Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Provably Secure Networks: Methodology and Toolset for Configuration Management

Cornelius Diekmann

Dissertation

Network Architectures and Services NET-2017-07-2

TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste

Provably Secure Networks: Methodology and Toolset for Configuration Management

Cornelius Hermann Diekmann

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Tobias Nipkow, Ph.D.
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. Georg Carle
	2.	Prof. Steven M. Bellovin (Columbia University)

Die Dissertation wurde am 28.03.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.07.2017 angenommen.

Cataloging-in-Publication Data
Cornelius Diekmann
Provably Secure Networks: Methodology and Toolset for Configuration Management
Dissertation, Juli 2017
Network Architectures and Services, Department of Computer Science
Technische Universität München

ISBN 978-3-937201-57-3
ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)
DOI 10.2313/NET-2017-07-2
Network Architectures and Services NET-2017-07-2
Series Editor: Georg Carle, Technische Universität München, Germany
© 2017, Technische Universität München, Germany

Abstract

Network management and administration is an inherently complex task, in particular when it comes to security. Configuration complexity in this domain leads to human error, which is often only uncovered when it is too late: after a successful attack.

This thesis focuses on the security of network configurations, i.e., network-level access control and network-level information flow security. The objective is to employ formal methods to prevent, uncover, and prove lack of security-related configuration errors. We contribute methods and tools to translate between security components on various abstraction levels and to verify their conformance. We prove correctness of our tools with the Isabelle interactive proof assistant.

First, we propose a method to construct new networks from scratch. We present our tool *topoS* which enables automation of the design, requiring only a specification of the security requirements. Second, we present a method to understand and analyze existing network security device configurations, focusing on the iptables firewall. We present our fully automated tool *fffuu* for this task. Finally, we show how both approaches can interact with each other.

Our experience has shown that a solution to the presented problems must be usable, must not expose over-formalism to the administrator, must leave the administrator in full low-level control, must support legacy configurations, and must be non-invasive, i.e., must not require that an administrator completely relinquishes control to a tool. We demonstrate that our proposed tool-supported methodology fulfills its goals as follows: By its very nature, access control lists scale quadratically in the number of networked entities or roles. We propose a methodology to specify security requirements which can scale better than linear (depending on its usage). Our methodology works on well-defined intermediate results and gives the administrator full control over them. A policy computed by this approach can be deployed to a network directly. Or, our methodology can be used completely non-invasive: It can statically verify that an existing iptables ruleset conforms to the policy and requirements. In general, we provide a method to compute a clear overview of the policy enforced by an existing (legacy) iptables firewall. Both directions (synthesizing new policies vs. verifying existing policies) are compatible with each other and an administrator may freely choose to which extent she wants to migrate to our methodology and to which extent she wants to remain in full low-level control. Ultimately, it is possible to use our toolset in a full circle.

We evaluated our tools, among others, on an aircraft cabin data network, Android measurement app, and on the largest collection of public, real-world iptables dumps (made available by us). We showed further applicability in the domain of microservice management, SDN configuration, cyber physical systems, software architectures, and privacy.

Kurzfassung

Administrierung und Management eines Netzwerkes ist eine inhärent komplexe Aufgabe, insbesondere im Hinblick auf Security. Konfigurationskomplexität führt zu menschlichem Versagen, welches erst erkannt wird, wenn es zu spät ist: nach einem erfolgreichen Angriff.

Diese Dissertation beschäftigt sich mit der Sicherheit von Netzwerkkonfigurationen, d.h. Access Control und Information Flow Security auf Netzwerkebene. Das erklärte Ziel ist es formale Methoden einzusetzen, um sicherheitsrelevante Konfigurationsfehler zu verhindern, erkennen und deren Abwesenheit zu beweisen. Wir stellen Tools und Methoden bereit, um zwischen Sicherheitskomponenten auf verschiedenen Abstraktionsebenen zu übersetzen und deren Konformität zu verifizieren. Wir beweisen die Korrektheit unserer Tools mit dem interaktiven Theorembeweiser Isabelle.

Im ersten Teil der Arbeit schlagen wir eine Methode vor, um Netzwerke von Grund auf neu zu designen. Wir präsentieren unser Tool *topoS*, welches diesen Designprozess automatisiert und dafür nur eine Spezifikation der Sicherheitsanforderungen benötigt. Im zweiten Teil stellen wir eine Methode vor, um bestehende Netzwerksicherheitsgerätekonfigurationen zu verstehen und zu analysieren. Dabei fokussieren wir uns auf die iptables Firewall und stellen unser automatisiertes Tool *fffuu* vor. Im finalen Teil der Arbeit zeigen wir, wie beide Ansätze ineinandergreifen.

Unsere Erfahrung hat gezeigt, dass Lösungen, die den genannten Problemen gerecht werden wollen, benutzbar sein müssen, dem Administrator keine Überformalisierung aussetzen dürfen, dem Administrator low-level Kontrolle zugestehen müssen, Legacy-Konfigurationen unterstützen müssen und nicht invasiv sein dürfen, d.h. dass sie nicht fordern dürfen, dass ein Administrator komplett die Kontrolle an ein Tool abgibt. Wir zeigen, dass unser vorgeschlagener, toolgestützter Ansatz diese Ziele wie folgt erfüllt: Es liegt in der Natur der Sache, dass Access Control Listen quadratisch mit der Anzahl der Geräte bzw. Rollen skalieren. Wie schlagen eine Methode vor, die es erlaubt Sicherheitsanforderungen zu spezifizieren, welche besser als linear skaliert (abhängig von der Nutzung). Unsere Methode arbeitet auf wohldefinierten Zwischenergebnissen und überlässt dem Administrator die volle Kontrolle über diese. Eine so berechnete Policy kann direkt in einem Netzwerk ausgerollt werden, oder unsere Methode kann komplett nicht-invasiv eingesetzt werden: Die Übereinstimmung existierender iptables Regelsätze mit der Policy oder den Anforderungen kann statisch überprüft werden. Im Allgemeinen stellen wir eine Methode vor, um eine Übersicht über die Policy zu berechnen, welche eine (legacy) iptables Firewall umsetzt. Beide Richtungen (neue Policies synthetisieren vs. existierende Policies verifizieren) sind untereinander kompatibel und es obliegt dem Administrator zu entscheiden, zu welchem Grad sie auf unsere Methode umstellen möchte und zu welchem Grad sie die komplette low-level Kontrolle behalten möchte. Es ist möglich unsere Tools iterativ einzusetzen.

Wir haben unsere Tools unter Anderem in einem Flugzeugkabinennetzwerk, einer Android Messapp und der größten öffentlichen iptables Kollektion (die von uns bereitgestellt wurde), getestet. Wir zeigten weitere Anwendbarkeit im Bereich des Microservicemanagement, SDN Konfiguration, Cyberphysicalsystems, Softwarearchitekturen und Privacy.

Contents

Chapter 1

Problem Statement & Goals

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

E. W. Dijkstra, On the nature of Computing Science (1984) [1].

1.1 Introduction

Network administration is a challenging task and requires competent network administrators. Handling user complaints, improving performance, reacting to hardware failures, account management, low-level troubleshooting, complying with high-level corporate policies, standard conformance, . . . , and security are among the daily tasks of a network administrator [2, 3]. Traditionally, network management is a low-level, manual, ad-hoc task. It is said that our networks are kept running by "Masters of Complexity" [4, 5]. Yet, "controlling complexity is a core problem in information security" [6]. Unsurprisingly, security issues exist in many networks [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17].

Network segmentation, isolation, and controlled access are the fundamental building blocks for the baseline security of a computer network [18, cf. B 4.1, M 5.111]. Administrating these security-related aspects of a network is a highly complex task. Human error, in particular configuration errors, are a central cause for network problems [10, 19, 20]. Configuration errors which lead to security problems are sometimes attributed to the (accidental) complexity of the low-level languages which are used to configure network security mechanisms [21, 22]. For example, the default Linux firewall iptables [23] features more than 200 matching features [24]. The firewall configuration language cannot be simplified by removing features because they are actively used [25, 26, 27]. In general, administrators need low-level control over their rulesets since, often, performance and other network-related issues apart from security must also be implemented in a firewall ruleset.

In addition, legacy configurations of enormous complexity have evolved over time. For example, the iptables firewall is over ten years old and there are also rulests of that age which are still deployed on core firewalls but are no longer understood by the administrator [25]. Even though simple, high-level languages for network configuration have been proposed [28, 29, 30, 31, 32, 33, 21, 14, 34, 35, 36], the question of how to deal with legacy configuration often remains unanswered.

In this thesis, we address research questions about the security of network configurations. We focus on network-level access control and network-level information flow security. The declared goal of this thesis is to provide means to help administrators to increase the security of their network configuration. We begin by designing a high-level language for security requirements which can be translated in several steps to configurations for network security mechanisms, e.g., iptables. This process is unique in that it still allows low-level control for the administrator as well as guaranteeing soundness. In the second part, we take the opposite direction and translate legacy iptables firewall configurations of enormous complexity to a simplified, high-level view.

• *The ipta(b)les – There and Back Again* •

The complete formal theory, as well as executable tools, have been machine-verified with the interactive theorem prover Isabelle/HOL [37]. Several contributions to the archive of formal proofs have been made [38, 39, 40, 41, 42, 43]. During his research, the author advanced the state of the art, both in the world of formal methods [44] as well as in the world of computer networks [45]. While the theoretical work is "substantial" [46] and "shiny" [47], the practical applicability has also been demonstrated to hundreds of hackers [48][1] [49, 50, 51].

1.2 Research Objectives

The declared goal of this thesis is to improve the situation in the field of network security administration. At the end of the day, in order to minimize attack surface, we want an answer to the question *"Which machines should be allowed to speak to each other?"* and we want to know whether the answer to the question is also practically enforced. To put this overall goal statement in concrete terms, we first present a model of security components and afterwards split the overall goal into several research questions according to the model.

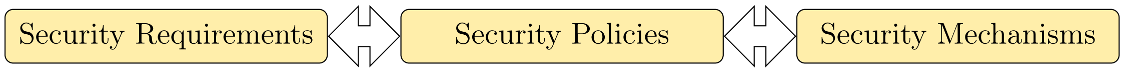


Figure 1.1: Security Components[2]

Security Components Security can be divided into three components [52], as illustrated by Figure 1.1: The *security requirements* specify on a high level of abstraction the scenario-specific security goals. The *security policy* specifies rules which implement the requirements. Finally, the *security mechanisms* enforce the policy; requiring low-level configuration.

> **Example.** We can imagine the security requirements as a text document written in natural language. The security policy could be expressed by an access control matrix. A firewall, the security mechanism, can be configured with a ruleset to implement the desired policy.

[1]Around 500 people attended the talk on-site; as of November 2016, the video recording of the talk has over 4000 views.

[2]The image was inspired by Bishop [52] and first appeared in the author's master's thesis [53]. Since 2013, it is also used in the lecture "Network Security" at TUM.

Security problems arise if, on the one hand, the components are not consistent with each other, e. g., a policy does not correctly reflect some security requirements or a security mechanism is misconfigured and does not implement the policy. On the other hand, security problems may also arise if the specification of the security requirements does not express the desired security properties.

Research Questions The overall research question is

"How can we provide means to help the administrator to configure secure networks and verify the security of existing network configurations?"

Given the model of security components and the scope of this thesis, a secure system must fulfill two properties: First, the security requirements must express the desired security properties and, second, the three components must be consistent with each other. We consider the notion of *"secure"* in the overall question by these two aspects. The posed question contains further aspects as it asks about developing new configurations vs. analyzing existing configurations. We further divide the question into these aspects. This yields the following two questions: First, we ask the question (**Q1**) *"How can we design secure networks from scratch?"*. Second, we ask the question (**Q2**) *"How can we analyze and verify existing configurations?"*.

Finally, we need to consider the last aspect of the overall question *"How can we provide means to help the administrator?"*. This last aspect corresponds to additional, generic, non-functional quality requirements which restrain the possible outcome of **Q1** and **Q2**. Hence, we will state them first.

We divide the non-functional requirements (**NF**) into the following aspects:

NF1 *Can we provide automated tools for the solutions to **Q1** and **Q2**?*

A theory or abstract process which answers **Q1** and **Q2** is helpful from a scientific point of view. However, to actually help administrators, working tools are required [17].

NF2 *Can the correctness of the tools be justified?*

For a tool to be useful, it must be trustworthy. In particular, if security-critical decisions and processes are offloaded to a tool, its correctness is crucial. Therefore, we require a formal, machine-verifiable correctness proof of our tools and, consequently, the theory they are built upon.

NF3 *Is over-formalism exposed to the administrator?*

Possible tools must usable. While the focus of this thesis is not on user studies and usability, by evaluating related work, we discovered anecdotally that tools which expose an excessive amount of formalism are easily rejected by our administrator.

NF4 *Are the solutions to **Q1** and **Q2** compatible?*

A framework which takes away low-level control from the administrators and takes control over config files is not desired. In particular, it is generally inadvisable to touch an administrator's configuration [54], and administrators need the possibility to manually apply low-level modifications to configurations. Therefore, it must be possible to go back and forth between the solutions to **Q1** and **Q2**. For example, it must be possible that an administrator makes low-level changes to rules which are generated by high-level requirements and it must be verified again that the low-level changes do not violate high-level requirements. In different scenarios, rules generated from high-level requirements must co-exist with legacy rules without negative security implications.

We now detail on the first question (**Q1**) *"How can we design secure networks from scratch?"*. This corresponds to the left-to-right direction of Figure 1.1. We divide it into the following aspects:

Q1.1 *How can the security requirements be specified?*

A language to specify security requirements is required. For the definition of *"secure"*, some means for the administrator to check that the specified requirements express the desired meaning is necessary. A solution which also satisfies **NF3** must expose low manual configuration overhead and little formalism to the administrator.

Q1.2 *How can a security policy be derived from the requirements?*

To satisfy **NF1**, a process which is completely automatic is required. In addition, to satisfy **NF4**, it should also be possible to verify a policy w.r.t. the requirements.

Q1.3 *How can a policy be deployed to real network security mechanisms?*

Also for this step, to satisfy **NF1**, a process which is completely automatic is required. The model assumptions which need to be fulfilled by the real-world security mechanism to enforce the policy need to be explicitly stated. Different possible choices for security mechanisms need to be evaluated, e.g., firewalls, OpenFlow-enabled switches, and containers.

We now detail on the second question (**Q2**) *"How can we analyze and verify existing configurations?"*. This corresponds to the right-to-left direction of Figure 1.1. We divide it into the following aspects:

Q2.1 *What are the semantics of a security mechanism?*

The behavior of a network security mechanism needs to be described. This behavior can be very complicated as the example of iptables shows. Yet, to fulfill **NF2**, a precise and formal model about the low-level behavior is required.

Q2.2 *How does an entity in a security mechanism configuration correspond to an entity in a policy?*

A policy may use symbolic names for entities whereas an entity in a security mechanism is usually identified by a network address. Network addresses, e.g., IP addresses, can be easily spoofed whereas a symbolic name in a policy is assumed to genuinely name an entity. Therefore, to lift raw network addresses as they occur in a mechanism's configuration to entities in a policy, in an additional step, it must be ensured that addresses cannot be spoofed.

Q2.3 *How can a high-level policy be derived from a low-level security mechanism configuration?*

Given a low-level configuration of a security mechanism, a high-level policy which abstracts over all unnecessary low-level details needs to be derived. For example, given an iptables ruleset with its over 200 different matching features and complex chain semantics, it needs to be simplified to a simple access control matrix.

Q2.4 *Can a derived high-level policy be verified w.r.t. a given set of security requirements?*

This question does not ask about deriving the requirements from a policy, since this process is not possible without guessing the intention of a policy author. Because of **NF2**, we refrain from guessing. Note that **Q1.2** has been strengthened such that a successful answer to it must already entail an answer to this question. We ask this question to ultimately ensure that the required answers to **Q1** and **Q2** do not exist in isolation, but must be compatible in both directions.

1.3 Structure of this Thesis

This thesis is structured to follow the research questions. Question **Q1** is answered in Part I and question **Q2** is answered in Part II. We conclude, demonstrate applicability, and combine the answers to both questions in Part III.

Part I We answer **Q1** by contributing a method to specify security requirements with low manual configuration effort and present the first fully verified translation of high-level security requirements to low-level security mechanism configurations.

Part II We answer **Q2** by contributing the first fully verified tool to analyze existing iptables filtering rules which understands all match conditions and can extract a high-level policy overview.

Part III We demonstrate the interplay of our answers to **Q1** and **Q1**, summarize applicability, and conclude.

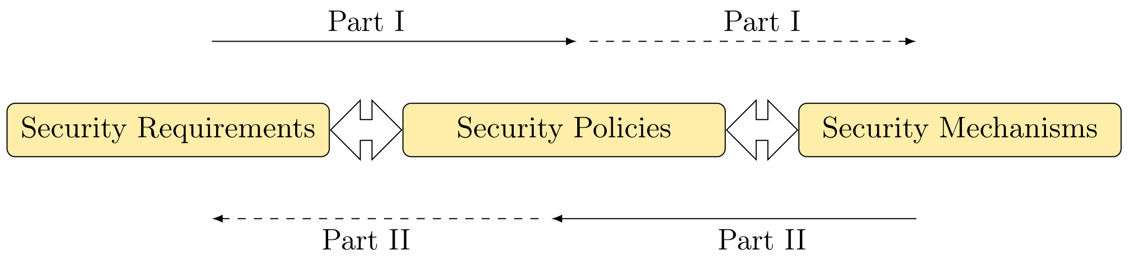


Figure 1.2: Overview of the Parts of this Thesis.

The overall structure, close to Figure 1.1, is illustrated in Figure 1.2. The solid lines mean that these translation steps are fully verified in Isabelle/HOL. The dashed line below

Part I indicates that there is a final, small, syntactic rewriting step which is not formally verified. However, we will use Part II to verify the results of this step afterwards. The dashed line above Part II means that we cannot compute security requirements, given only a policy. Such an attempt would correspond to reverse engineering and ultimately lead to guessing a user's intent. We provide means to verify a policy given the security requirements, but we make no attempt of any reverse engineering. We describe the individual chapters in the following.

Part 0: Introduction

Chapter 2 provides an overview of the current situation and hints at the relevance of our research questions.

Chapter 3 gives a brief overview of Isabelle, the interactive proof assistant used to machine-verify the results of this thesis (**NF2**).

Part I: Green-Field Approach. A detailed overview of this part can be found in Chapter 4.

Chapter 5 presents a method to formalize security requirements, answering **Q1.1**. We show how a specification can be securely auto-completed, which increases usability and decreases exposed formalism (**NF3**). To give an administrator feedback about the specified requirements, we show how they can be directly visualized as policy or how a policy can be verified, given a set of requirements and visualizing all possible violations. This directly answers **Q1.2**.

Chapter 6 presents a library of ready-to-use templates to prevent exposing any formalism. Only attributes need to be assigned to define security requirements (**NF3**).

Chapter 7 finally presents our tool, a case study, and further demonstrates applicability in an example. It also provides an outlook to provide a forward reference to introduce the problems which are not solved until Chapter 7.

Chapter 8 discusses a weakness of the automated policy construction method and subsequently improves it with regard to completeness and performance (**Q1.2**, **NF1**).

Chapter 9 does one step towards automatic (**NF1**) translation to security mechanisms (**Q1.3**). It shows how to translate connection-level policies to stateful network-level policies.

Chapter 10 finally presents deployment to a real network. Different security mechanisms are presented. This answers **Q1.3**.

Part II: Understanding Existing Configurations. A detailed overview of this part can be found in Chapter 11.

Chapter 12 presents a formal semantics of the filtering behavior of iptables, providing an answer to **Q2.1**.

Chapter 13 presents a novel algorithm to certify spoofing protection of a firewall configuration, providing an important part of the answer to question **Q2.2**.

Chapter 14 presents an algorithm to partition the complete IPv4 and IPv6 address space into classes with equal access rights. This provides the missing piece to the answer for question **Q2.2**. Building on this partitioning, we also present a method to translate a complex low-level iptables filtering ruleset with arbitrary match conditions to a simple firewall model and abstract it to an access control matrix which only considers IP addresses. This answers question **Q2.3**.

Part III: Applicability & Conclusion. A detailed overview of this part can be found in Chapter 15.

Chapter 16 introduces the applicability and compatibility of our developed solutions (**NF4**) by a simple example. It shows how our tools help operating a Docker-based environment.

Chapter 17 presents the interplay of our tools (**NF4**) in a real-world case study. It shows a privacy audit of the MeasrDroid platform.

Chapter 18 summarizes our answers to the scientific questions and summarizes the achieved results of this thesis.

Chapter 19 defines a list of criteria for tools which help in managing network access control. Based on these criteria, it then compares this work to the state of the art.

Chapter 20 summarizes applicability of our work with regard to generic policy management and reasoning, iptables firewall analysis, and software-defined networking.

Meta Structure All chapters start with a short abstract which summarizes the chapter's contributions in the big picture of this thesis. All parts which are based on joint work have an explicit statement on the author's contributions. If no such statement exists, the part is the single-handed contribution of Cornelius Diekmann.

1.4 Publications in the Context of this Thesis

Chapter 5 Cornelius Diekmann, Stephan-A. Posselt, Heiko Niedermayer, Holger Kinkelin, Oliver Hanka, and Georg Carle. *Verifying Security Policies using Host Attributes.* In FORTE – 34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems, volume 8461, pages 133-148, Berlin, Germany, June 2014. Springer.

Chapter 9 Cornelius Diekmann, Lars Hupel, and Georg Carle. *Directed Security Policies: A Stateful Network Implementation.* In Engineering Safety and Security Systems, volume 150 of Electronic Proceedings in Theoretical Computer Science, pages 20-34, Singapore, May 2014. Open Publishing Association.

Chapter 10 Cornelius Diekmann, Andreas Korsten, and Georg Carle. *Demonstrating topoS: Theorem-Prover-Based Synthesis of Secure Network Configurations.* In 2nd International Workshop on Management of SDN and NFV Systems, manSDN/NFV, Barcelona, Spain, November 2015.

Chapter 12 Cornelius Diekmann, Lars Hupel, and Georg Carle. *Semantics-Preserving Simplification of Real-World Firewall Rule Sets.* In 20th International Symposium on Formal Methods, pages 195-212, Oslo, Norway, June 2015. Springer.

Chapter 13 Cornelius Diekmann, Lukas Schwaighofer, and Georg Carle. *Certifying Spoofing-protection of Firewalls.* In 11th International Conference on Network and Service Management, CNSM, Barcelona, Spain, November 2015.

Chapter 14 Cornelius Diekmann, Julius Michaelis, Maximilian Haslbeck, and Georg Carle, *Verified iptables Firewall Analysis.* In IFIP Networking 2016, Vienna, Austria, May 2016.

Chapter 17 Marcel von Maltitz, Cornelius Diekmann and Georg Carle, *Taint Analysis for System-Wide Privacy Audits: A Framework and Real-World Case Studies.* In 1st Workshop for Formal Methods on Privacy, Limassol, Cyprus, November 2016. Note: no proceedings published.

Our formalization has been published in the Archive of Formal Proofs in the following entries:

- Cornelius Diekmann, *Network Security Policy Verification.*
- Cornelius Diekmann, Julius Michaelis and Lars Hupel, *IP Addresses.*
- Cornelius Diekmann, Julius Michaelis and Max Haslbeck, *Simple Firewall.*
- Cornelius Diekmann and Lars Hupel, *Iptables Semantics.*
- Julius Michaelis and Cornelius Diekmann, *Routing.*
- Julius Michaelis and Cornelius Diekmann, *LOFT – Verified Migration of Linux Firewalls to SDN*

Chapter 2

Current Situation & Problem Analysis

In this chapter, we describe the current state of network administration, configuration, and management with regard to security issues. Afterwards, we analyze the root cause of configuration complexity by comparing networks to software.

2.1 Evaluation of the Situation

A 2009 whitepaper by Netcordia [19] describes that networks "often fail, at great expense, not because of underlying equipment problems, but because of human error in setting them up and running them." The document concludes that "a primary (if not the primary) cause" for network downtime is human error. A 2003 survey [20] indicates that most Internet services fail because of human operator error, where configuration errors are the largest category of those human errors. A study [7], featuring 37 enterprise firewall configurations from the years 2000 and 2001, reveals that many firewalls are misconfigured. The study also reveals that the firewalls' configuration quality improves with new releases of the firewall product, which is mainly attributed to better default rule sets. Hence, better default settings provide less surface for human error. However, the study concludes "that there are no good high-complexity rule sets" [7]. Several years later, the situation has not actually improved [8]. Mansmann et al. [12] also hint that historically grown firewall rule sets are insufficiently understood. In 2007, a survey of 70 large ISPs revealed that management of access control lists were considered as the "most critical missing or limited vendor security feature" for infrastructure protection [55, 3]. In 2016, the same report series [56] still lists access control lists as one of the most widely and most actively used technique.[1] A 2012 survey [10] of 57 enterprise network administrators confirms that a "majority of administrators stated [estimated] misconfiguration as the most common cause of failure" [10]. A large 2013 study [57], conducted over two years across more than 10 large datacenters, reveals that there exists a variety of misconfigurations in network management. Based on Wool's findings [7], Casadoet al. [58] also conclude that "most networks today require substantial manual configuration by trained operators [. . .] to achieve even moderate security". Burns et al. [59] predict that "the scope of management is rapidly exceeding human

[1]The 2016 version of the report does no longer include an "Infrastructure Shortcomings" section.

capabilities because of the acceleration of changes in technology and topology" [59]. They see the need to eliminate low-level technical device configuration and focus on the desired behavior of a network. "Policies should define the intent of the administrator independently of the mechanisms used to implement the policy." [59].

This implies that the manual configuration complexity in network security management is a key aspect for failure. Advanced tools to support the administrator with the configuration complexity are barely deployed. "Paradoxically, most mission critical IT networks are configured and managed with little help from automated systems, but by humans working with few tools" [19]. But "administrators desire for newer, more sophisticated tools." [17]. However, "the security requirements of distributed systems are hard to specify and hard to formalize" [60]. A recent Dagstuhl seminar on "Formal Foundations for Networking" concludes that "[t]here is a growing need for tools and methodologies that provide rigorous guarantees about performance, reliability, and security" [61].

A study [15] conducted by Verizon from 2004 to 2009 and the United States Secret Service during 2008 and 2009 reveals that data leaks are also often caused by configuration errors [62]. The authors estimate that this might be due to the fact that "attackers know most users are over-privileged" [15]. In 2016 [63], privilege misuse (and misconfiguration) are still among the top causes of data breaches, which has also been demonstrated in a very concrete example [64]. This indicates that the complexity in network access policies which define who can communicate with whom cannot just be simplified, but on the contrary, should be expanded to reduce the attack surface by stricter, hence more complicated, access control policies.

Many vendor-specific devices with their own configuration interface exist [65]. A survey among enterprise administrators confirms that "typical enterprise networks are a complex ecosystem of firewalls, IDSes, web proxies, and other devices." [10]. "Managing many heterogeneous devices requires broad expertise and consequently a large management team." [10] A 2006 study [9] with 38 network administrators reveals the configuration complexity of security devices that require setting up low-level security policies, such as IPsec gateways, increases the probability of human error. The study finds that "even the expert administrators created policy conflicts" [9].

2.2 A Problem Classification

In this section, we try to trace back the symptoms of management complexity in networks to their root causes.

We know that even a set of simple switches which support round-robin load balancing is Turing-Complete [66]. But computer networks are becoming increasingly more software-defined: SANE [67] inspired Ethane [58], which itself inspired OpenFlow [68, §3.2 Example 1], which is now the de facto standard for Software-Defined Networking (SDN), which is used in the industry [69, 70]. With networks resembling more and more to software, we compare network management complexity to a field where complexity is well-studied since many decades: software engineering.

2.2.1 Types of Complexity in Software Engineering

In 1987, Brooks published his thousandfold-cited paper “No Silver Bullet: Essence and Accidents of Software Engineering” [71] in the IEEE Computer magazine. In software engineering nowadays, a good understanding about the complexity challenges exists [72, §1.2. The Inherent Complexity of Software]. In this section, we discuss the complexity challenges in software engineering as identified by Brooks [71] and later convey the results to the challenges in network management.

