. 2019. URL: http://ipset.netfilter.org/iptables.man.html.

[108] Netfilter. *Manual page for iptables-extensions*. Last access on April 23rd, 2025. 2019. URL: http://ipset.netfilter.org/iptables-extensions.man.html.

[109] *Next Generation Access Control – Functional Architecture (NGAC-FA)*. ANSI INCITS Standard 499-2018. American National Standard for Information Technology (ANSI), Jan. 2018.

[110] Matunda Nyanchama and Sylvia L. Osborn. "Modeling Mandatory Access Control in Role-Based Security Systems". In: *9th Annual IFIP WG11 Working Conference on Database Security (DBSec)*. Vol. 51. Rensselaerville, New York, USA, 1995, pp. 129–144.

[111] Ricardo M. Oliveira, Sihyung Lee, and Hyong S. Kim. "Automatic detection of firewall misconfigurations using firewall and network routing policies". In: *IEEE/IFIP Workshop on Proactive Failure Avoidance, Recovery, and Maintenance (PFARM)*. Lisbon, Portugal, 2009.

[112] Sylvia L. Osborn, Ravi S. Sandhu, and Qamar Munawer. "Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies". In: *ACM Transactions on Information and System Security (TISSEC)* 3.2 (2000), pp. 85–106. DOI: 10.1145/354876.354878.

[113] Aurojit Panda et al. "Verifying Reachability in Networks with Mutable Datapaths". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, USA, 2017, pp. 699–718. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths.

[114] Benedikt Pfaff et al. "SDN/NFV-enabled Security Architecture for Fine-grained Policy Enforcement and Threat Mitigation for Enterprise Networks". In: *Posters and Demos of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Los Angeles, CA, USA, 2017, pp. 15–16. DOI: 10.1145/3123878.3131970.

[115] Benjamin Plewka. "Automatische Umsetzung von Sicherheitspolicies in Rechnernetzen". Master's Thesis. University of Potsdam, Germany, 2021.

[116] Gentoo Linux Project. *SELinux/Role-based access control*. Last access on May 17th, 2023. URL: https://wiki.gentoo.org/wiki/SELinux/Role-based_access_control.

[117] *Puppet*. Last access on April 23rd, 2025. 2020. URL: https://puppet.com/docs/.

[118] Qasim Mahmood Rajpoot, Christian Damsgaard Jensen, and Ram Krishnan. "Attributes Enhanced Role-Based Access Control Model". In: *12th International Conference on Trust, Privacy and Security in Digital Business (TrustBus)*. Vol. 9264. Valencia, Spain, 2015, pp. 3–17. DOI: 10.1007/978-3-319-22906-5_1.

[119] Dinesha Ranathunga et al. *ForestFirewalls: Getting Firewall Configuration Right in Critical Networks*. Tech. rep. University of Adelaine, 2018.

[120] Amir Rawdat. *Testing the Performance of NGINX and NGINX Plus Web Servers*. Last access on April 26th, 2024. URL: https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/.

[121] RedHat. *SELinux and Mandatory Access Control (MAC)*. Last access on May 17th, 2023. URL: https://access.redhat.com/documentation/de-de/red_hat_enterprise_linux/7/html/virtualization_security_guide/sect-virtualization_security_guide-svirt-mac.

[122] Yakov Rekhter et al. *Address Allocation for Private Internets*. RFC 1918. IETF, Feb. 1996.

[123] *Role Based Access Control*. ANSI INCITS Standard 359–2004. American National Standard for Information Technology (ANSI), 2004.

[124] Scott Rose et al. *Zero Trust Architecture*. Special Publication SP800-207. US National Institute of Standards and Technology (NIST), 2020. DOI: 10.6028/NIST.SP.800-207.

[125] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. Tech. rep. Meta Inc., 2024. DOI: 10.48550/arXiv.2308.12950.

[126] Amina Saâdaoui, Nihel Ben Youssef Ben Souayeh, and Adel Bouhoula. "Formal approach for managing firewall misconfigurations". In: *IEEE 8th International Conference on Research Challenges in Information Science (RCIS)*. Marrakech, Morocco, 2014, pp. 1–10. DOI: 10.1109/RCIS.2014.6861044.

[127] Amina Saâdaoui, Nihel Ben Youssef Ben Souayeh, and Adel Bouhoula. "FARE: FDD-based Firewall Anomalies Resolution Tool". In: *Journal of Computational Science* 23 (2017), pp. 181–191. DOI: 10.1016/j.jocs.2017.09.003.

[128] Ravi S. Sandhu, Venkata Bhamidipati, and Qamar Munawer. "The ARBAC97 Model for Role-Based Administration of Roles". In: *ACM Transactions on Information and System Security (TISSEC)* 2.1 (1999), pp. 105–135. DOI: 10.1145/300830.300839.

[129] Ravi S. Sandhu and Qamar Munawer. "How to Do Discretionary Access Control Using Roles". In: *3rd ACM Workshop on Role-Based Access Control (RBAC)*. Fairfax, VA, USA, 1998, pp. 47–54. DOI: 10.1145/286884.286893.

[130] Alexander W. Schneider et al. "Empirical Results for Application Landscape Complexity". In: *48th IEEE Hawaii International Conference on System Sciences, (HICSS)*. Kauai, HI, USA, 2015, pp. 4079–4088. DOI: 10.1109/HICSS.2015.490.

[131] *Security and Privacy Controls for Information Systems and Organizations*. Special Publication SP800-53. US National Institute of Standards and Technology (NIST), Sept. 2020.

[132] Ehab Al-Shaer et al. "Conflict Classification and Analysis of Distributed Firewall Policies". In: *IEEE Journal on Selected Areas in Communications (JSAC)* 23.10 (2005), pp. 2069–2084. DOI: 10.1109/JSAC.2005.854119.

[133] Ehab S. Al-Shaer and Hazem H. Hamed. "Discovery of Policy Anomalies in Distributed Firewalls". In: *IEEE Conference on Computer Communications (INFOCOM)*. Vol. 4. Hong Kong, China, 2004, pp. 2605–2616. DOI: 10.1109/INFCOM.2004.1354680.

[134] Morris Sloman. "Policy Driven Management for Distributed Systems". In: *Journal of Network and Systems Management* 2.4 (1994), pp. 333–360. DOI: 10.1007/BF02283186.

[135] Daniel Sokolov. *Potsdam offline: Stadt spricht von Gefahr durch Hive-Bande*. Last access on April 23rd, 2025. 2021. URL: https://www.heise.de/news/Potsdam-offline-Stadt-spricht-von-Gefahr-durch-Hive-Bande-7477366.html.

[136] Sooel Son et al. "Model Checking Invariant Security Properties in OpenFlow". In: *IEEE International Conference on Communications (ICC)*. Budapest, Hungary, 2013, pp. 1974–1979. DOI: 10.1109/ICC.2013.6654813.

[137] Pyda Srisuresh and Matt Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663. IETF, Aug. 1999.

[138] Radu Stoenescu et al. "SymNet: Scalable symbolic execution for modern networks". In: *Conference of the ACM Special Interest Group on Data Communication, (SIGCOMM)*. Florianopolis, Brazil, 2016, pp. 314–327. DOI: 10.1145/2934872.2934881.

[139] Eva Stoica. "The Complexity of an IT Landscape". Master's Thesis. University of Twente, Netherlands, 2024.

[140] Alan Tang et al. "Lightyear: Using Modularity to Scale BGP Control Plane Verification". In: *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. New York, NY, USA: ACM, 2023, pp. 94–107. DOI: 10.1145/3603269.3604842.

[141] *The 2011 Standard of Good Practice for Information Security*. ISF Standard. Information Security Forum (ISF), June 2011.

[142] Marco Thomas, Claas Lorenz, and Alf Zugenmaier. "Optimal Deployment of High-Level Access Control Policies in Heterogeneous Enforcement Infrastructures". In: *29th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. Osaka, Japan, 2024, pp. 173–179. DOI: 10.1109/PRDC63035.2024.00029.

[143] Jeff Tully et al. "Healthcare Challenges in the Era of Cybersecurity". In: *Health Security* 18.3 (2020), pp. 228–231. DOI: 10.1089/hs.2019.0123.

[144] Artem Voronkov, Leonardo A. Martucci, and Stefan Lindskog. "Measuring the Usability of Firewall Rule Sets". In: *IEEE Access* 8 (2020), pp. 27106–27121. DOI: 10.1109/ACCESS.2020.2971093.

[145] Margaret Wasserman and Fred Baker. *IPv6-to-IPv6 Network Prefix Translation*. RFC 6296. IETF, June 2011.

[146] Frank Wolter and Michael Zakharyaschev. "Qualitative spatiotemporal representation and reasoning: a computational perspective". In: *Exploring Artificial Intelligence in the New Millennium*. 2003, pp. 175–215. ISBN: 15-5860-811-7.

[147] Hongkun Yang and Simon S. Lam. "Real-Time Verification of Network Properties Using Atomic Predicates". In: *IEEE/ACM Transactions on Networking (ToN)* 24.2 (2016), pp. 887–900. DOI: 10.1109/TNET.2015.2398197.

[148] Yi Yin et al. "An Analysis Method for IPv6 Firewall Policy". In: *IEEE 21st International Conference on High Performance Computing and Communications (HPCC)*. Zhangjiajie, China, 2019, pp. 1757–1762. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00241.

[149] Yi Yin et al. "An Inconsistency Detection Method for Security Policy and Firewall Policy Based on CSP Solver". In: *3rd International Conference on Cloud Computing and Security (ICCCS)*. Nanjing, China, 2017, pp. 147–161. DOI: 10.1007/978-3-319-68542-7_13.

[150] Yi Yin et al. "Consistency Decision Between IPv6 Firewall Policy and Security Policy". In: *4th International Conference on Information Communication and Signal Processing (ICICSP)*. Shanghai, China, 2021, pp. 577–581. DOI: 10.1109/ICICSP54369.2021.9611983.

[151] Farnaz Yousefi et al. "Liveness Verification of Stateful Network Functions". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Santa Clara, CA, USA, 2020, pp. 257–272. URL: https://www.usenix.org/conference/nsdi20/presentation/yousefi.

[152] Chaode Yu et al. *A Base YANG Data Model for Network Inventory*. Last access on January 2nd, 2025. URL: https://www.ietf.org/archive/id/draft-ietf-ivy-network-inventory-yang-04.html.

[153] Lihua Yuan et al. "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis". In: *IEEE Symposium on Security and Privacy (S&P)*. Berkeley/Oakland, CA, USA, 2006, pp. 199–213. DOI: 10.1109/SP.2006.16.

[154] Yifei Yuan et al. "NetSMC: A Custom Symbolic Model Checker for Stateful Network Verification". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Santa Clara, CA, USA, 2020, pp. 181–200. URL: https://www.usenix.org/conference/nsdi20/presentation/yuan.

[155] Hongyi Zeng et al. "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Seattle, WA, USA, 2014, pp. 87–99. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zeng.

[156] Peng Zhang et al. "APKeep: Realtime Verification for Real Networks". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Santa Clara, CA, USA, 2020, pp. 241–255. URL: https://www.usenix.org/conference/nsdi20/presentation/zhang-peng.

Appendix A

A.1 Access Control Frameworks

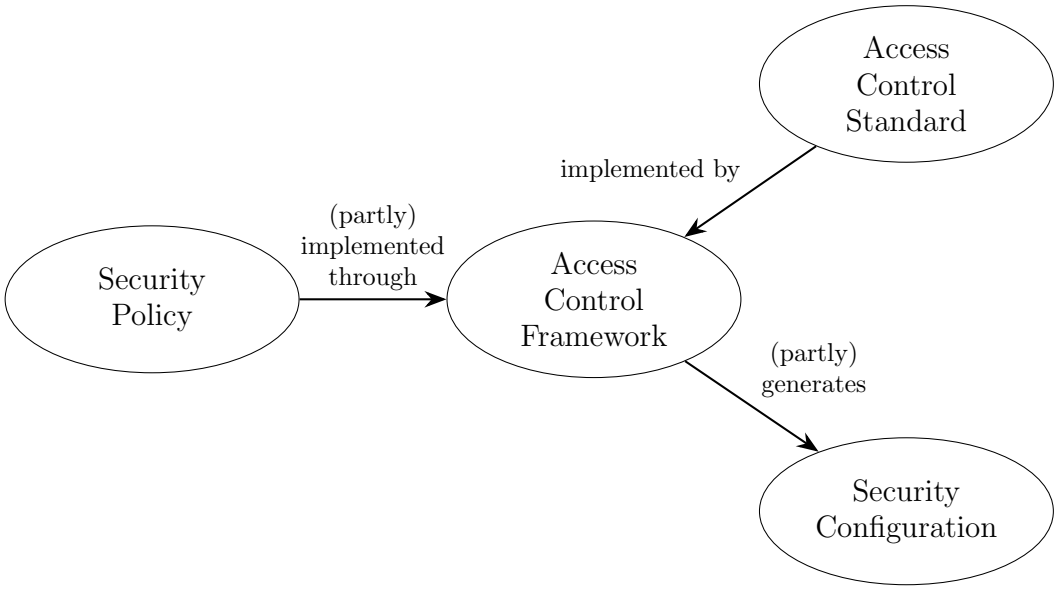


Fig. A.1.: Relations between *access control standard*, *access control frameworks*, security policies, and security configurations.

Access control frameworks have been proposed for a long time, e.g., the *Bell-LaPadula* formalism which forms the basis for *Mandatory Access Control* (MAC [35]) was publishd in 1973 [13]. In general, access control determines under which conditions some subject may interact with some object. E.g., if some user may write to some file. As shown in Figure A.1, these frameworks aim to fill the gap between security policies, security specification, and security configuration by providing:

- a (generic) formal security model to specify access policies,
- means to configure the system to be secured, and
- some workflows that enable the system's users to interact with the system.

Depending on its scope and expressiveness, the access control framework can be used to realize parts or all of an organization's security policies. I.e., its features cover all requirements stated in the organizational security policy. Consequently, the

security specification may be given partly or comprehensively by the framework's formalism and the security configuration may be generated partly or in full. E.g., *Mandatory Access Control* provides a simple mechanism to determine if some user may read or write some file by comparing the user's clearance level with the file's confidentiality level [13].

Governing IT security through an access control framework yields some benefits. Its formal model gives some degree of certainty that the security policy is specified soundly and that the automatically generated part of the security configuration is correct and effective. Hence, it is beneficiary for organizations that implement an ISM to introduce some access control framework, too.

In the following, we give a short overview of the most common families of access control frameworks – namely, *Mandatory Access Control* (MAC), *Discretionary Access Control* (DAC [35]), *Role Based Access Control* (RBAC [123]), and *Attribute Based Access Control* (ABAC [63]). In addition, we provide a detailed overview of NIST RBAC.

Discretionary Access Control (DAC) models base their access decisions on the identity of the subject. Subjects administer the access rights, e.g., reading or writing data, for the object that they create, e.g. files. Ownership may be passed as well but this leads to the loss of the creator's administrative privileges. Sometimes, owners can permit access by groups of subjects which simplifies administration. A prominent example for a DAC-based security system are the file permissions found in UNIX-like operating systems (cf. [65]).

Mandatory Access Control (MAC) models classify subjects as well as objects and base access decisions on generic rules that take the relationship between the classifications of subjects and object into consideration. For example, the US american Department of Defense uses a Bell-LaPadula model (cf. [13]) with confidentiality levels from `restricted` to `top secret` and general rules that apply access restrictions based on clearance levels. I.e., subjects are allowed to read classified data with a confidentiality level up to their own clearance level and they could write data to files of their own level or higher. Another example is the MAC implementation in SELinux [121].

Role Based Access Control (RBAC) decouples the subject-object relation by grouping users in a hierarchy of roles. Roles, in turn, grant access and modification

permissions for objects. RBAC has been standardized by NIST [123] and is broadly supported in real-world implementations, e.g., in Microsoft's Active Directory [102] or in SELinux [116]. There exist several extensions to the standardized RBAC which introduce well defined semantics to administrative roles (ARBAC97 [128]) or the notion of attributes (AERBAC [118]). The latter introduces attributes for users, objects, and the users' runtime context to match the abilities of ABAC. We give a detailed explanation of NIST RBAC in Appendix A.1.1.

Attribute Based Access Control (ABAC) The introduction of attributes to RBAC can be regarded as a reaction to the latest prominent access control model which is ABAC [63]. The general idea is to use attributes to freely characterize users, objects, context, and actions. In ABAC, access rules are defined as boolean predicates over attributes that are active within a context. Two well known examples are XACML [41] and NGAC [109].

Comparison

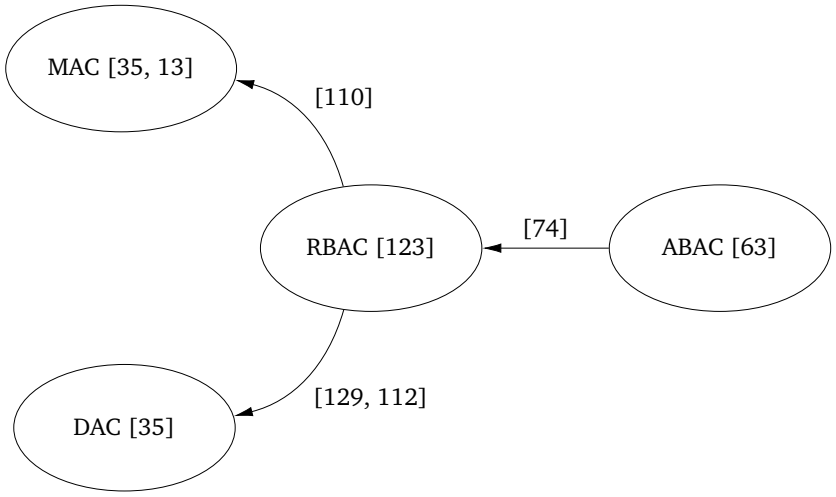


Fig. A.2.: Relations between common *access control frameworks*. The direction of the arrows symbolize the ability of one formal model to simulate the model pointed to, i.e., $A \rightarrow B$ means that A can simulate B.