In software engineering, one distinguishes between *accidental* difficulties and *essential* difficulties[2]. Essential difficulties are the difficulties which are inherent in the nature of software, whereas accidental difficulties are those that are not inherent. For example, designing, conceptualizing, and defining the requirements and interfaces of a business application is an inherently complex task. The domain-specific challenges, which are inherently complex, must be imaged by the application and a huge amount of relationships between data items, business processes, and algorithms must be specified. In contrast, implementing the application in the `C` programming language and dealing with memory errors is an accidental complexity that could have been avoided by selecting a memory-safe programming language.

Essential difficulties in software engineering refer to [71]:

- The *complexity* of software itself and in particular the complexity challenges in the problem domain that are mirrored by the software.
- The *conformity* that software must comply with existing or legacy system interfaces.
- The *changeability* of software and that software is often used beyond its original purpose.
- The *invisibility* of software and the fact that it cannot be adequately visualized.

While accidental difficulties can be tackled, essential difficulties are inherently hard to overcome and Brooks projects that “there is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity”.

2.2.2 Types of Complexity in Network Security Management

In computer networks, administrators are “touching low-level configurations all the time” [73]. But problems with low-level configuration languages are comparable to the problems which arise from the use of the `C` programming language. Thus, low-level languages can be classified as accidental complexity of network management.

We now interpret the essential difficulties of software engineering in the context of network security management. We classify the essential difficulties as

- The *complexity* of security requirements themselves.
- The *conformity* with legacy systems in a network and the understanding of legacy configuration.

[2]sometimes also referred to inherent difficulties

- The *changeability* of network traffic and that attackers may exploit weaknesses in unforeseeable ways.
- The *invisibility* of network configurations and the fact that it cannot be adequately visualized.

In addition, compared to software engineering, our tooling for networks is "pathetic" [5].

Related Work In a closely related work, Benson et al. [74] present metrics to measure "inherent complexity" of network (router) configurations, "abstracting away all the details of the underlying configuration language" [74]. In other words, they measure essential difficulties, explicitly abstracting over accidental difficulties.

In a study and interview with several system administrators, they empirically uncover the following essential difficulties, which are very close to the difficulties we have identified: The inherent *complexity* itself, expressed as a network's reachability policy. A network's evolution over time and legacy configuration parts, which is related to *conformity*. Finally, the interviews with the administrators reveal that complexity metrics are "helping operators visualize and understand networks" [74, §7], which relates to the *invisibility* of network configuration. They do not identify the *changeability* of network traffic but identify that some networks are more complex than necessary because they are optimized for monetary cost.

Chapter 3

Brief Introduction to Isabelle and Notation

We implemented our theory and the formal proofs in the Isabelle/HOL theorem prover [37]. Isabelle is a generic and interactive proof assistant. We use its standard Higher-Order Logic (HOL).

Internally, Isabelle is an LCF-style theorem prover. This means, a fact can only be proven if it is accepted by a mathematical inference kernel. Proof steps can be done by either the user or by (embedded or external) automated proof tactics and solvers. All proof steps must pass this kernel, hence, a faulty prover does not introduce unsoundness because the kernel would reject unsound steps which it cannot reproduce. The correctness of a proof only depends on the correctness of the kernel. This architecture makes the system highly trustworthy, because the proof kernel consists only of little code, is widely used (and has been for over a decade) and is rigorously manually checked. This makes errors very unlikely, which has been demonstrated by Isabelle's success over the past 20 years. In fact, there has not been a known bug in the Isabelle kernel in the past 20 years which affected a user's proof.[1]

Standards such as Common Criteria [75] require formal verification for their highest *Evaluation Assurance Level* (EAL7) and the Isabelle/HOL theorem prover is suitable for this purpose [75, §A.5]. Therefore, our approach is not only suitable for verification, but also a first step towards certification.

To stay focused, we usually only present the intuition behind proofs or even omit a proof completely. Whenever we omit a proof for a claim which is not obvious, we add a footnote that points to our formalization. In addition, for better readability and brevity, we will not present all proven statements as theorem but present some facts in natural language within a sentence. We point the interested reader to the proof or definition by a footnote. For example, when the text states within a sentence that foo[42] holds, the machine-verified proof for the claim 'foo' can be found by following the corresponding footnote.

[1]But there have been bugs (which were all fixed) for artificially constructed corner cases.

3.1 Notation

We will now explain the notational conventions we apply throughout this thesis. In general, we use pseudo code close to SML, Haskell, and Isabelle.

Functions A total function from type $\mathcal{A}$ to type $\mathcal{B}$ is denoted by $\mathcal{A} \Rightarrow \mathcal{B}$. In contrast, the logical implication is written with a long arrow "$\Longrightarrow$". Function application is written without parentheses: $\mathsf{f}\ x\ y$ denotes function "f applied to parameter x and parameter y".

Lists We write :: for prepending a single element to a list, e.g., $a :: b :: [c,\ d] = [a,\ b,\ c,\ d]$, and ::: for appending lists, e.g., $[a,\ b] ::: [c,\ d] = [a,\ b,\ c,\ d]$. The empty list is written as $[]$. We write list comprehension as $[\mathsf{f}\ a.\ a \leftarrow l]$, which denotes applying f to every element a of list l. Also, $[\mathsf{f}\ x\ y.\ \ x \leftarrow l_1,\ y \leftarrow l_2]$ denotes the list comprehension where f is applied to each combination of elements of the lists l_1 and l_2. For $\mathsf{f}\ x\ y = (x,\ y)$, this returns the cartesian product of l_1 and l_2.

Types The set of Boolean values is denoted by the symbol $\mathbb{B} = \{\mathsf{True},\ \mathsf{False}\}$. To explicitly write down the type of an object, we annotate it with '::'. The two colons for type annotations have more spacing that the list operations; they can usually be distinguished by the context. For example, $\mathsf{True} :: \mathbb{B}$ or $\mathsf{f} :: \mathcal{A} \Rightarrow \mathcal{B}$ are type annotations. We use polymorphic types, e.g., f could be applied to integers and return a Boolean but it could also be applied to graphs and return a string.

Definitions Whenever applicable, we write definitions with '$\cdot \equiv \cdot$' to distinguish the operator from the mathematical equality operator '$\cdot = \cdot$'. This increases readability since the equality operator may also occur in definitions. We only use '$\cdot \equiv \cdot$' for formulas, not for types.

Control Statements Control statements, for example **if** · **then** · **else**, are set in bold font.

Typesetting To further increase readability, we stick to the following typesetting. Polymorphic types, whenever applicable, are set calligraphic, e.g., $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$. Usually, $\mathcal{G}$ denotes a graph where the nodes may be of arbitrary type. Specific types are set in italic or in normal text, depending on the context, e.g., *firewall-rule list* or $\mathcal{A}$ *list* which is a list over arbitrary types. Functions and constants, in general everything that is not a free variable, are set in sans serif font, e.g., f, True. Variables and locally-bound objects are set italic, e.g., a, v_1. Within an example, we also set entity names which are only valid for the example in italic. Linux shell commands are set in `typewriter` font.

Example. Let graph G of type $\mathcal{G}$ be $(\{v_1, v_2\},\ \emptyset)$. G only contains two vertices and no edges. The vertices could be of arbitrary type, e.g., represented by strings or integers. We will say the vertices are of arbitrary type $\mathcal{V}$, then $\mathcal{G} = (\mathcal{V}\ \mathit{set}) \times ((\mathcal{V}\times\mathcal{V})\ \mathit{set})$. For all examples, we assume that the entities referenced in the example are distinct, here, $v_1 \neq v_2$. We can have a function f which maps any graph to True. Then, $\mathsf{f} :: \mathcal{G} \Rightarrow \mathbb{B}$. We could implement f by the lambda expression which ignores its first argument and always returns True as follows: $(\lambda\_.\ \mathsf{True})$. Then, $\mathsf{f}\ (\{v_1, v_2\},\ \emptyset)$ holds.

3.2 Availability of our Formalization

The Archive of Formal Proofs (AFP) [76] is the de-facto place to find Isabelle theories. It is organized similar to a scientific journal and all submissions are peer reviewed. The peer review assures that the submitted theories conform to the Isabelle style rules and that the proofs are properly accepted by Isabelle. Once a submission is accepted in the AFP, it is maintained and updated by the community for future Isabelle releases.

Technically, an entry which is accepted in the AFP is only guaranteed to contain sound proofs, it does not guarantee that actually something useful has been proven. The meaning and applicability of our theory are demonstrated in this thesis and in the non-AFP publications. However, the AFP entries (which are both manually-reviewed and machine-verified) created during this thesis provide a very strong guarantee about the correctness of our proofs and ensure that the theoretical results are easily reproducible and accessible, even with future versions of Isabelle.

The following entries have been created during this thesis with major contributions by Cornelius Diekmann [38, 39, 40, 41]:

- Cornelius Diekmann, *Network Security Policy Verification.*
- Cornelius Diekmann, Julius Michaelis and Lars Hupel, *IP Addresses.*
- Cornelius Diekmann, Julius Michaelis and Max Haslbeck, *Simple Firewall.*
- Cornelius Diekmann and Lars Hupel, *Iptables Semantics.*

The following entries have been created during this thesis with contributions by Cornelius Diekmann [42, 43]:

- Julius Michaelis and Cornelius Diekmann, *Routing.*
- Julius Michaelis and Cornelius Diekmann, *LOFT – Verified Migration of Linux Firewalls to SDN*

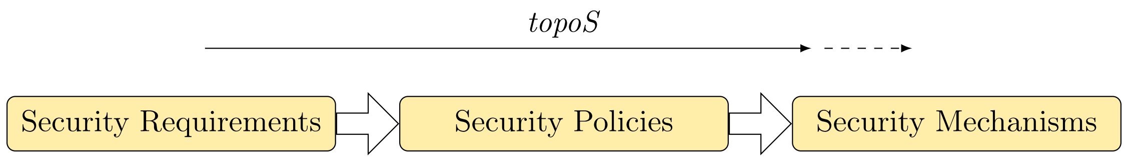
Part I

Green-Field Approach

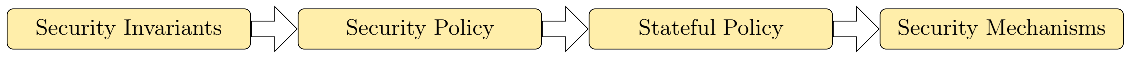
Chapter 4

Overview

As shown in Figure 1.1, this thesis focuses on the consistency between security requirements, a security policy, and security mechanisms. This part presents the left-to-right direction: Given the security requirements, in a greenfield approach, we provide a method to construct the security policy and the configuration of security mechanisms. Our tool to support the process is called *topoS*.

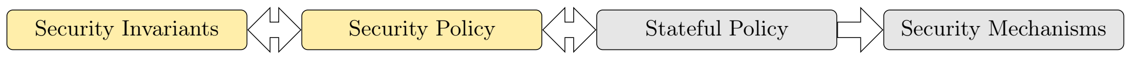


For this task, we have divided the three security components into four components.

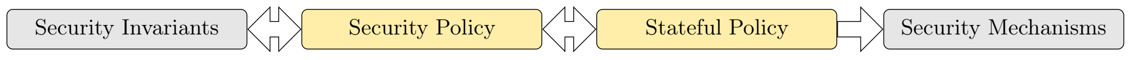


A set of *security invariants* formalizes the security requirements. The security policy has been split into the actual *security policy* and the *stateful policy*. This is motivated by the world of computer networks: The security policy expresses who may set up new connections and the stateful policy answers the question whether packets which belong to such an established connection are allowed bidirectionally. Finally, the *security mechanisms* are network components; we will demonstrate our tool for the Linux iptables firewall and an OpenFlow-enabled switch.

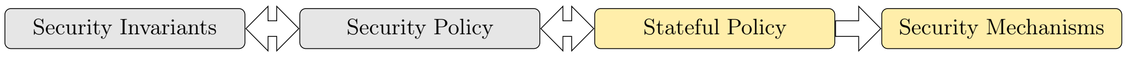
This part is structured as follows. In Chapter 5, we focus on the consistency between security invariants and a security policy. Given a specification of the security invariants, we show how to verify an existing policy or how to construct a new policy from scratch. Chapter 6 presents a library of ready-to-use security invariant templates. Chapter 7 presents a case study. Chapter 8 will improve the policy construction method.



In Chapter 9, we will cover the conformity of a security policy with a stateful policy, given the security invariants. We also show how to automatically compute a stateful policy from the security policy and the security invariants.



In Chapter 10, we put everything together and additionally demonstrate the translation to real network security devices.



Availability

Our Isabelle/HOL theory files with the formalization and the referenced correctness proofs and our tool *topoS* are available at

`https://github.com/diekmann/topoS` and the AFP [38]

Chapter 5

Verifying Security Policies using Host Attributes

This chapter, Chapter 6, and Section 7.2 in Chapter 7 are an extended version of the following paper [11]:

- Cornelius Diekmann, Stephan-A. Posselt, Heiko Niedermayer, Holger Kinkelin, Oliver Hanka, and Georg Carle. *Verifying Security Policies using Host Attributes.* In FORTE – 34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems, volume 8461, pages 133-148, Berlin, Germany, June 2014. Springer.

The following major improvements and new contributions were added:

- A full documentation of the security invariant library: Chapter 6.
- Several improvements and more configuration options for several invariant templates.
- New invariant templates.
- Generalized the Φ-structure, requiring reworking of all corresponding proofs.
- Improved the algorithm for policy construction: Chapter 8.
- Re-Implementation of the *topoS* tool in Isabelle: Section 7.1.
- New full-stack example: Section 7.3.
- Updated and extended related work, added analogy to software engineering.

Statement on author's contributions All improvements with regard to the paper are the work of the author of this thesis. For the original paper, the author of this thesis provided major contributions for the ideas, realization, formalization, analysis, and proof of the overall model and the invariant templates. He implemented the prototypical tool, researched related work, and conducted the user feedback session. The case study (A Cabin Data Network) was designed with the help of Oliver Hanka. It was evaluated and formalized by the author of this thesis.

The following ideas have been previously presented in the author's master's thesis [53]: Security requirements modeled as Boolean predicates are composable, the idea of offending flows, the secure default parameter, and the observation that the security strategy may be linked to a Boolean value related to the offending host. While the core ideas of the master's thesis remain, the complete formal foundations have been reworked for this Ph.D. thesis. The author's master's thesis relied on many inconvenient assumptions. For this Ph.D. thesis, the author completely re-implemented his master's thesis to improve and rework the formalization, and discovered many new insights which allowed getting rid of all unpleasant assumptions. For many parts of the author's master's thesis (e.g., composition), only the idea or prototypical, unverified code was presented. For this Ph.D. thesis, everything has been completely formalized and proven with Isabelle/HOL, requiring also a large rework or rebuild of existing theory. A preliminary prototype of *topoS* in Scala was developed during the author's master's thesis. The final *topoS* tool presented in this Ph.D. thesis is a completely new implementation (based on the new results of this thesis) in Isabelle/HOL.

Abstract In this chapter, we focus on the relationship between *Security Invariants* and a connection-level *Security Policy*, as illustrated in Chapter 4. We present a formalization of security invariants and show how they can be used to verify a policy and how a policy can be constructed from scratch, given only the security invariants.

5.1 Introduction

A distributed system, from a networking point of view, is essentially a set of interconnected hosts. Its connectivity structure comprises an important aspect of its overall attack surface, which can be dramatically decreased by giving each host only the necessary access rights. Hence, it is common to protect networks using firewalls and other forms of enforcing network-level access policies. Such access policies can be seen as means to describe which flows between hosts are allowed, and which are not. However, raw sets of such policy rules e.g., firewall rules, ACLs, or access control matrices, scale quadratically with the number of hosts and "controlling complexity is a core problem in information security" [6]. A case study, conducted in Chapter 7, reveals that even a policy with only 10 entities may cause difficulties for experienced administrators. Expressive policy languages can help to reduce the complexity. However, the question whether a policy fulfills certain security invariants and how to express these often remains.

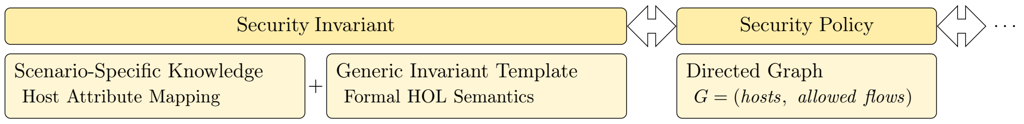


Figure 5.1: Formal Objects: Security Invariant and Security Policy.

Using an attribute-based [77] approach, we model simple, static, positive security policies with expressive, Mandatory Access Control (MAC) security invariants. The formal objects, illustrated in Figure 5.1, are carefully constructed for their use-case. The policy is simply a graph, which can for example be extracted from or translated to firewall rules (cf. Chapter 9, Chapter 10, and Part II). The security invariants are split into the formal semantics, accessible to formal analysis, and scenario-specific knowledge, easily configurable by the end user. This model landscape enables verification of security policies. Primarily, we contribute the following universal insights for constructing security invariants.

1. Both provably *secure* and *permissive* default values for host attributes can be found. This enables auto completion when specifying security invariants and hence decreases the user's configuration effort.

2. The security strategy, information flow or access control, determines whether a security violation occurs either at the sender's or at the receiver's side.

3. A violated invariant can always be repaired by tightening the policy *if and only if* the invariant holds for the deny-all policy.

In this Chapter 5, we focus on theoretical aspects: We formally introduce the underlying model in Section 5.2 and conduct a formal analysis in Section 5.3. Ultimately, we derive an algorithm to construct a security policy automatically in Section 5.4. In the subsequent chapters, we focus on application: We present our security invariant template library in Chapter 6. Our implementation, a case study, and an example are presented in Section 7.1, Section 7.2, and Section 7.3. Related work is described in Section 7.4. We conclude in Section 7.5.

5.2 Formal Model

In this section, we formally introduce the underlying model.

5.2.1 Terminology

This research intersects with the field of study of policies, which may lead to a clash of terminology. Whenever there is an ambiguity, we will use network terminology. In particular, we will use the term *host* for any entity which may appear in a policy, e.g., a host may be a collection of IP addresses, a name, or even a role. In contrast to common policy terminology [78, 79], we do not differentiate between subjects and objects (sometimes called targets) as they are usually indistinguishable on the network level and a host may act as both. For example, data may be written to an entity A. Then A could be interpreted as object. However, A may not be a traditional file, but A could also be a process which is analyzing the data and probably writing it to a different location. Then A could be interpreted as subject. Since our entities are usually networked hosts, it is natural that they may act as both, subject and object. This model assumption is in line with McIlroy and Reed's data flow model [80]. They notice that interprocess communication makes it necessary "to identify some subjects also as objects" [80].

We justify this choice with the another example, inspired by the goal that we ultimately want to enforce the policy on a network. A common terminology in the field of policies for filesystems is that users are subjects, files are targets, and access rights are either *read*, *write*, or *execute*. For example, a subject performs a read access to a target. However, when considering access rights from a network administrator's point of view who is setting up a simple router ACL, there is only the choice between allowing or disallowing the communication. It is possible to distinguish between sending and receiving hosts. On the application layer, network communication may cause read, write, or execute actions. However, a packet from one host to another as seen by a router could be a request to read a file, the contents of a file, the instruction to write to a file, or even be executable code itself. This is application layer information, which is not available on the network layer. Hence, a distinction between those three actions is not possible, which also means that a distinction between subjects and targets is not possible. Consequently, there is only one kind of generic entity: a host.

Having only the generic notion of a host and packets exchanged between hosts, it is still possible to distinguish between sender and receiver. Likewise, considering connections between hosts, it is possible to distinguish between initiator and accepting host. Hence, the distinction between client and server is not lost.

5.2.2 A Model of Security Policies and Security Invariants

According to Bishop, a *security policy* is "a specific statement of what is and is not allowed" [52]. Narrowing its scope to network access control, a security policy is a set of rules which state the allowed communication relationships between hosts. It can be represented as a directed graph. This view is consistent with the model used in seL4: "An access control policy is essentially a directed graph [...]" [81]. We will write $G = (V, E)$, with the hosts V and the allowed flows E.

Definition 1 (Security Policy). A security policy is a directed graph $G = (V, E)$, where the hosts V are a set of type $\mathcal{V}$ and the allowed flows E are a set of type $\mathcal{V}\times\mathcal{V}$. The type of G is abbreviated by $\mathcal{G} = (\mathcal{V}\ \textit{set}) \times ((\mathcal{V}\times\mathcal{V})\ \textit{set})$.

The policy we consider is on the abstraction level of connections or possibly application-level message flows. This abstraction level is important for unidirectional flows. For example, v_1 may write messages to a socket and v_2 reads them from a socket. Our policy may express that v_1 can send messages to v_2, but not the other way round. The unidirectional nature of this communication is adequate for reasoning on the connection level. However, when this policy is translated in a later chapter to a configuration for a network security mechanism, we need to consider network-level packet exchange. From the view of the network level, v_1 may transmit the message to v_2 using several packets over a TCP connection. While our connection-level policy models unidirectional message flow, a network-level TCP implementation requires bidirectional flow of packets between v_1 and v_2 for connection setup and acknowledgements. In an unusual manner, it is also possible to use only unidirectional UDP. We discuss the network-level implementation, the different cases, and how they are solved in Chapter 9. For both levels of abstraction—connection level and network level—a *directed* graph has proven to be the appropriate model. In this chapter, the policy is just on the connection level.

A policy defines rules (*"how?"*). It does not justify the intention behind these rules (*"why?"*). To reflect the *why?*-question, we note that depending on a concrete scenario, hosts may have varying security-relevant attributes. We model a host attribute of arbitrary type Ψ and establish a total mapping from the hosts V to their scenario-specific attribute. Security invariants can be constructed by combining a *host mapping* with a *security invariant template*. Latter two are defined together because the same Ψ is needed for a related host mapping and security invariant template; they are polymorphic over type Ψ. Different Ψ may appear across several security invariants.

Definition 2 (Host Mapping)**.** For scenario-specific attributes of type Ψ, a host mapping P is a total function which maps a host to an attribute. P is of type $\mathcal{V} \Rightarrow \Psi$.

Definition 3 (Security Invariant Template)**.** A security invariant template m is a predicate[1] of type $\mathcal{G} \Rightarrow (\mathcal{V} \Rightarrow \Psi) \Rightarrow \mathbb{B}$, defining the formal semantics of a security invariant. Its first argument is a security policy, its second argument a host attribute mapping. The predicate $\mathsf{m}\ G\ P$ holds iff the security policy G fulfills the security invariant specified by m and P.

Example (BLP). We have the goal to formalize a very simple invariant template to serve as example throughout this chapter. While the model is very simple, it has certain uses as Denning exemplifies for a "government or military system" [82] and we also show an example in Section 7.3. We model label-based information flow security inspired by the Bell-LaPadula model [83, 84, 85, 86, 87], but with simplifications outlined by Bishop's introductory informal description [88]. We exclude need-to-know vectors [83, 84] (also called categories or compartments [83, 84, 88, 86, 89]). Hence, labels consist only of "clearance levels" for subjects and "classifications" of objects. Since we do not distinguish between subjects and objects in our theory, in our simplified model the label of an entity directly maps to its current *security level.* The security levels model the host attributes for our security invariant template $\Psi = \{\mathsf{unclassified}, \mathsf{confidential}, \mathsf{secret}, \mathsf{topsecret}\}$. The Bell-LaPadula's no read-up and no write-down rules can be summarized by requiring that the security level of a receiver r should be greater-equal than the security level of the sender s, for all $(s, r) \in E$. With a total order '$\leq$' on Ψ, the security invariant template can be defined as $\mathsf{m}\ (V,\ E)\ P \equiv \forall (s, r) \in E.\ P\ s \leq P\ r$.

For simplicity, we decided for a total order, as opposed to the partial order of a lattice [82]. While a lattice structure is more expressive, we argue that using multiple simple invariants achieves the same expressiveness with better modularity. Theorem 5 will later underline this claim. Another lattice-based invariant template will be shown in Section 6.8.

Let the scenario-specific knowledge be that database $db_1 \in V$ is $\mathsf{confidential}$ and all other hosts are $\mathsf{unclassified}$. Using lambda calculus, the total function P can be defined as $(\lambda h.\ \textbf{if}\ h = db_1\ \textbf{then}\ \mathsf{confidential}\ \textbf{else}\ \mathsf{unclassified})$. Hence $P\ db_1 = \mathsf{confidential}$. If a host v_1 is $\mathsf{unclassified}$, it may send data to db_1, but not the other way round. This is appropriate for the abstraction level of our policy.[2]

For any policy G, the predicate $\mathsf{m}\ G\ P$ holds if db_1 does not leak confidential information (i. e., there is no non-reflexive outgoing edge from db_1). Independent of any policy, $\mathsf{m}\ \_\ P$ is a security invariant enriched with scenario-specific knowledge.

[1]A predicate is a total, Boolean-valued function.

[2]However, on the network level, this implies that it is impossible that v_1 and db_1 establish a TCP connection. Two hosts need exactly the same security label to establish a TCP connection [90]. Other invariants (in particular ACS, see next section) do not have this limitation. Chapter 9 details on how we derive a network-level implementation for a policy. Examples and evaluation will show that this simplified Bell-LaPadula template—which primarily serves as example—is barely useful once considering the network level and TCP. While there are some use cases where purely unidirectional packet flow is suitable (for example, we will use UDP in Section 10.3), trusted entities (Section 6.3) will permit bidirectional data flow between different security levels.

Security invariants formalize security goals. A template contributes the formal semantics. A host mapping contains the scenario-specific knowledge. This makes the scenario-independent semantics available for formal reasoning by treating P and G as unknowns. Even reasoning with arbitrary security invariants is possible by additionally treating m (of type $\mathcal{G} \Rightarrow (\mathcal{V} \Rightarrow \Psi) \Rightarrow \mathbb{B}$) as unknown.

With this modeling approach, the end user needs not to be bothered with the formalization of m, but only needs to specify G and P. In the course of this chapter, we present a convenient method for specifying P.

5.3 Properties and Semantic Analysis of Security Invariants

5.3.1 Security Strategies and Monotonicity

In IT security, one distinguishes between two main classes of security strategies: *Access Control Strategies* (ACS) and *Information Flow Strategies* (IFS) [89, §6.1.4]. An IFS focuses on confidentiality and an ACS on integrity or controlled access. We require that m is in one of these classes.

Conventionally, IT security "rests on confidentiality, integrity, and availability" [88, §1.1]. By limiting m to IFS or ACS, we emphasize that availability is not in the scope of this work. Availability requires reasoning on a lower abstraction level, for example, to incorporate network hardware failure. Availability invariants could be expressed similarly, but would require inverse monotonicity (see below).

The two security strategies IFS and ACS have one thing in common: they prohibit illegal actions. From an integrity and confidentiality point of view, prohibiting more never has a negative side effect. Removing edges from the policy cannot create new accesses and hence cannot introduce new access control violations. Similarly, for an IFS, by statically prohibiting flows in the network, no new direct information leaks nor new side channels can be created. In brief, prohibiting more does not harm security. From this, it follows that if a policy (V, E) fulfills its security invariant, for a stricter policy rule set $E' \subseteq E$, the policy (V, E') must also fulfill the security invariant. We call this property *monotonicity*.

Definition 4 (Monotonicity of Security Invariant Templates)**.** A security invariant template m is *monotonic* if and only if for all V, E, P, and $E' \subseteq E$, if $\mathsf{m}\ (V, E)\ P$ holds, it must also hold that $\mathsf{m}\ (V, E')\ P$. Formally:

$$\forall V\ E\ P.\ \mathsf{m}\ (V, E)\ P \longrightarrow (\forall E' \subseteq E.\ \mathsf{m}\ (V, E')\ P)$$

We require that all security invariant templates are monotonic.

Example. We refer to the BLP example on page 25 and will prove that it is monotonic. The invariant template was defined as $\forall (s, r) \in E.\ P\ s \leq P\ r$ and we assume the invariant holds. Consequently, for any $E' \subseteq E$, it holds that $\forall (s, r) \in E'.\ P\ s \leq P\ r$. $\square$

Therefore, if no information leak exists in the first place, disallowing additional flows cannot create a new information leak.

5.3.2 Offending Flows

Since m is monotonic, if an IFS or ACS security invariant is violated, there must be some flows in G that are responsible for the violation. By removing them, the security invariant should be fulfilled (if possible). We call a minimal set of such flows the *offending flows*. Minimality is expressed by requiring that every single flow in the offending flows bears responsibility for the security invariant's violation.

Definition 5 (Set of Offending Flows)**.**

$$\begin{aligned}\textsf{set-offending-flows}\ G\ P \equiv \big\{F \subseteq E \mid \neg\,\mathsf{m}\ G\ P\ \ \wedge\ \mathsf{m}\ (V,\, E \setminus F)\ P\ \ \wedge \\ \forall (s,r) \in F.\ \neg\,\mathsf{m}\ \ (V,\, (E \setminus F) \cup \{(s,r)\})\ \ P\big\}\end{aligned}$$

Hence, the *set of offending flows* for a policy $G = (V,\, E)$ with host attributes P consists of all subsets of the graph's flows $F \subseteq E$, such that if the security requirement is violated, removing the flows F renders the security invariant valid. In addition, F must be minimal, i.e., every single flow in F can violate the security invariant when added to $E \setminus F$ again.

> **Example.** The definition does not require that the offending flows are uniquely defined. This is reflected in its type since it is a set of sets. For example, for $G = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_2, v_3)\})$ and a security invariant that v_1 must not transitively access v_3, the invariant is violated: v_2 could forward requests. The set of offending flows is $\{\{(v_1, v_2)\}, \{(v_2, v_3)\}\}$. This ambiguity tells the end user that there are multiple options to fix a violated security invariant. The policy can be tightened by prohibiting one of the offending flows, e.g.,$\{(v_1, v_2)\}$.

If $\mathsf{m}\ G\ P$ holds, the set of offending flows is always empty[3]. Also, for every element in the set of offending flows, it is guaranteed that prohibiting these flows leads to a fulfilled security invariant[4]. Without the minimality requirement of the offending flows, they are monotonic.

Lemma 1 (Monotonicity of non-minimal Offending Flows)**.** *Assume* $\neg\,\mathsf{m}\ (V,\, E)\ P$ *and* F *is an offending flow i.e.,* $\mathsf{m}\ (V,\, E \setminus F)\ P$*, then any* F' *can be added to* F *and* $F \cup F'$ *still repairs the policy violation:* $\mathsf{m}\ (V,\, E \setminus (F \cup F'))\ P$.

It is not guaranteed that the set of offending flows is always non-empty for a violated security invariant. Depending on m, it may be possible that no set of flows satisfies Definition 5. For our security invariant templates, we require that whenever there is a violation, the set of offending flows is non-empty. This means, a violated invariant can always be repaired by tightening the policy. Theorem 1 proves[5] an important insight: this is possible if and only if the invariant holds for the deny-all policy.

Theorem 1 (No Edges Validity)**.** *For* m *monotonic, arbitrary* V*,* E*, and* P*, let* $G = (V,\, E)$ *and* $G_{\text{deny-all}} \equiv (V, \emptyset)$*. If* $\neg\,\mathsf{m}\ G\ P$ *then*

$$\mathsf{m}\ G_{\text{deny-all}}\ P \ \longleftrightarrow\ \textsf{set-offending-flows}\ G\ P \neq \emptyset$$

[3] sinvar-no-offending
[4] removing-offending-flows-makes-invariant-hold
[5] valid-empty-edges-iff-exists-offending-flows

We demand that all security invariants fulfill $\mathsf{m}\ G_{\mathit{deny\text{-}all}}\ P$. This means that violations are always fixable.

We call a host responsible for a security violation the *offending host*. Given one offending flow, the violation either happens at the sender's or the receiver's side. The following difference between ACS and IFS invariant can be observed. If m is an ACS, the host that initiated the request provokes the violation by violating an access control restriction. If m is an IFS, the information leak only occurs when the information reaches the unintended receiver. This distinction is essential as it renders the upcoming Definition 7 and Definition 8 provable.

Definition 6 (Offending Hosts)**.** For $F \in \mathsf{set\text{-}offending\text{-}flows}\ G\ P$

$$\mathsf{offenders}\ F \equiv \begin{cases} \{\, s \mid (s, r) \in F\} & \textbf{if } \text{ACS} \\ \{\, r \mid (s, r) \in F\} & \textbf{if } \text{IFS} \end{cases}$$

5.3.3 Secure Auto Completion of Host Mappings

Since P is a *total* function $\mathcal{V} \Rightarrow \Psi$, a host mapping for *every* element of $\mathcal{V}$ must be provided. However, an end user might only specify the *security-relevant* host attributes. Let $P_C \subseteq \mathcal{V} \times \Psi$ be a finite, possibly incomplete host attribute mapping specified by the end user. For some $\bot \in \Psi$, the total function P can be constructed by $P\ v \equiv (\textbf{if}\ (v, \psi) \in P_C\ \textbf{then}\ \psi\ \textbf{else}\ \bot)$. Intuitively, if no host attribute is specified by the user, $\bot$ acts as a default attribute.

Given the user specified all security-relevant attributes, we observe that the default attribute can never solve an existing security violation. Therefore, we conclude that for a given security invariant m, a value $\bot$ can securely be used as a default attribute if it cannot mask potential security risks.

We denote an update to P which returns $\bot$ for v by $P_{v \mapsto \bot}$, i.e., $P_{v \mapsto \bot} \equiv (\lambda x.\ \textbf{if}\ x = v\ \textbf{then}\ \bot\ \textbf{else}\ P\ x)$. In other words, a default attribute $\bot$ is secure w.r.t. the given information P if for all offenders v, replacing v's attribute by $\bot$ has the same amount of security-relevant information as the original P.

Definition 7 (Secure Default Attribute)**.** We call $\bot$ a secure default attribute if and only if for a fixed m and for arbitrary G and P that cause a security violation, replacing the host attribute of any offenders by $\bot$ must guarantee that no security-relevant information is masked. Formally,

$$\forall\, G\ P.\ \forall\, F \in \mathsf{set\text{-}offending\text{-}flows}\ G\ P.\ \ \forall v \in \mathsf{offenders}\ F.\ \neg\, \mathsf{m}\ G\ P_{v \mapsto \bot}$$

Example. In the simple Bell-LaPadula model, an IFS, let us assume information is leaked. The predicate 'information leaks' holds, no matter to which lower security level the information is leaked. In general, if there is an illegal flow, it is from a higher security level at the sender to a lower security level at the receiver. Replacing the security level of the receiver with the lowest security level, the information about the security violation is always preserved. Thus, $\mathsf{unclassified}$ is the secure default attribute[6]. In summary, if all

[6]interpretation BLPbasic: SecurityInvariant_IFS

classified hosts are labeled correctly, treating the rest as unclassified prevents information leakage.

To elaborate on Definition 7, it can be restated as follows. It focuses on the available security-relevant information in the case of a security violation. The attribute of an offending host v bears no information, except for the fact that there is a violation. A secure default attribute $\bot$ cannot solve security violations. Hence $P\ v$ and $\bot$ are equal w.r.t. the security violation. Thus, P and $P_{v \mapsto \bot}$ must be equal w.r.t. the information about the security violation. Requiring this property for all policies, all possible security violations, all possible choices of offending flows, and all candidates of offending hosts, this definition justifies that $\bot$ never hides a security problem.

Example. Definition 7 can be specialized to the exemplary case in which a new host x is added to a policy G without updating the host mapping. Consulting an oracle, x's real host attribute is $P\ x = \psi$. In reality, the oracle is not available and x is mapped to $\bot$ because it is new and unknown. Let x be an attacker. With the oracle's ψ-attribute, x causes a security violation. We demand that the security violation is exposed even without the knowledge from the oracle. Definition 7 satisfies this demand: if x mapped by the oracle to ψ causes a security violation, x mapped to $\bot$ does not mask the security violation.

Note that our theory assumes that a user specifies all security-relevant attributes. Hence, our default attribute is not designed to be secure if a user does not provide this information. This is in line with our goal that security invariants provide a language to describe scenario-specific security requirements: If a requirement is not listed in the requirement specification, our formalization provides no means to uncover this. However, our algorithm for automated policy construction (Section 5.4) may provide a feedback by computing an overly permissive policy which may hint at the missing requirement.

A 'deny-all' default attribute is easily proven secure. Definition 7 reads the following for this case: if an offender v does something that violates $\mathsf{m}\ G\ P$, then removing all of v's rights ($P_{v \mapsto \text{deny-all}}$), a violation must persist. Hence, designing whitelisting security invariant templates with a restrictive default attribute is simple. However, to add to the ease-of-use, more permissive default attributes are often desirable since they reduce the manual configuration effort. In particular, if a security invariant only concerns a subset of a policy's hosts, no restrictions should be imposed on the rest of the policy. This is also possible with Definition 7, but may require a comparably difficult proof.

Example. In the BLP example on page 25, no matter how many (unconfigured) hosts are added to the policy, it is sufficient to only specify that db_1 is $\mathsf{confidential}$. This confidentiality is guaranteed while no restrictions are put on hosts that do not interact with db_1.

Definition 8 (Default Attribute Uniqueness)**.** A default attribute $\bot$ is called unique iff it is secure (Definition 7) and there is no $\bot' \neq \bot$ s.t. $\bot'$ is secure.

We demand that all security invariants fulfill Definition 7 and Definition 8. This means that there is only one unique secure default attribute $\bot$.

Example. In the simple Bell-LaPadula model, since the security levels form a total order, the lowest security level is uniquely defined.

With the experience of proving that default attributes of about 20 invariant templates fulfill Definition 7 and Definition 8, the connection between offending host and security strategy was discovered. During our early research, we realized that a Boolean variable, fixed for m, indicating the offending host was necessary to make Definition 7 and Definition 8 provable. A classification of the different invariants revealed the important connection.

5.3.4 Φ-Structured Security Invariant Templates

The Bell-LaPadula security invariant template was defined as $\forall (s, r) \in E.\ P\ s \leq P\ r$. That means, for all flows, a predicate over the sender's and receiver's security level is evaluated. Here, the predicate is $(\lambda sl_s\ sl_r.\ sl_s \leq sl_r)$. We found that a similar structure is also found in most other security invariant templates. Many security invariant templates presented in Section 6 have a simple, common structure: a predicate is evaluated for all flows over the sender's and receiver's scenario-specific attributes. Let Φ be this predicate. Φ is of type $\Psi \Rightarrow \Psi \Rightarrow \mathbb{B}$. Then the security invariant template corresponds to $\forall (s, r) \in E.\ \Phi\ (P\ s)\ (P\ r)$. Consequently, we call all security invariant templates that follow this structure Φ-structured.

Definition 9 (Φ-Structured Security Invariant Templates)**.** A security invariant template is called Φ-*structured* iff for some predicate Φ, the invariant template can be expressed as $\forall (s, r) \in E.\ \Phi\ (P\ s)\ (P\ r)$.

It is possible to express lemmas and theorems generically for all Φ-structured invariant templates. This simplifies proofs about specific invariant templates since the generic lemmas can be reused. In addition, this avoids proof duplication.

Example. All Φ-structured invariant templates immediately fulfill monotonicity[7].

Some security invariant templates slightly deviate from the Φ-structured definition. However, the important results which hold for Φ-structured invariant templates also hold for templates with the following structure. All have in common that some predicate is evaluated for all flows (i.e., policy rules):

- The predicate also takes the name of the entity into account:
  $\forall (s, r) \in E.\ \Phi\ (P\ s)\ s\ (P\ r)\ r$
- Reflexive flows are excluded:
  $\forall (s, r) \in E.\ s \neq r \longrightarrow \Phi\ (P\ s)\ (P\ r)$
- A combination of both:
  $\forall (s, r) \in E.\ s \neq r \longrightarrow \Phi\ (P\ s)\ s\ (P\ r)\ r$

For the sake of brevity, we only consider Φ-structured invariant templates according to Definition 9. We refer the reader to the proof document for the slightly deviated versions.

[7]monotonicity-sinvar-mono

5.3.5 Unique and Efficient Offending Flows

We assume that m is a Φ-structured invariant template. Definition 5 is defined over all subsets. Consequently, the naive computational complexity to compute the set of offending flows of is in $\boldsymbol{NP}$. This section shows that – with knowledge about a concrete security invariant template m – it can be computed in linear time. For Φ-structured invariants, the offending flows are always uniquely defined and can be described intuitively[8]. The same holds for templates with a similar structure[9].

Theorem 2 (Φ Set of Offending Flows)**.** *If* m *is* Φ*-structured, i.e.,* $\mathsf{m}\ G\ P \equiv \forall (s,r) \in E.\ \Phi\ (P\ s)\ (P\ r)$*, then*

$$\textsf{set-offending-flows}\ G\ P \equiv \begin{cases} \{\{(s,r) \in E \mid \neg\ \Phi\ (P\ s)\ (P\ r)\}\} & \textbf{if}\ \neg\ \mathsf{m}\ G\ P \\ \emptyset & \textbf{if}\ \mathsf{m}\ G\ P \end{cases}$$

Example. For the Bell-LaPadula model, if no security violation exists the set of offending flows is $\emptyset$, else $\{\{(s,r) \in E \mid P(s) > P(r)\}\}$,[10] i.e., all flows from a higher to a lower security level.

Offending flows for templates similar to Φ-structured invariants In general, if no violation exists, the set of offending flows is always $\emptyset$[11]. If there exists a violation and the structure of the template corresponds to one of the structures similar to a Φ-structured invariant, the following holds. If the structure of the template is $\forall (s,r) \in E.\ \Phi\ (P\ s)\ s\ (P\ r)\ r$, then $\textsf{set-offending-flows}\ G\ P$ is $\{\{(s,r) \in E \mid \neg\ \Phi\ (P\ s)\ s\ (P\ r)\ r)\}\}$[12]. If the structure is $\forall (s,r) \in E.\ s \neq r \longrightarrow \Phi\ (P\ s)\ (P\ r)$, then $\textsf{set-offending-flows}\ G\ P$ is $\{\{(s,r) \in E \mid s \neq r\ \wedge \neg\ \Phi\ (P\ s)\ (P\ r)\}\}$[13]. If the structure is $\forall (s,r) \in E.\ s \neq r \longrightarrow \Phi\ (P\ s)\ s\ (P\ r)\ r$, then $\textsf{set-offending-flows}\ G\ P$ is $\{\{(s,r) \in E \mid s \neq r \wedge \neg\ \Phi\ (P\ s)\ s\ (P\ r)\ r\}\}$[14].

5.3.6 Composition of Security Invariants

Usually, there is more than one security invariant for a given scenario. However, composition and modularity is often a non-trivial problem. For example, access control lists that are individually secure can introduce security breaches under composition [91]. Also, information flow security of individually secure processes, systems, and networks may be subverted by composition [92]. This is known as the *composition problem* [87].

With our formalization, composability and modularity are enabled by design. For a fixed policy G with k security invariants, let m_i be the security invariant template and P_i the host mapping, for $i \in \{1 \ldots k\}$. The predicate $\mathsf{m}_i\ G\ P_i$ holds if and only if the security invariant i holds for the policy G. With this modularity, composition of all security invariants is straightforward[15]: all security invariants must be fulfilled. The monotonicity guarantees

[8]ENF-offending-set
[9]ENFsr-offending-set, ENFnr-offending-set, ENFnrSR-offending-set
[10]BLP-offending-set
[11]sinvar-no-offending
[12]ENFsr-offending-set
[13]ENFnr-offending-set
[14]ENFnrSR-offending-set
[15]all-security-requirements-fulfilled

that having more security invariants provides greater or equal security.

$$\mathsf{m}_1\ G\ P_1\ \wedge \ldots \wedge\ \mathsf{m}_k\ G\ P_k$$

This composition works due to two careful design decisions. First, a security policy can only have positive rules. This implies that there cannot be contradictory allow and deny rules; the policy is represented only by allow rules. Second, monotonicity ensures that the algorithms and definitions of the following sections behave as expected, also under composition.

Care must be taken with regard to the type when composing multiple security invariants. Each invariant m_i may use its own type Ψ_i, hence, each invariant template is of type $\mathcal{G} \Rightarrow (\mathcal{V} \Rightarrow \Psi_i) \Rightarrow \mathbb{B}$. Some m_j may be of type $\mathcal{G} \Rightarrow (\mathcal{V} \Rightarrow \Psi_j) \Rightarrow \mathbb{B}$, where $\Psi_i \neq \Psi_j$. For example, Ψ_i could be security levels while Ψ_j could be access control lists. Therefore, it is not possible to store several different security invariant templates in the same list. The list $[\mathsf{m}_i,\ \mathsf{m}_j]$ is not well-typed. However, a security invariant template applied to a policy and a host mapping, e.g., $\mathsf{m}_i\ G\ P_i$, is of type $\mathbb{B}$. Therefore, $[\mathsf{m}_1\ G\ P_1,\ \ldots,\ \mathsf{m}_i\ G\ P_i,\ \mathsf{m}_j\ G\ P_j,\ \ldots,\ \mathsf{m}_k\ G\ P_k]$ is a well-typed list of Booleans. It is also possible to partially apply the scenario-specific knowledge P_i to m_i, i.e., $(\lambda G.\ \mathsf{m}_i\ G\ P_i)$. This leaves a function of type $\mathcal{G} \Rightarrow \mathbb{B}$. We call this a *configured security invariant*. This means, for a specific scenario, all security invariants with their scenario-specific knowledge can be specified independently of the policy G. The list $[(\lambda G.\ \mathsf{m}_1\ G\ P_1),\ \ldots,\ (\lambda G.\ \mathsf{m}_i\ G\ P_i),\ (\lambda G.\ m_j\ G\ P_j),\ \ldots,\ (\lambda G.\ \mathsf{m}_k\ G\ P_k)]$ is a well-typed list of predicates over a security policy.

The set-offending-flows was defined for a policy and a host attribute mapping. Its type is $\mathcal{G} \Rightarrow (\mathcal{V} \Rightarrow \Psi) \Rightarrow (\mathcal{V} \times \mathcal{V})\ \mathit{set\ set}$. It can also be defined for a configured security invariant $(\lambda G.\ \textsf{set-offending-flows}\ G\ P)$. This leaves a function of type $\mathcal{G} \Rightarrow (\mathcal{V} \times \mathcal{V})\ \mathit{set\ set}$. To avoid notational overhead, we will apply set-offending-flows to both invariant templates and configured invariants interchangeably.

Notationally, we will write m for a security invariant template and m_c for a configured security invariant template.

5.4 Policy Construction

In this section, we combine the results of the previous sections to present a simple algorithm to construct a security policy, given the list of configured security invariants.

A policy that fulfills all security invariants can be constructed by removing all offending flows from an arbitrary starting policy. This approach is sound[16] for arbitrary m. In general, $G_{\text{all}} \equiv (V, V \times V)$ is a good starting point.

To simplify the following algorithm, we specify a helper function.

$$\textsf{delete-edges}\ (V,\ E)\ E' \equiv (V,\ E \setminus E')$$

Then, the following function constructs a security policy for a given list of configured invariants.

$\textsf{generate-valid-topology}\ :: (\mathcal{G} \Rightarrow \mathbb{B})\ \text{list} \Rightarrow \mathcal{G} \Rightarrow \mathcal{G}$

[16] generate-valid-topology-sound

$$\begin{aligned}
&\textsf{generate-valid-topology}\ []\ G = G\\
&\textsf{generate-valid-topology}\ (m_c :: Ms)\ G =\\
&\qquad \textsf{delete-edges}\ \ (\textsf{generate-valid-topology}\ Ms\ G)\ \left(\bigcup \textsf{set-offending-flows}\ m_c\ G\right)
\end{aligned}$$

In each iteration, the algorithms removes the offending flows of one configured invariant and calls itself recursively with the updated policy. A different approach would be to compute the offending flows for all configured invariants for the starting policy and remove them at once. Both approaches yield the same result.

Lemma 2.

$$\begin{gathered}
\textsf{generate-valid-topology}\ Ms\ G =\\
\textsf{delete-edges}\ G\ \left(\bigcup \left(\bigcup m_c \in \textsf{set}\ Ms.\ \textsf{set-offending-flows}\ m_c\ G\right)\right)
\end{gathered}$$

Theorem 3 (Soundness of Policy Construction)**.** *Let M be a set of configured security invariants. This means, each element of M is of type $\mathcal{G} \Rightarrow \mathbb{B}$, is monotonic, and always has defined offending flows. Then*

$$\forall m_c \in M.\ \ m_c\ \ (\textsf{generate-valid-topology}\ M\ G)$$

For Φ-structured security invariant templates and when starting with the allow-all policy $G_{\text{all}} = (V, V \times V)$, the algorithm is also complete[17]. Completeness tells us that the constructed policy is most permissive, i.e., it is not possible to add additional allowed flows to the policy.

Theorem 4 (Completeness of Policy Construction for Φ-Structured Invariants)**.** *Let M be a set of configured, Φ-structured security invariants. Let*

$$(V,\ E_{\text{max}}) = \textsf{generate-valid-topology}\ M\ (V,\ V \times V)$$

Then all configured security invariants hold for the constructed policy $(V,\ E_{\text{max}})$ but for all $e \in (V \times V) \setminus E_{\text{max}}$, if e is added to the constructed policy, at least one configured security invariant is violated.

For the rest of this thesis, we will always assume that we call our automated policy construction algorithm with G_{all}.

Example. We refer to the BLP example on page 25 where only db_1 was labeled confidential. Constructing a valid policy for this setting means to remove all flows where db_1 is sending out information (except for the reflexive flow where db_1 sends to itself). This means that any other hosts $\neq db_1$ can freely interact and even send data to db_1.

Example. For Φ-structured invariants, Theorem 2 tells that the offending flows are uniquely defined. Lemma 2 tells that a valid policy is constructed by removing all those uniquely defined offending flows. Consequently, for Φ-structured invariants, a maximum policy is uniquely defined.

[17] generate-valid-topology-max-topo

Example. If completely contradictory security invariants are given, the resulting (maximum) policy is the deny-all policy $G_{\text{deny-all}} = (V, \emptyset)$.

Example. We assume both a valid, manually-specified policy $G_{\text{manual}} = (V,\ E)$ and a specification of the security invariants M are available. We can compute $G_{\text{computed}} =$ $\mathsf{generate\text{-}valid\text{-}topology}\ M\ (V,\ V \times V)$.

With this, one can test whether the specified invariants 'mean' the right thing by comparing G_{computed} with G_{manual}.

If all invariants are Φ-structured, any valid G_{manual} is a sub-policy of G_{computed}.[18] This means, if the specified invariants encode the 'right' requirements and encode all requirements, G_{computed} should be equal to G_{manually}.

Consequently, G_{computed} gives feedback about whether the specified security invariants carry the right meaning.

For Φ-structured invariants, the algorithm has all properties on could wish for: It is simple, sound, complete, and fast. In Chapter 8, we will improve this algorithm for non-Φ-structured invariants.

[18]enf-all-valid-policy-subset-of-max

Chapter 6

Security Invariant Template Library

The invariants of Section 6.2, 6.3, 6.8, and a simplified version of Section 6.11 have been presented in the following paper [11]:

- Cornelius Diekmann, Stephan-A. Posselt, Heiko Niedermayer, Holger Kinkelin, Oliver Hanka, and Georg Carle. *Verifying Security Policies using Host Attributes.* In FORTE – 34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems, volume 8461, pages 133-148, Berlin, Germany, June 2014. Springer.

The invariants of Section 6.15 and Section 6.16 have been presented in the following paper [93]:

- Marcel von Maltitz, Cornelius Diekmann and Georg Carle, *Taint Analysis for System-Wide Privacy Audits: A Framework and Real-World Case Studies.* In 1st Workshop for Formal Methods on Privacy, Limassol, Cyprus, November 2016. Note: no proceedings published.

Statement on author's contributions The formalization of all the presented invariant templates are the work of the author of this thesis. The following security invariants are based on the author's master's thesis [53]: Section 6.2, 6.3, 6.7, 6.8, 6.13, 6.14, 6.11, 6.12, and 6.10. They have been re-implemented for the new underlying theory, extended, and improved.

The following security invariants are based on joint work with Marcel von Maltitz: Section 6.15 and Section 6.16. Von Maltitz contributed to the literature research and survey of the conceptualization of privacy. From this, von Maltitz derived a working (informal) definition of privacy. Both, the author of this thesis and von Maltitz, contributed to the research of related work with regard to taint analysis. Those sections are only outlined very briefly in the paragraph 'Background' of Section 6.15. The author of this thesis provided major contributions for deriving the formulas, the implementation, realization, and proofs of the formalizations.

Abstract The previous chapter discussed the theoretic foundations of security invariant templates. In this chapter, we present our library of pre-defined security invariant templates and show some examples.

6.1 Introduction

In this chapter, we present all security invariant templates which are defined at the time of this writing. Our implementation currently features fifteen templates and grows. Common networking scenarios such as subnets, non-interference invariants, or access control lists are available. All can be inspected in the published theory files. They are summarized in Table 6.1 and will be illustrated in this chapter. The first column of the tables gives the template's name, the second column the section in which the template will be defined, the third column lists whether the template is Φ-structured (cf. Section 5.3.4), the fourth column refers to the security strategy (Section 5.3.1), finally, column five provides a summarizing description of the template's intended use cases.

Table 6.1: Security Invariant Templates

Name	§	Φ	Stgy	Description
Simple BLP	6.2	✓	IFS	Simplified Bell-LaPadula
Bell-LaPadula	6.3	✓	IFS	Label-based Information Flow Security with trusted entities
Comm. Partners	6.4	✓	ACS	Simple ACLs (Access Control Lists)
Comm. With	6.5	✗	ACS	White-listing transitive ACLs
Not Comm. With	6.6	✗	ACS	Black-listing transitive ACLs
Dependability	6.7	✗	ACS	Limit dependence on certain hosts
Domain Hierarchy	6.8	✓	ACS	Hierarchical control structures
NoRefl	6.9	✓	ACS*	Allow/deny reflexive flows. Can lift symbolic policy identifiers to role names (e.g., symbolic host name corresponds to an IP range.)
NonInterference	6.10	✗	IFS	Transitive non-interference properties
PolEnforcePoint	6.11	✓	ACS	Central application-level policy enforcement point. Master/Slave relationships.
Sink	6.12	✓	IFS	Information sink. Hosts (or host groups) must not publish any information
Subnets	6.13	✓	ACS	Collaborating, protected host groups
SubnetsInGW	6.14	✓	ACS	Simple, collaborating, protected or accessible host groups
Simple Tainting	6.15	✓	IFS	Simplified label-based Privacy
Tainting	6.16	✓	IFS	Label-based Privacy with untainting

§= Section. Stgy = Strategy: Access Control (ACS) or Information Flow (IFS)

Default Attributes The default attributes of all the templates were designed (whenever possible) to be as permissive as possible. This means, the default attributes should at least allow flows between all hosts which are set to the default attribute. This greatly adds to the ease-of-use, since the scope of an invariant is limited only to the explicitly configured hosts. The unconcerned parts of a security policy are not negatively affected. For example, for a policy with 100 hosts, one can easily express an invariant for only three hosts; the connectivity of the other 97 hosts among each other should not be negatively influenced since those 97 machines are set to $\bot$. For certain strict invariants, this is not always possible due to the proof obligations imposed by Definition 8.

Meta Invariant We present a meta security invariant template to model system boundaries in Section 6.17. We call it a *meta* invariant because internally, it is implemented by two of the invariants of Table 6.1.

6.2 Simple Bell-LaPadula

A simplified version of the Bell-LaPadula model served as guiding example throughout the last chapter. In this section, it will be completely formalized.

The following two paragraphs about the Bell-LaPadula Model in the literature and its history are based on two previously published paragraphs in the author's master's thesis [53].

The Bell-LaPadula Model in the Literature The *Bell-LaPadula* Model [83, 84, 86] (BLP) defines 5 universal access rights: read-only, append, execute, read-write, and control [84, 89]. It resembles to military-style classifications [88]. Each subject is assigned a *clearance level* and objects are assigned a *classification* [83, 84]. In this thesis, we call them both security levels. The classifications form an ordering with higher values representing more classified information. In simplified terms, the model introduces the famous no-read-up and no-write down rules.[1] With these system-global rules, BLP is a MAC model. No-read-up means that subjects are not allowed to access objects with a higher classifications than their own. No-write-down means that a subject with a high clearance is not allowed to write to objects with a lower classification. Note that the Bell-LaPadula model can also be classified as information flow security model. By enforcing only these two rules, objects are successively assigned ever increasing classification levels. Thus, the model introduces so-called *trusted processes* which correspond to trusted subjects, allowed to decrease the classification of objects [89].

History Elliot Bell and Leonard J. La Padula produced the first volumes of the BLP Model [83, 84] "during two fiscal years" [87]. After one year of modeling work, a gap between practical implementation and the model still existed. Engineers had problems utilizing the model to build secure prototypes, which resulted in simplifications of the model. The concept of a subject's current security level and trusted subjects were introduced [87]. After two more years of work, the BLP model was published in its final version [86]. After 11 years, an interpretation of the BLP model for networks was developed [87]. It featured "hosts" as active entities and "connections" as resources [87].