Figure A.2 shows how the different access control frameworks relate to each other in terms of their ability to simulate one another. While MAC and DAC cannot simulate each other, i.e., there exist policies that can be expressed in one but not in the other, RBAC and ABAC can simulate both. In addition, ABAC is able to simulate RBAC, i.e, all RBAC policies can be expressed in ABAC as well, while the opposite is – to the best of our knowledge – not known as of today.

A.1.1 NIST RBAC

The US american NIST has standardized a flavor of Role Based Access Control (RBAC) for usage in government agencies as well as in the general public [63]. RBAC offers granularity suitable for network scenarios, e.g., including (sub) networks, hosts, or services, and the standardization paved the way for broad adoption in practice. In Appendix A.1.2, we will show full compatibility of policies expressed with our policy language FPL and RBAC systems that follow the NIST standard.

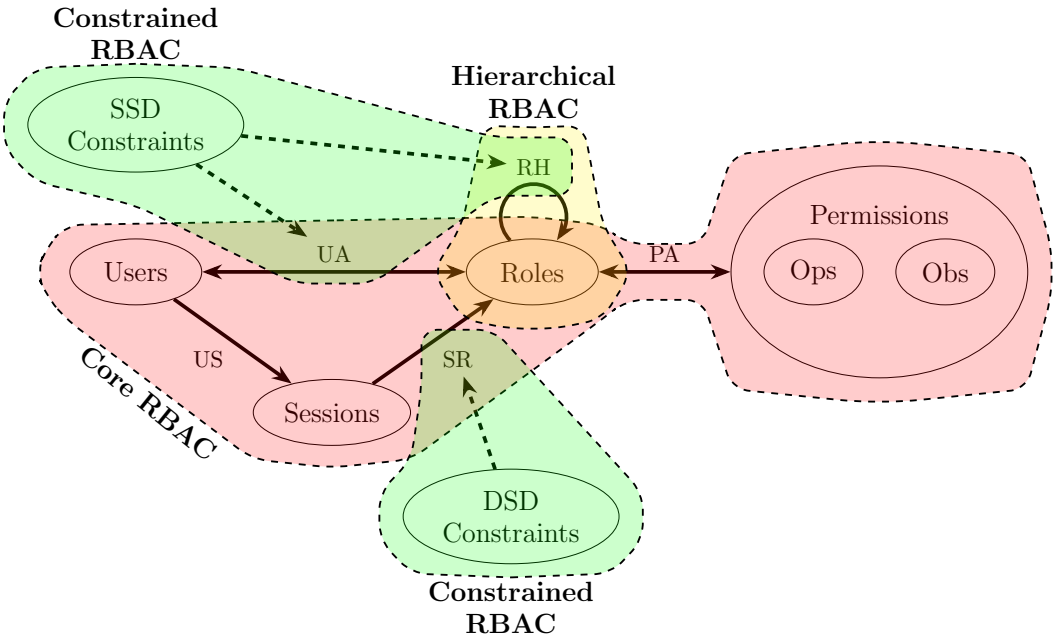


Fig. A.3.: Overview of NIST RBAC (after [63, pp. 9, 10]).

In general, NIST RBAC [123] formulates requirement profiles for RBAC systems, i.e., *Core*, *Hierarchical*, and *Constrained* RBAC. Figure A.3 gives an overview of the different sets, relations, and functions that are involved. Core RBAC consists of a set of *Users*, *Roles*, *Sessions*, and *Permissions* whereas the latter comprise of *Operations* (Ops) on *Objects* (Ops). In addition, there are relations to assign users to roles (UA) as well as to sessions (US), to activate roles in sessions (SR), and to grant permissions to roles (PA). Hierarchical RBAC introduces a *Role Hierarchy* (RH) relation that imposes restrictions on and allow inheritance for roles. Constrained RBAC introduces constraints for *Static Separation of Duties* (SSD) that restricts user assignments as well as the role hierarchy and *Dynamic Separation of Duties* (DSD) that restricts the role activation per session.

In the following, we use a simple organization to illustrate the different aspects of NIST RBAC. As depicted in Figure A.4, the organization consists of an *Office* network

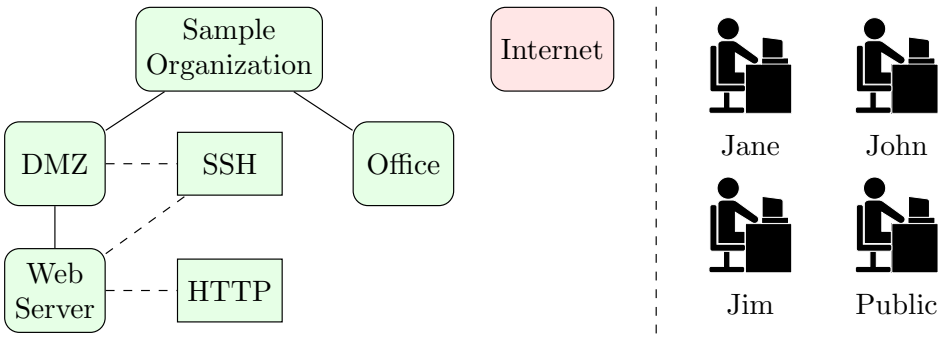


Fig. A.4.: Example organization and users.

and a *DMZ* that includes a *WebServer*. In addition, hosts in the DMZ offer *SSH* remote access and the WebServer runs an *HTTP* service. Also, the *Internet* is included as an external entity in order to simulate governance of interactions with the general public. In our example, the users are the administrator *Jane*, the procurator *John*, and the clerk *Jim*. In addition, the general public is represented by the user *Public*.

Core RBAC

RBAC's basic building blocks are sets for *Users*, *Objects*, *Roles*, *Operations*, and *Sessions*. These sets form *relations* and RBAC systems implement *functions* to make access decision. Concerning our example the sets are the following:

- *Users* = {John, Jane, Jim, Public}
- Objects *Obs* = {WebServer.HTTP, WebServer.SSH, DMZ.SSH}
- *Roles* = {SampleOrganization, DMZ, WebServer, Office, Internet}
- Operations *Ops* = {may access}
- *Sessions* = $\{s_{John}, s_{Jane}, s_{Jim}, s_{Public}\}$

The objects and operations form the *Permissions* relation – formally $Permissions : 2^{(Ops \times Obs)}$. The permissions are assigned to the roles via the *permission assignment* relation $PA \subseteq Permissions \times Roles$ and the roles, in turn, are assigned to users: $UA \subseteq Users \times Roles$. In our example, these relations are:

- $Permissions = \{$
  (may access, WebServer.HTTP),
  (may access, WebServer.SSH),
  (may access, DMZ.SSH)
  $\}$

- $PA = \{$
  ((may access, DMZ.SSH), Office),
  ((may access, WebServer.SSH), Office),
  ((may access, WebServer.HTTP), Office),
  ((may access, WebServer.HTTP), Internet)
  $\}$

- $UA = \{$
  (John, Office),
  (Jane, Office),
  (Jane, DMZ),
  (Jane, WebServer),
  (Jim, Office),
  (Public, Internet)
  $\}$

Now, the RBAC system is required to implement functions to determine the *authorized users* (AU) per role, formally:

$$AU(r : Roles) \rightarrow 2^{Users} = \{u \in Users \mid (u, r) \in UA\}$$

and the *authorized permissions* (AP) per role:

$$AP(r : Roles) \rightarrow 2^{Permissions} = \{p \in Permissions \mid (p, r) \in PA\}$$

For instance, the Internet role is assigned to the user Public, i.e., AU(Internet) = {Public} whereas this role may grant access to the public HTTP service offered by the WebServer, i.e, AP(Internet) = {(may access, WebServer.HTTP)}.

These assignments and authorizations determine a user's potential to access some object. In addition, at runtime the actual access is regulated through the user's session. First, the active user is determined, i.e., $SU(s : Sessions) \rightarrow Users$, and, second, the active roles are derived, formally:

$$SR(s : Sessions) \rightarrow 2^{Roles} = \{r \in Roles \mid (SU(s), r) \in UA\}.$$

Concerning our example, $SU(s_{Public})$ = Public and $SR(s_{Public})$ = {Internet}. Finally, the *available permissions* are calculated through

$$ASP(s : Sessions) \rightarrow 2^{Permissions} = \bigcup_{r \in SR(s)} AP(r).$$

In our example, this enables the Public to access the HTTP service offered by the WebServer at runtime, i.e., ASP(s_{Public}) = {(may access, WebServer.HTTP)}.

Hierarchical RBAC

Hierarchical RBAC allows the structuring of roles by introducing relations for hierarchies and inheritance of access rights. In comparison, structuring and inheritance offer a natural representation of real-world relations in organizations and they enable a more concise overall description than the default role listings.

There are two notions of the hierarchy: *limited* and *general*. Their main difference lies in the shape of the role graph, i.e., the limited hierarchy is structured as a tree whereas the roles of the general hierarchy form an *Acyclic Directed Graph* (DAG). Formally, the general role hierarchy relation is defined as $RH \subseteq Roles \times Roles$ (briefly annotated as $r_1 \succeq r_2$ with $r_1, r_2 \in Roles$) where role r_2 is more general than role r_1. In our example, the role hierarchy could be encoded as

$$RH = \{\text{(Office, SampleOrga), (DMZ, SampleOrga), (WebServer, DMZ)}\}.$$

The inheritance relation allows the transitive passing of access rights from higher positioned roles to subsequent roles. Given two roles r_1 and r_2 with $(r_1, r_2) \in RH$, then the inheriting role r_1 holds at least all access rights that are associated with the more general role r_2 – formally:

$$r_1 \succeq r_2 \rightarrow AP(r_2) \subseteq AP(r_1).$$

Concerning our example, the user assignments and permission assignments are

$$\begin{aligned} UA = \{ & \\ & \text{(John, Office), (Jane, Office),} \\ & \text{(Jane, DMZ), (Jim, Office),} \\ & \text{(Public, Internet)} \\ \} & \end{aligned}$$

resp.

$$PA = \{$$
$$\quad \text{(Office, (may access, DMZ.SSH))},$$
$$\quad \text{(Office, (may access, WebServer.HTTP))},$$
$$\quad \text{(Internet, (may access, WebServer.HTTP))}$$
$$\}$$

Note that now the UA contains five elements instead of six and the PA has one element less, ergo, the representation is more concise.

In addition, the functions for the authorized users (AU) and authorized permissions (AP) are enhanced to deal with hierarchies. Users that are authorized for more general roles are also authorized for subsequent roles. Formally,

$$AU(r : Roles) \rightarrow 2^{Users} = \{u \in Users \mid r' \succeq r, (u, r') \in UA\}.$$

For example, the administrator Jane who is authorized for the DMZ role is also authorized for the WebServer role, i.e., AU(WebServer) = {Jane}. Respectively, permissions that are assigned to more general roles are also assigned to subsequent roles. Formally,

$$AP(r : Roles) \rightarrow 2^{Permissions} = \{p \in Permissions \mid r' \succeq r, (p, r') \in PA\}$$

For instance, the SSH access to the DMZ role is also assigned to the WebServer role, i.e.,

$$AP(\text{WebServer}) = \{$$
$$\quad \text{(may access, WebService.HTTP)},$$
$$\quad \text{(may access, WebService.SSH)}$$
$$\}.$$

Constrained RBAC

Separation of Duty (SoD) can be achieved by applying additional restrictions on the authorization of users and the activation of roles in a session. For this purpose, RBAC offers two variants: static SoD which governs user authorization and dynamic SoD which restricts sessions.

The static SoD limits the amount of roles that a user may be authorized for. This is realized through a relation $SSD \subseteq 2^{Roles} \times \mathbb{N}$ which is constrained by

$$\forall (rs, n) \in SSD, \forall t \subseteq rs : |t| \geq n \rightarrow \bigcap_{r \in t} AU(r) = \emptyset$$

Intuitively, a user may be authorized for an amount of roles lower than n. For instance, SSD = {({SampleOrga, Internet}, 2)} holds whereas SSD = {({Office, DMZ}, 2)} does not due to Jane who is authorized for both roles. SSD = {({Office, DMZ, Internet}, 3)}, on the other hand, holds because no user is authorized for all three roles.

Dynamic SoD is defined as a relation $DSD \subseteq 2^{Roles} \times \mathbb{N}$ with two constraints:

(1) $\forall rs \in 2^{Roles}, n \in \mathbb{N}, (rs, n) \in DSD \rightarrow n \geq 2 : |rs| \geq n$

(2) $\forall s \in Sessions, \forall rs \in 2^{Roles}, \forall rsub \in 2^{Roles}, \forall n \in \mathbb{N}, (rs, n) \in DSD, rsub \subseteq rs :$
$rsub \subseteq SR(s) \rightarrow |rsub| < n$

Intuitively, these require that less than n roles may be activated in a session. For example, for $n = 3$, Jane may activate at most two roles in her session s_{Jane}. Hence,

DSD = {({Office, DMZ}, 2), ({Office, WebServer}, 2), ({DMZ, WebServer}, 2)}.

The standard allows to define further restrictions beyond these quantitative constraints.

A.1.2 Compatibility of FPL with RBAC

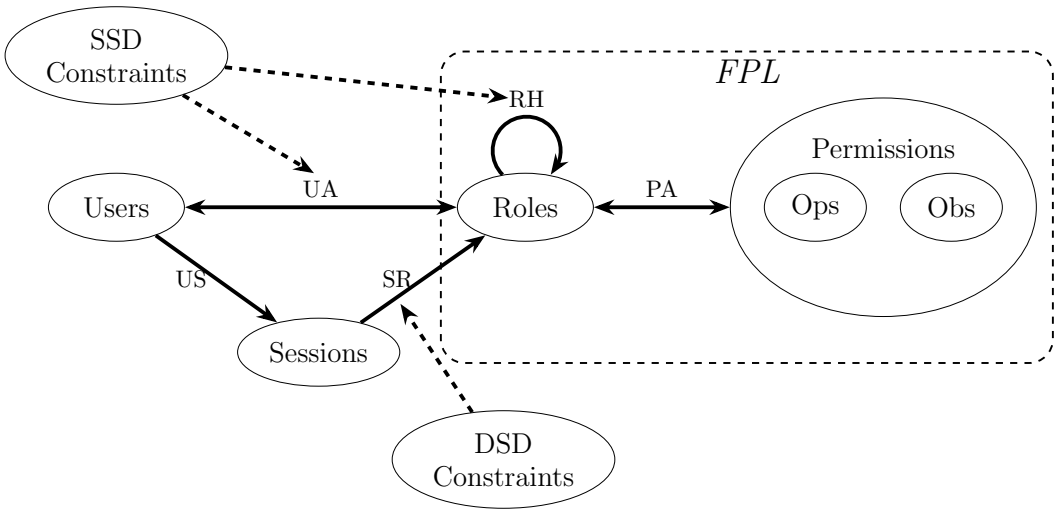


Fig. A.5.: RBAC covered by FPL.

The role concept in FPL shows similarities to RBAC and, indeed, FPL is compatible with NIST RBAC [123] which has been described in Appendix A.1.1. RBAC regulates the relations between users and access permissions through roles. User access (UA)

to roles and session management (US and SR) are beyond FPL's scope and can be omitted in this discussion. Figure A.5 shows RBAC's concepts covered by FPL and, in the following, we will discuss FPL's compatibility in detail based on RBAC's standard taxonomy.

Core RBAC User and session handling (UA resp. US and SR) are out of FPL's scope. FPL access operators correspond to the set of operations Ops in RBAC. The technical attributes specified for FPL roles and services form the set of objects Obs in RBAC. Therefore, FPL rules of the form

```
Subject Operator Object
```

correspond to the permission assignment PA which means that the authorized permissions AP can be derived. Hence, FPL rules can be integrated into a core RBAC framework.

Hierarchical RBAC Super role relations in FPL are realized as a DAG and therefore, they can form general role hierarchies in RBAC. FPL's resolution mechanism corresponds to the heritage relation (RH) and authorized permission function (AP) in RBAC. Therefore, FPL can be integrated into a hierachical RBAC framework. Again, user access is out of FPL's focus.

Contrained RBAC Again, user access to roles and session management is beyond FPL's scope. Any standard RBAC framework that introduces constraints, can also apply these to FPL roles.

All in all, FPL implements all necessary requirements for *Core* as well as *Hierarchical* RBAC while *Constrained* RBAC deals with issues separate from FPL's scope. Hence, FPL is fully compatible with NIST RBAC and can be integrated into standard RBAC frameworks.

A.2 Header Space Analysis

Besides Header Space Algebra (cf. Section 2.2.1), Header Space Analysis builds on *transfer functions* and custom analysis algorithms for reachability and loop detection.