Our Security Invariant Template As noticed, entities may act as subjects as well as objects. Since we do not distinguish the two, we only have sending entities and receiving entities. In addition, by looking at a single (possibly encrypted) connection, one cannot distinguish different access rights (such as read-only, append, execute, read-write, and control). For this interpretation, referring to the BLP example on page 25, the no-read-up and no-write-down rules are fulfilled for packets with arbitrary requests if flows from higher to lower security levels are prevented. Therefore, the simplified Bell-LaPadula invariant template was introduced as $\mathsf{m}\ (V,\ E)\ P \equiv \forall (s, r) \in E.\ P\ s \leq P\ r$.

[1]The BLP model also features an additional access matrix.

For the sake of presentiveness, the security levels were defined as $\{\mathsf{unclassified}, \mathsf{confidential}, \mathsf{secret}, \mathsf{topsecret}\}$. Technically, there is no reason that $\mathsf{topsecret}$ is the highest security level. We lift this restriction of a maximum level by modeling security levels as natural numbers: $\Psi = \mathbb{N}$. Hence, the template does not impose an upper limit on the "confidentialness".

The lowest security level is $0 = \bot$, which can be understood as $\mathsf{unclassified}$. Consequently, $1 = \mathsf{confidential}$, $2 = \mathsf{secret}$, $3 = \mathsf{topsecret}$, $\ldots$. The total order of the security levels now corresponds to the total order of the natural numbers '$\leq$'. It is important that there is a lowest security level (i.e., 0), otherwise, a unique and secure default parameter could not exist. Hence, it is not possible to extend the security levels to $\mathbb{Z}$ to model unlimited "un-confidentialness".

The formula of m, $\bot$, and the efficient version of the offending flows have already been presented. It is an information flow security strategy.

Alternative Definition For the following sections, we first introduce a helper function to simplify definitions. We call the function $\mathsf{succ\text{-}tran}$ and it represents the nodes that are transitively reachable in a graph by a starting node. Roughly speaking, $\mathsf{succ\text{-}tran}\ G\ v$ corresponds to all nodes reachable from v. It is defined with the help of the transitive closure of the edges of G.

Definition 10 (reachable hosts)**.**

$$\mathsf{succ\text{-}tran}\ (V,\, E)\ v \equiv \{v'.\ (v, v') \in E^{+}\}$$

Executable code for $\mathsf{succ\text{-}tran}$ over lists is available in an AFP entry [94]. Note that this executable code is not used for the simplified Bell-LaPadula model since the first definition without the transitive closure is way more efficient. However, it will be used for later definitions.

With this helper, the formula of the simple Bell-LaPadula invariant template can now be expressed differently. It requires that for any host $v \in V$, for any reachable host v', the security level of v' is greater or equal to the one of v.

$$\mathsf{m}\ (V,\, E)\ P\ \equiv\ \big(\forall v \in V.\ \ \forall v' \in \mathsf{succ\text{-}tran}\ (V,\, E)\ v.\ \ (P\ v) \leq (P\ v')\big)$$

Both definitions are equal[2]. The original definition only requires a condition for the edges in the graph while the new definition imposes a global condition. Since both are equal, it can be said that the second definition is also Φ-structured (although syntactically, it clearly is not) and that the offending flows are efficiently computable.

In general, this is not the case for all invariant templates and the simple Bell-LaPadula template is a notable exception. When an invariant template is defined with the help of $\mathsf{succ\text{-}tran}$, all paths between two hosts may need to be considered. In this case, since the set of offending flows is defined over all subsets of the edges in the graph, it may not be possible to simplify this definition. Consequently, the set of offending flows may be of exponential size. Hence, it becomes infeasible to efficiently execute algorithms built on top of it, most notably, the policy construction. On the other hand, all templates which are marked as Φ-structured in Table 6.1 are safe for the use in any algorithm.

[2] sinvar-BLPbasic-tancl

6.3 Simplified Bell-LaPadula with Trust

A simplified version of the Bell-LaPadula model was outlined the previous section. In this section, we extend this template with a notion of trust by adding a Boolean flag *trust* to the host attributes. This is a refinement to represent real-world scenarios more accurately and analogously happened to the original Bell-LaPadula model [87]. For a host v, let $\mathsf{level}\ (P\ v)$ denote v's security level and $\mathsf{trust}\ (P\ v)$ whether v is trusted. A trusted host can receive information of any security level and may declassify it, i. e., distribute the information with its own security level. For example, a trusted host is allowed to receive any information and with the $\mathsf{unclassified}$ level, it is allowed to reveal it to anyone. The template is thus formalized as follows.

$$\mathsf{m}\ (V,\, E)\ P\ \equiv\ \forall (s,r) \in E.\ \begin{cases} \mathsf{True} & \textbf{if}\ \ \mathsf{trust}\ (P\ r) \\ \mathsf{level}\ (P\ s)\ \leq\ \mathsf{level}\ (P\ r) & \textbf{otherwise} \end{cases}$$

The default attribute is $\bot = (\mathsf{unclassified}, \mathsf{untrusted})$,[3] where $\mathsf{untrusted}$ simply means False. It is Φ-structured, where $\Phi = (\lambda c_1\ c_2.\ \ \textbf{if}\ \mathsf{trust}\ c_2\ \textbf{then}\ \mathsf{True}\ \textbf{else}\ \mathsf{level}\ c_1 \leq \mathsf{level}\ c_2)$. As its simplified version, it is an information flow security strategy.

6.4 Communication Partners

This security invariant template can be understood as an implementation of Access Control Lists (ACLs). Traditional ACLs, for example represented as full access control matrix, scale quadratically in the number of hosts. This is hard to manage manually. To contain this quadratic configuration effort, this security invariant template was carefully designed to be contained to a subset of all policy entities. Three host attributes are defined:

$$\Psi = \mathsf{DontCare} \quad | \quad \mathsf{Care} \quad | \quad \mathsf{Master}\ \ \mathcal{V}\ \mathit{list}$$

The host attributes can be understood as follows. For all hosts labeled as DontCare, as long as they do not interact with Master hosts, no restriction is imposed on them. They are not part of the security requirement which is formalized by this invariant and correspond to the invariant's default attribute $\bot$. Hosts with a Master attribute are hosts which have an ACL enabled. The parameter of the Master attribute is the ACL, i. e., a list of hosts which are allowed to access this host. Finally, the Care attribute tells that the host itself participates in the security requirement which is formalized, but does not have ACL restrictions by itself.

Example. Hosts labeled with Master [] cannot be accessed at all.

Usually, if a host u is labeled with Master $[v]$, then v should be labeled as either Care or Master. This means that v is able to access u.

If v is labeled as DontCare, it cannot access u, though it is in u's access list. This covers the scenario that u is temporarily disabled but one does not want to update all access lists which may mention u. For this scenario, DontCare can be used as a simple Boolean flag which temporarily disables u.

[3]interpretation BLPtrusted: SecurityInvariant-IFS

A security requirement which is formalized with the Communication Partners invariant template focuses around Master hosts. Hosts with the $\mathsf{DontCare}$ attribute are completely out of focus for the formalized requirement. Hosts with Care attribute sit in-between: Master hosts may grant special rights to access them but they are also completely unconstrained on how they interact with the rest of the world, i.e., $\mathsf{DontCare}$ hosts.

Example. Let db_1 have the attribute Master $[h_1, h_2]$ and db_2 the attribute Master $[db_1]$. Let both h_1 and h_2 have the Care attribute. Let all other hosts have the default attribute of $\mathsf{DontCare}$. Then, h_1 and h_2 can access db_1 and can also access all other hosts, excluding db_2. db_1 can access all hosts. All hosts which have not been mentioned can freely access each other and h_1 and h_2.

$$\begin{tikzcd}
 & db_1 \arrow[loop above] \arrow[rr] \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] \arrow[drrr] & & db_2 \arrow[loop above] \arrow[dlll] \arrow[dl] \arrow[dr] & \\
h_1 \arrow[loop below] \arrow[rr, leftrightarrow] \arrow[rrrr, leftrightarrow, bend right] & & h_2 \arrow[loop below] \arrow[rr, leftrightarrow] & & h_3 \arrow[loop below]
\end{tikzcd}$$

We formalize the invariant. The invariant template does not restrict reflexive accesses, i.e., accesses from a host to itself. It can be expressed as one of the Φ-structured invariants. It is necessary that the predicate (here $\mathsf{allowed\text{-}flow} :: \Psi \Rightarrow \mathcal{V} \Rightarrow \Psi \Rightarrow \mathcal{V} \Rightarrow \mathbb{B}$) has access to both the host attribute and the name of the host in order to look up the name of a host in the access control list.

$$\mathsf{m}\ (V,\ E)\ P \equiv \forall (s,r) \in E.\ s \neq r \longrightarrow \mathsf{allowed\text{-}flow}\ (P\ s)\ s\ (P\ r)\ r$$

The following table implements the access control restrictions $\mathsf{allowed\text{-}flow}$. The first column Ψ_s is the host attribute of the sender, the second column the name of the sender s. Likewise, the third column is the host attribute of the receiver Ψ_r, followed by the name of the receiver r. Next, the result (rslt), and an explanation are given. Whenever the name of the sender, the receiver, or the ACL itself is not important for a rule, it is left out with an underscore '_'.

The use of the $\mathsf{DontCare}$ attribute has an additional positive effect. It prevents that stale access list entries have an undesired effect.

Example. Let u be labeled with Master $[v]$. Now, host v is removed but the access list at u is not updated because it is expected that v will be re-added again soon. However, it was forgotten about v and the stale access list entry at v remains. Of course, it is easy to warn about stale access list entries, yet, they may be desirable for a transition period. Somewhere in the future, a new host with accidentally the same name v is added. Because v is not part of the security requirement formalized by this invariant, it is set to $\bot = \mathsf{DontCare}$. Therefore, though the stale access list entry still exists, v is not granted access to u and the stale access list can thus cause no immediate harm.

Ψ_s	s	Ψ_r	r	rslt	explanation
DontCare	_	DontCare	_	✓	No restrictions
DontCare	_	Care	_	✓	No restrictions
DontCare	_	Master _	_	✗	Hosts which are not part of the formalized security requirement must not access hosts with ACL restrictions
Care	_	Care	_	✓	No restrictions
Care	_	DontCare	_	✓	No restrictions
Care	s	Master M	_	$s \in M$	s must be in the ACL M
Master _	s	Master M	_	$s \in M$	– " –
Master _	_	Care	_	✓	No restrictions
Master _	_	DontCare	_	✓	No restrictions

6.5 Comm. With

The "Communicate With" security invariant template is an experimental, white-listing, transitive access control list model. It is not Φ-structured and we are not aware of an efficient implementation for the offending flows. Therefore, it serves primarily for demonstration purposes. With the improvements demonstrated in Chapter 8, the invariant can be practically applied but there is no guarantee that a computed policy may be maximum. The same holds for all following non-Φ-structured invariant templates.

The type for the host attributes is a list of hosts, i.e., $\Psi = \mathcal{V}$ *list* and represents to the only hosts that may be accessed, even transitively. The default attribute is the empty list $\bot = []$. The formula for m states that for any host v, all nodes a which are reachable from v must be in v's access list.

$$\mathsf{m}\ (V,\ E)\ P\ \equiv\ \forall v \in V.\ \ (\forall a \in \mathsf{succ\text{-}tran}\ \ (V,\ E)\ \ v.\ \ a \in P\ v)$$

Alternatively, the invariant can be expressed as for all edges in the transitive closure over E, i.e., all transitive accesses, the accessed host v_2 must be in the access list of the accessing host v_1.

$$\mathsf{m}\ (V,\ E)\ P\ \equiv\ \forall (v_1, v_2) \in E^+.\ \ v_2 \in (P\ v_1)$$

Both definitions are equal[4].

Example. For example, assume $V = \{v_1, v_2, v_3\}$ and let $P\ v_1 = [v_2,\ v_3]$. This means v_1 must only access v_2 and v_3 transitively. Let $E = \{(v_1, v_2),\ (v_2, v_3)\}$, then v_1 accesses v_3 transitively. With v_1's host attributes, a direct access (v_1, v_3) would be allowed. In order for the invariant to be fulfilled for E, additionally, v_2 must be allowed to access v_3.

$$\begin{tikzcd} v_1 \arrow[r] & v_2 \arrow[r] & v_3 \end{tikzcd}$$

Assume there is another node v_4 and there are additional edges from v_1 to v_3 and from v_3 to v_4.

[4] ACLcommunicateWith-sinvar-alternative

$$\begin{tikzcd} v_1 \arrow[r] \arrow[rr, bend right] & v_2 \arrow[r] & v_3 \arrow[r] & v_4 \end{tikzcd}$$

Assume v_1 must not access v_4. Then any combination of edges on any path from v_1 to v_4 which disconnects v_1 and v_4 may be an offending flow. Hence, the offending flows are not uniquely defined. The choice that the invariant incorporates transitive accesses results in the fact that the set of offending flows may be even exponential. For example, let $E = \{(v_1, v_2),\ (v_1, v_3),\ (v_2, v_3),\ (v_3, v_4)\}$ and let P be such that every host can access all other hosts, except that v_1 must not access v_4. Then the set of offending flows is $\{\{(v_1, v_2), (v_1, v_3)\}, \{(v_1, v_3), (v_2, v_3)\}, \{(v_3, v_4)\}\}$.

$$\begin{tikzcd} v_1 \arrow[r, dashed] \arrow[rr, bend right, dashed] & v_2 \arrow[r] & v_3 \arrow[r] & v_4 \end{tikzcd}$$

$$\begin{tikzcd} v_1 \arrow[r] \arrow[rr, bend right, dashed] & v_2 \arrow[r, dashed] & v_3 \arrow[r] & v_4 \end{tikzcd}$$

$$\begin{tikzcd} v_1 \arrow[r] \arrow[rr, bend right] & v_2 \arrow[r] & v_3 \arrow[r, dashed] & v_4 \end{tikzcd}$$

6.6 Not Comm. With

The "Not Communicate With" security invariant template is an experimental, black-listing, transitive access control list model. It is not Φ-structured and we are not aware of an efficient implementation for the offending flows. Therefore, it serves primarily for demonstration purposes. It complements the previously presented "Communicate With" template.

The type for the host attributes is a set of hosts, i.e., $\Psi = \mathcal{V}\ \mathit{set}$ and corresponds to the hosts that a host must not transitively access. For example, let $P\ v = \{v_1,\ v_2,\}$, then v must not access v_1 and v_2 transitively. The default attribute is the universe, i.e., the set of all elements, $\bot = \mathsf{UNIV}$. The universe is represented symbolically and executable code can be generated for any (finite or infinite) type $\mathcal{V}$ of nodes. Analogously to the previous invariant, the formula states that any hosts which are accessible from some starting host v must not be in v's "not-access-list".

$$\mathsf{m}\ (V,\ E)\ P\ \equiv\ \forall v \in V.\ \ (\forall a \in \mathsf{succ\text{-}tran}\ \ (V,\ E)\ \ v.\ \ a \notin P\ v)$$

Technically, this invariant is exactly the inverse of the previous invariant.[5] If all host attributes are inversed, i.e., $\forall v.\ P_{\text{new}}\ v = \mathsf{UNIV} \setminus (P\ v)$, then the two invariants are equal. All insights about the "Communicate With" template also apply to this template. The main difference it that in this invariant, Ψ is a set to represent UNIV, which is not possible in the previous invariant.

6.7 Dependability

Hosts provide a service on which other hosts might depend. The "Dependability" invariant template was designed to limit a network's dependence on a service. Note that we do not model availability; this access control model describes a limit of how many hosts may at most access one specific host. Every host is assigned a dependability level, represented as natural number: $\Psi = \mathbb{N}$. This number encodes the maximal number of hosts which may transitively depend on a host.

Let card be the cardinality of a set. Then, the invariant can be expressed as follows:

$$\mathsf{m}\ (V,\ E)\ P\ \equiv\ (\forall v \in V.\ \ \mathsf{card}\ \ (\mathsf{succ\text{-}tran}\ (V,\ E)\ v) \leq (P\ v))$$

[5]ACLcommunicateNotWith-inverse-ACLcommunicateWith

The template can also be expressed as a predicate over all edges. Note that it is not Φ-structured since the predicate depends on the graph.

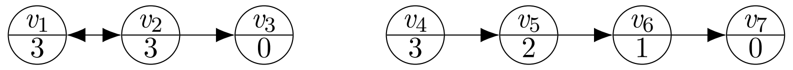
$$\mathsf{m}\ (V,\ E)\ P\ \equiv\ (\forall (v_1, v_2) \in E.\ \ \mathsf{card}\ \ (\mathsf{succ\text{-}tran}\ (V,\ E)\ v_1) \leq (P\ v_1))$$

The dependability level corresponds to the number of hosts a host may transitively access. Hence, it is an access control strategy. By default, a host does not have any access rights: $\bot = 0$.

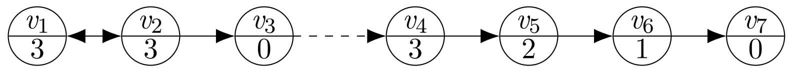
A high dependability level of a host may represent that many other hosts depend on the service provided by this hosts. Hence, high numbers are indicators for important hosts which may require special care.

The invariant is hard to use for the following two reasons. First, the default dependability level means that hosts may not communicate at all. Consequently, every host needs to be assigned a dependability level manually. Second, due to the use of succ-tran, in case of a violation, the number of offending flows may be exponential. Therefore, we propose the following use of the invariant. The dependability levels can be set automatically to a valid host attribute configuration and the invariant should be excluded from automated policy construction. Setting the level for each host v to card (succ-tran $(V,\ E)\ v$) provides a valid configuration[6]. This avoids manual configuration and makes sure that the offending flows are always empty. Whenever there is an update to the policy, the dependability levels can be recomputed. If a significant change in the levels due to the update has occurred, this raises a pointer for further manual inspection. This may uncover unintended side-effects of a policy change.

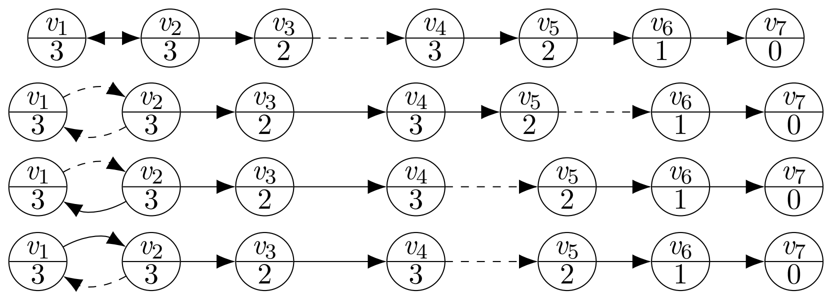
Example. Assume $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and $E = \{(v_1, v_2), (v_2, v_1), (v_2, v_3), (v_4, v_5), (v_5, v_6), (v_6, v_7)\}$. The nodes v_1, v_2, v_3 are disconnected from the other part of the graph. Below, the dependability levels are visualized for every node beneath its name. Noteworthy, v_1 needs a dependability level of at least 3 because it can access v_2, v_3, and can also transitively access itself.



Now assume we connect v_3 and v_4. Due to v_3's dependability level, this only causes one offending flow, namely exactly the flow we added.

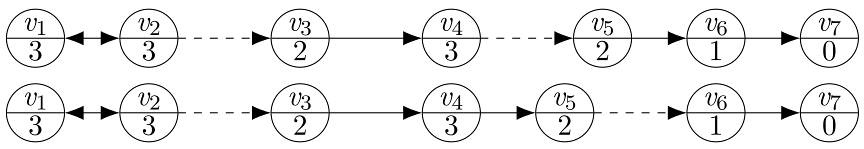


An approach to fix the violation by increasing v_3's dependability level fails. Due to the use of succ-tran, there are multiple possibilities for the offending flows. The following offending flows are created by this:

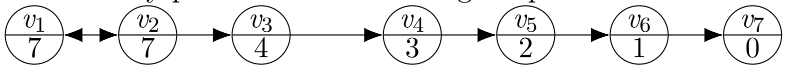


---

[6]dependability-fix-nP-impl-correct



Calculating new dependability levels yields the following results. Though only the edge (v_3, v_4) was added, the vast increase in the level at for example v_1 indicates that this simple additionally permitted flow has large impact on other areas of the policy.



Dependability Non-Refl As the previous example may have hinted, it may be unexpected that a host can depend on itself. The invariant template can also be expressed to exclude accesses from a host to itself.

$$\mathsf{m}\ (V,\ E)\ P\ \equiv\ (\forall v \in V.\ \ \mathsf{card}\ \ ((\mathsf{succ\text{-}tran}\ (V,\ E)\ v) \setminus \{v\}) \leq (P\ v))$$

Apart from reflexive accesses, this version of the dependability templates behaves as the original version.

Example. The default Dependability invariant also counts an access from a node to itself. Hence, the host in the illustration below (left) needs a dependability level of at least one. With the version of the Dependability invariant which excludes reflexive accesses, the node can have the default dependability level (illustrated below, right).

6.8 Domain Hierarchy

The domain hierarchy template mirrors hierarchical access control structures.

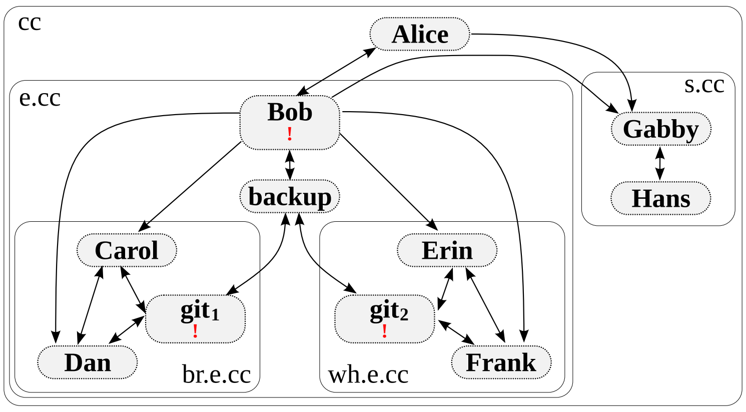


Figure 6.1: Policy and host mapping of **cc**.

It is best introduced by example. The tiny car company (*cc*) consists of the two sub-departments engineering (*e*) and sales (*s*). The engineering department itself consists of the brakes (*br*) and the wheels (*wh*) department. This tree-like organizational structure is illustrated in Figure 6.2. We denote a position by the fully qualified domain name, e.g., *wh.e.cc* uniquely identifies the wheels department. Let '$\sqsubseteq$' denote the '*is below or at the*

$$\begin{tikzcd} & & \mathbf{cc} & \\ & \mathbf{e} \arrow[ur, phantom, "\sqsubseteq"] & & \mathbf{s} \arrow[ul, phantom, "\sqsupseteq"] \\ \mathbf{br} \arrow[ur, phantom, "\sqsubseteq"] & & \mathbf{wh} \arrow[ul, phantom, "\sqsupseteq"] & \end{tikzcd}$$

Figure 6.2: Organizational structure of **cc**.

same hierarchy level' relation, e.g., $wh.e.cc \sqsubseteq wh.e.cc$, $wh.e.cc \sqsubseteq e.cc$, and $wh.e.cc \sqsubseteq cc$. However, $wh.e.cc \not\sqsubseteq br.e.cc$ and $br.e.cc \not\sqsubseteq wh.e.cc$. The '$\sqsubseteq$' relation denotes a partial order[7]. The company's command structures are strictly hierarchical, i.e., commands are either exchanged in the same department or travel from higher departments to their sub-departments. Formally, the receiver's level $\sqsubseteq$ sender's level. For a host v, let $\mathsf{level}\ (P\ v)$ map to the fully qualified domain name of v's department. For example in Figure 6.1, $\mathsf{level}\ (P\ Bob) = e.cc$.

As in many real-world applications of a mathematical model, exceptions exist. Those are depicted by exclamation marks in Figure 6.1. For example, Bob as head of engineering is in a trusted position. This means he can operate as if he were in the position of Alice. This implies that he can communicate on par with Alice, which also implies that he might send commands to the sales department. We model such exceptions by assigning each host a trust level. This trust level specifies up to which position in the hierarchy this host may act. For example, Bob in $e.cc$ with a trust level of 1 can act as if he were in cc, which means he has the same command power as Alice. Let $\mathsf{trust}\ (P\ v)$ map to v's trust level. We define a function chop which takes two parameters: a domain name called *level* and a natural number called *trust*. It returns the domain name *level* with *trust* sub-domains chopped off. For example, $\mathsf{chop}\ br.e.cc\ 1 = e.cc$ and $chop\ br.e.cc\ 2 = cc$. With this, the security invariant template can be formalized as follows.

$$\mathsf{m}\ (V,\, E)\ P \;\equiv\; \forall (s,r) \in E.\ \ \mathsf{level}\ (P\ r)\ \sqsubseteq\ \mathsf{chop}\ \big((\mathsf{level}\ (P\ s))\ (\mathsf{trust}\ (P\ s))\big)$$

In the Domain Hierarchy, the default attribute is[8] a special value $\bot$ with a trust of zero and which is at the lowest point in the hierarchy, i.e., $\forall\ l.\ \bot \sqsubseteq l$. Finally, it is worth mentioning that the $\sqsubseteq$-relation forms a lattice[9], which is a desirable structure for security classes [82].

6.9 NoRefl

Entities in a policy, called hosts throughout this thesis, may correspond to several network entities. For example, what occurs as a single host in the policy may correspond in the implementation to a complete network subnet or another group of physical or virtual hosts. As long as there is a one-to-one mapping between policy entities and entities in the implementation, reflexive policy rules (i.e., rules of the form (v, v)) correspond to in-host communication and may be disregarded from a network point of view. However, if a host in the policy corresponds to a group of hosts in the implementation, a reflexive policy rule

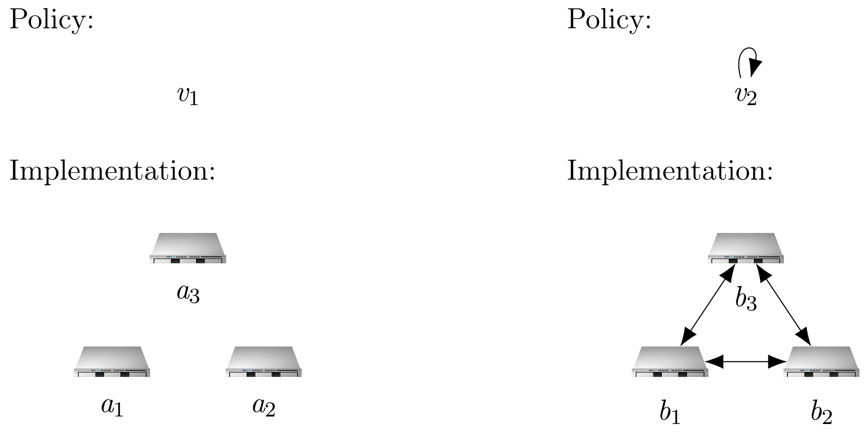
[7]instantiation domainNameDept :: order

[8]interpretation DomainHierarchyNG: SecurityInvariant-ACS

[9]instantiation domainName :: lattice

dictates for the implementation whether the hosts in the group may communicate among themselves. Details will be discussed in Section 10.3.4.1.

Example. In the illustration below, let v_1 and v_2 be nodes in the security policy. In the network, let v_1 and v_2 correspond to 3 physical machines each. The reflexive policy rule (v_2, v_2) decides that the machines which correspond to v_2 in the policy (illustrated on the right) can communicate with each other, while the machines corresponding to v_1 cannot.



Effectively, hosts in the policy can be lifted to role descriptions by having them represent several physical hosts. A host in the policy can now be understood as a role. In a physical network implementation, many physical hosts may have the same role. They inherit the access rights of their role. Reflexive edges in the policy hence encode whether hosts of the same role may communicate with each other.

Note that this understanding of roles does not correspond to RBAC [95]. With our simple lifting, physical hosts cannot inherit multiple roles or switch them dynamically, but roles can have several host attributes.