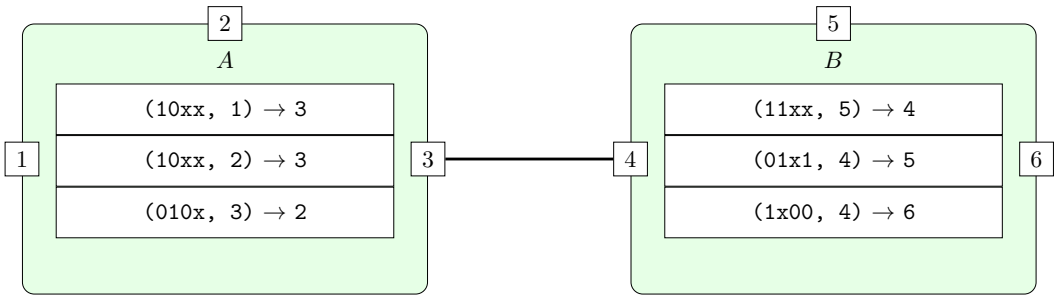


Fig. A.6.: Example network with two switches A and B.

We detail these concepts in Appendix A.2.1 resp. Appendix A.2.2 and provide a comparison with NetPlumber in Appendix A.2.3.

In the following, we will use a simple network which is depicted in Figure A.6 to illustrate the definition and application of the transfer functions. The network consists of two switches A and B with three ports 1, 2, and 3 resp. 4, 5, and 6, one bidirectional link between the ports 3 and 4, and three forwarding rules per switch.

A.2.1 Transfer Functions

HSA's analysis builds upon a set of transfer functions – namely switch, network, and topology transfer. These are used to simulate hops and network traversal which, in turn, build the basis for reachability analysis and loop detection (cf. Appendix A.2.2). Subsequently, we provide details for these functions.

Switch Transfer Function

The switch transfer function is used to model the behaviour of network devices. It takes header-port pairs as inputs and yields sets of header-port pairs, i.e.,

$$T : \mathcal{N} \to 2^{\mathcal{N}}$$
$$T(h,p) : (h,p) \to \{(h_1,p_1),(h_2,p_2),\dots\}$$

where h is a header space object that represents sets of packets.

For example, switch A forwards packets matching `10xx` and coming in at port 1 via port 3 without changing the packets. So, $T_A(\texttt{10xx},1) \to \{(\texttt{10xx},3)\}$.

In our example, the switch transfer functions for A and B are:

$$T_A(h,p) := \begin{cases} \{(h \cap \texttt{10xx}, 3)\}, \text{ if } p = 1, h \cap \texttt{10xx} \neq \emptyset \\ \{(h \cap \texttt{10xx}, 3)\}, \text{ if } p = 2, h \cap \texttt{10xx} \neq \emptyset \\ \{(h \cap \texttt{010x}, 2)\}, \text{ if } p = 3, h \cap \texttt{010x} \neq \emptyset \\ \emptyset, \text{ else} \end{cases}$$

resp.

$$T_B(h,p) := \begin{cases} \{(h \cap \texttt{11xx}, 4)\}, \text{ if } p = 5, h \cap \texttt{11xx} \neq \emptyset \\ \{(h \cap \texttt{01x1}, 5)\}, \text{ if } p = 4, h \cap \texttt{01x1} \neq \emptyset \\ \{(h \cap \texttt{1x00}, 6)\}, \text{ if } p = 4, h \cap \texttt{1x00} \neq \emptyset \\ \emptyset, \text{ else} \end{cases}$$

These functions encode a *first-match-wins* semantics where the upmost matching rule is applied.

Network Transfer Function

The network transfer function maps ports to switch transfer functions and applies the latter. Formally:

$$\Psi : \mathcal{N} \to 2^{\mathcal{N}}$$

$$\Psi(h,p) := \begin{cases} T_1(h,p), \text{ if } p \in switch_1 \\ T_2(h,p), \text{ if } p \in switch_2 \\ \dots \\ T_n(h,p), \text{ if } p \in switch_n \end{cases}$$

In our example, the ports 1, 2, and 3 belong to switch A and, hence, T_A is applied. The ports 4, 5, and 6, on the other hand, dispatch to the application of T_B. I.e.,

$$\Psi(h,p) := \begin{cases} T_A(h,1), \text{ if } p \in \{1,2,3\} \\ T_B(h,4), \text{ if } p \in \{4,5,6\} \end{cases}$$

Topology Transfer Function

The topology transfer function links ports and passes packets over these connections. Formally,

$$\Gamma : \mathcal{N} \to 2^{\mathcal{N}}$$
$$\Gamma(h, p) := \begin{cases} \{(h, p^*)\}, \text{ if } p \text{ is connected to } p^* \\ \emptyset, \text{ else} \end{cases}$$

If there is no outgoing link, the packets are dropped by the device. Concerning our example, the topology transfer function is the following since the ports 3 and 4 are connected bidirectionally:

$$\Gamma(h, p) := \begin{cases} \{(h, 4)\}, \text{ if } p = 3 \\ \{(h, 3)\}, \text{ if } p = 4 \\ \emptyset, \text{ else} \end{cases}$$

Simulation of Hops and Network Traversal

The simulation of the traversal of single packets in a network is performed by repeatedly applying the network and topology transfer functions. For this purpose, two more functions are defined: the *hop traversal* function $\Phi : \mathcal{N} \to \mathcal{N}$ and the *network traversal* function $\Phi^k : \mathcal{N} \to \mathcal{N}$.

Note: The simulation works with single packets, i.e., points in the Network Space. Hence, the transfer functions yield sets with at most one element (i.e., if the packet is not dropped). For the sake of simplicity, the following definitions treat the transfer functions as if they yield points in the Network Space instead of regions, i.e., $T : \mathcal{N} \to \mathcal{N}$ instead of $T : \mathcal{N} \to 2^{\mathcal{N}}$, $\Psi : \mathcal{N} \to \mathcal{N}$ instead of $\Psi : \mathcal{N} \to 2^{\mathcal{N}}$, and $\Gamma : \mathcal{N} \to \mathcal{N}$ instead of $\Gamma : \mathcal{N} \to 2^{\mathcal{N}}$. For this purpose, we silently unpack the single element from the resulting sets and treat empty sets as packet drops, respectively. This also resembles to the descriptions and definitions from the original paper [78].

Hop Traversal

To simulate the traversal of a packet h arriving at some port p, first, packet transformations as well as forwarding of a single hop is calculated by applying the network

transfer function Ψ. Then, the resulting packet is forwarded over the port's outgoing link using the topology transfer function Γ. Formally,

$$\begin{aligned}&\Phi : \mathcal{N} \to \mathcal{N}\\&\Phi(h, p) := \Gamma(\Psi(h, p))\end{aligned}$$

For example, we can simulate one hop for a packet $h = \texttt{1000}$ arriving at port $p = 1$ as follows:

$$\begin{aligned}\Phi(\texttt{1000}, 1) &= \Gamma(\Psi(\texttt{1000}, 1))\\&= \Gamma(\texttt{1000, 3})\\&= (\texttt{1000}, 4)\end{aligned}$$

Network Traversal

The simulation of a packet traversing the network is performed by hop-wise forwarding. For this purpose, for a packet-port pair the curret hop is simulated. Then, from there, the next hop can be simulated and so on. The network traversal is formally defined as

$$\begin{aligned}&\Phi^k : \mathcal{N} \to \mathcal{N}\\&\Phi^k(h, p) := \overbrace{\Phi(\Phi(\ldots(\Phi}^{k \text{ times}}(h, p))\ldots)) = \overbrace{\Gamma(\Psi(\ldots(\Gamma(\Psi}^{k \text{ times}}(h, p)))\ldots))\end{aligned}$$

where k denotes the amount of simulation steps.

For example, we simulate the packet $h = \texttt{1000}$ arriving at port $p = 1$ for $k = 1$:

$$\Phi^1(\texttt{1000}, 1) = \Phi(\texttt{1000}, 1) = (\texttt{1000}, 4)$$

For $k = 2$, the simulation is as follows:

$$\begin{aligned}\Phi^2(\texttt{1000}, 1) &= \Phi(\Phi(\texttt{1000}, 1))\\&= \Phi(\texttt{1000}, 4)\\&= \Gamma(\Psi(\texttt{1000}, 4))\\&= \Gamma(\texttt{1000}, 6)\\&= \text{drop}\end{aligned}$$

This example shows that the simulation yields the packet's position *within* the network after exactly k steps but not less. Simulating two hops resulted in a dropped packet while the simulation of one hop did not.

A.2.2 Network Verification with HSA

HSA's hop-wise network simulation shows that we can model networks and adequately process single packets using transfer functions. In order to verify useful properties packet-wise simulation does not scale well since the enumeration of the exponentially large Header Space is infeasible. Instead, HSA leverages its ability to model and process arbitrary regions of the Network Space to analyze the behaviour of several packets at once. In the original paper [78], the verifiably properties were *network reachability*, *loop detection*, and *slice isolation*. In the following, we will explain reachability analysis and loop detection but omit slice isolation. In this work, we do not need explicit slicing to determine network security compliance. Slicing, e.g., based on virtual networks using VLAN or VXLAN, is a measure to segment networks and is a common approach to implement network security in practice. Nevertheless, our approach checks compliance in an end-to-end manner and configuration for virtual networks is part of our network model. Breaches of virtual network isolation may also violate compliance and will be detected by our approach along other possible reasons, e.g., misconfigured firewall rules. Hence, we skip HSA's detection of slice isolation breaches here for the sake of brevity.

Checking the reachability between hosts and networks enables verification of security compliance. For each compliance rule, reachability between the rule's subjects and objects is determined and checked for conformance. Loops, on the other hand, pose a general threat due to the fact that an attacker with the ability to trigger the loop over the network, is able to consume vast amounts of network resources which may lead to denial-of-service.

Reachability

To check if a host resp. network can reach another host resp. network, HSA performs a path-wise reachability analysis. First, all paths between two ports a and b are calculated with device-level granularity, e.g., $\{a \rightarrow S_1 \rightarrow ... \rightarrow S_n \rightarrow b\}$ (a and b serve as representatives for the hosts resp. networks). For instance, assuming that a is connected with port 1 in our example network and b to port 6, then $R_{a \rightarrow b} = \{a \rightarrow A \rightarrow B \rightarrow b\}$ because there is just a single path over A and B. Each path analysis yields a, possibly empty, set of packets that reach b over this path and the union over all path packet sets is the set of all packets that can reach b. Note that the packets arriving at b are not necessarily in the same shape as sent by a as is due to the fact that switch transfer functions may change packets, e.g., a NAT may rewrite addresses and ports.

Per path, the set of propagated packets is calculated using

$$T_n(\Gamma(T_{n-1}(...(\Gamma(T_1(h,a)))...)))$$

where h can be any packet set, e.g., the set of all packets encoded as `xx...xx` or some packet set that describes the host resp. network. In addition, since the switch transfer functions T yields a set of header-port pairs, its output needs to be filtered according to the respective path under inspection. I.e., only the tuple (h,p) (if present) where $\Gamma(h,p) = \{(h,p^*)\}$ and p^* belongs to the next device on the path is considered for further analysis.

Concerning our example, the path between port 1 and port 6 is calculated as:

$$\begin{aligned} R_{a\to b} &= \bigcup\{T_B(\Gamma(T_A(\texttt{xxxx},1)))\} \\ &= \bigcup\{T_B(\Gamma(\{(\texttt{10xx},3)\}))\} \\ &= \bigcup\{T_B(\Gamma(\texttt{10xx},3))\} \\ &= \bigcup\{T_B(\texttt{10xx},4)\} \\ &= \bigcup\{\{(\texttt{1000},6)\}\} \\ &= \{(\texttt{1000},6)\} \end{aligned}$$

The original HSA paper [78] additionally performs analysis concerning the set of packets that is actually send by a and possibly changed during transfer. We omit these details since they are not necessary for compliance verification. Only the packet sets arriving at the destination may need further analysis, e.g., if some service is reached or not. For details on compliance rules, refer to Chapter 4.

Loop Detection

The network has a loop if there exists a packet that traverses the same (ingress) port at least twice. In order to find loops, for each ingress port HSA performs a depth-first search along the propagation graph by injecting the set of all possible packets, i.e., encoded as `xx...xx`. The search aborts in four cases:

- The packets leave the network which is indicated that the topology transfer function Γ drops the packet, i.e., there is no link to some other device within the network.
- Some previously visited port is visited again.
- No packets are left to pass the port.

- The starting port is visited again. Note that this case will be reported but not the one from the previous case. This is due to the fact that HSA performs loop analysis for all ports in the network and reporting the previous case could result in reporting the same loop multiple times.

The above search can be formalized as follows

$$loop(h,p,s,P) = \begin{cases} P \cup \{p\}, \text{ if } p = s \text{ and } P \neq \emptyset \\ P, \text{ if } p \in P \\ \emptyset, \text{ if } h = \emptyset \\ \bigcup\{ \\ \quad loop(h'',p'',s,P \cup \{p\}) \mid (h',p') \in \Psi(h,p), (h'',p'') \in \Gamma(h',p') \\ \}, \text{ else} \end{cases}$$

where s is the start port and P is the set of previously visited (ingress) ports. Analysis for some ingress port s is initialized as $loop(\texttt{xx...xx}, s, s, \emptyset)$.

For instance, if we want to analyse port 1 from our example for loops, we run $loop(\texttt{xxxx}, 1, 1, \emptyset)$. Since P is empty and, thus, also port 1 is not in P, the first two cases do not apply. Therefore, the traffic needs to be propagated through the device, i.e., switch A, over the outgoing links to the next devices, i.e., switch B, and loop detection needs to be performed for the ingress ports thereof. I.e., $\Psi(\texttt{xxxx}, 1) = \{(\texttt{10xx}, 3)\}$ and $\Gamma(\texttt{10xx}, 3) = (\texttt{10xx}, 4)$. Now loop detection needs to be run for 10xx at port 4: $loop(\texttt{110xx}, 4, 1, \{1\})$. As port 4 is not the starting port 1 and has not been visited before, we need to propagate through the device and try the next devices. Since $\Psi(\texttt{10xx}, 4) = \{(\texttt{1000}, 6)\}$ but $\Gamma(\texttt{1000}, 6) = \emptyset$ there are no further devices and the loop detection returns the empty set indicating that no loop has been found.

Let us assume that the forwarding tables in our example network are configured slightly different. In particular, the last rule in switch A matches packets with $h = \texttt{1000}$ and forwards these via port 3. In switch B, the last rule now forwards via port 4 (the changes are highlighted):

$$T_A(h,p) = \begin{cases} \{(h \cap \texttt{10xx}, 3)\}, \text{ if } p = 1, h \cap \texttt{10xx} \neq \emptyset \\ \{(h \cap \texttt{10xx}, 3)\}, \text{ if } p = 2, h \cap \texttt{10xx} \neq \emptyset \\ \{(h \cap \texttt{1000}, 3)\}, \text{ if } p = 3, h \cap \texttt{1000} \neq \emptyset \end{cases}$$

and

$$T_B(h,p) = \begin{cases} \{(h \cap \texttt{11xx}, 4)\}, \text{ if } p = 5, h \cap \texttt{11xx} \neq \emptyset \\ \{(h \cap \texttt{01x1}, 5)\}, \text{ if } p = 4, h \cap \texttt{01x1} \neq \emptyset \\ \{(h \cap \texttt{1x00}, 4)\}, \text{ if } p = 4, h \cap \texttt{1x00} \neq \emptyset \end{cases}$$

The packet 1000 injected at port 1 or 4 will cycle infinitely between the switches A and B. To find this loop, we start the analysis at port 4: $loop(\texttt{xxxx}, 4, 4, \emptyset)$. Since P is empty and, thus, also port 4 is not in P, the first and second case does not apply. Hence, we need to propagate traffic through the device and try the next devices. I.e., $\Psi(\texttt{xxxx}, 4) = \{(\texttt{01x1}, 5), (1x00, 4)\}$. Applying the topology transfer results in $\Gamma(\texttt{01x1}, 5) = \emptyset$ and $\Gamma(\texttt{1x00}, 4) = (\texttt{1x00}, 3)$. The first set of packets leaves the network but the second is forwarded over port 4 to port 3 where loop analysis is continued: $loop(\texttt{1x00}, 3, 4, \{4\})$. Since port 3 is neither the starting port 4 nor in the set of previously visited ports P, we need to propagate traffic to the next devices. I.e., $\Psi(\texttt{1x00}.3) = \{(1000, 3)\}$ and $\Gamma(\texttt{1000, 3}) = \{(\texttt{1000, 4})\}$. At port 4 the loop detection, finally, detects the loop, i.e., $loop(\texttt{1000}, 4, 4, \{4, 3\}) = \{4, 3\}$, since port 4 is also the starting port. The function yields the set of ports that form the loop.

The original HSA paper [78] additionally performs further analysis to distinguish between infinite and finite loops, e.g., looping packets that are dropped when their TTL reaches 0. We do not elaborate on this distinction because 1) also finite loops consume network ressources and can be used in DoS attacks and 2) it is neither discussed by the NetPlumber paper [79] nor implemented by the tool.

A.2.3 Comparison of NetPlumber with HSA

In this section, we compare modeling and analysis of NetPlumber with HSA and reveal similarities and – sometimes subtle – differences.

Ports The set of ports in NetPlumber corresponds with $\mathcal{P}$ in HSA while the set of links $L \subseteq \mathcal{P} \times \mathcal{P}$ does not have a direct correspondence but is used for purposes in NetPlumber similar to the topology transfer function Γ in HSA. Nevertheless, links in HSA work bidirectionally whereas links in NetPlumber do not. Therefore, NetPlumber allows to model special scenarios and devices more intuitively, e.g., data diodes with one-way fibre transmissions[1].

[1] cf. https://csrc.nist.gov/glossary/term/data_diode

Tables NetPlumber's tables implement limited versions of HSA's switch transfer functions. A table may implement one transfer function $T : \mathcal{N} \rightarrow 2^{\mathcal{N}}$ where desired packet transformations must be achieveable by rewriting only. HSA, on the other hand, allows more generic transformations including simple arithmetics, e.g., to decrement the TTL, or bit shift operations, e.g., for checksum functions. In order to model more complex transfer functions and network devices in NetPlumber, tables need to be chained using links. In addition, rules in NetPlumber only match wildcard expressions whereas transfer functions in HSA deal with arbitrary header space objects. To achieve the same behaviour, multiple NetPlumber rules can be used to represent portions of the Header Space that cannot be represented with a single wildcard expression. This might result in vastly larger memory consumption and processing times.

Sources and Probes NetPlumber's source nodes and their attachment to ports resemble the starting points, i.e., $(h, p) \in \mathcal{N}$, of HSA's reachability analysis and loop detection. Probe nodes, on the other hand, realize a structured and highly optimized approach to verify properties including reachability but also other that have no pendant in HSA, e.g., waypoint traversal.

Overall Analysis NetPlumber implements analysis through a flow-oriented depth-first search. In comparison to HSA's path-wise approach, this is more efficient due to the fact that common parts of a flow's path are calculated in one go instead of possibly several times as in HSA. Concerning reachability, NetPlumber performs backwards analysis for incoming flows at probe nodes using quantified regular expressions which is more expressive than HSA's simple reachability analysis (cf. next section).

NetPlumber's loop detection is performed on-the-fly in a depth-first manner which is similar to HSA. Nevertheness, NetPlumber uses tables instead of ports for loop detection which is more coarse grained and, argumentatively, more efficient. Intuitively, defining loops at the device level seems reasonable. Conceptually, it is possible to refactor NetPlumber's loop detection to use ports instead of tables.

A.3 Detailed Descriptions of other Dataplane Verification Approaches

ConfigChecker The tool *ConfigChecker* [5] enables the analysis of reachability and the detection of loops. For this purpose, it models networks of stateless firewalls based on a simple data model which includes source and destination addresses resp. ports, and device IDs but notably not protocol fields. Queries are formulated using the *Computational Tree Logics* (CTL) [23]. For instance, the reachability of a network $a2$ from a network $a1$ is specified as:

$$(src = a1 \wedge dest = a2 \wedge loc(a1)) \rightarrow AF(src = a1 \wedge dest = a2 \wedge loc(a2))$$

Using BDDs for the representation of packet sets, ConfigChecker supports incremental updates of its models.

In comparison, FPL and FaVe have a clear advantage in terms of accessibility and expressiveness. First, as seen above, CTL is hardly usable by non-academically skilled staff. For example, in FPL, reachability between some network roles $a1$ and $a2$ would simply be $a1$ `--->` $a2$. Second, DHSA allows FaVe to model device configurations with arbitrary header fields, stateful firewalls, and dynamic protocols like IPv6. Finally, FaVe's source code is publicly available.

VeriFlow The central idea of VeriFlow [81] is to identify sets of packets with identical forwarding behaviour – the so-called *equivalence classes* (EC) – and the independent analysis of EC-specific forwarding graphs.

An equivalence class is defined as a set of packets where for two arbitrary packets from the EC, each network device behaves identical, i.e., performs the same action like dropping or forwarding via the same port. VeriFlow uses multidimensional, ternary prefix trees – so called *tries* – over header bits (0, 1, don't care *) to represent sets of packets. A path through the trie represents a set of packets where the header bits are set according to the edge's bit annotations.

For a network, the trie is constructed by representing all rules as tries and merging these into a single trie. This trie's leaves are associated with the rules that match the packet set of the incoming path. Since these rules process the respective packets identically, they form the equivalence classes. Now, VeriFlow constructs EC-wise forwarding graphs that only contain the associated forwarding rules and abide the network's topology. Since within an EC, forwarding is identical for all packets,

analysis can be performed through generic graph algorithms without further consideration of the concrete packets. In the case of a rule update, VeriFlow is able to split or merge ECs and it reconstructs forwarding graphs if necessary. The tool, then, performs reverification only for the changed ECs and forwarding graphs which further optimizes performance.

VeriFlow[2] is primarily evaluated concerning its update performance and loop detection. The authors use synthetic workloads with 87 routers and an initial FIB (BGP Forwarding Information Base) with five million entries. The RIB (BGP Routing Information Base) entries are changed based on simulated BGP replays. Nevertheless, the concrete numbers of forwarding rules in the RIBs are not stated. In addition to IPv4-only forwarding, they perform experiments with three and six header fields. On average, VeriFlow processes rule updates in 0.38 milliseconds and stays below one millisecond in more than 90% of the cases.

FaVe, on the other hand, offers similar performance but is evaluated for more complex networks including dynamic protocols like IPv6. Also, stateful packet filters are supported whereas VeriFlow is limited to router ACLs with fixed header fields.

Libra By limiting its scope to IPv4 forwarding, the data plane checker Libra [155] scales to tremendously large networks with hundreds of millions of forwarding rules. The authors follow a divide-and-conquer approach that is implemented using a Map-Reduce data processing architecture. First, the mappers calculate independent slices and assign related forwarding rules to these slices. Then, the reducers construct and analyze forwarding graphs for sub networks and related slices. Supported invariants include reachability, loop freeness, black hole detection, and waypoint traversal. The jobs of the mapping and reducing phase can be performed in parallel respectively and, hence, allow phase-wise scale-out. In the case of a rule update, first, the slices are checked for consistency and, then, only those subnets and slices are recreated and verified which are affected by the rule. Hence, Libra also offers incremental verification.

Libra is evaluated with synthetic data center workloads where the largest network consists of ten thousand switches and ca. 265 million IPv4 forwarding rules. Since the authors estimated that they need 20,000 machines for this workload, they performed reduced experiments and extrapolated the runtime to 15 minutes. Real measurements using 50 machines for analyzing a workload of 316 switches with ca. 150 million forwarding rules took 93 seconds.

[2]There is a public implementation available at [6]. We were able to compile the code but the program segfaults directly after startup. We did not investigate into the issue further.

FaVe, in contrast, is not limited to IPv4 forwarding. Instead, it supports an arbitrary amount of header fields and dynamic protocols like IPv6. Further, FaVe is not limited to L3-switches but offers a large variety of predefined and complex device models like routers or stateful firewalls. Our evaluation with realistic workloads (cf. Section 7.2) show FaVe's scalability for large networks without being limited to datacenter networks.

SecGuru, RCDC, and NoD Three works have been published by groups with close affiliations to Microsoft research: *NoD* [90], *SecGuru* [14], and *RCDC* [72]. They put an emphasis on the applicability to data center networks and explore approaches in the dimensions of generality of applicability and performance. In summary, NoD provide a more generic approach to network verification whereas SecGuru and RCDC achieve scalability to data center workloads through specialization to the network architecture.

Network-optimized Datalog (NoD, [90]) adapts Datalog – a Prolog dialect for database operations [47] – to the domain of network verification. They provide modeling of packet headers, network rules, devices, and network topologies in terms of Datalog rules and enable queries that can be used to express invariants. For instance, given a symbolic packet injected at some point in the network, where can it go? Queries like this allow to check for network reachability.

In particular, NoD is based on two ideas. First, it represents sets of packets concisely as *differences of cubes* based on ternary bit-vectors which is similar to HSA's header space objects (cf. Section 2.2.1). Second, internally NoD avoids costly packet enumeration by introducing an optimized *SELECT-PROJECT* operator for Datalog. The Z3-based prototype[3] shows good performance, e.g., its verifies the Stanford workload in 5.9 seconds.

NoD is limited in terms of state in network functions like stateful packet filters as offered by FaVe. Also, NoD is not capable to process dynamic protocols like IPv6 even though some dynamic has been implemented for MPLS.

SecGuru [14] takes a different approach by focusing on the verification of Azure cloud data center networks and taking their high degree of structure into consideration. They verify a series of invariants called *cloud contracts*, e.g., *DNS ports on DNS servers are accessible from tenant devices over both TCP and UDP* (Cloud Contract 1, [14]) that are checked in parallel per router.

[3] Inititally, they published their implementation here: `http://web.ist.utl.pt/nuno.lopes/netverif/`. But since their code hoster CodePlex is out of business, it is not publicly available anymore. Nevertheless, parts of NoD, e.g., the ternary bit vectors, have been upstreamed to Z3.

The central idea lies in the formulation of assumptions about a router's environment that allow local verification in isolation. Locality allows fast and parallel analysis instead of more complex network wide verification. The assumptions hold since cloud data center networks are highly structured. For instance, routers are hierarchically ordered and each adjacent level provides some degree of redundancy. E.g., top-of-the-rack switches are grouped into *clusters* by a set of routers that provide redundant routes within the respective cluster. Given this uniformity in architecture, the configurations are very similar, e.g., they may only differ in IP prefixes but not ports or links. Also, different Azure data centers vary in their size but not so much in structure which allows easy adaption within the domain of data center networks.

On the technical level, packet sets, forwarding rules, and ACL rules are modeled as ternary bit vectors and the contracts are formulated using a simplified version of NoD (see above). Each contract can be verified either per router, e.g., Contract 1, or for pairs of routers, e.g., Contract 5.

Instead of providing a structured performance evaluation, the authors prove their tools relevance through real-world usage in Azure. In cooperation with the cloud engineers, routing related contracts are checked on a daily basis while ACL related contracts are verified per ACL update which sums up to 40 thousand runs per month.

In contrast, FaVe is not limited to data center networks since it does not require assumptions about the network architecture while being highly scalable as shown in Section 7.2.2. Also, local verification is inherently limited in terms of loop detection and end-to-end state checking which are both offered by FaVe. Further, FPL offers a more flexible way to specify security policies for organizations. In essence, SecGuru buys its scalability by being limited to structured data center networks and handcrafted cloud contracts.

The *Reality Checker for Data Centers* (RCDC, [72]) builds on SecGuru and overcomes some of its limitations. It still relies on the structure of cloud data centers but enables the verification of end-to-end invariants and flexibilizes contract definition.

The central idea is to formulate end-to-end contracts which are divisable into locally checkable smaller contracts which, in sum, give insights into the validity of the end-to-end contract. As for SecGuru, the local contracts can be verified in parallel. For instance, tenants in Azure can define *network security groups* (NSGs) which group technical ressources and limit access. RCDC is able to verify the conformity of ACL configuration with the NSGs. Since NSGs are user provided, this also reflects that RCDC offers more flexible contracts than SecGuru. Nevertheless, since they

build upon local contracts, also the end-to-end contracts are limited to cloud data center use cases. According to the authors, RCDC is actively used in production and delivers similar performance like SecGuru.

FaVe, on the other hand, offers the verification of more fine grained policies, arbitrary network topologies, and stateful devices. Further, unlike RCDC, FaVe is not limited to IPv4 prefixes, quintuples, or VLANs, but allows an arbitrary header space and the verification of dynamic protocols like IPv6.

AP-Verifier This data plane checker [147] promises low millisecond verification runtimes for large router networks.

The tool models network rules and packets in terms of boolean predicates and achieves its high performance through a concise and highly efficient representation of the header space – the so-called *atomic predicates* (AP). These are disjunct and non-empty sets of packets which are globally unique and composable in order to represent all predicates in the network. Since they are globally unique, a label, i.e., an integer, can be assigned to each AP and a rule's matching predicate can be represented as a set of labels, i.e., integers, instead of the direct encoding of the matched packets, e.g., a SAT formula or a BDD. Further, algorithms for analysis can be implemted based on regular set operations instead of data structures that are more time or space demanding, e.g., BDDs.

AP-Verifier models networks by precomputing the packet processing of routers and switches as AP-represented predicates and by annotating the topology graph with these predicates. Particularly, a router's ACLs and forwarding tables are represented using sets of APs. Then, the filtering ACL predicates are assigned to ingress ports in order to filter incoming packets throughout analysis. Further, ACL predicates are used to filter traffic via egress ports, i.e., the forwarding predicates that forward through the respective ports. The verification of network invariants like loop freeness or reachability is performed by using AP-aware graph algorithms over forwarding graphs that are calculated for each access port. AP-Verifiers performance is based on the fact that set operations over sets of labels, i.e., integers, are much more efficient than those of more fine grained packet set representations like BDDs. The evaluation shows great performance and scalability for workloads like the Stanford or Internet2 networks with overall runtimes below one second. AP-Verifier's calculation of the atomic predicates takes 196.68 ms for the Stanford and 142.95 ms for the Internet2 workload. The computation of a single reachability tree takes 0.95 ms resp. 0.27 ms on average. Total run times, e.g., for pairwise reachability, have not been reported.

In comparison, FaVe offers support for more complex network devices like stateful packet filters and dynamic protocols like IPv6. In general, it is not clear if AP-Verifier has been evaluated for more complex header spaces, i.e., beyond quintuples of source and destination IPs, the protocol field, and source and destination ports. Furthermore, in a master's thesis [82], we reevaluated the public prototype for the Stanfort and Internet2 workloads (cf. Section 7.2).

In comparison with the run times from the paper – which are 202.24 ms resp. 148.28 ms – our results are in the same order of magnitude. As seen in Section 7.2.2, FaVe's initialization phase takes 2.08 s for the Stanford and 47.85 s for the Internet2 workload which is ca. 10 times resp. 322 times slower. Nevertheless, FaVe's overall run time is reasonably fast and the tool offers a rich variety of network devices and accessible compliance specification.

Delta-net By limiting itself to IP routed networks instead of arbitrary packet header spaces Delta-net [60] achieves sub-millisecond reverification times for updates of networks with hundreds of millions of forwarding rules.

Its central concept is based on the division of the address space into disjunct labeled intervals, i.e., atoms, which are used to represent packet sets and model forwarding rules. They construct a forwarding graph where an edge is annotated with a set of atoms that is forwarded over the edge. Then, atom-aware graph algorithms are used for the verification of network invariants like reachability or loop freeness. This idea is is similar to AP-Verifier but limited to the IPv4 address space.

Delta-net is evaluated with a set of eight synthetic workloads with up to 300 routers and 125 million IPv4 forwarding rules. The authors run experiments that install and remove millions of rules and measure the reverification run times. On average, rule update runtimes range between 3 and 41μs with medians between 1 and 5μs. They do not provide total runtimes, i.e., the duration to set up the largest amount of rules in a network, but we can approximate this by multiplying the average rule insertion runtime with the maximum amount of network rules. For the largest network – the INET workload – this would be 125 million rule insertions with an average runtime of 41μs. This results in a total runtime of approximately 85 minutes.

In contrast, FaVe is able to process arbitrary header fields and much more complex devices, e.g., routers with ACLs or stateful packet filters. Further, FaVe offers support for dynamic protocols like IPv6. Delta-net achieves its impressive performance by introducing limitions, e.g., only unfiltered IPv4 forwarding, that are not realistic for a broad variety of networking scenarios, e.g., corporate networks or data centers. FaVe,

on the other hand, does not impose such limitations and offers splendid performance in realistic benchmarks (cf. Section 7.2).

APKeep The data plane verification tool APKeep [156] successfully combines several concepts from previous approaches, i.e., [147, 81, 60], and achieves very low reverification runtimes for network updates. In essence, the tool models networks based on composable network functions that, in turn, comprise of composable and predefined modeling *elements*. A modeling element offers a set of input and output ports and comprises a set of rules. The three predefined element types offer IP-based *forwarding*, quintuple-based *filtering*, and quintuple-based *rewriting*. In the formal *Port-Predicate-Model* (PPM), a boolean predicate that represents a set of packets is calculated for each output port. The predicate abides the element's rules so that the represented packet set comprises those packets that are emitted via the assigned port. Further, the global header space is divided into a set of mutual exclusive equivalence classes where any packet of the same class is processed identically by all modeling elements. This way, each predicate can be represented by a set of equivalence classes. Further, upon rule updates, APKeep automatically splits or merges equivalence classes if necessary. In order to model network functions and devices, the output ports of elements are connected unidirectionally to input ports of other elements. Networks are modeled by connecting output ports of device models to input ports of other device models. In addition, the edges are annotated using the equivalence classes of the emitting port. This way, a network model is an annotated, directed graph that can be analyzed using graph algorithms.

APKeep shows an outstanding update performance in comparison with other approaches, i.e., AP-Verifier, VeriFlow, NetPlumber, and Delta-net. A network update is processed in sub-milliseconds – often in the range of microseconds. For instance, on average, APKeep processes an update in the Stanford workload in 127 μs. This is a performance advantage of a factor of more than 13 over AP-Verifier and a factor of 75 over NetPlumber which were the 2nd resp. 3rd best approaches in class.

APKeep's performance is based on three key concepts. First, the graph analysis based on equivalence classes as representatives of packet sets which resembles AP-Verifier's approach. Second, the use of a *delta*-graph which is a reduced sub-graph that consists of only those equivalence classes and model elements that are affected by an update. This improves reverification efforts and resembles Delta-net's approach. Third, identical elements are shared among all device models which minimizes recalculation efforts upon updates.

In comparison, FaVe offers arbitrary header spaces beyond simple quintuples, i.e., source and destination addresses, the protocol field, and source and destination ports. Further, FaVe enables modeling dynamic protocols like IPv6. APKeep is limited to model stateless devices and lacks support for stateful devices like stateful packet filters. Finally, FaVe verifies compliance with high-level FPL policies whereas APKeep offers predicates and graph-traversing state machines to administrators and security officials who often lack technical expertise.