The decision whether reflexive flows are allowed can be encoded with the following security invariant template. We distinguish between two host attributes. Either reflexive flows are allowed or they are denied: $\Psi = \{\mathsf{Refl},\ \mathsf{NoRefl}\}$. By default, reflexive flows are denied, $\bot = \mathsf{NoRefl}$. Then, the security invariant template can be formalized as follows. For all reflexive flows, the corresponding host must have the Refl attribute.

$$\mathsf{m}\ (V,\ E)\ P\ \equiv\ \forall (s, r) \in E.\ \ s = r \longrightarrow P\ s = \mathsf{Refl}$$

Noteworthy, this invariant can be interpreted as both, ACS and IFS strategy.[10] Recalling that the strategy was defined by whether the offending host is the sender or the receiver (Definition 6), and for this special invariant where sender and receiver are equal, the peculiarity becomes obvious. We chose it to be an access control strategy. This will have the additional advantage, later when translating to a stateful policy, that the requirements for access control security strategies are less strict (cf. Chapter 9), which makes the resulting stateful policy more permissive and hence easier to implement in real-world scenarios.

[10] NoRefl-SecurityInvariant-IFS

6.10 NonInterference

Parts of the following paragraph about the non-interference model in the programming languages literature have been previously published in the author's master's thesis [53].

History of NonInterference for IFS in Programming Languages Information flow strategies define valid information channels between subjects [89]. It originated from programming language research. The concept of *Information Flow Security* (IFS) is to control the flow of information within programs. The original idea behind IFS is that a programmer labels confidential and non-confidential information. The compiler subsequently verifies that no information flow from confidential to non-confidential channels occurs. The confidential information must not interfere with non-confidential information. Originally, non-interference was introduced by Goguen and Meseguer [96] describing that a subject's possible actions must be invisible to another subject to fulfill non-interference. However, this model was recognized incorrect and several efforts were made to correct the model [97]. A satisfactory model finally was constructed featuring the "complete sequence of actions performed subsequent to a given action" [97]. The now prevalent [98] IFS model can be described as follows: An observer with a low security level observes the public output (the output the observer can legally obtain with her security level) of a run of a system. When the same observer now observes a similar run, which differs to the first run only in actions of higher security levels, the observer must observe exactly the same output. Intuitively, this model compares any two runs of a program which only differ in their secret input and states for the program to be secure, the observable public output must be equal. This model has the downside that it cannot publish any computation result based on private data. Consider for example a login function which checks whether a user-supplied password matches the user's deposited password. We expect that a system can tell any unauthenticated user that her login attempt failed. It is thus necessary that the return value of the login function is public information. As the function is working with the private database of user passwords, the return value of this function is yet private. This renders IFS unsuitable or impractical for many applications. Another scenario is a statistical computation which aggregates many private user records and performs an anonymous statistical aggregation which can be published without any privacy concerns. However, this scenario can also not be modeled by traditional IFS. Myers and Liskov [99] address these issues by presenting a new IFS model. The model is called decentralized as each user may impose individual restrictions on her data. In their model, users can declassify data – that is, removing restrictions – only if they are or indirectly act for the owner of the data. The authors admit that their model "does not work well in large, networked systems, where varying levels of trust exist among nodes in the network" [99]. As the presented models focus mainly on actions which are invoked on data, Broberg and Sands [98] describe these models as flow insensitive. They present a flow sensitive technique (flow locks) and introduce a new representation of an attacker's knowledge.

Our Security Invariant Template Our NonInterference invariant template for computer networks formalizes the requirement that two hosts must not interact with each other. It is a very strict information flow security strategy. There are only two possible host attributes: $\Psi = \{\mathsf{Interfering}, \mathsf{Unrelated}\}$. All hosts which are marked as $\mathsf{Interfering}$ must not be able to interfere with each other, by any means. The invariant is very strict and by default, all hosts are assumed as $\bot = \mathsf{Interfering}$. It is not Φ-structured and we are not aware of an efficient implementation for the offending flows.

For a starting node v, and a graph G, we define all nodes which are reachable from v, assuming all edges in the graph were bidirectional.

Definition 11 (reachable hosts in an undirected graph)**.**

$$\mathsf{undirected\text{-}reachable}\ (V,\ E)\ v \equiv (\mathsf{succ\text{-}tran}\ (V,\ E \cup \{(r, s) \mid (s, r) \in E\})\ v) \setminus \{v\}$$

The definition excludes v from the result, otherwise, the invariant can never be fulfilled because any interfering host would interfere with itself. With this, the invariant can be formalized as follows: For all $\mathsf{Interfering}$ hosts, they must only be able to reach hosts which are not $\mathsf{Interfering}$.

$$\begin{aligned}\mathsf{m}\ (V,\ E)\ P \ \ &\equiv\ \forall v \in \{v' \in V \mid P\ v' = \mathsf{Interfering}\}.\\ &\qquad\qquad \{P\ v' \mid v' \in \mathsf{undirected\text{-}reachable}\ G\ v\} \subseteq \{\mathsf{Unrelated}\}\end{aligned}$$

Example. For example, assume $V = \{v_1, v_2, v_3, v_4\}$ and let $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4)\}$. Assume both v_1 and v_4 are $\mathsf{Interfering}$ and v_2 and v_3 are $\mathsf{Unrelated}$. This setting is similar to the example of "Comm. With" in Section 6.5: Any combination of edges on any path from v_1 to v_4 which disconnects v_1 and v_4 may be an offending flow. The set of offending flows is $\{\{(v_1, v_2), (v_1, v_3)\}, \{(v_1, v_3), (v_2, v_3)\}, \{(v_3, v_4)\}\}$.

$$\begin{tikzcd} v_1 \arrow[r, dashed] \arrow[rr, dashed, bend right] & v_2 \arrow[r] & v_3 \arrow[r] & v_4 \end{tikzcd} \qquad \begin{tikzcd} v_1 \arrow[r] \arrow[rr, dashed, bend right] & v_2 \arrow[r, dashed] & v_3 \arrow[r] & v_4 \end{tikzcd} \qquad \begin{tikzcd} v_1 \arrow[r] \arrow[rr, bend right] & v_2 \arrow[r] & v_3 \arrow[r, dashed] & v_4 \end{tikzcd}$$

However, NonInterference is a stricter invariant than "Comm. With". If direction of the flow (v_3, v_4) is reversed, the new policy can be visualized as follows. The "Comm. With" (example in Section 6.5) invariant is fulfilled for the new policy.

$$\begin{tikzcd} v_1 \arrow[r] \arrow[rr, bend right] & v_2 \arrow[r] & v_3 & v_4 \arrow[l] \end{tikzcd}$$

However, NonInterference interprets the graph as if it were undirected and still concludes that v_1 and v_4 are interfering. The offending flows are illustrated below.

$$\begin{tikzcd} v_1 \arrow[r, dashed] \arrow[rr, dashed, bend right] & v_2 \arrow[r] & v_3 & v_4 \arrow[l] \end{tikzcd} \qquad \begin{tikzcd} v_1 \arrow[r] \arrow[rr, dashed, bend right] & v_2 \arrow[r, dashed] & v_3 & v_4 \arrow[l] \end{tikzcd} \qquad \begin{tikzcd} v_1 \arrow[r] \arrow[rr, bend right] & v_2 \arrow[r] & v_3 & v_4 \arrow[l, dashed] \end{tikzcd}$$

6.11 Policy Enforcement Point

Hosts may belong to a certain domain. Sometimes, a pattern where intra-domain communication between domain members must be approved by a central instance is required.

Example. Let several virtual machines belong to the same domain and a secure hypervisor manage intra-domain communication. As another example, inter-device communication of slave devices in the same domain is controlled by a central master device.

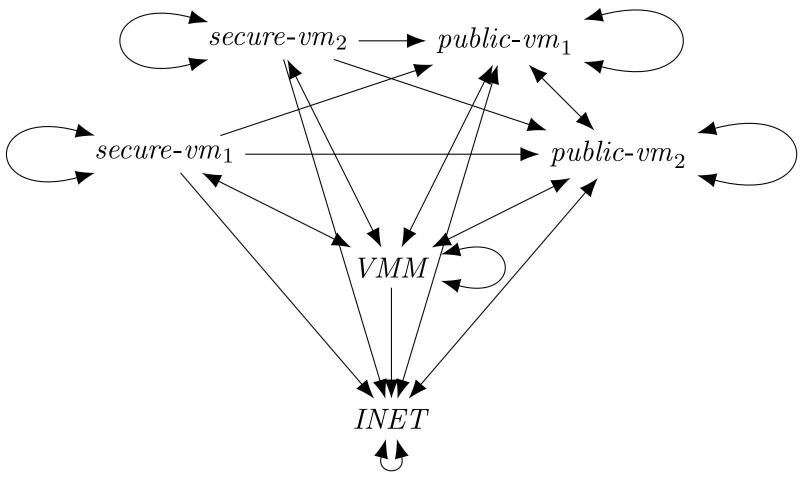
We call such a central instance an application-level 'policy enforcement point' and present a template for this architecture. Five host roles are distinguished: $\Psi = \{\mathsf{PolEnforcePoint}, \mathsf{PolEnforcePointyIN}, \mathsf{DomainMember}, \mathsf{AccessibleMember}, \mathsf{Unassigned}\}$. A PolEnforcePoint, a policy enforcement point accessible from the outside (PolEnforcePointIN), a DomainMember, a less-restricted AccessibleMember which is accessible from the outside world, and a default value $\bot$ = Unassigned that reflects 'none of these roles'. The following table implements the access control restrictions. The role of the sender (snd), role of the receiver (rcv), the result (rslt), and an explanation are given.

snd	rcv	rslt	explanation
PolEnforcePoint	_	✓	Can send to the world.
PolEnforcePointIN	_	✓	— "—
DomainMember	DomainMember	✗	Must not communicate directly. May communicate via PolEnforcePoint or PolEnforcePointIN.
DomainMember	_	✓	No restrictions for direct access to outside world. Outgoing accesses are not within the invariant's scope.
AccessibleMember	DomainMember	✗	Must be approved.
AccessibleMember	_	✓	No further restrictions, accessible members are also accessible among each other.
Unassigned	Unassigned	✓	No restrictions.
Unassigned	PolEnforcePointIN	✓	Accessible from outside.
Unassigned	PolEnforcePoint	✗	Not accessible from outside.
Unassigned	AccessibleMember	✓	Directly accessible from the outside.
Unassigned	DomainMember	✗	Protected from outside world.

This template is minimalistic in that it only restricts accesses to members (from other members or the outside world), whereas accesses from members to the outside world are unrestricted. It can be implemented by a simple table lookup. In-host communication is allowed by adding $s \neq r$.

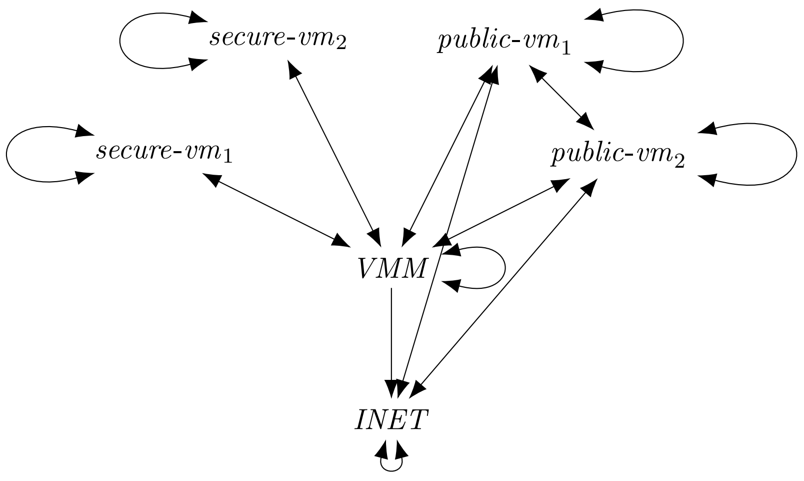
$$\mathsf{m}\ (V,\, E)\ P \quad \equiv \quad \forall (s,\, r) \in E,\ \ s \neq r.\ \ \mathsf{table}\ (P\ s)\ (P\ r)$$

Example. In a cloud environment, for brevity, let *VMM* denote a secure hypervisor. The hypervisor has VM-specific security policies installed and acts as a broker between virtual machines. The *VMM* is configured as PolEnforcePoint. The VMs $\textit{secure-vm}_1$ and $\textit{secure-vm}_2$ run security-critical tasks and accesses between them and to them is mediated by the *VMM*. They are assigned the DomainMember host attribute. In contrast, $\textit{public-vm}_1$ and $\textit{public-vm}_2$ do not have special security requirements and are not mediated by the *VMM*. They are assigned the AccessibleMember host attribute, which makes them also directly accessible from the *INET*. Still, the two public VMs have slightly more access rights than an arbitrary host in the *INET*: $\textit{public-vm}_1$ and $\textit{public-vm}_2$ may access the *VMM* and hence the secure VMs if the *VMM* permits it. The corresponding maximum policy for this setting is visualized below.



This invariant template only controls how DomainMembers can be accessed. It does not restrict the information flow from DomainMembers. Therefore, the secure VMs may freely access anyone but may not be accessed by anyone, except the *VMM*.

As an additional security requirement, let the secure VMs have confidential data. This is formalized with the Simplified Bell-LaPadula with Trust invariant (Section 6.3). The *VMM* is trusted and may declassify this data (security level unclassified and trusted). This prevents that the secure VMs may initiate any connections to the outside by themselves and must have everything approved by the *VMM*.



6.12 Sink

Some hosts may be information sinks. That means, no information must leave those hosts. The information flow security strategy presented in this section formalizes the notion of information sinks.

Example. Assume logging information should not leave a central log server. `syslog` messages are usually sent via UDP (RFC 5426 [100]), which enables a purely unidirectional channel, even without TCP acknowledgements. In cyber-physical systems, some actuators without sensors may also only receive commands and do not transmit any answers. They can also be considered information sinks.

An information sink may not be limited to a single host but a pool of hosts. Though no information must leave the pool, it may be desirable that the pool can cooperate.

For this security invariant, we distinguish between three types of hosts. The type of host attributes is defined as $\Psi = \{\mathsf{Sink}, \mathsf{SinkPool}, \mathsf{Unassigned}\}$. This models strict information Sinks where no information must leave this host, pools of information sinks (SinkPool) which may collaborate but no information must leave the pool, and Unassigned which does not impose any restrictions. The default value is $\bot = \mathsf{Unassigned}$.

The following table formalizes the meaning of the individual host attributes.

snd	rcv	rslt	explanation
Sink	_	✗	No data must leave a Sink
SinkPool	SinkPool	✓	The pool can communicate with its members
SinkPool	Sink	✓	The pool can also send to individual sinks (but not the other way round)
SinkPool	_	✗	The pool cannot send data to the outside
Unassigned	_	✓	Everything else is not constrained

With the table, the security invariant can be expressed as:

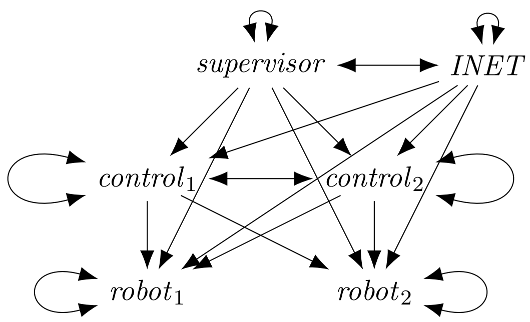
$$\mathsf{m}\ (V,\, E)\ P \;\;\equiv\;\; \forall (s,\, r) \in E,\;\; s \neq r.\;\; \mathsf{table}\ (P\ s)\ (P\ r)$$

Example. In this example, we model the information flow of trade secrets in a simplified SCADA factory network. This example is not concerned with the access control permissions, but only with information flow. The hosts in the example are a *supervisor* for the complete production process, two *control*lers which control the production units, and two production units (called *robot*s). In addition, there is a host that represents the Internet.

The supervisor may both access the Internet and send commands to the control units. The control units drive the production units. Since each control unit may send commands to both production units, the control units might need to synchronize for this task. The graph for the described scenario can be visualized as follows.

$$\begin{tikzcd}
 & \mathit{supervisor} \arrow[dl] \arrow[dr] \arrow[rr, leftrightarrow] & & \mathit{INET} \\
\mathit{control}_1 \arrow[rr, leftrightarrow] \arrow[d] \arrow[drr] & & \mathit{control}_2 \arrow[d] \arrow[dll] & \\
\mathit{robot}_1 & & \mathit{robot}_2 &
\end{tikzcd}$$

Let the trade secrets be the production process of a product, i.e., how the robots are driven. The main concern is that these secrets must not leak. The secret production steps are only encoded in the *control* units, not the *supervisor*. To prevent information leakage, the *control* units are information sinks, however, they need to cooperate, hence, they are labeled as SinkPool. The *robot*s only need to receive commands and must not leak any information, therefore, they are just Sinks. The *supervisor* and *INET* may be left unconfigured, i.e., set to $\bot$. The policy visualized above fulfills this requirement. However, the following (maximum) policy also fulfills the formalized requirement.



The picture shows that the control units and robots do not leak any information. However, any node, in particular *INET*—though it cannot receive an answer—may send arbitrary commands to the robots. This is because the *Sink* invariant is an information flow security invariant and does not restrict access control.

6.13 Subnets

This invariant template formalizes the logical partition of a network into different segments. We call each segment a subnet.

A host can either be a member of a specific subnet, can be a border router between subnets, or may not be part of the formalized security goal. Many different subnets can be formalized with one instance of this invariant. The type for host attributes is formalized as $\Psi = \{\mathsf{Subnet}\ \mathbb{N},\ \mathsf{BorderRouter}\ \mathbb{N},\ \mathsf{Unassigned}\}$. The types Subnet and BorderRouter have as parameter a natural number. This number indicates the specific subnet. By default, a host carries the $\bot = \mathsf{Unassigned}$ attribute.

Example. Let $P\ v_1 = \mathsf{Subnet}\ 8$, $P\ v_2 = \mathsf{Subnet}\ 8$, $P\ v_3 = \mathsf{Subnet}\ 42$, and $P\ v_4 = \bot$. Then v_1 and v_2 are in the same subnet, v_3 is in a different subnet, and v_4 is in a completely different network segment (which is out of scope for the security requirement that is formalized with P).

All subnets in this model are identified by a natural number. Hence, two subnets with a different number are distinct by definition. This means that this invariant does not permit different subnets to have overlapping IP address ranges. Since all entities are distinct and a subnet only consists of the entities in it, overlapping IP ranges cannot occur.[11]

[11] In general, we did not model IP address ranges yet. So far, entities are polymorphic over type $\mathcal{V}$. For example, $\mathcal{V}$ can be the set of all IPv4 addresses and entities are thus individual IP addresses. But since all elements of $\mathcal{V}$ must be pairwise distinct, representing entities by overlapping IP ranges is not possible. One abusive workaround would be to define $\mathcal{V}$ as a set of strings and encode (possibly overlapping) IP addresses in CIDR notation into these strings. While at the abstraction level of our policy, two different strings represent different policy entities, soundness problems on the network level occur if a firewall does not interpret them as strings but overlapping IP ranges. We discuss the relation between policy entities and sets of IP addresses in detail in Part II: Overlapping IP address ranges with different policy actions (allow or deny) bring up the question whether the intersection should be allowed or denied. In contrast to our policy with only positive rules where order does not matter, we discuss the first-matching semantics of firewalls in Chapter 12. When entities correspond to IP addresses, we consider IP address spoofing in Chapter 13. In Chapter 14, when inferring a policy from a network-level firewall ruleset, entities will correspond to sets of IP addresses. Theorem 22 will conclude that the IP address ranges of all such computed entities are indeed disjoint. Therefore, it is possible to have entities with overlapping IP address ranges in a policy (for example with the string workaround), but one needs to verify that the network-level connectivity enforced by a firewall corresponds to the desired policy. Our tool *fffuu* (Section 14.7) is suitable for this task.

Border routers may connect different subnets by communicating with each other. The design decision for this invariant is that they are extremely restricted in their access rights. A core idea is that a host in a specific subnet must not be reachable from "unconfigured" (i.e., $\bot$) parts of the network, not even indirectly. Also, a host must not be reachable from other subnets, also, not even indirectly. Since border routers can communicate with each other, to fulfill the design goals, a border router must not be allowed to establish a connection to a member of a subnet.

Example. Let $P\ v_1 = \mathsf{Subnet}\ 1$, $P\ v_2 = \mathsf{Subnet}\ 2$, $P\ v_3 = \mathsf{BorderRouter}\ 1$, and $P\ v_4 = \mathsf{BorderRouter}\ 2$. The border routers v_3 and v_4 may communicate. The host v_2 may send packets to its border router v_4. Since v_1 and v_2 are in different subnets, v_2 should not be able to reach v_1. The possible path $v_2 \longrightarrow v_4 \longrightarrow v_3 \longrightarrow v_1$ is prohibited by disallowing that border routers can establish connections to subnet hosts – including hosts of their own subnet.

Note that v_1 could set up a connection to v_3. If this is a stateful connection which also permits replies from v_3 back to v_1 (cf. Chapter 9), a channel between v_1 and v_2 is established. Since this requires the consent of v_1 which needs to set up the stateful connection, v_2 is still not permitted to access v_1 on its own and this setting does not contradict the intentions of the invariant template.

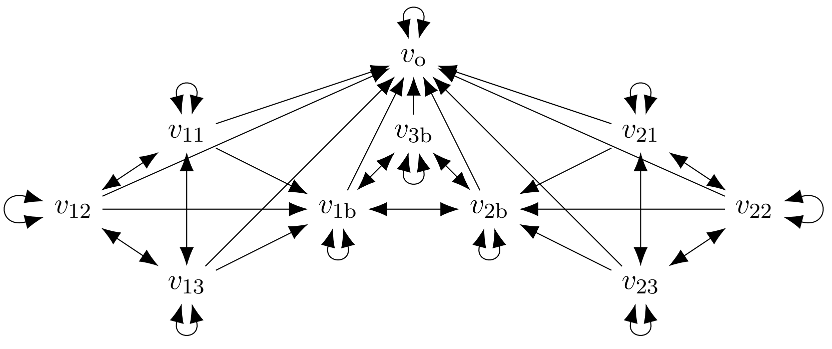
Similar to previous templates, the invariant template is formalized with the help of a table.

$$\mathsf{m}\ (V,\, E)\ P \quad \equiv \quad \forall (s,\, r) \in E.\ \ \texttt{table}\ (P\ s)\ (P\ r)$$

The access rights of the different host attributes are formalized as follows.

A violation of the invariant template occurs exactly iff[12] $\bot$ is trying to connect to a non-$\bot$ host, or a host is trying to connect to a host or a border router of a different subnet, or a border router is trying to access a host.

Example. Let the hosts v_{11}, v_{12}, v_{13} be members of subnet 1 and let $v_{1\mathrm{b}}$ be the border router of subnet 1. Likewise, let the hosts v_{21}, v_{22}, v_{23} be members of subnet 2 and let $v_{2\mathrm{b}}$ be the border router of subnet 2. Let $v_{3\mathrm{b}}$ be the border router of subnet 3 and let v_o be an arbitrary $\bot$ host. The maximum policy for this setting can be visualized as follows.



It can be seen that the hosts in each subnet have full connectivity among themselves. Everyone can access the outside-world host v_o. The border routers have full connectivity

[12]SINVAR_Subnets.violating-configurations-exhaust

snd	rcv	rslt	explanation
Subnet s_1	Subnet s_2	$s_1 = s_2$	Two hosts in a subnet can communicate if they are in the same subnet.
Subnet s_1	BorderRouter s_2	$s_1 = s_2$	A host in a subnet can communicate with its border router.
Subnet s_1	Unassigned	✓	No restriction for communication outside of the subnet.
BorderRouter s_1	Subnet s_2	✗	A router should not establish connections to any hosts. This is due to the design decision that hosts must not be (even indirectly) accessible from the outside. Considering the generic requirements for invariants, it would be possible to change the condition from ✗ to ($s_1 = s_2$), but this would violate this specific invariant's design decision.
BorderRouter s_1	BorderRouter s_2	✓	Border routers may communicate with each other.
BorderRouter s_1	Unassigned	✓	No restriction
Unassigned	Unassigned	✓	No restriction
Unassigned	_	✗	Must not set up connection to subnet or border routers

among themselves but cannot access hosts in a subnet. Even transitively, hosts of one subnet cannot access hosts of other subnets.

With this invariant template, it is impossible for some host in a subnet to be somehow accessible by the outside world. However, it is often desirable that a host is accessible, but only for connection which are routed over a border router. For this, we have extended the template[13] with two additional host attributes:

- A BorderRouter′ which can access members of its own subnet.
- An InboundRouter which is accessible by anyone, regardless of the subnet, but which cannot access hosts of a subnet.

With this setting, a $\bot$ host can set up connections to an InboundRouter, which can set up connections to a subnet's BorderRouter′, which can set up connections to a host in this subnet.

However, configuring this invariant template may get very complex. In addition, since this template may make assertions about a global subnet structure, it may no longer encode exactly one security requirement but may collect several. This may break modularity. Therefore, we decided to develop a simpler template which only considers one subnet. This will be formalized with the next invariant template. Statements about several subnets can be expressed with several instances of the template.

[13]SINVAR_Subnets2.thy

6.14 SubnetsInGW

This invariant template encodes the following security goal for one subnet or group of hosts: Members of a subnet may freely communicate with each other but are not accessible from the outside world. It may be possible from the outside world to connect to a Member over an $\mathsf{InboundGateway}$. The template considers only one single subnet. The host attributes are $\Psi = \{\mathsf{Member},\ \mathsf{InboundGateway},\ \mathsf{Unassigned}\}$ where $\bot = \mathsf{Unassigned}$.

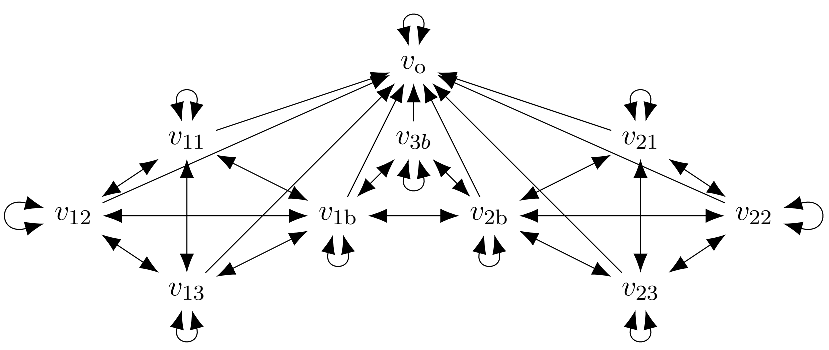
With the following table, the invariant template is formalized as follows.

snd	rcv	rslt	explanation
Member	_	✓	Accesses to members are restricted, but not the other way round.
InboundGateway	_	✓	No restrictions, may even connect to *Member*s.
Unassigned	Unassigned	✓	No restrictions.
Unassigned	InboundGateway	✓	This is the only way to indirectly access *Member*s.
Unassigned	Member	✗	Direct access is prohibited.

$$\mathsf{m}\ (V,\, E)\ P \quad \equiv \quad \forall (s,\, r) \in E.\ \ \mathsf{table}\ (P\ s)\ (P\ r)$$

With this invariant, a screened subnet architecture (sometimes called DMZ or demilitarized zone architecture) can be built. A Member is in the protected internal network and not accessible by external hosts. The servers in the DMZ are labeled $\mathsf{InboundGateway}$. With this setting, a relaxed screened subnet architecture is encoded because the servers in the DMZ can access the internal machines. This is sometimes desirable. To build an actual DMZ architecture where the servers in the DMZ must not access the internal hosts, the DMZ servers may simply be set to $\bot$.

Example. We emulate the example of Section 6.13 (Subnets) with the help of this invariant. Let $V = \{v_{11}, v_{12}, v_{13}, v_{1b}, v_{21}, v_{22}, v_{23}, v_{2b}, v_{3b}, v_{\mathrm{o}}\}$. Referring to the previous example, we will call the nodes $v_{*\mathrm{b}}$ border routers. We split the security goals into five configured security invariants. First, let the hosts v_{11}, v_{12}, v_{13} be Members and let $v_{1\mathrm{b}}$ be the $\mathsf{InboundGateway}$. Likewise, we create a second configured security invariant for the hosts v_{21}, v_{22}, v_{23}, and $v_{2\mathrm{b}}$. These two invariants would allow the border routers to be globally accessible. To restrict accesses for each border router, we attach an ACL (using the Communication Partners template) to each router. We create one instance for each router. The ACLs specify that the border router is only accessible by the other routers and the members of its subnet. The maximum policy for this setting can be visualized as follows.