A.4 FPL Grammar

The grammar of the inventory and policy specification in FPL is defined as follows (in eBNF notation, we list only relevant whitespaces and newlines):

```
fpl ::= COMMENT | inventory newlines policy

policy ::= definition 'policies' '(' 'default:' default_policy ')'
           newlines rules
           newlines 'end'

default_policy ::= 'allow' | 'deny'

rules ::= rules_item | rules_item newlines rules

rules_item ::= COMMENT | rule

rule ::= subject operator object

subject ::= NAME
object ::= NAME | NAME '.' NAME | NAME '.*'

operator ::= '--->' | '<-->' | '<->>' | '--/->' | '<-/->' | '-/->>'

inventory ::= inventory_item | inventory_item inventory

inventory_item ::= COMMENT | service | role

service = definition 'service' NAME
          newlines service_body
          newlines 'end'

service_body ::= proto | proto newlines port

proto ::= 'protocol' '=' QUOTE [a-zA-Z]+ QUOTE
```

```
port ::= 'port' '=' [0-9]+

role ::= definition 'role' NAME
         newlines role_elems
         newlines 'end'

role_elems ::= role_elem | role_elem newlines role_elems

role_elem ::= COMMENT | field | offer | include

offer ::= 'offers' NAME

include ::= 'includes' NAME

definition ::= 'define' | 'def' | 'describe') | 'desc'

field ::= NAME '=' field_values

field_values ::= single_value | '[' value_list ']'

value_list ::= single_value | single_value ',' value_list

single_value ::= VALUE_WORD | quoted_value

quoted_value ::= QUOTE VALUE_TEXT QUOTE

VALUE_TEXT ::= [a-zA-Z0-9.:/ -_]+

VALUE_WORD ::= [a-zA-Z0-9.:/-_]+

NAME ::= [a-zA-Z][a-zA-Z0-9]*

WS ::= [ \t]+
newlines ::= NL | NL newlines
NL ::= [\r\n] | '\r\n'
QUOTE ::= ["']
COMMENT ::= '#'.*$
```

A.5 Proofs of the algebraic Properties of DHSA

In conjunction with Section 5.4, we prove that DHSA forms a distributive boolean lattice.

Union Commutativity

Theorem 3. *DHSA's union operation is commutative, i.e.:*

$$\forall x, y \in \mathcal{M} : x \cup y = y \cup x$$

We prove this theorem by proving the commutativity of the union of match sets.

Lemma 9. *The union of match sets is commutative, i.e.:*

$$\forall M_1, M_2 \in \mathcal{M} : M_1 \cup M_2 = M_2 \cup M_1.$$

Proof. As defined in Definition 11, DHSA's match set union reuses the regular set union operation which is commutative. □

Corollar 1. *The union of single matches is commutative, i.e.:*

$$\forall \{m_1\}, \{m_2\} \in \mathcal{M} : \{m_1\} \cup \{m_2\} = \{m_2\} \cup \{m_1\}.$$

Proof.

$$\{m_1\} \cup \{m_2\} = \{m_1, m_2\} = \{m_2, m_1\} = \{m_2\} \cup \{m_1\}$$

Since unioning match sets is commutative (Lemma 9).

□

Intersection Associativity

Theorem 4. *DHSA's intersection operation is associative, i.e.:*

$$\forall x, y, z \in \mathcal{M} : x \cap (y \cap z) = (x \cap y) \cap z.$$

We prove this theorem by proving the associativity of the intersection of tuples, single matches, and match sets.

Lemma 10. *Intersecting tuples is associative, i.e.:*

$$\forall t_1^h, t_2^h, t_3^h \in \mathcal{H}^D \times \mathcal{V}_{\mathcal{H}^D} : t_1^h \cap (t_2^h \cap t_3^h) = (t_1^h \cap t_2^h) \cap t_3^h.$$

Proof.

$$
\begin{aligned}
t_1^h \cap (t_2^h \cap t_3^h) &= (h, v_1^h) \cap ((h, v_2^h) \cap (h, v_3^h)) \\
&= (h, v_1^h) \cap ((h, v_2^h \cap v_3^h)) && \mid \text{cf. Definition 6} \\
&= (h, v_1^h \cap (v_2^h \cap v_3^h)) \\
&= (h, (v_1^h \cap v_2^h) \cap v_3^h) && \mid \text{since value sets are regular} \\
& && \mid \text{sets (Def. 1) and the regular} \\
& && \mid \text{set intersection is associative} \\
&= (h, (v_1^h \cap v_2^h)) \cap (h, v_3^h) \\
&= (t_1^h \cap t_2^h) \cap t_3^h
\end{aligned}
$$

□

Lemma 11. *Intersecting single matches is associative, i.e.:*

$$\forall \{m_1\}, \{m_2\}, \{m_3\} \in \mathcal{M} : (m_1 \cap m_2) \cap m_3 = m_1 \cap (m_2 \cap m_3).$$

Proof.

$$
\begin{aligned}
(m_1 \cap m_2) \cap m_3 &= \{t_1^{h_i} \cap t_2^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_1^{h_i} \in m_1, t_2^{h_i} \in m_2\} \cap m_3 && \mid \text{cf. Def. 7} \\
&= \{(t_1^{h_i} \cap t_2^{h_i}) \cap t_3^{h_i} \mid \\
&\quad \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_1^{h_i} \in m_1, t_2^{h_i} \in m_2, t_3^{h_i} \in m_3 \\
&\quad \} \\
&= \{t_1^{h_i} \cap (t_2^{h_i} \cap t_3^{h_i}) \mid && \mid \text{since the} \\
&\quad \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_1^{h_i} \in m_1, t_2^{h_i} \in m_2, t_3^{h_i} \in m_3 && \mid \text{tuple} \\
&\quad \} && \mid \text{intersection} \\
& && \mid \text{is} \\
& && \mid \text{associative} \\
& && \mid \text{(Lemma 10)} \\
&= m_1 \cap \{t_2^{h_i} \cap t_3^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_2^{h_i} \in m_2, t_3^{h_i} \in m_3\} \\
&= m_1 \cap (m_2 \cap m_3)
\end{aligned}
$$

□

Lemma 12. *Intersecting match sets is associative, i.e.:*

$$\forall M_1, M_2, M_3 \in \mathcal{M} : (M_1 \cap M_2) \cap M_3 = M_1 \cap (M_2 \cap M_3).$$

Proof.

$$
\begin{aligned}
(M_1 \cap M_2) \cap M_3 &= \{m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\} \cap M_3 && \mid \text{cf. Def. 8} \\
&= \{(m_1 \cap m_2) \cap m_3 \mid m_1 \in M_1, m_2 \in M_2, m_3 \in M_3\} \\
&= \{m_1 \cap (m_2 \cap m_3) \mid m_1 \in M_1, m_2 \in M_2, m_3 \in M_3\} && \mid \text{since the match} \\
& && \mid \text{intersection} \\
& && \mid \text{is associative} \\
& && \mid \text{(Lemma 11)} \\
&= M_1 \cap \{m_2 \cap m_3 \mid m_2 \in M_2, m_3 \in M_3\} \\
&= M_1 \cap (M_2 \cap M_3)
\end{aligned}
$$

□

Union Associativity

Theorem 5. *DHSA's union operation is associative, i.e.:*

$$\forall x, y, z \in \mathcal{M} : x \cup (y \cup z) = (x \cup y) \cup z.$$

We prove this theorem by proving the associativity of the union of match sets.

Lemma 13. *Unioning match sets is associative, i.e.:*

$$\forall M_1, M_2, M_3 \in \mathcal{M} : M_1 \cup (M_2 \cup M_3) = (M_1 \cup M_2) \cup M_3.$$

Proof. As defined in Definition 11, the union of match sets reuses the regular set union operation which is associative. □

Corollar 2. *Unioning single matches is associative, i.e.:*

$$\forall \{m_1\}, \{m_2\}, \{m_3\} \in \mathcal{M} : \{m_1\} \cup (\{m_2\} \cup \{m_3\}) = (\{m_1\} \cup \{m_2\}) \cup \{m_3\}.$$

Proof.

$$
\begin{aligned}
\{m_1\} \cup (\{m_2\} \cup \{m_3\}) &= \{m_1, m_2\} \cup \{m_3\} \\
&= \{m_1, m_2, m_3\} \\
&= \{m_1\} \cup \{m_2, m_3\} && \mid \text{since unioning match sets} \\
& && \mid \text{is associative (Lemma 13)} \\
&= (\{m_1\} \cup \{m_2\}) \cup \{m_3\}
\end{aligned}
$$

□

Intersection Idempotency

Theorem 6. *DHSA's intersection operation is idempotent, i.e.:*

$$\forall x \in \mathcal{M} : x \cap x = x.$$

We prove the idempotency of the intersection operation by proving that intersecting tuples, matches, and match sets is idempotent.

Lemma 14. *The tuple intersection is idempotent, i.e.:*

$$\forall t \in \mathcal{H}^D \times \mathcal{V}_{\mathcal{H}^D} : t^h \cap t^h = t^h.$$

Proof.

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

□

Lemma 15. *The match intersection is idempotent, i.e.:*

$$\forall \{m\} \in \mathcal{M} : m \cap m = m.$$

Proof.

$$\begin{aligned}
m \cap m &= \{t^{h_i} \cap t^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t^{h_i} \in m\} && \text{| cf. Def. 7} \\
&= \{t^{h_i} \mid \forall i = 1..|\mathcal{H}^D|, h_i \in \mathcal{H}^D : t^{h_i} \in m\} && \text{| since intersecting tuples is} \\
& && \text{| idempotent (Lemma 14)} \\
&= m && \text{| cf. Def. 2}
\end{aligned}$$

□

Lemma 16. *The intersection of match sets is idempotent, i.e.:*

$$\forall M \in \mathcal{M} : M \cap M = M.$$

Proof.

$$\begin{aligned}
M \cap M &= \{m \cap m \mid m \in M\} && \text{| cf. Def. 8} \\
&= \{m \mid m \in M\} && \text{| since intersecting matches is} \\
& && \text{| idempotent (Lemma 15)} \\
&= M && \text{| Def. 4}
\end{aligned}$$

□

Union Idempotency

Theorem 7. *DHSA's union operation is idempotent, i.e.:*

$$\forall x \in \mathcal{M} : x \cup x = x.$$

We prove the idempotency of the union operation by proving that unioning match sets is idempotent.

Lemma 17. *The union of match sets is idempotent, i.e.:*

$$\forall M \in \mathcal{M} : M \cup M = M.$$

Proof. As defined in Definition 11, the union of match sets reuses the regular set union operation which is idempotent. □

Corollar 3. *The match union is idempotent, i.e.:*

$$\forall \{m\} \in \mathcal{M} : \{m\} \cup \{m\} = \{m\}.$$

Proof.

$$\{m\} \cup \{m\} = \{m, m\} = \{m\}$$

(Due to Lemma 17.) □

Distributivity of Intersections over Unions

Theorem 8. *DHSA's intersection operation distributes over the union operation, i.e.:*

$$\forall x, y, z \in \mathcal{M} : x \cap (y \cup z) = (x \cap y) \cup (x \cap z).$$

We prove the distributivity of the intersection operation over the union operation by proving that these operations are distributive for match sets.

Lemma 18. *The intersection operation distributes over unions for match sets, i.e.:*

$$\forall M_1, M_2, M_3 \in \mathcal{M} : M_1 \cap (M_2 \cup M_3) = (M_1 \cap M_2) \cup (M_1 \cap M_3).$$

Proof.

$$\begin{aligned}
M_1 \cap (M_2 \cup M_3) &= M_1 \cap \{m_2, m_3 \mid m_2 \in M_2, m_3 \in M_3\} && \mid \text{cf. Def. 11} \\
&= \{m_1 \cap m_2, m_1 \cap m_3 \mid m_1 \in M_1, m_2 \in M_2, m_3 \in M_3\} && \mid \text{cf. Def. 8} \\
&= \{m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\} \cup && \mid \text{cf. Def. 11} \\
&\quad \{m_1 \cap m_3 \mid m_1 \in M_1, m_3 \in M_3\} \\
&= (M_1 \cap M_2) \cup (M_1 \cap M_3) && \mid \text{cf. Def. 4}
\end{aligned}$$

□

Corollar 4. *The intersection operation distributes over unions for matches, i.e.:*

$$\forall \{m_1\}, \{m_2\}, \{m_3\} \in \mathcal{M} : \{m_1\} \cap (\{m_2\} \cup \{m_3\}) = (\{m_1\} \cap \{m_2\}) \cup (\{m_1\} \cap \{m_3\}).$$

Proof.

$$\begin{aligned}
\{m_1\} \cap (\{m_2\} \cup \{m_3\}) &= \{m_1\} \cap \{m_2, m_3\} && \mid \text{cf. Def. 11} \\
&= \{m_1 \cap m_2, m_1 \cap m_3\} && \mid \text{cf. Def. 8} \\
&= \{m_1 \cap m_2\} \cup \{m_1 \cap m_3\} && \mid \text{cf. Lemma 18} \\
&= (\{m_1\} \cap \{m_2\}) \cup (\{m_1\} \cap \{m_3\})
\end{aligned}$$

□

Distributivity of Unions over Intersections

Theorem 9. *DHSA's union operation distributes over the intersection operation, i.e.:*

$$\forall x, y, z \in \mathcal{M} : x \cup (y \cap z) = (x \cup y) \cap (x \cup z).$$

We prove the distributivity of the union operation over the intersection operation by proving that these operations are distributive for match sets.

Lemma 19. *The union operation distributes over intersections for match sets, i.e.:*

$$\forall M_1, M_2, M_3 \in \mathcal{M} : M_1 \cup (M_2 \cap M_3) = (M_1 \cup M_2) \cap (M_1 \cup M_3).$$

Proof.

$$
\begin{array}{lll}
M_1 \cup (M_2 \cap M_3) & = M_1 \cup \{m_2 \cap m_3 \mid m_2 \in M_2, m_3 \in M_3\} & \mid \text{cf. Def. 8} \\
& = \{m_1, m_2 \cap m_3 \mid m_1 \in M_1, m_2 \in M_2, m_3 \in M_3\} & \mid \text{cf. Def. 11} \\
& = \{m_1 \cap m_1, m_1 \cap m_2, m_1 \cap m_3, m_2 \cap m_3 \mid & \mid \text{we can safely add} \\
& \quad m_1 \in M_1, m_2 \in M_2, m_3 \in M_3 & \mid m_1 \cap m_2 \text{ resp. } m_1 \cap m_3 \\
& \quad \} & \mid \text{since all their possible} \\
& & \mid \text{values are already} \\
& & \mid \text{present in } m_1 \text{ and,} \\
& & \mid \text{hence, do not add any} \\
& & \mid \text{new values to the match} \\
& & \mid \text{set} \\
& = \{m_1, m_2 \mid m_1 \in M_1, m_2 \in M_2\} \cap & \mid \text{cf. Def. 8} \\
& \quad \{m_1, m_3 \mid m_1 \in M_1, m_3 \in M_3\} & \\
& = (M_1 \cup M_2) \cap (M_1 \cup M_3) & \mid \text{cf. Def. 11}
\end{array}
$$

□

Corollar 5. *The union operation distributes over intersections for matches, i.e.:*

$$\forall \{m_1\}, \{m_2\}, \{m_3\} \in \mathcal{M} : \{m_1\} \cup (\{m_2\} \cap \{m_3\}) = (\{m_1\} \cup \{m_2\}) \cap (\{m_1\} \cup \{m_3\}).$$

Proof.

$$
\begin{array}{lll}
\{m_1\} \cup (\{m_2\} \cap \{m_3\}) & = \{m_1\} \cup \{m_2 \cap m_3\} & \mid \text{cf. Def. 8} \\
& = \{m_1, m_2 \cap m_3\} & \mid \text{cf. Def. 11} \\
& = \{m_1 \cap m_1, m_1 \cap m_2, m_1 \cap m_3, m_2 \cap m_3\} & \mid \text{cf. Lemma 19} \\
& = \{m_1, m_2\} \cap \{m_1, m_3\} & \\
& = (\{m_1\} \cup \{m_2\}) \cap (\{m_1\} \cap \{m_3\}) &
\end{array}
$$

□

Absorbtion of Unions by Intersections

Theorem 10. *DHSA's intersection operation absorbs the union operation, i.e.:*

$$\forall x, y \in \mathcal{M} : x \cap (x \cup y) = x.$$

We prove the absorbtion of the union operation by the intersection operation by proving that intersections absorb unions for match sets.

Lemma 20. *The intersection operation absorbs unions for match sets, i.e.:*

$$\forall M_1, M_2 \in \mathcal{M} : M_1 \cap (M_1 \cup M_2) = M_1.$$

Proof.

$$\begin{array}{lll} M_1 \cap (M_1 \cup M_2) & = M_1 \cap \{m_1, m_2 \mid m_1 \in M_1, m_2 \in M_2\} & \mid \text{cf. Def. 11} \\ & = \{m_1 \cap m_1, m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\} & \mid \text{cf. Def. 8} \\ & = \{m_1 \mid m_1 \in M_1\} = M_1 & \mid \text{since for each} \\ & & \mid m_1 \cap m_2 \text{ there exists} \\ & & \mid \text{a superset } m_1 \text{ and,} \\ & & \mid \text{hence, the former} \\ & & \mid \text{can be omitted safely} \end{array}$$

□

Corollar 6. *The intersection operation absorbs unions for matches, i.e.:*

$$\forall \{m_1\}, \{m_2\} \in \mathcal{M} : \{m_1\} \cap (\{m_1\} \cup \{m_2\}) = \{m_1\}.$$

Proof.

$$\begin{array}{lll} \{m_1\} \cap (\{m_1\} \cup \{m_2\}) & = \{m_1\} \cap \{m_1, m_2\} & \mid \text{cf. Def. 11} \\ & = \{m_1, m_1 \cap m_2\} & \mid \text{cf. Def. 8} \\ & = \{m_1\} & \mid \text{cf. Lemma 20} \end{array}$$

□

Absorbtion of Intersections by Unions

Theorem 11. *DHSA's union operation absorbs the intersection operation, i.e.:*

$$\forall x, y \in \mathcal{M} : x \cup (x \cap y) = x.$$

We prove the absorbtion of the intersection operation by the union operation by proving that unions absorb intersections for match sets.

Lemma 21. *The union operation absorbs intersections for match sets, i.e.:*

$$\forall M_1, M_2 \in \mathcal{M} : M_1 \cup (M_1 \cap M_2) = M_1.$$

Proof.

$$\begin{array}{lll} M_1 \cup (M_1 \cap M_2) & = M_1 \cup \{m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\} & \mid \text{cf. Def. 8} \\ & = \{m_1, m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\} & \mid \text{cf. Def. 11} \\ & = \{m_1 \mid m_1 \in M_1\} = M_1 & \mid \text{since for each } m_1 \cap m_2 \\ & & \mid \text{there exists a superset } m_1, \\ & & \mid \text{the former can be omitted} \\ & & \mid \text{safely} \end{array}$$

□

Corollar 7. *The union operation absorbs intersections for matches, i.e.:*

$$\forall \{m_1\}, \{m_2\} \in \mathcal{M} : \{m_1\} \cup (\{m_1\} \cap \{m_2\}) = \{m_1\}.$$

Proof.

$$\begin{array}{lll} \{m_1\} \cup (\{m_1\} \cap \{m_2\}) & = \{m_1\} \cup \{m_1 \cap m_2\} & \mid \text{cf. Def. 8} \\ & = \{m_1, m_1 \cap m_2\} & \mid \text{cf. Def. 11} \\ & = \{m_1\} & \mid \text{cf. Lemma 21} \end{array}$$

□

A.6 Neutral Elements in DHSA

We show that the empty set and the set of all values may serve as neutral elements for DHSA's union resp. intersection operations.

Theorem 12. *The set of all values behaves neutrally when used in an intersection, i.e.:*

$$\forall M \in \mathcal{M} : M \cap M_\Omega = M.$$

We proof overall neutrality by proving neutrality for tuples, matches, and match sets.

Lemma 22. *The tuple with the value domain, i.e.,* $t^h_\Omega := (h, \mathcal{V}_h)$*, behaves neutrally when used in tuple intersections, i.e.:*

$$\forall t^h \in \mathcal{H}^D \times \mathcal{V}^D_\mathcal{H} : t^h \cap t^h_\Omega = t^h.$$

Proof.

$$t^h \cap t^h_\Omega = (h, v_h) \cap (h, \mathcal{V}_h) = (h, v_h \cap \mathcal{V}_h) = (h, v_h) = t^h.$$

□

Lemma 23. *The match consisting of tuples with full value domains, i.e.,* $m_\Omega := \{t^{h_i}_\Omega \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D\}$*, behaves neutrally when used in match intersections, i.e.:*

$$\forall \{m\} \in \mathcal{M} : m \cap m_\Omega.$$

Proof.

$$\begin{aligned} m \cap m_\Omega &= \{t^{h_i}_m \cap t^{h_i}_\Omega \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t^{h_i}_m \in m, t^{h_i}_\Omega \in m_\Omega\} && \text{| cf. Def. 7} \\ &= \{t^{h_i}_m \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t^{h_i}_m \in m\} = m && \text{| since } t^{h_i}_\Omega \text{ is the} \\ & && \text{| neutral element for} \\ & && \text{| tuple intersections} \\ & && \text{| (Lemma 22)} \end{aligned}$$

□

Lemma 24. *The match set consisting of the full match, i.e.,* $M_\Omega := \{m_\Omega\}$*, behaves neutrally when used in match set intersections, i.e.:*

$$\forall M \in \mathcal{M} : M \cap M_\Omega.$$

Proof.

$$\begin{aligned} M \cap M_\Omega &= \{m \cap m_\Omega \mid m \in M\} && \text{| cf. Def. 8} \\ &= \{m \mid m \in M\} && \text{| since } m_\Omega \text{ is the neutral element for the} \\ & && \text{| match intersection (Lemma 23)} \\ &= M \end{aligned}$$

□

Theorem 13. *The empty set serves as a neutral element for the union operation, i.e.:*

$$\forall M \in \mathcal{M} : M \cup \emptyset = M.$$

We proof neutrality for match sets and subsequentially for single matches.

Lemma 25. *The empty match set $M_\emptyset := \emptyset$ behaves as neutral element when used in match set unions, i.e.:*

$$\forall M \in \mathcal{M} : M \cup M_\emptyset = M.$$

Proof. Since match sets use the regular set union (cf. Definition 11) where the empty set serves as neutral element, neutrality also holds for match sets. □

Lemma 26. *The empty match $m_\emptyset := \{t_\emptyset^{h_i} \mid \forall i = 1..|\mathcal{H}^D| : h_i \in \mathcal{H}^D, t_\emptyset^{h_i} := (h_i, \emptyset)\}$ behaves as neutral element when used in match unions, i.e.:*

$$\forall \{m\} \in \mathcal{M} : \{m\} \cup \{m_\emptyset\} = \{m\}.$$

Proof.

$$\begin{aligned} \{m\} \cup \{m_\emptyset\} &= \{m, m_\emptyset\} \\ &= \{m\} \qquad \mid \text{since } m_\emptyset \text{ is always a subset of } m, \\ &\qquad\quad \mid \text{i.e., } m_\emptyset \subseteq m \Leftrightarrow m_\emptyset \cap m = m_\emptyset, \text{ the} \\ &\qquad\quad \mid \text{former can be omitted safely} \end{aligned}$$

□

A.7 Supremum and Infimum in DHSA

Theorem 14. *The empty match set $M_\emptyset := \emptyset$ is the supremum in DHSA, i.e.:*

$$\nexists M \in \mathcal{M} : M \subset M_\emptyset.$$

Proof.

$$\begin{aligned} & \nexists M \in \mathcal{M} : M \subset M_\emptyset \\ \Leftrightarrow\ & \forall M \in \mathcal{M} : M \not\subset M_\emptyset \\ \Leftrightarrow\ & \forall M \in \mathcal{M} : M_\emptyset \subseteq M \\ \Leftrightarrow\ & \forall M \in \mathcal{M}, m_1 \in M_\emptyset : \exists m_2 \in M : m_1 \subseteq m_2 \quad \mid \text{cf. Def. 10} \end{aligned}$$

Since there is no m_1 in $M_\emptyset$ by definition, the above statement holds trivially. □

Theorem 15. *The set of all values M_Ω is the infimum in DHSA, i.e.:*

$$\nexists M \in \mathcal{M} : M_\Omega \subset M.$$

Proof.

$$\begin{aligned}
& \nexists M \in \mathcal{M} : M_\Omega \subset M \\
\Leftrightarrow\ & \forall M \in \mathcal{M} : M_\Omega \not\subset M \\
\Leftrightarrow\ & \forall M \in \mathcal{M} : M \subseteq M_\Omega \\
\Leftrightarrow\ & \forall M \in \mathcal{M}, m_1 \in M : \exists m_2 \in M_\Omega : m_1 \subseteq m_2 \quad | \text{ cf. Def. 10} \\
\Leftrightarrow\ & \forall M \in \mathcal{M}, m_1 \in M : \exists m_1 \in M_\Omega : m_1 \subseteq m_1 \quad | \text{ cf. Def. 5}
\end{aligned}$$

Since m_1 is always also contained in M_Ω, the above statement holds trivially. □

A.8 FPL Inventory and Policy of the IFI Benchmark

The following snippet contains the FPL inventory and policy used in the IFI benchmark.

```
def role external
    description = 'Services for external view'
    hosts       = ['cs.uni-potsdam.de']
    ipv4        = '141.89.48.0/24'
    ipv6        = '2001:638:807:30::0/64'
    gateway     = '141.89.48.254'
    vlan        = 48
end

def role internal
    description = 'Services for internal view'
    hosts       = ['internal.cs.uni-potsdam.de']
    ipv4        = '10.3.12.0/23'
    gateway     = '10.3.13.254'
    vlan        = 463
end

def role admin
    description = 'Hosts of administrators'
    hosts       = ['admin.cs.uni-potsdam.de']
    ipv4        = '10.3.14.0/23'
    gateway     = '10.3.15.254'
    vlan        = 464
end

```

```
26 def role office
27     description = 'Hosts of secretariats'
28     hosts       = ['office.cs.uni-potsdam.de']
29     ipv4        = '10.3.16.0/23'
30     gateway     = '10.3.17.254'
31     vlan        = 465
32 end
33
34 def role staff_1
35     description = 'Hosts of employee'
36     hosts       = ['staff.cs.uni-potsdam.de']
37     ipv4        = '10.3.18.0/23'
38     gateway     = '10.3.19.254'
39     vlan        = 466
40 end
41
42 def role staff_2
43     description = 'Hosts of employee'
44     hosts       = ['staff.cs.uni-potsdam.de']
45     ipv4        = '10.3.20.0/23'
46     gateway     = '10.3.21.254'
47     vlan        = 467
48 end
49
50 def role pool
51     description = 'Hosts of the computer pools'
52     hosts       = ['pool.cs.uni-potsdam.de']
53     ipv4        = '10.3.22.0/23'
54     gateway     = '10.3.23.254'
55     vlan        = 468
56 end
57
58 def role lab
59     description = 'Hosts of the laboratories'
60     hosts       = ['lab.cs.uni-potsdam.de']
61     ipv4        = '10.3.24.0/23'
62     gateway     = '10.3.25.254'
63     vlan        = 469
64 end
65
66 def role hpc_mgt
67     description = 'Management network of the hpc'
68     hosts       = ['mgt.hpc']
69     ipv4        = '10.3.26.0/23'
70     vlan        = 470
71 end
72
73 def role hpc_ic
```

```
    description = 'Interconnect network of the hpc'
    hosts       = ['ic.hpc']
    ipv4        = '10.3.28.0/23'
    vlan        = 471
end

def role slb
    description = 'Server load balancing network'
    hosts       = ['slb.cs.uni-potsdam.de']
    ipv4        = '10.3.30.0/23'
    gateway     = '10.3.31.254'
    vlan        = 472
end

def role mgt
    description = 'Management network'
    hosts       = ['mgt.cs.uni-potsdam.de']
    ipv4        = '10.3.32.0/23'
    gateway     = '10.3.33.254'
    vlan        = 473
end

def role san
    description = 'Storage area network'
    hosts       = ['san.cs.uni-potsdam.de']
    ipv4        = '10.3.34.0/23'
    vlan        = 474
end

def role vmo
    description = 'VMware vMotion network'
    hosts       = ['vmo.cs.uni-potsdam.de']
    ipv4        = '10.3.36.0/23'
    vlan        = 475
end

def role prt
    description = 'Printer network'
    hosts       = ['prt.cs.uni-potsdam.de']
    ipv4        = '10.3.38.0/23'
    vlan        = 476
end

def role cam
    description = 'Camera network'
    hosts       = ['cam.cs.uni-potsdam.de']
    ipv4        = '10.3.40.0/23'
    vlan        = 477
```

```
122 end
123
124
125 def role access_to_external
126     description = 'Access to the external services'
127     includes Internet
128     includes admin
129     includes office
130     includes staff_1
131     includes staff_2
132     includes pool
133     includes lab
134     includes slb
135 end
136
137 def role access_to_internal
138     description = 'Access to the internal services'
139     includes admin
140     includes office
141     includes staff_1
142     includes staff_2
143     includes pool
144     includes lab
145     includes slb
146 end
147
148 def role access_from_admin
149     description = 'Access from the admin network'
150     includes internal
151     includes office
152     includes staff_1
153     includes staff_2
154     includes pool
155     includes lab
156     includes slb
157 end
158
159 def role access_to_internet
160     description = 'Access to the internet'
161     includes admin
162     includes office
163     includes staff_1
164     includes staff_2
165     includes pool
166     includes lab
167 end
168
169
```

```
def policies (default: deny)
    # Access to the external services
    access_to_external  <->>  external

    # Access to the internal services
    access_to_internal  <->>  internal

    # Access from the admin network
    admin  <->>  access_from_admin

    # Access to the internet
    access_to_internet  <->>  Internet
end
```