It can be seen that the resulting policy is almost equal to the one of Section 6.13, with the only difference that border routers are now allowed to access their subnet members.

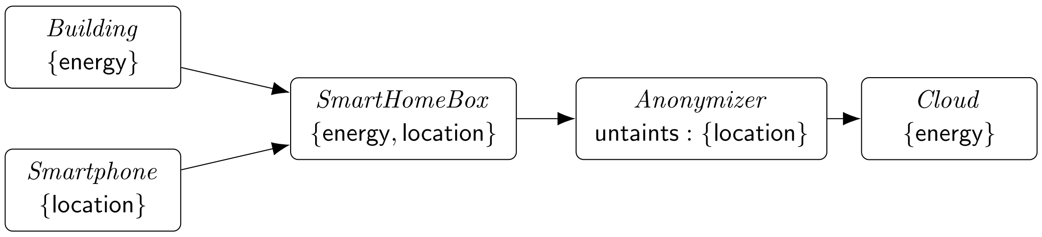
6.15 Simple Tainting

While previous invariants primarily focused on security in terms of access control and information flow, we will now present two invariants which focus on privacy. We will start with motivating background information and related work. Afterwards, we present our invariant formalization. Naturally, these invariants are information flow security strategies. Afterwards, we analyze the invariant and show that static taint analysis is as expressive as the Bell-LaPadula model.

Background Recently, dynamic taint analysis [101] has been used successfully in the Android world to enhance user privacy [102, 103, 104]. Based on those ideas, the two invariants presented in this section and the following section formalize static taint analysis. We demonstrate that coarse-grained taint analysis is also applicable to the analysis and auditing of distributed architectures, can be done completely static (preventing runtime failures), while providing strong formal guarantees.

We base our understanding of privacy on the operationalization performed by Pfitzmann and Rost [105] and further elaborated on by Bock and Rost [106]. Their proposal has been adapted by the European Union Agency for Network and Information Security (ENISA) [107] and by the German Standardized Data Protection Model [108], showing wide acceptance of their approach. In summary, this set of related work bases the understanding of privacy upon the data protection goals of *unlinkability*, *transparency*, *intervenability*, and *data minimization*. A detailed discussion of these aspects can be found in the extended version of our paper [109]. Our model of static taint analysis was designed to make these aspects of privacy more tangible. In brief, we address unlinkability by making it possible to see whether two taint labels are ever assigned to the same entity. In addition, the taint labels of an entity are a measure to promote data minimization. Transparency and intervenability are user-facing protection goals and our model provides a first step towards this by making the relevant information explicit.

Example. We introduce the concepts of taint analysis by a simple, fictional example: A house, equipped with a smart meter to measure its energy consumption. The owner also provides location information via her smartphone to allow the system to turn off the lights when she leaves the home. Once every month, the aggregated energy consumption is sent over the Internet to the energy provider for billing.



We are interested in the privacy implications of this setup and perform a taint tracking analysis. The system architecture is visualized in the above figure. The *Building* produces information about its energy consumption. Therefore, we label the *Building* as taint source and assign it the energy label. Likewise, the *Smartphone* tracks the location of its owner. Both data is sent to the *SmartHomeBox*. Since the *SmartHomeBox* aggregates all data, it is assigned the set $\{\mathsf{energy}, \mathsf{location}\}$ of taint labels. The user wants to transmit only the energy information, not her location to the energy provider's *Cloud*. Therefore, the *Anonymizer* filters the information and removes all location information. We call this process *untainting*. Let '$\cdot \setminus \cdot$' denote the minus operation on sets. With the *Anonymizer* operating correctly, since $\{\mathsf{energy}, \mathsf{location}\} \setminus \{\mathsf{location}\} = \{\mathsf{energy}\}$, only energy-related information ends up in the energy provider's *Cloud*.

We now formalize a simplified version of the security invariant template, based on the above ideas. For clarity, it does not provide the untainting feature. This feature will be added explicitly in Section 6.16.

The host attributes are sets of taint labels. For simplicity, we model taint labels as strings: $\Psi = \textit{string set}$. For example, $P\ \textit{SmartHomeBox} = \{\mathsf{energy}, \mathsf{location}\}$. By default, an entity does not have any taint label, i.e., $\bot = \emptyset$.

Intuitively, information flow security according to the taint model can be understood as follows. Information leaving a node v is tainted with v's taint labels, hence every receiver r must have the respective taint labels to receive the information. In other words, for every node v in the policy, all nodes r which are reachable from v must have at least v's taint labels. Representing reachability by the transitive closure (i.e., succ-tran), the invariant can be formalized as follows:

$$\mathsf{m}\ (V,\ E)\ P \quad \equiv \quad \forall v \in V.\ \forall r \in \mathsf{succ\text{-}tran}\ (V,\ E)\ v.\ \ P\ v \subseteq P\ r$$

Analysis: Tainting vs. Bell-LaPadula Model The Bell-LaPadula model is the traditional, de-facto standard model for label-based information flow security. The question arises whether we can justify our taint model using BLP. We will compare our model to the simple Bell-LaPadula model (Section 6.2).

We need to give names to the invariant templates which have always been called m. We call the tainting invariant template tainting and we call the Bell-LaPadula template blp.

Inspired by BLP, we show an alternative definition for our tainting invariant:

Lemma 3 (Localized Definition of Tainting)**.**

$$\mathsf{tainting}\ (V, E)\ t = \forall (v_1, v_2) \in E.\ \ t\ v_1 \subseteq t\ v_2$$

Lemma 3 also provides[14] a computational efficient formula, which only iterates over all edges and never needs to compute a transitive closure.

We now show[15] that one $\mathsf{tainting}$ invariant is equal to BLP invariants for every taint label. We define a function $\mathsf{project}\ a\ Ts$, which translates a set of taint labels Ts to a security level depending on whether a is in the set of taint labels. Formally, $\mathsf{project}\ a\ Ts \equiv \textbf{if}\ a \in Ts\ \textbf{then}\ \mathsf{confidential}\ \textbf{else}\ \mathsf{unclassified}$. Using function composition, the term $\mathsf{project}\ a\ \circ\ P$ is a function which first looks up the taint labels of a node and projects them afterwards.

Theorem 5 (Tainting and Bell-LaPadula Equivalence)**.**

$$\mathsf{tainting}\ G\ P \longleftrightarrow \forall a.\ \mathsf{blp}\ G\ (\mathsf{project}\ a\ \circ\ P)$$

In the context of privacy, the '→'-direction of our theorem shows that one $\mathsf{tainting}$ invariant guarantees individual privacy according to Bell-LaPadula for each taint label. This implies that every user of a system can obtain her personal privacy guarantees. This is one step towards transparency and intervenability.

The '←'-direction shows that $\mathsf{tainting}$ is as expressive as the simple Bell-LaPadula model. This justifies the theoretic foundations w.r.t. the well-studied BLP model. These findings are in line with Denning's lattice interpretation [82]; however, to the best of our knowledge, we are the first to discover and formally prove this connection in the presented context.

The theorem can be generalized[16] for arbitrary (but finite) sets of taint labels A. The $\mathsf{project}$ function then maps to a numeric value of a security level by taking the cardinality of the intersection of A with Ts. For example, if we want to project $\{\mathsf{location}, \mathsf{temp}\}$, then $\{\mathsf{name}\}$ is $\mathsf{unclassified}$, $\{\mathsf{name}, \mathsf{location}, \mathsf{zodiac}\}$ is $\mathsf{confidential}$, and $\{\mathsf{name}, \mathsf{location}, \mathsf{zodiac}, \mathsf{temp}\}$ is secret.

6.16 Tainting

We continue our efforts of the previous section to formalize an invariant template for privacy based on taint analysis.

Real-world application requires the need to untaint information, for example, when data is encrypted or properly anonymized. This was also demonstrated in the example of the previous section.

We extend the host attributes to carry both taint labels and untainting information. The taint labels now consist of two components: the labels a node taints and the labels it untaints: $\Psi = \textit{string set}\ \times\ \textit{string set}$. We define the helper accessor which allows us to conveniently access the individual taint label sets. Let taints return the first element of the tuple and let $\mathsf{untaints}$ return the second element of the tuple.[17]

We extend the simple $\mathsf{tainting}$ invariant to support untainting:

$$\mathsf{m}\ (V, E)\ P\ \equiv\ \forall (v_1, v_2) \in E.\ \ \mathsf{taints}\ (P\ v_1) \setminus \mathsf{untaints}\ (P\ v_1) \subseteq \mathsf{taints}\ (P\ v_2)$$

[14]SINVAR-Tainting.sinvar-preferred-def

[15]tainting-iff-blp

[16]tainting-iff-blp-extended

[17]Actually, in our implementation `SINVAR-TaintingTrusted`, we create a new type `taints-raw` to avoid accidental type errors and confusion of taint labels with other tuples.

To abbreviate a node's labels, we will write X—Y, where X corresponds to the taints and Y corresponds to the untaints. In the example of the previous section, we have $P\ Anonymizer = \{\mathsf{energy}\}$—$\{\mathsf{location}\}$.

We impose the type constraint that $Y \subseteq X$, i.e., untaints $Ts \subseteq$ taints Ts.[18] We implemented the datatype such that X—Y is extended to $X \cup Y$—Y. Regarding previous section's example, this merely appears to be a very convenient abbreviation. In particular, *P Anonymizer* now corresponds to $\{\mathsf{energy}, \mathsf{location}\}$—$\{\mathsf{location}\}$, for which the improved tainting invariant holds and which also corresponds to our intuitive understanding of untainting. However, this is a fundamental requirement for the overall soundness of the invariant. Without the constraint, there can be dead untaints, i.e., untaints which can never have any effect. This would violate the uniqueness property required by the secure default parameters and can cause further problems in pathological corner cases. Yet, with this type constraint, all insights obtained for the simple model now follow analogously for this model.

Analysis: Tainting vs. Bell-LaPadula Model We now compare our improved model to the Bell-LaPadula model with trust (Section 6.3).

We give names to the invariant templates which have always been called m. We call the tainting invariant template tainting′ and we will call the Bell-LaPadula template blp′.

In the context of Bell-LaPadula, a trusted entity is allowed to declassify information, i.e., receive information of any security level and redistribute with its own level (which may be lower than the level of the received information). This concept is comparable to untainting.

Recall that in Bell-LaPadula, trust extracts the trusted flag from an entity's attributes and level extracts the security level.

Our insights about the equality follow analogously to the simple tainting invariant.[19] Let project a (X—Y) be a function which translates taints (X) and untaints (Y) labels to security levels and trust of the Bell-LaPadula model. The function translates the security level as follows: **if** $a \in (X \setminus Y)$ **then** confidential **else** unclassified. It translates the trust flag as follows: $a \in Y$.

Theorem 6 (Tainting and Bell-LaPadula Equivalence)**.**

$$\mathsf{tainting}'\ G\ t \longleftrightarrow \forall a.\ \mathsf{blp}'\ G\ (\mathsf{project}\ a \circ t)$$

Similarly to the version without trust, the theorem can be generalized for arbitrary (but finite) sets of taint labels.[20]

6.17 System Boundaries

Our formalism provides a number of useful analyses. For example, given a security invariant specification, it allows to compute all permitted flows which is invaluable for validating a given specification. However, our formalism might lack knowledge about architectural constraints which leads to the computation of an unrealistic amount of flows. For example,

[18] taints-wellformedness
[19] tainting-iff-blp-trusted
[20] tainting-iff-blp-trusted-extended

several logical entities may correspond to individual programs which are running on the same machine. Some programs may only communicate via IPC and are not externally reachable. We want to provide this knowledge to our formalism.

Therefore, we model systems with clearly defined boundaries. We define *internal* components as nodes which are only accessible from inside the system. We define *passive system boundaries* to be boundaries which only accept incoming connections. Analogously, *active system boundaries* are boundaries which only establish outgoing connections. A *boundary* may be both.

A security invariant in our formalism must either be an access control strategy or information flow security strategy. An access control invariant restricts accesses *from* the outside, an information flow invariant restricts leakage *to* the outside. However, internal components of a system require both: they should neither be accessible from components outside of the system nor leak data to outside components. We overcame this limitation of our framework by constructing a model for system boundaries which is internally translated to two invariants: an access control invariant (SubnetsInGW, Section 6.14) and an information flow invariant (Bell-LaPadula, Section 6.3). We have integrated the concept of system boundaries into our formalization and proven that the two configured invariants which are generated by a specification of a system boundary yield exactly the desired behavior.

Chapter 7

Evaluation & Case Studies

Abstract We present a tool which implements the presented theory and evaluate our theory and tool in the case study of a cabin data network and at the example of an imaginary factory network.

7.1 Stand-Alone Tool: *topoS*

Version 1: Scala Implementation Typically, a network or policy administrator does not want to depend on a theorem prover and does not want to learn the formal language of a theorem prover. Therefore, we build a prototypical stand-alone tool to demonstrate applicability of our theory. Our tool is a stand-alone Java `jar` file and only requires a JVM to run; no external dependencies exist. The stand-alone tool also serves to emphasize the following:

- Our tool does not depend on Isabelle/HOL being installed on a user's system.
- No automated provers are run in the background. Hence, our algorithms do not rely on unpredictable heuristics to solve proof obligations, which is known to fail (timeout) at runtime for certain problems; our algorithms directly solve the problems.
- A typical user does not need to prove anything to use our tool.
- Our tool runs on common, off-the-shelf hardware, without special requirements or dependencies.

We call our prototypical tool *topoS*. The Scala implementation supports most of the features presented in the previous chapters. Its core reasoning logic consists of code generated by Isabelle/HOL. This guarantees the correctness of all results computed by *topoS*'s core [110, 111]. Of course, only the core is generated by Isabelle; errors in the user interface cannot be prevented by this approach. In addition, the code generation only guarantees partial correctness. This means, termination is not guaranteed; however, if there is a result, it is correct. Our empirical evaluation shows that the code always terminates for reasonably-sized input.

Version 2: Isabelle/ML Implementation However, we realized that the tool's Scala code was diverging from the theory files over time.

The Scala tool was based on code originally written for the author's master's thesis [53]. For the preliminary prototype of the master's thesis, about 5000 lines of theory were used

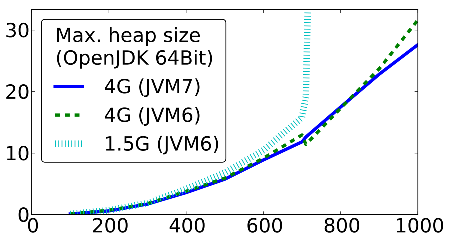


Figure 7.1: Runtime of the policy construction algorithm for 100 Φ-structured invariants on an i7-2620M CPU (2.70GHz), Java Virtual Machine. X-axis: $|V|$, Y-axis: runtime in minutes.

to generate 2000 lines of Scala code. In total, the Scala tool consisted of about 6500 lines of code. Consequently, there was an amount of more than three times unverified code involved than there was verified code. During the Ph.D. thesis, the theory (and consequently the generated code) and the manually-written code of the Scala tool grew. However, the ratio of about more than three times unverified code vs. verified code remained.

For this Ph.D. thesis, the theory has been completely reworked and the Scala tool diverged more and more. We decided that the tool and the theories must be better kept in sync. Therefore, we decided to re-implement the tool completely in Isabelle. This has the nice advantage that the thousand lines of unverified Scala code could be replaced by only few hundred lines of ML [112] code. Some lines of unverified code will always remain for the visualization. For our Isabelle re-implementation, they are narrowed down to the visualization with graphviz [113]. This choice allowed to vastly improve the ratio between verified and unverified code. In total, after refactoring and a large cleanup, there are approximately 12000 lines of theory (excluding examples) and only about 400 lines of unverified ML code (only used for the visualization). This means the ratio of unverified vs. verified code has improved from approximately 300% to approximately 3%. An additional advantage to implement our tool within Isabelle is that it is automatically maintained in the AFP [38]. Furthermore, the Isabelle/ML implementation is feature complete.

The Isabelle-based version of *topoS* can be used similarly to the Scala-based version. The main difference is that a user must start Isabelle. The commands which could be invoked from the Scala tool can be invoked within Isabelle. Otherwise, the same properties as for the Scala-based tool are provided. In particular, a user does not need to prove anything at runtime. For a user without a formal background, Isabelle can be used as an interactive text editor (not as a theorem prover).

The results presented in this Ph.D. thesis are based on the Isabelle version of *topoS*. For the original paper where the case study of Section 7.2 appeared first, the case study has been conducted with the Scala-based version of *topoS*. For this Ph.D. thesis, we have re-checked the case study with the Isabelle version of *topoS*.

7.1.1 Computational Complexity

topoS performs linear in the number of security invariants and quadratic in the number of hosts for Φ-structured invariants. For scenarios with less than 100 hosts, it responds interactively in less than 10 seconds. With the grouping approach presented in [114], we argue that often, setups with more than 100 nodes are unlikely as large groups of identical hosts can be pooled into a small representative set, e.g., instead of modeling 1000 identical workstation PCs, only 2 representatives of this large set are necessary for reasoning. For the presented case study (Section 7.2), all results are immediately available. Therefore, we provide an interactive convenient tool for designing and conceptualizing a network.

A benchmark of the automated policy construction, the most expensive algorithm, is presented in Figure 7.1. The benchmark is based on the Scala-based version of *topoS* to exclude overhead induced by Isabelle and to control the JVM heap size. For $|V|$ hosts, $|V|^2/4$ flows were created. With reasonable memory consumption, policies with up to 250k flows can be processed in less than half an hour.

topoS contains a lot of machine-generated code that is not optimized for performance but correctness. However, the overall theoretical and practical performance is sufficient for real-world usage. During our work with Airbus Group, we never encountered any performance issues.

7.2 Case Study: A Cabin Data Network

In this section, we present a slightly more complex scenario: a policy for a cabin data network for the general civil aviation. This example was chosen as security is very important in this domain and it provides a challenging interaction of different security invariants. It is a small imaginary toy example, developed in collaboration with Airbus Group. To make it self-contained and accessible to readers without aeronautical background knowledge, it does not obey aeronautical standards (such as ARINC specification 664P5 [115]). However, the scenario is plausible, i.e., a real-world scenario may be similar. During our research, we also evaluated real-world scenarios in this domain. With this experience, we try to present a small, simplified, self-contained, plausible toy scenario that, however, preserves many real-world snares.

The network consists of the following hosts.

CC The Cabin Core Server, a server that controls essential aircraft features, such as air conditioning and the wireless and wired telecommunication of the crew.

C1, *C2* Two mobile devices for the crew to help them organize, e.g., communicate, make announcements.

Wifi A wifi hotspot that allows passengers to access the Internet with their own devices. Explicitly listed as it might also be responsible for billing passenger's Internet access.

IFEsrv The In-Flight Entertainment server with movies, Internet access, etc. Master of the IFE displays.

IFE1, *IFE2* Two In-Flight Entertainment displays, mounted at the back of passenger seats. They provide movies and Internet access. Thin clients, everything is streamed from the IFE server.

P1, *P2* Two passenger-owned devices, e.g., laptops, smartphones.

SAT A satellite uplink to the Internet.

The following three security invariants are specified.

Security Invariant 1, Domain Hierarchy. Four different security domains exist in the aircraft, cf. Figure 7.2. They separate the **crew** domain, the **entertain**ment domain, the passenger-owned devices (**POD**) domain and the Internet (**INET**) domain. All devices belong to a domain.

The following devices are in the **entertain** domain: *IFEsrv*, *IFE1*, *IFE1*.

The Cabin Core Server, which is located in the **crew** domain, may also send to the entertain domain. Hence, it is trusted. Possible use cases include: Stewards coordinate food distribution or an announcement from the crew is send to the In-Flight Entertainment system (via *CC*) and distributed there to the IFE displays.

The *Wifi* is located in the **POD** domain to be reachable by PODs. It is trusted to send to the entertain domain. Possible use cases include: A passenger subscribes a film from the IFE server to her notebook or establishes connections to the Internet.

In the **INET** domain, the *SAT* is isolated to prevent accesses from the Internet into the aircraft.

$$
\begin{array}{lll}
\mathit{CC} & \mapsto & (\mathsf{level}:\ \mathit{crew.aircraft},\ \ \mathsf{trust}:\ 1)\\
\mathit{C1} & \mapsto & (\mathsf{level}:\ \mathit{crew.aircraft},\ \ \mathsf{trust}:\ 0)\\
\mathit{C2} & \mapsto & (\mathsf{level}:\ \mathit{crew.aircraft},\ \ \mathsf{trust}:\ 0)\\
\mathit{IFEsrv} & \mapsto & (\mathsf{level}:\ \mathit{entertain.aircraft},\ \ \mathsf{trust}:\ 0)\\
\mathit{IFE1} & \mapsto & (\mathsf{level}:\ \mathit{entertain.aircraft},\ \ \mathsf{trust}:\ 0)\\
\mathit{IFE2} & \mapsto & (\mathsf{level}:\ \mathit{entertain.aircraft},\ \ \mathsf{trust}:\ 0)\\
\mathit{SAT} & \mapsto & (\mathsf{level}:\ \mathit{INET.entertain.aircraft},\ \ \mathsf{trust}:\ 0)\\
\mathit{Wifi} & \mapsto & (\mathsf{level}:\ \mathit{POD.entertain.aircraft},\ \ \mathsf{trust}:\ 1)\\
\mathit{P1} & \mapsto & (\mathsf{level}:\ \mathit{POD.entertain.aircraft},\ \ \mathsf{trust}:\ 0)\\
\mathit{P2} & \mapsto & (\mathsf{level}:\ \mathit{POD.entertain.aircraft},\ \ \mathsf{trust}:\ 0)
\end{array}
$$

Security Invariant 2, Policy Enfrocement Point. The IFE displays are thin clients and strictly bound to their server. Peer to peer communication is prohibited. The Policy Enforcement Point template directly provides the respective access control restrictions.

$$
\begin{array}{lll}
\mathit{IFEsrv} & \mapsto & \mathsf{PolEnforcePointIN}\\
\mathit{IFE1} & \mapsto & \mathsf{DomainMember}\\
\mathit{IFE2} & \mapsto & \mathsf{DomainMember}
\end{array}
$$

Security Invariant 3, Bell-LaPadula with Trust. Requirement 1 and 2 encode access control restrictions. Invariant 3 defines information flow restrictions by labeling confidential information sources according to the Bell-LaPadula model with trust. To protect the passenger's privacy when using the IFE displays, it is undesirable that the IFE displays communicate with anyone, except for the *IFEsrv*. Therefore, the IFE displays are marked as confidential. Note that requirement 2 dictates what can access the IFE displays and this requirement dictates what the IFE displays can send out. The *IFEsrv* is considered a central trusted device. To enable passengers to surf the Internet on the IFE displays by forwarding the packets to the Internet or forward announcements from the crew, it must be allowed to declassify any information to the default (i.e., unclassified) security level. Finally, the crew communication is considered more critical than the convenience features, therefore, *CC*, *C1*, and *C2* are considered secret. As the *IFEsrv* is trusted, it can receive and forward announcements from the crew.

CC	$\mapsto$	(level : secret, trust : False)
C1	$\mapsto$	(level : secret, trust : False)
C2	$\mapsto$	(level : secret trust : False)
IFE1	$\mapsto$	(level : confidential, trust : False)
IFE2	$\mapsto$	(level : confidential, trust : False)
IFEsrv	$\mapsto$	(level : unclassified, trust : True)

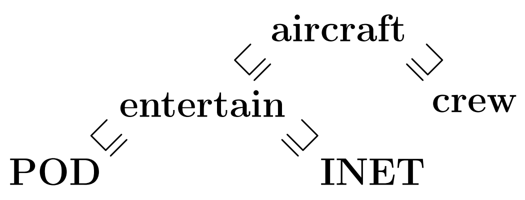


Figure 7.2: Security domains of the cabin data network.

This case study illustrates that this complex scenario can be divided into three security invariants that can be represented with the help of the previously presented templates. It also reveals that very few host attributes must be manually specified; the automatically added secure default attributes complete the configuration.

The policy is illustrated in Figure 7.3. The different domains are illustrated and trusted devices according to the Domain Hierarchy are marked with an exclamation mark. The *declassify* host attribute belongs to the Bell-LaPadula invariant and corresponds to a trusted host with the unclassified security level.

Verifying the Policy Our tool verified that all security invariants are fulfilled.

Analyzing the Invariant Specification Focusing in *topoS*'s analysis capabilities, the following results were obtained. With the automated policy construction algorithm, an alternative policy was calculated. In Figure 7.3, the solid edges combined with the dashed edges[1] correspond to the uniquely defined policy with the maximum number of allowed

[1]combined with all reflexive edges, i.e., in-host communication

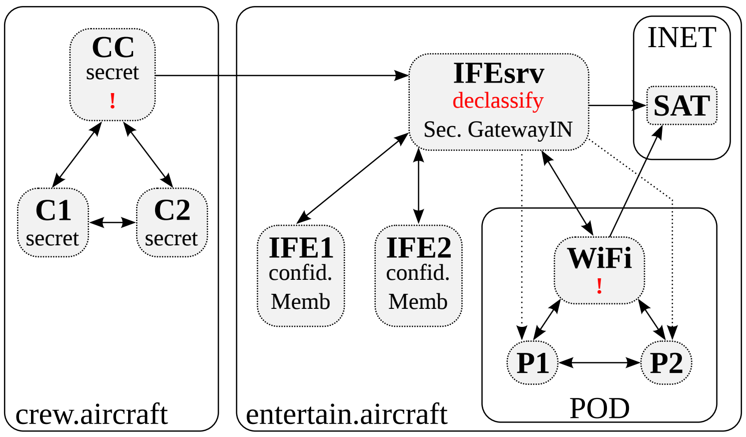


Figure 7.3: Cabin network policy and hosts' attributes.

Experience	Participants	Complexity	Valid	Violations	Missing	Errors
Expert	5	medium but tricky	16.0/15.8/1.7	1.0/3.2/4.4	5.0/ 5.6/2.1	9.0/ 8.8/3.8
Intermediate	5	medium	14.0/14.0/1.4	1.0/1.6/1.9	7.0/ 7.4/1.0	8.0/ 9.0/2.6
Novice	5	medium	12.0/10.6/5.7	4.0/6.6/4.9	11.0/11.2/6.2	17.0/17.8/9.1
Total	15	medium but tricky	15.0/13.5/4.1	2.0/3.8/4.5	7.0/ 8.1/4.5	9.0/11.9/7.2
Legend: median/arithmetic mean/std deviation						

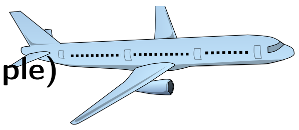
Table 7.1: Results of user feedback session.

flows. The solid lines were given by the policy, the dashed lines were calculated from the invariants. These 'diffs' are computed and visualized automatically by *topoS*. They provide the end user with helpful feedback regarding '*what do my invariants require?*' vs. '*what does my policy specify?*'. This results in a feedback loop we used extensively during our research to refine the policy and the invariants. It provides a 'feeling' for the invariants.

In this case study, two insights were obtained from this analysis. First, according to the security invariants, the IFE server could possibly connect to the passenger-owned devices directly. Presenting the question to an engineer, this would raise the question about whether this is actually acceptable. If this were not acceptable, it would indicate a bug in the specified security invariants. In this scenario however, it is acceptable and impossible by hardware constraints anyway. The second insight is that – disregarding the two flows from the IFE server to the passenger-owned devices – the policy only inferred from the invariants and the policy designed by hand coincide exactly. This is a strong indicator that the specified invariants 'mean' the right thing.

7.2.1 End-User Feedback Session

To estimate the scenario's complexity, we asked some network professionals to design the scenario's policy. In total, 15 volunteered to participate in our study. No private or behavioral data was collected during this short study. The scenario description of Figure 7.4 was handed out to the participants. It was emphasized that no security requirement must be violated, but the participants should try to put the maximum number of flows in the network to fulfill as much as possible of the use cases. Therefore, the task was to maximize the allowed flows without violating any security invariant. This is a purely technical task. Afterwards, we asked the participants to rate the complexity of this exercise. The questions and answers



Imaginary Aircraft Cabin Data Network (Toy Example)

Devices in our Aircraft

CC The Cabin Core Server, a server that controls essential aircraft features, such as air conditioning and the wireless and wired telecommunication of the crew.

C1, C2 Two mobile devices for the crew to help them identifying passenger calls or make announcements.

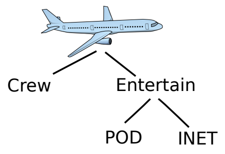
IFEsrv The In-Flight Entertainment server with movies, etc.