A.9 ACL configuration of the IFI-Benchmark

The following configuration snippet contains the ACL rules for the Cisco router in the IFI benchmark.

```
access-list 100 permit ip 10.0.14.0 0.0.1.255 123.123.48.0 0.0.0.255
access-list 100 permit ip 10.0.16.0 0.0.1.255 123.123.48.0 0.0.0.255
access-list 100 permit ip 10.0.18.0 0.0.1.255 123.123.48.0 0.0.0.255
access-list 100 permit ip 10.0.20.0 0.0.1.255 123.123.48.0 0.0.0.255
access-list 100 permit ip 10.0.22.0 0.0.1.255 123.123.48.0 0.0.0.255
access-list 100 permit ip 10.0.24.0 0.0.1.255 123.123.48.0 0.0.0.255
access-list 100 permit ip 10.0.30.0 0.0.1.255 123.123.48.0 0.0.0.255
# deny all other internal accesses
access-list 100 deny ip 10.0.0.0 0.0.255.255 123.123.48.0 0.0.0.255
# access from the internet
access-list 100 permit ip 0.0.0.0 0.0.0.0 123.123.48.0 0.0.0.255
access-list 100 deny ip any any

access-list 101 permit ip 10.0.14.0 0.0.1.255 10.0.12.0 0.0.1.255
access-list 101 permit ip 10.0.16.0 0.0.1.255 10.0.12.0 0.0.1.255
access-list 101 permit ip 10.0.18.0 0.0.1.255 10.0.12.0 0.0.1.255
access-list 101 permit ip 10.0.20.0 0.0.1.255 10.0.12.0 0.0.1.255
access-list 101 permit ip 10.0.22.0 0.0.1.255 10.0.12.0 0.0.1.255
access-list 101 permit ip 10.0.24.0 0.0.1.255 10.0.12.0 0.0.1.255
access-list 101 permit ip 10.0.30.0 0.0.1.255 10.0.12.0 0.0.1.255
access-list 101 deny ip any any

access-list 102 permit ip 10.0.12.0 0.0.1.255 10.0.14.0 0.0.1.255
access-list 102 permit ip 10.0.16.0 0.0.1.255 10.0.14.0 0.0.1.255
access-list 102 permit ip 10.0.18.0 0.0.1.255 10.0.14.0 0.0.1.255
access-list 102 permit ip 10.0.20.0 0.0.1.255 10.0.14.0 0.0.1.255
```

```
access-list 102 permit ip 10.0.22.0 0.0.1.255 10.0.14.0 0.0.1.255
access-list 102 permit ip 10.0.24.0 0.0.1.255 10.0.14.0 0.0.1.255
access-list 102 permit ip 10.0.30.0 0.0.1.255 10.0.14.0 0.0.1.255
access-list 102 permit ip 123.123.48.0 0.0.0.255 10.0.14.0 0.0.1.255
# deny all other internal accesses
access-list 102 deny ip 10.0.0.0 0.0.255.255 10.0.14.0 0.0.1.255
# access from the internet
access-list 102 permit ip 0.0.0.0 0.0.0.0 10.0.14.0 0.0.1.255
access-list 102 deny ip any any

access-list 103 permit ip 10.0.12.0 0.0.1.255 10.0.16.0 0.0.1.255
access-list 103 permit ip 10.0.14.0 0.0.1.255 10.0.16.0 0.0.1.255
access-list 103 permit ip 123.123.48.0 0.0.0.255 10.0.16.0 0.0.1.255
# deny all other internal accesses
access-list 103 deny ip 10.0.0.0 0.0.255.255 10.0.16.0 0.0.1.255
# access from the internet
access-list 103 permit ip 0.0.0.0 0.0.0.0 10.0.16.0 0.0.1.255
access-list 103 deny ip any any

access-list 104 permit ip 10.0.12.0 0.0.1.255 10.0.18.0 0.0.1.255
access-list 104 permit ip 10.0.14.0 0.0.1.255 10.0.18.0 0.0.1.255
access-list 104 permit ip 123.123.48.0 0.0.0.255 10.0.18.0 0.0.1.255
# deny all other internal accesses
access-list 104 deny ip 10.0.0.0 0.0.255.255 10.0.18.0 0.0.1.255
# access from the internet
access-list 104 permit ip 0.0.0.0 0.0.0.0 10.0.18.0 0.0.1.255
access-list 104 deny ip any any

access-list 105 permit ip 10.0.12.0 0.0.1.255 10.0.20.0 0.0.1.255
access-list 105 permit ip 10.0.14.0 0.0.1.255 10.0.20.0 0.0.1.255
access-list 105 permit ip 123.123.48.0 0.0.0.255 10.0.20.0 0.0.1.255
# deny all other internal accesses
access-list 105 deny ip 10.0.0.0 0.0.255.255 10.0.20.0 0.0.1.255
# access from the internet
access-list 105 permit ip 0.0.0.0 0.0.0.0 10.0.20.0 0.0.1.255
access-list 105 deny ip any any

access-list 106 permit ip 10.0.12.0 0.0.1.255 10.0.22.0 0.0.1.255
access-list 106 permit ip 10.0.14.0 0.0.1.255 10.0.22.0 0.0.1.255
access-list 106 permit ip 123.123.48.0 0.0.0.255 10.0.22.0 0.0.1.255
# deny all other internal accesses
access-list 106 deny ip 10.0.0.0 0.0.255.255 10.0.22.0 0.0.1.255
# access from the internet
access-list 106 permit ip 0.0.0.0 0.0.0.0 10.0.22.0 0.0.1.255
access-list 106 deny ip any any

access-list 107 permit ip 10.0.12.0 0.0.1.255 10.0.24.0 0.0.1.255
access-list 107 permit ip 10.0.14.0 0.0.1.255 10.0.24.0 0.0.1.255
```

```
access-list 107 permit ip 123.123.48.0 0.0.0.255 10.0.24.0 0.0.1.255
# deny all other internal accesses
access-list 107 deny ip 10.0.0.0 0.0.255.255 10.0.24.0 0.0.1.255
# access from the internet
access-list 107 permit ip 0.0.0.0 0.0.0.0 10.0.24.0 0.0.1.255
access-list 107 deny ip any any

access-list 108 permit ip 10.0.12.0 0.0.1.255 10.0.30.0 0.0.1.255
access-list 108 permit ip 10.0.14.0 0.0.1.255 10.0.30.0 0.0.1.255
access-list 108 permit ip 123.123.48.0 0.0.0.255 10.0.30.0 0.0.1.255
access-list 108 deny ip any any

# prevent source address spoofing for external service address space
# other well known internal address spaces 10.0.0.0/8 and
# 192.168.0.0/16 are blocked by default
access-list 112 deny ip 123.123.48.0 0.0.0.255 0.0.0.0 0.0.0.0
access-list 112 permit ip any any

# permit subnets with internet access
access-list 113 permit ip 123.123.48.0 0.0.0.255 0.0.0.0 0.0.0.0
access-list 113 permit ip 10.0.14.0 0.0.1.255 0.0.0.0 0.0.0.0
access-list 113 permit ip 10.0.16.0 0.0.1.255 0.0.0.0 0.0.0.0
access-list 113 permit ip 10.0.18.0 0.0.1.255 0.0.0.0 0.0.0.0
access-list 113 permit ip 10.0.20.0 0.0.1.255 0.0.0.0 0.0.0.0
access-list 113 permit ip 10.0.22.0 0.0.1.255 0.0.0.0 0.0.0.0
access-list 113 permit ip 10.0.24.0 0.0.1.255 0.0.0.0 0.0.0.0
access-list 113 deny ip any any

access-list 110 permit ip any any
access-list 111 deny ip any any

interface vlan 4095
  description Internet
  ip address 0.0.0.0 0.0.0.0
  ip access-group 113 out
  ip access-group 112 in

interface vlan 48
  description external
  ip address 123.123.48.0 255.255.255.0
  ip access-group 100 out
  ip access-group 110 in

interface vlan 463
  description internal
  ip address 10.0.13.254 255.255.254.0
  ip access-group 101 out
  ip access-group 110 in
```

```
123
124 interface vlan 464
125   description admin
126   ip address 10.0.15.254 255.255.254.0
127   ip access-group 102 out
128   ip access-group 110 in
129
130 interface vlan 465
131   description office
132   ip address 10.0.17.254 255.255.254.0
133   ip access-group 103 out
134   ip access-group 110 in
135
136 interface vlan 466
137   description staff_1
138   ip address 10.0.19.254 255.255.254.0
139   ip access-group 104 out
140   ip access-group 110 in
141
142 interface vlan 467
143   description staff_2
144   ip address 10.0.21.254 255.255.254.0
145   ip access-group 105 out
146   ip access-group 110 in
147
148 interface vlan 468
149   description pool
150   ip address 10.0.23.254 255.255.254.0
151   ip access-group 106 out
152   ip access-group 110 in
153
154 interface vlan 469
155   description lab
156   ip address 10.0.25.254 255.255.254.0
157   ip access-group 107 out
158   ip access-group 110 in
159
160 interface vlan 470
161   description hpc_mgt
162   no ip address
163   ip access-group 111 out
164   ip access-group 111 in
165
166 interface vlan 471
167   description hpc_ic
168   no ip address
169   ip access-group 111 out
170   ip access-group 111 in
```

```
171
172 interface vlan 472
173   description slb
174   ip address 10.0.31.254 255.255.254.0
175   ip access-group 108 out
176   ip access-group 110 in
177
178 interface vlan 473
179   description mgt
180   no ip address
181   ip access-group 111 out
182   ip access-group 111 in
183
184 interface vlan 474
185   description san
186   no ip address
187   ip access-group 111 out
188   ip access-group 111 in
189
190 interface vlan 475
191   description vmo
192   no ip address
193   ip access-group 111 out
194   ip access-group 111 in
195
196 interface vlan 476
197   description prt
198   no ip address
199   ip access-group 111 out
200   ip access-group 111 in
201
202 interface vlan 477
203   description cam
204   no ip address
205   ip access-group 111 out
206   ip access-group 111 in
207
208 # binds upstream port to internet vlan
209 interface 1
210   switchport access vlan 4095
```

A.10 Example of the State Shell Interweaving

The following rule set implements the policy from the introductory example in Section 4.1.2 (`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::110/124 -p tcp --dport 80 \
      -j ACCEPT
(5) ip6tables -A FORWARD -s 2001:db8::200/120 -d 2001:db8::100/120 \
      -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:

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

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 moved to the end of our rule list:

$$R_{forward} = \{$$

$$
\begin{aligned}
&r_1 : \{ \\
&\quad (\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
&\quad (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\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{state}, \mathcal{V}_{\texttt{state}}) \\
&\} \rightarrow \mathit{drop}, \\
&r_2 : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
&\quad (\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{state}, \{\texttt{ESTABLISHED}\}) \\
&\} \rightarrow \mathit{accept}, \\
&r_3 : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}), \\
&\quad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\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{state}, \mathcal{V}_{\texttt{state}}) \\
&\} \rightarrow \mathit{accept}, \\
&r_4 : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
&\quad (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\}), \\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}), \\
&\quad (\texttt{state}, \mathcal{V}_{\texttt{state}}) \\
&\} \rightarrow \mathit{accept}, \\
&r_5 : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
&\quad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \{\texttt{2001:db8::100/120}\}), \\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}), \\
&\quad (\texttt{state}, \mathcal{V}_{\texttt{state}}) \\
&\} \rightarrow \mathit{accept}, \\
&r_6 : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
&\quad (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),
\end{aligned}
$$

$$
\begin{aligned}
&(\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&(\texttt{state}, \mathcal{V}_{\texttt{state}})\\
\} &\rightarrow \mathit{drop}\\
\}
\end{aligned}
$$

Next, we collect the state checking rules:

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

1. General Reverse State Shell Derivation

First, we have to subtract the state checking rule (r_2) from the initial rule set. Then, by reversing the directional fields and setting the *backwards* flag we obtain the general reverse state shell:

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

$$
\begin{aligned}
&\} \rightarrow accept,\\
&r_4 : \{\\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&\quad (\texttt{sip}, \{\texttt{2001:db8::110/124}\}), (\texttt{dip}, \mathcal{V}_{\texttt{sip}}),\\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}),\\
&\quad (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\})\\
&\} \rightarrow accept,\\
&r_5 : \{\\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&\quad (\texttt{sip}, \{\texttt{2001:db8::100/120}\}),\\
&\quad (\texttt{dip}, \{\texttt{2001:db8::200/120}\}),\\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \{22\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}),\\
&\quad (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\})\\
&\} \rightarrow accept,\\
&r_6 : \{\\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&\quad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&\quad (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \{1\})\\
&\} \rightarrow drop\\
&\}
\end{aligned}
$$

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:

$$
\begin{aligned}
&S_1^R = \{\\
&\quad r_{(2\cdot 1+1)\cdot 6+1=19} : \{\\
&\quad\quad (\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \{\texttt{eth0}\}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}),\\
&\quad\quad (\texttt{dip}, \{\texttt{2001:db8::0/32}\}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}),\\
&\quad\quad (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),
\end{aligned}
$$

$$
\begin{aligned}
&(\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
&\} \rightarrow \mathit{drop}, \\
&r_{(2\cdot1+1)\cdot6+3=21} : \{ \\
&\quad (\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), \\
&\quad (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}), \\
&\quad (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
&\quad (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
&\} \rightarrow \mathit{accept}, \\
&r_{(2\cdot1+1)\cdot6+4=22} : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), \\
&\quad (\texttt{sip}, \{\texttt{2001:db8::110/124}\}), (\texttt{dip}, \mathcal{V}_{\texttt{sip}}), \\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
&\quad (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
&\} \rightarrow \mathit{accept}, \\
&r_{(2\cdot1+1)\cdot6+5=23} : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
&\quad (\texttt{sip}, \{\texttt{2001:db8::100/120}\}), \\
&\quad (\texttt{dip}, \{\texttt{2001:db8::200/120}\}), \\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \{22\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}), \\
&\quad (\texttt{state}, \{\texttt{ESTABLISHED}\}), (\texttt{back}, \{1\}) \\
&\} \rightarrow \mathit{accept}, \\
&r_{(2\cdot1+1)\cdot6+6=24} : \{ \\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), \\
&\quad (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), \\
&\quad (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), (\texttt{state}, \{\texttt{ESTABLISHED}\}), \\
&\quad (\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:

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

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

$$
\begin{aligned}
B_2 = \{ & \\
& r_{2\cdot2\cdot6+3=27} : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}), \\
& \quad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}), \\
& \quad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}), \\
& \quad (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}}) \\
& \} \rightarrow \mathit{accept}, \\
& r_{2\cdot2\cdot6+4=28} : \{ \\
& \quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), \\
& \quad (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\}), \\
& \quad (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}), \\
& \quad (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}})
\end{aligned}
$$

$$
\begin{aligned}
&\} \rightarrow \mathit{accept},\\
&r_{2\cdot2\cdot6+5=29} : \{\\
&\qquad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&\qquad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}),\\
&\qquad (\texttt{dip}, \{\texttt{2001:db8::100/120}\}),\\
&\qquad (\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}),\\
&\qquad (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}})\\
&\} \rightarrow \mathit{accept},\\
&r_{2\cdot2\cdot6+6=30}\{\\
&\qquad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&\qquad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&\qquad (\texttt{state}, \mathcal{V}_{\texttt{state}}), (\texttt{back}, \mathcal{V}_{\texttt{back}})\\
&\} \rightarrow \mathit{drop}\\
&\}
\end{aligned}
$$

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:

$$
\begin{aligned}
R_S = \{\\
&r_{13} : \{\\
&\qquad (\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&\qquad (\texttt{sip}, \{\texttt{2001:db8::0/32}\}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&\qquad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&\qquad (\texttt{back}, \mathcal{V}_{\texttt{back}})\\
&\} \rightarrow \mathit{drop},\\
&r_{19} :\\
&\qquad (\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \{\texttt{eth0}\}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}),\\
&\qquad (\texttt{dip}, \{\texttt{2001:db8::0/32}\}), (\texttt{proto}, \mathcal{V}_{\texttt{proto}}),\\
&\qquad (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),
\end{aligned}
$$

$$
\begin{aligned}
&(\texttt{back}, \{1\})\\
&\} \rightarrow \mathit{drop},\\
&r_{21} : \{\\
&\quad (\texttt{iif}, \{\texttt{eth0}\}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}), (\texttt{sip}, \mathcal{V}_{\texttt{dip}}),\\
&\quad (\texttt{dip}, \{\texttt{2001:db8::200/120}\}),\\
&\quad (\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&\quad (\texttt{back}, \{1\})\\
&\} \rightarrow \mathit{accept},\\
&r_{22} : \{\\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{oif}}), (\texttt{oif}, \mathcal{V}_{\texttt{iif}}),\\
&\quad (\texttt{sip}, \{\texttt{2001:db8::110/124}\}), (\texttt{dip}, \mathcal{V}_{\texttt{sip}}),\\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \{80\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}),\\
&\quad (\texttt{back}, \{1\})\\
&\} \rightarrow \mathit{accept},\\
&r_{23} : \{\\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&\quad (\texttt{sip}, \{\texttt{2001:db8::100/120}\}),\\
&\quad (\texttt{dip}, \{\texttt{2001:db8::200/120}\}),\\
&\quad (\texttt{proto}, \{6\}), (\texttt{sport}, \{22\}), (\texttt{dport}, \mathcal{V}_{\texttt{sport}}),\\
&\quad (\texttt{back}, \{1\})\\
&\} \rightarrow \mathit{accept},\\
&r_{24} : \{\\
&\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}, \{1\})\\
&\} \rightarrow \mathit{drop},\\
&r_{27} : \{\\
&\quad (\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \{\texttt{eth0}\}),\\
&\quad (\texttt{sip}, \{\texttt{2001:db8::200/120}\}), (\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{accept},\\
&r_{28} : \{
\end{aligned}
$$

$$
\begin{aligned}
&(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&(\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \{\texttt{2001:db8::110/124}\}),\\
&(\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{80\}),\\
&(\texttt{back}, \mathcal{V}_{\texttt{back}})\\
\} \rightarrow \mathit{accept},\\
r_{29} : \{\\
&(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}),\\
&(\texttt{sip}, \{\texttt{2001:db8::200/120}\}),\\
&(\texttt{dip}, \{\texttt{2001:db8::100/120}\}),\\
&(\texttt{proto}, \{6\}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \{22\}),\\
&(\texttt{back}, \mathcal{V}_{\texttt{back}})\\
\} \rightarrow \mathit{accept},\\
r_{30}\{\\
&(\texttt{iif}, \mathcal{V}_{\texttt{iif}}), (\texttt{oif}, \mathcal{V}_{\texttt{oif}}), (\texttt{sip}, \mathcal{V}_{\texttt{sip}}), (\texttt{dip}, \mathcal{V}_{\texttt{dip}}),\\
&(\texttt{proto}, \mathcal{V}_{\texttt{proto}}), (\texttt{sport}, \mathcal{V}_{\texttt{sport}}), (\texttt{dport}, \mathcal{V}_{\texttt{dport}}),\\
&(\texttt{back}, \mathcal{V}_{\texttt{back}})\\
\} \rightarrow \mathit{drop}\\
\}
\end{aligned}
$$

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

A.11 Policy Specification of the UP Benchmark

In the UP policy file appear 71 different roles. Most of them represent sub-organizations, others stand for typical network components like a DMZ or a Perimeter Gateway Firewall (PGF). The corresponding `iptables` rule set consists of 1,035 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
```

A.12 Canonical IPv6 Firewall Ruleset Generation

This IPTables script from [66] incorporates best practices for IPv6 security and has been used as a blueprint for IPv6 rule sets throughout this thesis – particularly in the UP-Benchmark.

```
#!/bin/bash
# --- variable definitions -------------------------
PATH=/bin:/sbin:/usr/bin:/usr/sbin

# services that may be accessed publicly
TCP_SERVICES="22 80"
```

```
UDP_SERVICES=" "

# services that may be accessed privately
LOCAL_TCP_SERVICES=" "
LOCAL_UDP_SERVICES=" "
LOCAL_NETWORK=" "

# --- check ip6tables ------------------------------
if ! [ -x /sbin/ip6tables ]; then
    exit 0
fi

# --- set default policy: drop all ---------------
ip6tables -X
ip6tables -F
ip6tables -t filter -F
ip6tables -t mangle -F
ip6tables -P INPUT DROP
ip6tables -P FORWARD DROP
ip6tables -P OUTPUT DROP

# --- input ---------------------------------------
# --- allow loopback traffic
ip6tables -A INPUT -i lo -j ACCEPT

# --- allow icmp traffic
ip6tables -A INPUT -p icmpv6 -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type destination-unreachable \
    -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type packet-too-big -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type time-exceeded -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type parameter-problem \
    -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type echo-request -m limit \
    --limit 900/min -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type echo-reply -m limit \
    --limit 900/min -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type neighbour-solicitation \
    -j ACCEPT
ip6tables -A INPUT -p icmpv6 --icmpv6-type neighbour-advertisement \
    -j ACCEPT

# --- allow connection with state established or
# --- related
ip6tables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

# --- allow tcp connection
if [ -n "${TCP_SERVICES}" ] ; then
```

```
  for PORT in ${TCP_SERVICES}; do
    ip6tables -A INPUT -p tcp --dport ${PORT} -j ACCEPT
  done
fi
if [ -n "${LOCAL_TCP_SERVICES}" ] ; then
  for PORT in ${LOCAL_TCP_SERVICES}; do
    ip6tables -A INPUT -p tcp --src ${LOCAL_NETWORK} \
        --dport ${PORT} -j ACCEPT
  done
fi

# --- allow udp connection
if [ -n "${UDP_SERVICES}" ] ; then
  for PORT in ${UDP_SERVICES}; do
    ip6tables -A INPUT -p udp --dport ${PORT} -j ACCEPT
  done
fi
if [ -n "${LOCAL_UDP_SERVICES}" ] ; then
  for PORT in ${LOCAL_UDP_SERVICES}; do
    ip6tables -A INPUT -p udp --src ${LOCAL_NETWORK} \
        --dport ${PORT} -j ACCEPT
  done
fi

# --- output -------------------------------------
# --- allow loopback traffic
ip6tables -A OUTPUT -o lo -j ACCEPT

# --- allow icmp traffic
ip6tables -A OUTPUT -p icmpv6 -j ACCEPT

# --- allow connection with state
# --- new, established or related
ip6tables -A OUTPUT -m state --state NEW,ESTABLISHED,RELATED \
    -j ACCEPT
# --- forward ------------------------------------
# --- see scripts
ip6tables -A FORWARD -p icmpv6 \
    --icmpv6-type destination-unreachable -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type packet-too-big \
    -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type echo-request -m limit \
    --limit 900/min -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type echo-reply -m limit \
    --limit 900/min -j ACCEPT
ip6tables -A FORWARD -p icmpv6 \
    --icmpv6-type ttl-zero-during-transit -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type unknown-header-type \
```

```
    -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type unknown-option \
    -j ACCEPT

# special routing header chain
# for routing header type 0, allow only packets that
# have the 'segments left' field set to 0
# for routing header type 2, allow only packets that
# have the 'segments left' field set to 1
# for all other header types, allow only packets that
# have the 'segments left' field set to 0
# (default policy for all custom chains is RETURN)
ip6tables -N routinghdr
ip6tables -A routinghdr -m rt --rt-type 0 ! --rt-segsleft 0 -j DROP
ip6tables -A routinghdr -m rt --rt-type 2 ! --rt-segsleft 1 -j DROP
ip6tables -A routinghdr -m rt --rt-type 0 --rt-segsleft 0 -j RETURN
ip6tables -A routinghdr -m rt --rt-type 2 --rt-segsleft 1 -j RETURN
ip6tables -A routinghdr -m rt ! --rt-segsleft 0 --j DROP
ip6tables -A FORWARD -m ipv6header --header ipv6-route --soft \
    -j routinghdr

ip6tables -A FORWARD -p tcp --dport 80 -j ACCEPT
ip6tables -A FORWARD -p udp --dport 80 -j ACCEPT

# --- allow connection with state established or
# --- related
ip6tables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT

# --- block and log dropped packages
#ip6tables -A FORWARD -j LOG --log-prefix "DROPPED Packages:"
#ip6tables -A FORWARD -j DROP

echo "1" > /proc/sys/net/ipv6/conf/all/forwarding
```

A.13 Generated IPTables Rule Set

An IPTables rule set that was generated for the example FPL policy from Chapter 4.