IFE1, IFE2 Two In-Flight entertainment displays, mounted at the back of passenger seats. Movies and internet access. Slim devices, everything is streamed from the IFE server.

Wifi A wifi hotspot that allows passengers to access the Internet with their own devices.

Sat A satellite uplink to the Internet.

P1, P2 Two passenger owned devices, e.g. laptops, smartphones.



Security Requirements

Requirement One

In our aircraft, we have 4 different security domains. Higher domains can send to all lower domains, lower domains must not send to higher domains. Different domains on the same level must not send to each other (they are separate).

- **Crew Domain**
  A separate domain, very high security level. The mobile crew devices are in this domain. The cabin core server is also in there; however it is a special trusted device that may send to other domains (domain-spanning).
  Use cases: Stewards coordinate food distribution; Announcement from the crew is send to In-Flight Entertainment system.
- **Entertain Domain**
  A separate domain, same level as Crew Domain. The In-Flight Entertainment displays and the IFE server are in this domain.
  The Entertain Domain has several sub-domains of lower security levels:
  - **POD Domain**
    All passenger owned devices are in this domain. In addition, the wifi access is in this domain. It has (limited) trust, i.e. it is allowed to send into the Entertain Domain but not higher.
    Use case: Passenger subscribes a film from the IFE server to her notebook.
  - **INET Domain**
    The Satellite uplink is the only member of this domain.

Requirement Two

In our aircraft, we have some confidentiality requirements. The complete crew communication, including the cabin core server has the highest confidentiality level. This data must not leak to untrusted places. To protect the passenger's privacy when using the pre-installed devices, the IFE devices also have a confidentiality level, lower than the crew devices. The IFE server has a special role: It can declassify information (i.e. reveal to others).
Use Case: Announcement is send from a crew device and forwarded to the IFE displays via the IFE server.

Requirement Three

In our aircraft, the IFE displays are slim devices and strictly bound to their server.
Example: No peer to peer among the IFE displays; they are not directly reachable from 'the outside'.

Your Task: Design the network

Create a network topology. Put in as many flows as possible, do not violate the security requirements. Use a *directed* graph: Different meanings of 'directed' are allowed per edge. E.g., only send this direction, only establish connections (like a stateful packet filter), ...

Figure 7.4: Scenario and task description as handed out to the study participants.

Experience	utility tool	utility idea	acceptance idea	acceptance tool
Expert	3.0/3.4/0.5	3.0/3.2/0.7	100%	100%
Intermediate	3.0/3.2/0.4	4.0/4.0/0.0	100%	100%
Novice	4.0/3.2/1.2	4.0/3.2/1.2	80%	100%
Total	3.0/3.3/0.8	4.0/3.5/0.9	93%	100%

Utility measure: 0) counter-productive, 1) more counter-productive than helpful, 2) neutral, 3) helpful, 4) extremely helpful

Legend: median/arithmetic mean/std deviation

Table 7.2: User's thoughts about the prototypical tool.

were pre-formulated.

The results are illustrated in Table 7.1. It shows the perceived complexity of the task and the number of valid, violating, and missing flows the participants defined. We define the error count as the number of invalid plus the number of missing flows. Surprisingly, even expert network administrators made errors (both missing flows and security violations) when designing the policy.

Afterwards, we presented our prototypical Scala tool to the participants and asked the participants about their thoughts about our tool. The results are summarized in Table 7.2. Note that this part of the user feedback session is neither a controlled experiment nor a scientific study: the conditions were not randomized, it suffers from demand bias, and there was no control group. The main evaluation of this work are the formal correctness proofs. Our only goal of this part of the user feedback session was to collect a rough feedback and some user's first thoughts.

The overall feedback was that our tool is downright helpful (3.0/3.3/0.8). We also introduced the idea behind our tool and admitted that the user interface of our prototype can be vastly improved. We asked the participants to judge the idea behind our tool. An experienced participant raised concern that special training for novice administrators is necessary. However, the overall judgment about the idea was very positive and it was considered remarkably helpful (4.0/3.5/0.9). In addition, 93% of the participants consider that our idea might help to manage large networks over a long period with many responsible persons. The positive feedback and recurring question we received during the user field study about where, when, and how expensive to obtain our tool was very motivating. Our tool's graphical feedback was also much appreciated. Finally, 100% of the participants would want to use our tool[2] for similar tasks.

A detailed scenario description, the host attribute mappings, and raw data are available [116].

Related Study Johnson et al. [117] propose to split the process of policy authoring into three separate user roles. Though their policy framework and language differs from ours, their concepts can be abstracted to our system. They define *policy element authors*, who have domain knowledge and define the necessary elements a policy can use. For our scenario, definitions such as the cabin core server, the in-flight entertainment system, etc. would be defined by this user role. A *template author* is an experienced user who defines templates.

[2]or a competing product, we asked to assume that an intuitive user interface is available

For our scenario, the policy enforcement point invariant template and other templates would be defined by this user role. Finally, *policy authors* instantiate the templates to create the actual policies. In our scenario, this user role would correspond to the participants of our end-user feedback session.

In a user study with 20 experienced participants, Johnson et al. evaluate how well users can abstract over concrete policies by developing templates. Similar to our user feedback session, no control group was involved and demand bias was probably introduced. Nevertheless, the study shows that most users can successfully create templates. This hints that also our approach of splitting policies into generic templates and template instantiation may contribute to user-friendliness. The feature most used by the study participants was a policy preview feature. These results are in line with our user feedback session where the participants valued the automated policy construction in combination with the graph visualization. It shows that users in general value feedback about the meaning of a policy statement they have written.

In the study by Johnson et al. almost half of the participants were concerned that templates may permit a policy which is too permissive. We believe that the principles our system is built upon prevent this issue: First, we built our security invariant templates with the monotonicity principle of "prohibiting more is more or equally secure" (Definition 4). In addition, the composability of several invariants provides the same guarantee (Section 5.3.6). Second, the secure default parameter provides the incentive that adding more information makes the secure default parameter more secure (Definition 7).

7.3 Example: Imaginary Factory Network

In this section, we give an example of an imaginary factory network. The example was chosen to show the interplay of several security invariants and to demonstrate their configuration effort. The specified security invariants deliberately include some minor specification problems. These problems will be used to demonstrate the inner workings of the algorithms and to visualize why some computed results will deviate from the expected results. At this point, we also try to outline the big picture of this thesis by including some results of the following chapters in this example.

7.3.1 Scenario Description

The described scenario is an imaginary factory network. It consists of sensors and actuators in a cyber-physical system. The on-site production units of the factory are completely automated and there are no humans in the production area. Sensors are monitoring the building. The production units are two robots which manufacture the actual goods. The robots are controlled by two control systems.

The network consists of the following hosts which are responsible for monitoring the building.

Statistics A server which collects, processes, and stores all data from the sensors.

SensorSink A device which receives and collects data from the *PresenceSensor*, *Webcam*, *TempSensor*, and *FireSensor*. It sends the data to the *Statistics* server.

PresenceSensor A sensor which detects whether a human is in the building.

Webcam A camera which monitors the building indoors.

TempSensor A sensor which measures the temperature in the building.

FireSensor A sensor which detects fire and smoke.

The following hosts are responsible for the production line.

MissionControl1 An automation device which drives and controls the robots.

MissionControl2 An automation device which drives and controls the robots. It contains the logic for a secret production step, carried out only by *Robot2*.

Watchdog Regularly checks the health and technical readings of the robots.

Robot1 Production robot unit 1.

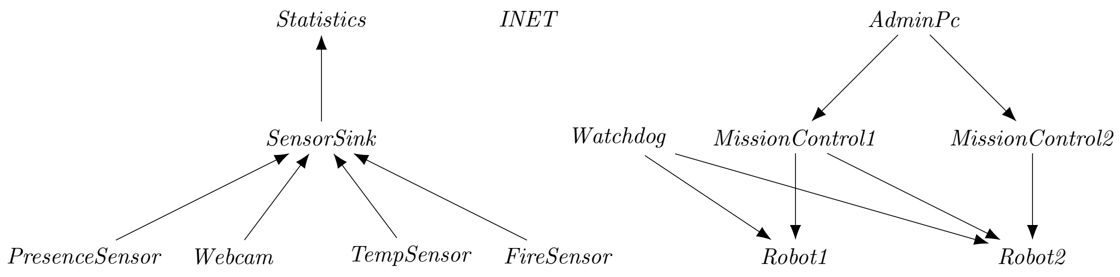
Robot2 Production robot unit 2. Performs a secret production step.

AdminPc A human administrator can log into this machine to supervise or troubleshoot the production.

We model one additional special host.

INET A symbolic host which represents all hosts which are not part of this network.

The security policy is visualized below.



The idea behind the policy is the following. The sensors on the left can all send their readings in a unidirectional fashion to the sensor sink, which forwards the data to the statistics server. In the production line, on the right, all devices will set up stateful connections. This means, once a connection is established, packet exchange can be bidirectional. This makes sure that the watchdog will receive the health information from the robots, the mission control machines will receive the current state of the robots, and the administrator can actually log into the mission control machines. The policy should only specify who is allowed to set up the connections. We will elaborate on the stateful implementation in Section 7.3.5.

7.3.2 Specification of Security Invariants

Several security invariants are specified.

Security Invariant 1, BLP Basic. The sensors in the building may record any employee. Due to privacy requirements, the sensor readings, processing, and storage of the data are treated with a high security level. The presence sensor does not allow do identify an individual employee, hence produces less critical data, hence has a lower level.

Statistics	$\mapsto$	3
SensorSink	$\mapsto$	3
PresenceSensor	$\mapsto$	2
Webcam	$\mapsto$	3

Security Invariant 2, BLP Basic. The production process is a corporate trade secret. The mission control devices have the trade secrets in their program. The important and secret step is done by *MissionControl2*.

MissionControl1	$\mapsto$	1
MissionControl2	$\mapsto$	2
Robot1	$\mapsto$	1
Robot2	$\mapsto$	2

Note that Invariant 1 and Invariant 2 are two distinct specifications. They specify individual security goals independent of each other. For example, in Invariant 1, *MissionControl2* has the default security level $\bot = 0$ and in Invariant 2, *PresenceSensor* has security level $\bot$. Consequently, both cannot interact.

Security Invariant 3, BLP Trusted. Monitoring the building while also ensuring privacy of the employees is an important goal for the company. While the presence sensor only collects the single-bit information whether a human is present, the webcam allows identifying individual employees. The data collected by the presence sensor is classified as secret while the data produced by the webcam is top secret. The sensor sink only has the *secret* security level, hence it is not allowed to process the data generated by the webcam. However, the sensor sink aggregates all data and only distributes a statistical average which does not allow identifying individual employees. It does not store the data over long periods. Therefore, it is marked as trusted and may thus receive the webcam's data. The statistics server, which archives all the data, is considered top secret.

Statistics	$\mapsto$	(level : topsecret, trust : False)
SensorSink	$\mapsto$	(level : secret, trust : True)
PresenceSensor	$\mapsto$	(level : secret, trust : False)
Webcam	$\mapsto$	(level : topsecret, trust : False)

Security Invariant 4, Communication Partners. *Robot2* carries out a mission-critical production step. For its integrity, it must be made sure that *Robot2* only receives packets from *Robot1*, the two mission control devices and the watchdog.

$$\begin{array}{lcl} \mathit{Robot2} & \mapsto & \mathsf{Master}\ [\mathit{Robot1}, \mathit{MissionControl1}, \\ & & \qquad \mathit{MissionControl2}, \mathit{Watchdog}] \\ \mathit{MissionControl1} & \mapsto & \mathsf{Care} \\ \mathit{MissionControl2} & \mapsto & \mathsf{Care} \\ \mathit{Watchdog} & \mapsto & \mathsf{Care} \end{array}$$

Note that *Robot1* is in the access list of *Robot2*, but it does not have the Care attribute. This means, *Robot1* can never access *Robot2*. A tool could automatically detect such inconsistencies and emit a warning. However, a tool should only emit a warning—not an error—because this setting could be intentional and desirable.

In our factory, this setting is currently desirable: Three months ago, *Robot1* had an irreparable hardware error and needed to be removed from the production line. When removing *Robot1* physically, all its host attributes were also deleted. The access list of *Robot2* was not changed. It was planned that *Robot1* will be replaced and later will have the same access rights again. A few weeks later, a replacement for *Robot1* arrived. The replacement is also called *Robot1*. The new robot arrived neither configured nor tested for the production. After carefully testing *Robot1*, *Robot1* has been given back the host attributes for the other security invariants. Despite the ACL entry of *Robot2*, when *Robot1* was added to the network, because of its missing Care attribute, it was not given automatically access to *Robot2*. This prevented that *Robot1* would accidentally impact *Robot2* without being fully configured. In our scenario, once *Robot1* will be fully configured, tested, and verified, it will be given back the Care attribute.

In general, this design choice of the invariant template prevents that a newly added host may inherit access rights due to stale entries in access lists. At the same time, it does not force administrators to clean up their access lists because a host may only be removed temporarily and wants to be given back its access rights later on. Note that managing access lists scales quadratically in the number of hosts. In contrast, the Care attribute can be considered as a Boolean flag which allows to temporarily enable or disable the access rights of a host locally without touching the carefully constructed access lists of other hosts. It also prevents that new hosts which have the name of hosts removed long ago (but where stale access rights were not cleaned up) accidentally inherit their access rights.

This design of the invariant template was motivated by the requirements for the secure default parameter.

Security Invariant 5, Domain Hierarchy. The production line is designed according to a strict command hierarchy. On top of the hierarchy are control terminals which allow a human operator to intervene and supervise the production process. On the level below, one distinguishes between supervision devices and control devices.

The watchdog is a typical supervision device whereas the mission control devices are control devices. Directly below the control devices are the robots. This is the structure that is necessary for the example. However, the company defined a few more sub-departments for future use. The full domain hierarchy tree is visualized below.

$$\begin{tikzcd}
 & & & \textbf{ControlTerminal} \arrow[dll, phantom, "\sqsupseteq" sloped] \arrow[drr, phantom, "\sqsubseteq" sloped] & & & \\
 & \textbf{ControlDevices} \arrow[dl, phantom, "\sqsupseteq" sloped] \arrow[d, phantom, "\sqcup\!\!|"] \arrow[dr, phantom, "\sqsubseteq" sloped] & & & & \textbf{Supervision} \arrow[dl, phantom, "\sqcup\!\!|"] \arrow[dr, phantom, "\sqsubseteq" sloped] & \\
\textbf{Robots} & \textbf{OtherStuff} & \textbf{Domain3} & & \mathbf{S_1} & & \mathbf{S_2}
\end{tikzcd}$$

Apart from the watchdog, only the following linear part of the tree is used: **Robots** $\sqsubseteq$ **ControlDevices** $\sqsubseteq$ **ControlTerminal**. Because the watchdog is in a different domain, it needs a trust level of 1 to access the robots it is monitoring.

MissionControl1	$\mapsto$	(level : *ControlTerminal.ControlDevices*, trust : 0)
MissionControl2	$\mapsto$	(level : *ControlTerminal.ControlDevices*, trust : 0)
Watchdog	$\mapsto$	(level : *ControlTerminal.ControlDevices.Robots*, trust : 0)
Robot1	$\mapsto$	(level : *ControlTerminal.Supervision*, trust : 1)
Robot2	$\mapsto$	(level : *ControlTerminal.ControlDevices.Robots*, trust : 0)
AdminPc	$\mapsto$	(level : *ControlTerminal*, trust : 0)

Security Invariant 6, Policy Enforcement Point. The sensors should not communicate with each other; all accesses must be mediated by the sensor sink.

SensorSink	$\mapsto$	PolEnforcePointIN
PresenceSensor	$\mapsto$	DomainMember
Webcam	$\mapsto$	DomainMember
TempSensor	$\mapsto$	DomainMember
FireSensor	$\mapsto$	DomainMember

Security Invariant 7, Sink. The actual control program of the robots is a corporate trade secret. The control commands must not leave the robots. Therefore, they are declared information sinks. In addition, the control command must not leave the mission control devices. However, the two devices could possibly interact to synchronize and they must send their commands to the robots. Therefore, they are labeled as sink pools.

MissionControl1	$\mapsto$	SinkPool
MissionControl2	$\mapsto$	SinkPool
Robot1	$\mapsto$	Sink
Robot2	$\mapsto$	Sink

Security Invariant 8, Subnets. The sensors, including their sink and statistics server are located in their own subnet and must not be accessible from elsewhere.

Also, the administrator's PC is in its own subnet. The production units (mission control and robots) are already isolated by the DomainHierarchy and are not added to a subnet explicitly.

Statistics	$\mapsto$	Subnet 1
SensorSink	$\mapsto$	Subnet 1
PresenceSensor	$\mapsto$	Subnet 1
Webcam	$\mapsto$	Subnet 1
TempSensor	$\mapsto$	Subnet 1
FireSensor	$\mapsto$	Subnet 1
AdminPc	$\mapsto$	Subnet 4

Security Invariant 9, SubnetsInGW. The statistics server is further protected from external accesses. Another, smaller subnet is defined with the only member being the statistics server. The only way it may be accessed is via that sensor sink.

Statistics	$\mapsto$	Member
SensorSink	$\mapsto$	InboundGateway

Security Invariant 10, NonInterference. Finally, there is a final constraint. The fire sensor is managed by an external company and has a built-in GSM module to call the fire fighters in case of an emergency. This additional, out-of-band connectivity is not modeled. However, the contract defines that the company's administrator must not interfere in any way with the fire sensor.

Statistics	$\mapsto$	Unrelated
SensorSink	$\mapsto$	Unrelated
PresenceSensor	$\mapsto$	Unrelated
Webcam	$\mapsto$	Unrelated
TempSensor	$\mapsto$	Unrelated
FireSensor	$\mapsto$	Interfering
MissionControl1	$\mapsto$	Unrelated
MissionControl2	$\mapsto$	Unrelated
Watchdog	$\mapsto$	Unrelated
Robot1	$\mapsto$	Unrelated
Robot2	$\mapsto$	Unrelated
AdminPc	$\mapsto$	Interfering
INET	$\mapsto$	Unrelated

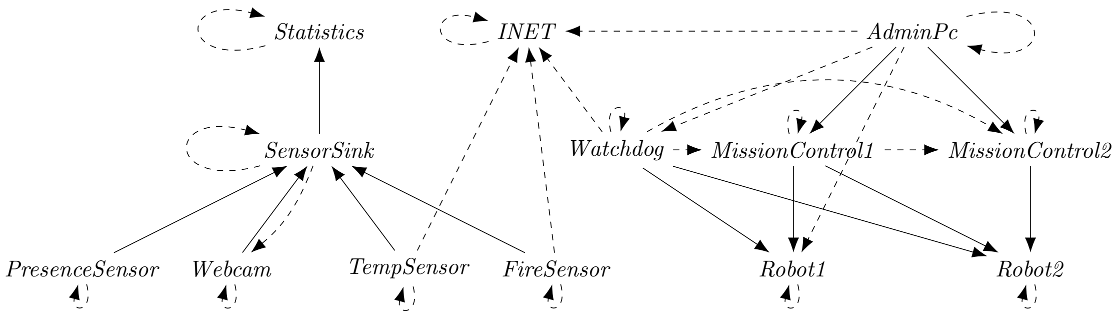
As discussed in Section 6.10, this invariant is very strict and rather theoretical. It is not Φ-structured and may produce an exponential number of offending flows. Therefore, we exclude it by default from our algorithms for now.

7.3.3 Policy Verification

The given policy fulfills all the specified security invariants. Also, including invariant 10 (NonInterference), the policy fulfills all security invariants.

The question, *"how good are the specified security invariants?"* remains. Therefore, we use the algorithm from Section 5.4 to generate a policy. Then, we will compare our manually-specified policy with the automatically generated one. If we exclude the NonInterference invariant from the policy construction, we know that the resulting policy must be maximal. Therefore, the computed policy reflects the view of the specified security invariants and, thus, gives a direct feedback whether the specified security invariants express the right thing. By maximality of the computed policy and monotonicity, we know that our manually-specified policy must be a subset of the computed policy. This allows comparing the manually-specified policy to the policy implied by the security invariants: If there are too many flows which are allowed according to the computed policy but which are not in our manually-specified policy, we can conclude that our security invariants are not strict enough.

We visualize this comparison below. The solid edges correspond to the manually-specified policy. The dashed edges correspond to the flows which would be additionally permitted by the computed policy.



The comparison reveals that the following flows would be additionally permitted. We will discuss whether this is acceptable or if the additional permissions indicates that we probably forgot to specify a security goal.

- All reflexive flows, i.e., all hosts can communicate with themselves. Since each host in the policy corresponds to one physical entity, there is no need to explicitly prohibit or allow in-host communication.
- The *SensorSink* may access the *Webcam*. Both share the same security level, there is no problem with this possible information flow. Technically, a bi-directional connection may even be desirable, since this allows the sensor sink to influence the video stream, e.g., request a lower bit rate if it is overloaded.
- Both the *TempSensor* and the *FireSensor* may access the Internet. No security level or other privacy concerns are specified for them. This may raise the question whether this data is indeed public. It is up to the company to decide that this data should also be considered confidential.
- *MissionControl1* can send to *MissionControl2*. This may be desirable since it was stated anyway that the two may need to cooperate. Note that the opposite direction

is definitely prohibited since the critical and secret production step only known to *MissionControl2* must not leak.

- The *Watchdog* may access *MissionControl1*, *MissionControl2*, and the *INET*. While it may be acceptable that the watchdog which monitors the robots may also access the control devices, it should raise a concern that the watchdog may freely send data to the Internet. Indeed, the watchdog can access devices which have corporate trade secrets stored but it was never specified that the watchdog should be treated confidentially. Note that in the current setting, the trade secrets will never leave the robots. This is because the policy only specifies a unidirectional information flow from the watchdog to the robots; the robots will not leak any information back to the watchdog. This also means that the watchdog cannot actually monitor the robots. Later, when implementing the scenario, we will see that the simple, hand-waving argument from the beginning that "*the watchdog connects to the robots and the robots send back their data over the established connection*" will not work because of this possible information leak.
- The *AdminPc* is allowed to access the *Watchdog*, *Robot1*, and the *INET*. Since this machine is trusted anyway, our fictional company does not see a problem with this.

7.3.4 Outlook: About NonInterference

The NonInterference template was deliberately selected for our scenario as one of the 'problematic' and rather theoretical invariants. Our framework allows to specify almost arbitrary invariant templates. We concluded that all non-Φ-structured invariants which may produce an exponential number of offending flows are problematic for practical use. This includes "Comm. With" (Section 6.5), "Not Comm. With" (Section 6.6), Dependability (Section 6.7), and NonInterference (Section 6.10). In this section, we discuss the consequences of the NonInterference invariant for automated policy construction. We will conclude that, though we can solve all technical challenges, said invariants are—due to their inherent ambiguity—not very well suited for automated policy construction.

The computed maximum policy does not fulfill invariant 10 (NonInterference). This is because the fire sensor and the administrator's PC may be indirectly connected over the Internet.

Since the NonInterference template may produce an exponential number of offending flows, it is infeasible to try our automated policy construction algorithm with it. We have tried to do so on a machine with 128 GB of memory but after a few minutes, the computation ran out of memory. On said machine, we were unable to run our policy construction algorithm with the NonInterference invariant for more than five hosts.

In Chapter 8, we will improve the policy construction algorithm. The new algorithm instantly returns a solution for this scenario with a very small memory footprint.

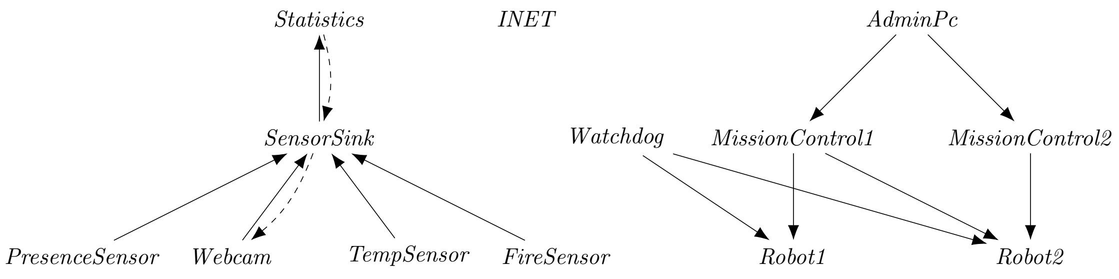
However, it is an inherent property of the NonInterferance template (and similar templates), that the set of offending flows is not uniquely defined. Consequently, since several solutions are possible, even our new algorithm may not be able to compute one maximum solution. It would be possible to construct some maximal solution, however, this would require to enumerate all offending flows, which is infeasible. Therefore, our algorithm can only return some (valid but probably not maximal) solution for non-Φ-structured invariants.

As a human, we know the scenario and the intention behind the policy. Probably, the best solution for policy construction with the NonInterferance property would be to restrict outgoing edges from the fire sensor. If we consider the policy above which was constructed without NonInterference, if we cut off the fire sensor from the Internet, we get a valid policy for the NonInterference property. Unfortunately, an algorithm does not have the information of which flows we would like to cut first and the algorithm needs to make some choice. In this example, the algorithm decides to isolate the administrator's PC from the rest of the world. This is also a valid solution. We could change the order of the elements to tell the algorithm which edges we would rather sacrifice than others. This may help but requires some additional input. The author personally prefers to construct only maximum policies with Φ-structured invariants and afterwards fix the policy manually for the remaining non-Φ-structured invariants. Though our new algorithm gives better results and returns instantly, the very nature of invariant templates with an exponential number of offending flows tells that these invariants are problematic for automated policy construction.

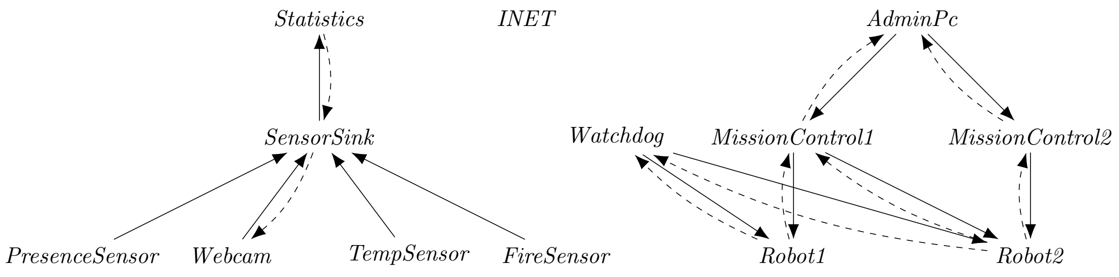
7.3.5 Outlook: Stateful Implementation

In this section, we will implement the policy and deploy it in a network. This requires discussing packet flow on the network level, which is usually bidirectional for TCP. However, our security policy is on the connection level (Def. 1) and includes unidirectional flows. As the scenario description stated, all devices in the production line should establish stateful connections which allows – once the connection is established – packets to travel in both directions. This is necessary for the watchdog, the mission control devices, and the administrator's PC to actually perform their task.

We compute a stateful implementation. We will elaborate on the criteria and the algorithms for this in Chapter 9. Below, the stateful implementation is visualized. It consists of the policy as visualized above. In addition, dashed edges visualize where answer packets are permitted.



As can be seen, only the flows between *SensorSink* ↔ *Webcam* and *Statistics* ↔ *SensorSink* are allowed to be stateful. This setup cannot be practically deployed because the watchdog, the mission control devices, and the administrator's PC also need to set up stateful connections. Previous section's discussion already hinted at this problem. The reason why the desired stateful connections are not permitted is due to information leakage. In detail: Security Invariant 2 (trade secrets) and Security Invariant 7 (robots information sink) are responsible. Both invariants prevent that any data leaves the robots and the mission control devices. To verify this suspicion, the two invariants are removed and the stateful flows are computed again. The result is visualized below.