```
iptables -P FORWARD DROP
iptables -A FORWARD -m conntrack --ctstate ESTABLISHED -j ACCEPT
ip6tables -P FORWARD DROP
ip6tables -A FORWARD -i eth1 -s 2001:db8::200/120 -j DROP
ip6tables -A FORWARD -i eth1 -s 2001:db8::100/120 -j DROP
ip6tables -A FORWARD -p icmpv6 --icmpv6-type destination-unreachable \
    -j ACCEPT
```

```
ip6tables -A FORWARD -p icmpv6 --icmpv6-type packet-too-big -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type echo-request -m limit \
    --limit 900/min -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type echo-reply -m limit \
    --limit 900/min -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type ttl-zero-during-transit \
    -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type unknown-header-type \
    -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type unknown-option -j ACCEPT
ip6tables -N routinghdr
ip6tables -A routinghdr -m rt --rt-type 0 ! --rt-segsleft 0 -j DROP
ip6tables -A routinghdr -m rt --rt-type 2 ! --rt-segsleft 1 -j DROP
ip6tables -A routinghdr -m rt --rt-type 0 --rt-segsleft 0 -j RETURN
ip6tables -A routinghdr -m rt --rt-type 2 --rt-segsleft 1 -j RETURN
ip6tables -A routinghdr -m rt ! --rt-segsleft 0 --j DROP
ip6tables -A FORWARD -m ipv6header --header ipv6-route --soft \
    -j routinghdr
ip6tables -A FORWARD -m conntrack --ctstate ESTABLISHED -j ACCEPT
ip6tables -A FORWARD -i eth1 --protocol tcp --dport 80 \
    -d 2001:db8::110/124 -m conntrack --ctstate NEW \
    -m comment --comment "Internet to WebServer" -j ACCEPT
ip6tables -A FORWARD --protocol tcp --dport 80 -s 2001:db8::200/120 \
    -d 2001:db8::110/124 -m conntrack --ctstate NEW \
    -m comment --comment "Office to WebServer" -j ACCEPT
ip6tables -A FORWARD --protocol tcp --dport 22 -s 2001:db8::200/120 \
    -d 2001:db8::110/124 -m conntrack --ctstate NEW \
    -m comment --comment "Office to WebServer" -j ACCEPT
ip6tables -A FORWARD -s 2001:db8::200/120 -d 2001:db8::200/120 \
    -m conntrack --ctstate NEW \
    -m comment --comment "Office to Office" -j ACCEPT
ip6tables -A FORWARD -s 2001:db8::110/124 -d 2001:db8::110/124 \
    -m conntrack --ctstate NEW \
    -m comment --comment "WebServer to WebServer" -j ACCEPT
```

A.14 Modeling Application Layer Gateways

We have explored the modeling of an *Application Layer Gateway* (ALG, cf. [137, Sec. 2.9]) and implemented a prototype thereof. Despite the fact that the prototype works well, we encountered different open challenges that restrained us from including a thorough evaluation in the main corpus of this thesis. The most prevalent issue is the lack of application specific policy specification capabilities in FPL because these are necessary for precise modeling and verification. Hence, our model and

implementation provides highly abstracted capabilities that need to be detailed in the future in order to better match the ALGs security processing semantics. In the remainder of this section, we explain how ALGs work and describe our ALG model as well as its prototype implementation.

Typically, ALGs are deployed in a *P-A-P* network perimeter architecture [55] where the ALG – the *A* in P-A-P – is enveloped by two packet filters – the *P*s in P-A-P. These packet filters handle lower layer traffic – particularly up to the transport layer – and the ALG concentrates on the layers above with an emphasis on the application layer. ALGs can be implemented on top of *network proxies* and serve as transparent endpoints towards parts of the application, e.g., the client or the server. For instance, assuming that the clients' application interactions should be secured, the proxy accepts connections from the clients on a listening socket and, afterwards, connects to the application's server with a separate network connection using another socket. Then, the application stream runs through the proxy's user space and may be inspected, filtered, or alternated by application specific security functions called *relays*. An example would be the inspection of SMTP traffic using a malware detection engine.

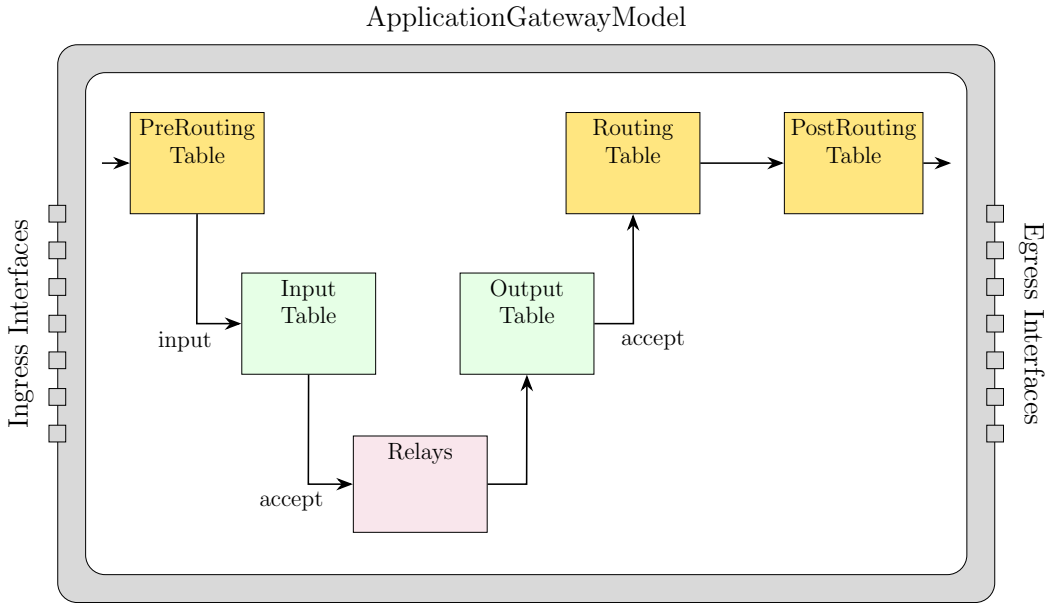


Fig. A.7.: Model of a proxy-based application layer gateway pipeline following the P-A-P structure. The green tables are configured using packet filter configuration, the yellow tables contain static rules or are derived from device configuration, and the red table is determined at runtime.

As shown in Figure A.7, we build our model of an ALG upon a stateless packet filter model. We chose the statelessness due to its simplicity as a first step. Since proxies

work in a stateful manner, a future step would be to switch to the stateful packet filter model.

In comparison to our packet filter model, there are two notable differences: The *Relay* table and the lack of a *Forward* table. We omitted forwarding in order to force traffic to flow over the relays as required by the characteristics of network proxies. The Relay table is used to mimic the deep inspections of the ALG. Currently, we do not do any inspection at all. Instead, we offer the instrumentation of connection snapshots. An entry consists of two TCP quintuples – one for the *client-to-proxy* connection and one for the *proxy-to-server* connection. We then instrument two rules in the Relay table which operate in the forth resp. back direction. For instance, the rule in forth direction works as follows:

```
match: (<client-to-proxy-quintuple>),
rewrite: (<proxy-to-server-quintuple>),
action: (accept)
```

The rule matches packets from the client to the proxy, rewrites their source and destination according to the proxy-to-server quintuple, and forwards them towards their new destintion, i.e., the server. The rule in backwards direction works similarly but matches the server-to-proxy quintuple, i.e., the reversed proxy-to-server quintuple, and rewrites to the proxy-to-client quintuple, i.e., the reversed client-to-proxy quintuple.

For now, we limited our model to correctly mimic the ALG's behaviour on the network level. In the future, it is necessary to improve the model with regard to application specific capabilities of an extended FPL, the change of the underlying packet filter model towards stateful filtering with state deduction, and efficient modeling of complex relays, e.g., by adapting state shell interweaving for the encoding and analysis of finite automata.

Abbreviations

AAD Automatic Anomaly Detection. 7, 8, 180, 195

ABAC Attribute-based Access Control. 223

ACL Access Control List. 54, 139, 141, 167

AI Artificial Intelligence. 208

ALG Application Layer Gateway. 207

AP Authorized Permissions. 226, 228, 230

AP Atomic Predicate. 55

API Application Programming Interface. 149

AR Attribute Relation. 91

AST Abstract Syntax Tree. 110, 116

AU Authorized Users. 226, 228

BDD Binary Decision Diagram. 46, 48, 56, 185, 191

BGP Border Gateway Protocol. 54, 56

BSI German Federal Office for Infomation Security. 2, 17

CC Common Criteria. 18

CRM Customer Relationship Management. 2

CSP Constraint Solving Problem. 48

CTL Computational Tree Logic. 53

CV Compliance Verification. 7

DAC Discretionary Access Control. 19, 222, 223

DAG Directed Acyclic Graph. 91, 171, 227

DHSA Domain-oriented Header Space Analysis. ix, xvii, 12, 15, 79, 86, 112, 118, 135, 191, 193, 205

DHSA Domain-oriented Header Space Algebra. ix, xii, 80, 86, 205

DMZ Demilitarized Zone. 61, 91, 95, 111, 136, 165, 173, 186, 225, 228

DNAT Destiation Network Address Translation. 33

DNS Domain Name System. 61, 74, 82

DSD Dynamic Separation of Duties. 224

eBNF Extended Backus-Naur Form. 247

eBPF Extended Berkeley Packet Filter. 161

EC Equivalence Class. 54

ERP Enterprise Resource Planning. 8

FaVe Fast Verification Framework. ix, xii, 12, 45, 59, 80, 107, 135, 180, 205

FDD Firewall Decision Diagram. 48, 191

FIB Forwarding Information Base. 54

FML Flow Management Language. 73, 76, 77

FPL FaVe Policy Language. ix, xii, 11, 59, 91, 112, 135, 141, 164, 170, 205, 208

FTP File Transfer Protocol. 74, 161

HOL Higher Order Logic. 48

HSA Header Space Analysis. ix, xii, 12, 20, 46, 79, 98, 112, 118, 185, 205

HSA Header Space Algebra. ix, xii, 20, 21, 27, 205

HTTP Hypertext Transfer Protocol. 63, 67, 74, 82, 95, 151, 225

HTTPS Hypertext Transfer Protocol Secure. 61, 74

IANA Internet Assigned Numbers Authority. 41

ICMP Internet Control Message Protocol. 41, 159

ICMPv6 Internet Control Message Protocol Version 6. 165, 171

IDS Intrusion Detection System. 46, 52, 76

IEC International Electrotechnical Commission. 2, 74

IFI University of Potsdam, Department of Computer Science. 141

IGP Interior Gateway Protocol. 56

IP Internet Protocol. 8, 11, 22, 33, 40, 60, 62, 71, 77, 81, 86, 88, 113, 123, 136, 139, 142, 144, 149, 173, 198

IPS Intrusion Prevention System. 51, 76

IPsec Internet Protocol Security. 14, 207

IPv4 Internet Protocol version 4. 12, 40, 169, 198, 202

IPv6 Internet Protocol version 6. ix, xii, xv, 10, 12, 40, 47, 56, 57, 169, 171, 198, 202, 205

ISF Information Security Forum. 2

ISM Information Security Management. viii, xi, 2, 17, 208

ISMS Information Security Management System. 2

ISO International Organization of Standards. 2

ISP Internet Service Provider. 63

LLM Large Language Model. 208

LTL Linear Time Logic. 52

MAC Mandatory Access Control. 222, 223

MAC Medium Access Code. 136

MPLS Multiprotocol Layer Switching. 15, 56

NAT Network Address Translation. 14, 77, 235

NFV Network Function Virtualization. 8

NGAC Next Generation Access Control. 223

NIST US National Institute for Standards in Technology. 2, 223, 224, 229

OS Operation System. 10

OSI Open Systems Interconnection. 144

OSPF Open Shortest Path First. 56

PA Permission Assignments. 224, 225, 230

PDCA Plan-Do-Check-Act. viii, xi, 2, 206

PGF Perimeter Gateway Firewall. 111

QoS Quality of Service. 77

RBAC Role-based Access Control. 19, 222, 223, 229

RFC Request for Comments. 41, 129

RH Role Hierarchy. 224, 230

RTP Real-time Transport Protocol. 161

SAT Boolean Satisfiability Problem. 9, 10, 46, 48, 56, 185, 191

SCADA Supervisory Control and Data Acquisition. 74

SDN Software Defined Networking. 8, 47, 53, 73

SEFL Symbolic Execution Friendly Language. 50

SFP Security Function Policy. 18

SFR Security Functional Requirements. 18

SIP Session Initiation Protocol. 161

SMT Satisfiability Modulo Theories (a class of theorem proving techniques). 48

SoD Separation of Duties. 228, 229

SR Service Relation. 91

SR Session Roles. 224, 230

SSD Static Separation of Duties. 224

SSH Secure Shell. 61, 63, 67, 68, 77, 95, 151, 153, 225, 228

STL Spatio Temporal Logics. 188, 191, 200

TCP Transmission Control Protocol. 41, 61, 63, 66, 148, 149, 158, 168

TLS Transport Layer Security. 14, 207

TOE Target of Evaluation. 18

TSF Target of Evaluation Security Functionality. 18

TTL Time to Live. 47, 186, 238, 239

TUM Technical University of Munich. 49, 163, 168, 199

UA User Assignments. 224, 225, 229

UDP User Datagram Protocol. 41, 61, 159

UP University of Potsdam. 10, 162, 164, 200

US User Sessions. 224, 230

VLAN Virtual Local Area Network. 11, 40, 56, 60, 62, 64, 113, 136, 137, 142, 169, 235

VXLAN Virtual Extensible Local Area Network. 235

WAF Web Application Firewall. 207

XACML eXtensible Access Control Markup Language. 223

ZTN Zero Trust Networking. 206

Definitions

Correlation Anomaly Two rules correlate if they overlap, i.e., they match a common set of packets, have different actions, and the higher prioritized rule does not shadow the other. 184

Firewall Anomaly Firewall anomalies occur if two or more filtering rules match the same packet. 180

Generalization Anomaly A rule is generalized by one or more rules with lower priorities if these handle the same packets. 182

High-level Policy Language A high-level policy language enables an abstract, concise, unambigious, and expressive specification of entities and their interactions. 59

Network Inventory A network inventory is a collection of data for network devices and their components managed by a specific management system. 8

Network Security Policy Network security policies specify the conditions of communication between subjects and objects. 19

Redundancy Anomaly Two rules are redundant if they handle the same packets and have the same action. 184

Security Compliance Security compliance is a state where organizational security requirements and their technical enforcement align. 3

Security Configuration The security configuration denotes a concrete implementation of the security policy. 19

Security Policy Security policies are sets of rules that govern the interaction of entities on the organizational level. 19

Security Specification The security specification denotes a formally specified version of the security policy. 19

Shadowing Anomaly A rule is shadowed if all packets that are relevant for this rule, i.e., that the rule matches, are processed by one rule or a combination of rules with higher priorities. 181

Unreachability Anomaly A rule is unreachable if all packets have been processed by rules with higher priorities. 183

Index

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `https://github.com/derric/cleanthesis`.