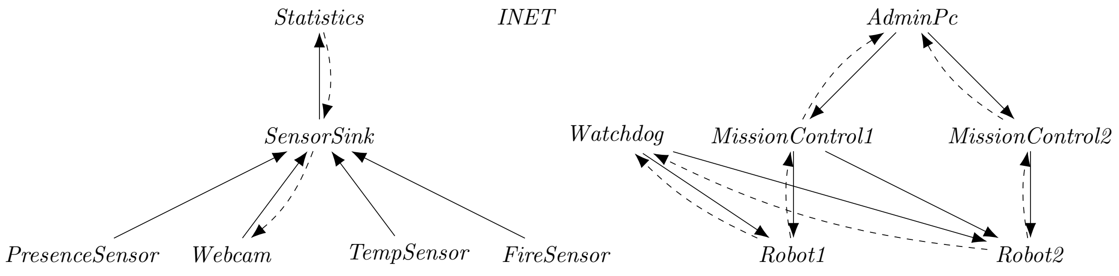
This stateful policy could be transformed into a fully functional implementation. However, there would be no security invariants specified which protect the trade secrets. Without those two invariants, the invariant specification is too permissive. For example, if we recompute the maximum policy, we can see that the robots and mission control can leak any data to the Internet. Even without the maximum policy, in the stateful policy above, it can be seen that *MissionControl1* can exfiltrate information from robot 2, once it establishes a stateful connection.

Therefore, the two invariants are not removed but repaired. The goal is to allow the watchdog, administrator's pc, and the mission control devices to set up stateful connections without leaking corporate trade secrets to the outside.

First, we repair invariant 2. On the one hand, the watchdog should be able to send packets both to *Robot1* and to *Robot2*. *Robot1* has a security level of 1 and *Robot2* has a security level of 2. Consequently, in order to be allowed to send packets to both, *Watchdog* must have a security lvel not higher than 1. On the other hand, the *Watchdog* should be able to receive packets from both. By the same argument, it must have a security level of at least 2. Consequently, it is impossible to express the desired meaning in the simple BLP template. There are only two solutions to the problem: Either the company installs one watchdog for each security level, or the watchdog must be trusted. We decide for the latter option and upgrade the template to the Bell-LaPadula model with trust. We define the watchdog as trusted entity with a security level of 1. This means, it can receive packets from and send packets to both robots but it cannot leak information to the outside world. We do the same for the *AdminPc*.

Then, we repair invariant 7. We realize that the following set of hosts forms one big pool of devices which must all somehow interact but where information must not leave the pool: The administrator's PC, the mission control devices, the robots, and the watchdog. Therefore, all those devices are configured to be in the same SinkPool.

The computed stateful policy with the repaired invariants is visualized below.



It can be seen that all connections which should be stateful are now indeed stateful. In addition, it can be seen that *MissionControl1* cannot set up a stateful connection to *Robot2*.

This is because *MissionControl1* was never declared a trusted device and the confidential information in *MissionControl2* and *Robot2* must not leak.

The improved invariant definition even produces a better (i.e., stricter) maximum policy.

7.3.6 Outlook: Iptables Implementation

In this section, we serialize the stateful policy to an iptables firewall ruleset. Our policy graph only contains positive (i.e., allow) rules. This means, the order in which the rules are installed is irrelevant. Therefore, we set the default policy (`-P`) to `DROP` and iterate over all edges in the policy and emit an `ACCEPT` iptables rule.

The translation for each rule is straightforward. Each rule has a sender and a receiver, which we can translate to iptables. To prevent IP spoofing, we assume that each device is connected to its own interface. Therefore, for the sender of a rule, we match on the input interface (`-i`) and source IP address (`-s`). For the receiver, we match on the output interface (`-o`) and destination IP address (`-d`). For the rules in the policy which are marked stateful, we additionally match on the `ESTABLISHED` state. The resulting ruleset can be seen in Figure 7.5.

We predict that more packets will be send in the 'answer-direction' than the 'connection-setup-direction'. For example, the watchdog will only send one monitoring command and afterwards, a robot will regularly send back health information. For performance reasons, we want the `ESTABLISHED` rules to be on top of the ruleset. To make the translation slightly more interesting, we will mix `-A` (append rule to the back) and `-I` (insert rule on top). We will later verify that the serialized iptables ruleset indeed reflects the desired policy.

To deploy the scenario, we assign each device an IP address according to Table 7.3. Now, the ruleset can be loaded by the Linux kernel. We use the results of Part II to verify the correctness of the generated ruleset. Therefore, we load the ruleset into our analysis tool (Part II) and compute a service matrix for an arbitrary service.

Table 7.3: IP Mapping

Variable	IP Address
\$*Statistics_ipv4*	10.0.0.1
\$*SensorSink_ipv4*	10.0.0.2
\$*PresenceSensor_ipv4*	10.0.1.1
\$*Webcam_ipv4*	10.0.1.2
\$*TempSensor_ipv4*	10.0.1.3
\$*FireSensor_ipv4*	10.0.1.4
\$*MissionControl1_ipv4*	10.8.1.1
\$*MissionControl2_ipv4*	10.8.1.2
\$*Watchdog_ipv4*	10.8.8.1
\$*Robot1_ipv4*	10.8.2.1
\$*Robot2_ipv4*	10.8.2.2
\$*AdminPc_ipv4*	10.8.0.1

The resulting matrix for `NEW` packets is visualized in Figure 7.6. The graph shows who is allowed to set up connections with whom.

```
iptables -P FORWARD DROP
iptables -A FORWARD -i $PresenceSensor_iface -s $PresenceSensor_ipv4                ↩
              -o $SensorSink_iface -d $SensorSink_ipv4 -j ACCEPT
iptables -A FORWARD -i $Webcam_iface -s $Webcam_ipv4                                ↩
              -o $SensorSink_iface -d $SensorSink_ipv4 -j ACCEPT
iptables -A FORWARD -i $TempSensor_iface -s $TempSensor_ipv4                        ↩
              -o $SensorSink_iface -d $SensorSink_ipv4 -j ACCEPT
iptables -A FORWARD -i $FireSensor_iface -s $FireSensor_ipv4                        ↩
              -o $SensorSink_iface -d $SensorSink_ipv4 -j ACCEPT
iptables -A FORWARD -i $SensorSink_iface -s $SensorSink_ipv4                        ↩
              -o $Statistics_iface -d $Statistics_ipv4 -j ACCEPT
iptables -A FORWARD -i $MissionControl1_iface -s $MissionControl1_ipv4              ↩
              -o $Robot1_iface -d $Robot1_ipv4 -j ACCEPT
iptables -A FORWARD -i $MissionControl1_iface -s $MissionControl1_ipv4              ↩
              -o $Robot2_iface -d $Robot2_ipv4 -j ACCEPT
iptables -A FORWARD -i $MissionControl2_iface -s $MissionControl2_ipv4              ↩
              -o $Robot2_iface -d $Robot2_ipv4 -j ACCEPT
iptables -A FORWARD -i $AdminPc_iface -s $AdminPc_ipv4                              ↩
              -o $MissionControl2_iface -d $MissionControl2_ipv4 -j ACCEPT
iptables -A FORWARD -i $AdminPc_iface -s $AdminPc_ipv4                              ↩
              -o $MissionControl1_iface -d $MissionControl1_ipv4 -j ACCEPT
iptables -A FORWARD -i $Watchdog_iface -s $Watchdog_ipv4                            ↩
              -o $Robot1_iface -d $Robot1_ipv4 -j ACCEPT
iptables -A FORWARD -i $Watchdog_iface -s $Watchdog_ipv4                            ↩
              -o $Robot2_iface -d $Robot2_ipv4 -j ACCEPT
# SensorSink -> Webcam (answer)
iptables -I FORWARD -m state --state ESTABLISHED -i $SensorSink_iface -s $SensorSink_ipv4 ↩
              -o $Webcam_iface -d $Webcam_ipv4 -j ACCEPT
# Statistics -> SensorSink (answer)
iptables -I FORWARD -m state --state ESTABLISHED -i $Statistics_iface -s $Statistics_ipv4 ↩
              -o $SensorSink_iface -d $SensorSink_ipv4 -j ACCEPT
# Robot1 -> MissionControl1 (answer)
iptables -I FORWARD -m state --state ESTABLISHED -i $Robot1_iface -s $Robot1_ipv4   ↩
              -o $MissionControl1_iface -d $MissionControl1_ipv4 -j ACCEPT
# Robot2 -> MissionControl2 (answer)
iptables -I FORWARD -m state --state ESTABLISHED -i $Robot2_iface -s $Robot2_ipv4   ↩
              -o $MissionControl2_iface -d $MissionControl2_ipv4 -j ACCEPT
# MissionControl2 -> AdminPc (answer)
iptables -I FORWARD -m state --state ESTABLISHED                                    ↩
              -i $MissionControl2_iface -s $MissionControl2_ipv4                    ↩
              -o $AdminPc_iface -d $AdminPc_ipv4 -j ACCEPT
# MissionControl1 -> AdminPc (answer)
iptables -I FORWARD -m state --state ESTABLISHED                                    ↩
              -i $MissionControl1_iface -s $MissionControl1_ipv4                    ↩
              -o $AdminPc_iface -d $AdminPc_ipv4 -j ACCEPT
# Robot1 -> Watchdog (answer)
iptables -I FORWARD -m state --state ESTABLISHED -i $Robot1_iface -s $Robot1_ipv4   ↩
              -o $Watchdog_iface -d $Watchdog_ipv4 -j ACCEPT
# Robot2 -> Watchdog (answer)
iptables -I FORWARD -m state --state ESTABLISHED -i $Robot2_iface -s $Robot2_ipv4   ↩
              -o $Watchdog_iface -d $Watchdog_ipv4 -j ACCEPT
```

Figure 7.5: iptables ruleset

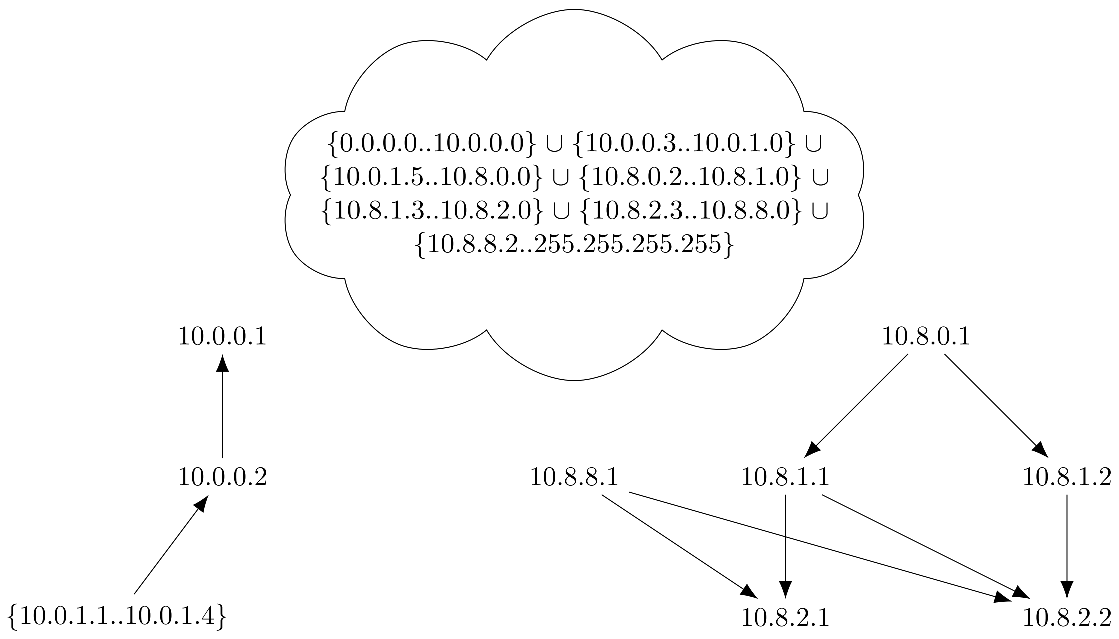


Figure 7.6: Analysis of firewall rules of Figure 7.5. New connections.

Mapping back the IP addresses to the names in the policy, we see that the iptables ruleset indeed corresponds to our desired policy. It can be seen that our analysis tool has pooled all sensors into one node because they all have the same access rights. The node at the top with the complicated IP range specification corresponds to all IP addresses which are not used in our factory: the *INET*.

Next, we verify that all `ESTABLISHED` connections are implemented as desired. The connectivity matrix is visualized in Figure 7.7.

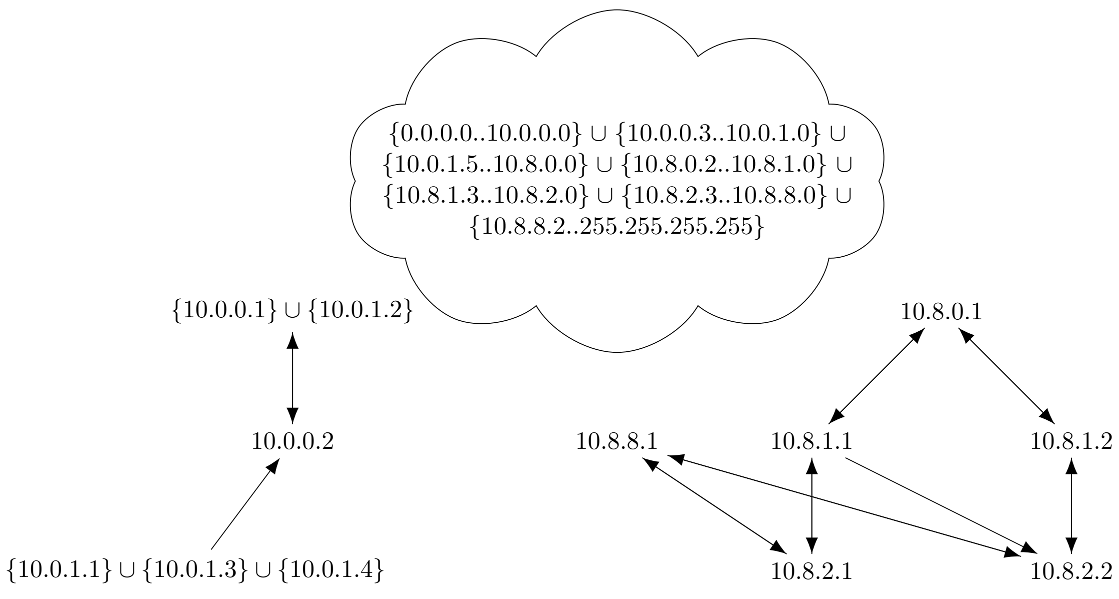


Figure 7.7: Analysis of firewall rules of Figure 7.5. Established connections.

It can be seen that the statistics server and webcam are condensed into one node. Comparing to the stateful policy, they indeed have the same access rights for stateful connections. The policy does not show a reflexive rule for this node, hence, they still cannot communicate directly. All other sensors are only allowed unidirectional information flows, as specified by the policy. All other edges (except for mission control one to robot two) are bidirectional. This corresponds to the desired connectivity structure for established connections.

Consequently, we have verified that the iptables implementation exactly corresponds to the desired stateful policy.

7.4 Related Work

In a field study with 38 participants, Hamed and Al-Shaer discovered that "even expert administrators can make serious mistakes when configuring the network security policy" [9]. Our user feedback session extends this finding as we discovered that even expert administrators can make serious mistakes when *designing* the network security policy.

Noteworthy, Ou et al. [114] summarize a monotonicity property that is very similar to ours but from the opposite point of view: "gaining privileges does not hurt an attacker[...]".

In the context of developing and implementing network protocols, Wang et al. [118] propose a framework called "Formally Verifiable Networking". They demonstrate the use of a theorem prover and leverage that this allows formalizing and verifying a specification and generate executable code out of certain specifications. The authors highlight that this approach enables two compatible ways to get a protocol implementation: First, a user can specify the protocol and (manually) verify the specification in the theorem prover and finally generate an executable code (if the specification allows it). The second way starts with an NDlog (Network Datalog) implementation of the protocol, which is translated into the theorem prover which can then be (manually) verified. Our approach provides an analogue advantage of two compatible ways to get a policy. First, a user can specify the security invariants and automatically derive a policy from it (cf. Section 5.4). Second, a user can define a policy and verify that it corresponds to the security invariants; the verification is automatic. Consequently, our approach never requires a manual proof from the user.

Cuppens et al. [21] propose a policy language to administrate firewalls. They follow the traditional approach of the policy community to differentiate between subjects, objects, and actions. Because they are very accurate about their definitions, we can show that their classification can be simplified to our graph-based model. First, we assume that we do not match on the content of a network packet. This assumption can be justified since a firewall is not a deep packet inspection system. In addition, for example, the payload of a packet may be encrypted and a firewall cannot decrypt it. Cuppens et al. model their subjects as the machines in the network. This corresponds to the vertices in our graph. Next, Cuppens et al. model the actions as the allowed services a machine may use, characterized by the protocol and ports. Our graph can be viewed as the projection for a single service or the overall access matrix for the universe of all services.[3] With this view, an action is reduced to the 'send' permission, which is a singleton and hence exactly corresponds to an edge in our graph. Finally, Cuppens et al. model objects as the packets in the network, which are only characterized by their receiver. Since the receiver is always a machine in the network, the objects also corresponds to the subjects, which correspond to the vertices in

[3]Note: Our approach may need several graphs to represent different access rights for different services.

our graph. Consequently, several graphs (according to our model) are as expressive as a triplet of subject, action, object in Cuppens' model. It is unclear which model is 'better'. The advantages of our approach are that it isolates exactly one aspect and is hence simpler, can be easily visualized, and provides elegant algorithms to work with.

In their inspiring work, Guttman and Herzog [6] describe a formal modeling approach for network security management. They suggest algorithms to verify whether configurations of firewalls and IPsec gateways fulfill certain security goals. These comparatively low-level security goals may state that a certain packet's path only passes certain areas or that packets between two networked hosts are protected by IPsec's ESP confidentiality header. This allows reasoning on a lower abstraction level at the cost of higher manual specification and configuration effort. Header space analysis [119] allows checking static network invariants such as no-forwarding-loops or traffic-isolation on the forwarding and middleboxes plane. It provides a common, protocol-agnostic framework and algebra on the packet header bits.

Firmato [33] was designed to ease management of firewalls. A firewall-independent entity relationship model is used to specify the security policy. With the help of a model compiler, such a model can be translated to firewall configurations. Ethane [58] is a link layer security architecture which evolved to the network operating system NOX [120]. They implement high-level security policies and propose a secure binding from host names to network addresses. In the long term, we consider *topoS* a valuable add-on on top of such systems for policy verification. For example, it could warn the administrator that a recent network policy change violates a security invariant, maybe defined years ago.

NetCore [28] is a language for packet-forwarding policies in software defined networks that abstracts from low-level hardware details. However, compared to the abstract graph utilized in our work, it can be considered rather technical and low-level. However, from an abstract point of view, both describe the network topology. Guha et al. [121] present a verified compiler to translate NetCore to a SDN controller. The authors use the Coq proof assistant [122] to verify the correctness of their SDN controller, demonstrate its suitable performance, and uncover bugs in other non-machine-verified controllers. As Guha et al. provide formally verified means to translate a network topology (NetCore) to real hardware and we provide formally verified means to verify the intention behind the topology (graph), we see great potential to expect provably correct networks from the abstract human intent down to the low-level hardware in the near future.

Expressive policy specification languages, such as Ponder [123], were proposed. Positive authorization policies (only a small aspect of Ponder) are roughly comparable to our policy graph. The authors note that e.g., negative authorization policies (deny-rules) can create conflicts. Policy constraints can be checked at compile time. Bera et al. [36] present a policy specification language (SPSL) with allow and deny policy rules. With this, a conflict-free policy specification is constructed. Conflict-free Boolean formulas of this policy description and the policy implementation in the security mechanisms (router ACL entries) are checked for equality using a SAT solver. One unique feature covered is hidden service access paths, e.g., http might be prohibited in zone1 but zone1 can ssh to zone2 where http is allowed. Craven et al. [124] focus on policies in dynamic systems and their analysis. These papers require specification of the verification goals and security goals and can thus benefit from our contributions.

This work's modeling concept is very similar to the Attribute Based Access Control (ABAC) model [77], though the underlying formal objects differ. ABAC distinguishes subjects, resources, and environments. Attributes may be assigned to each of these entities, similar to our host mappings. The ABAC policy model consists of positive rules which grant access based on the assigned attributes, comparably to security invariant templates. Therefore, our insights and contributions are also applicable to the ABAC model.

Analogy to Software Architectures Finally, we want to show parallels between our work and scientific results from the field of software development and software engineering. Roughly speaking, a software architecture can be abstractly understood as a high level specification of a software and its documentation. We will use the definition that "[a]rchitecture defines the components of a system and their dependencies" [125]. An architecture is realized by an actual program, i.e., code. For our analogy, we will equate a software architecture with security requirements and we equate code with a security policy.

In software development, researchers identified the problem that the specified software architecture and the actual code diverge over time [127]. In fact, "[o]ne problem with high-level models is that they are always inaccurate with respect to the system's source code" [126]. In real-world case studies, it was shown that documented architectures may vastly diverge from the actual code over time [125] and that "documentation becomes a dead artifact that is used very infrequently" [125]. The same insight may also apply to an informal, textual representation of security requirements.

In the field of software architectures, researches have built tools to visualize the divergence of a specified architecture and the actual implementation [125, 126]. An example can be found in Figure 7.8. Analogously, our method allows to visualize differences between formalized security invariants and an actual policy. Visualizations which are generated by *topoS* look remarkably similar to visualizations of divergence in software architectures. For example in Figure 7.8, in the context of our analogy, a *convergence* would correspond to a flow which is specified in the policy and allowed by the security invariants. A *divergence* would correspond to a flow which is present in the policy but prohibited by the security invariants. An *absence* would correspond to a flow which is not specified in the policy but would be accepted by the computed maximum policy.

Additionally, by the same means as architecture consistency checkers help to uncover errors and architecture drift, by formalizing security requirements with our approach, it can always be ensured that the requirements and the actual policy stay consistent.

Finally, it has been proposed [128] to consider architecture strictness, where strictness refers to a measure which reflects to which degree components may access each other. For example, in a very strict system, no component may access another. This corresponds to our monotonicity principle which could be translated in the context of this analogy as follows: Increasing the strictness of a system does not decrease its security.

7.5 Conclusion

After several hundred thousand changed lines of formal theory, our simple, yet powerful, model landscape emerged. Representing policies as graphs makes them visualizable. Describing security invariants as total Boolean-valued functions is both expressive and accessible to

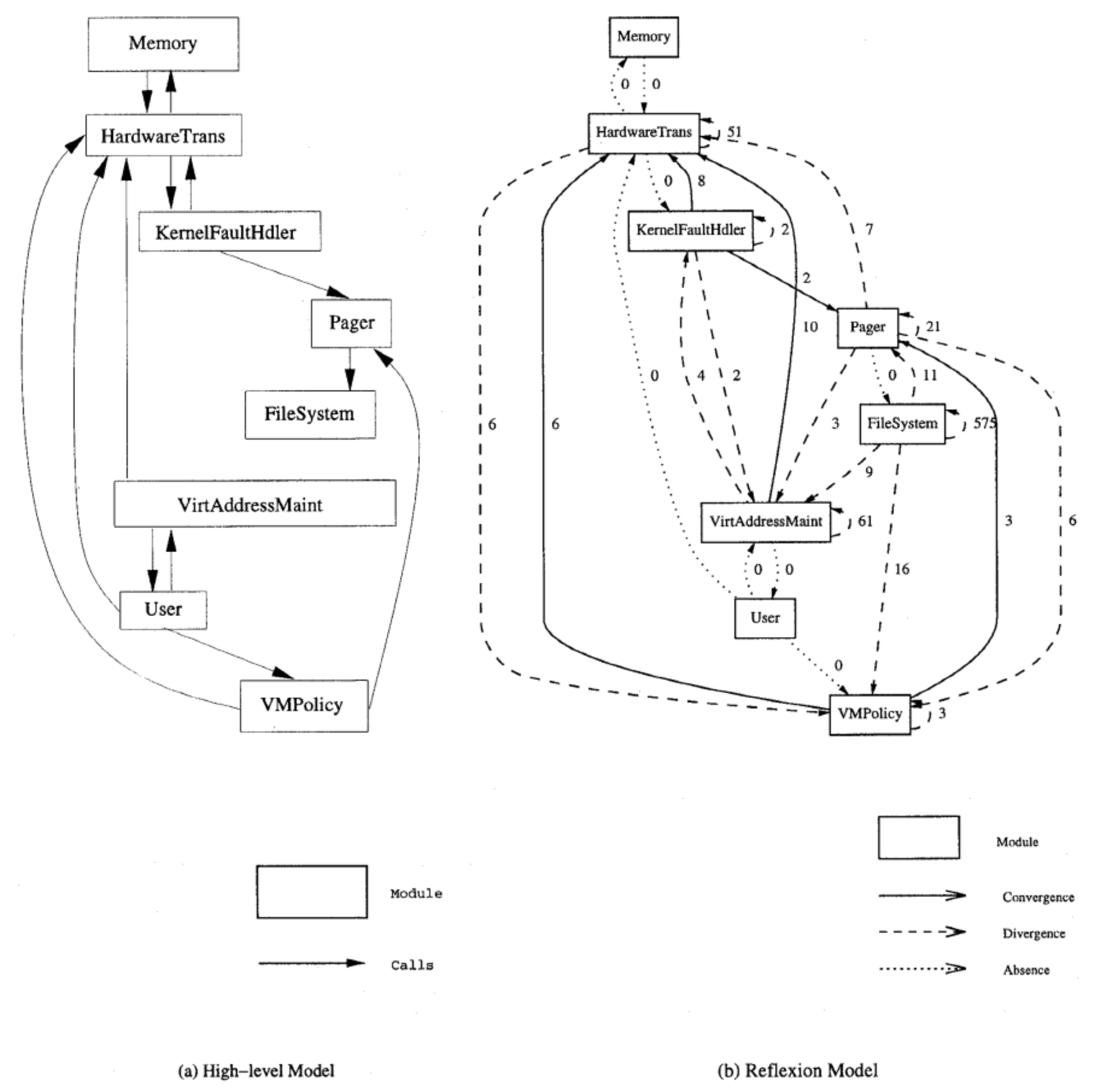


Figure 2: High-level and Reflexion Models for the NetBSD Virtual Memory Subsystem

Figure 7.8: By Gail C. Murphy, David Notkin, and Kevin Sullivan (1995) [126]

formal analysis. Representing host mappings as partial configurations is end-user-friendly, transforming them to total functions makes them handy for the design of templates. With this simple model, we discovered important universal insights on security invariants. In particular, the transformation of host mappings and a simple sanity check which guarantees that security policy violations can always be resolved. This provides deep insights about how to express verification goals. The full formalization in the Isabelle/HOL theorem prover provides high confidence in the correctness.

Chapter 8

Improved Policy Construction

Abstract The previous chapters show that our policy construction algorithm is only practically usable for Φ-structured invariant temaplates. In this chapter, we improve the policy construction algorithm to cope with arbitrary invariants.

8.1 Introduction

The algorithm presented Section 5.4 constructs a security policy which fulfills all security invariants. Summarizing the algorithm with the help of Lemma 2, for a list *Ms* of configured security invariants, the algorithm simply removes all offending flows:

$$\textsf{delete-edges}\ G \left(\bigcup \left(\bigcup m_c \in \textsf{set}\ Ms.\ \textsf{set-offending-flows}\ m_c\ G\right)\right)$$

The main concern with this algorithm is that it needs to construct the complete set of offending flows. This can be done efficiently for Φ-structured invariants. However, as has been shown in Section 6, there are some invariant templates which have a different structure and where the size of the set of offending flows can grow infeasible large. This is for example the case for NonInterference (Section 6.10), Comm. With (Section 6.5), Not Comm. With (Section 6.6), and Dependability (Section 6.7).

In this chapter, we present a new algorithm for policy construction. The main idea of the new algorithm is presented by the following formula.

$$\textsf{delete-edges}\ G \left(\bigcup m_c \in \textsf{set}\ Ms.\ \textbf{if}\ m_c\ G\ \textbf{then}\ \emptyset\ \textbf{else}\ \varepsilon\ (\textsf{set-offending-flows}\ m_c\ G)\right)$$

Here, ε corresponds to Hilbert's ε-operator. We use this operator in a simplified setting.[1] Its first argument is a set and it returns an element from the set. The element which ε returns is chosen non-deterministically: ε is an *indefinite* operator. For example, if $(\varepsilon\ \{1, 2, 3\}) = x$, then $x = 1 \vee x = 2 \vee x = 3$. Nothing can be said if ε is applied to the empty set. Because $\neg\ m_c\ G$ ensures that the set of offending flows is always defined,[2] ε can be safely applied here.

[1]In Isabelle, a more generic version of Hilbert's ε operator is available as one of the core axioms of HOL. The operator can be used with the keyword `SOME`. In the context of this thesis, we can use it in a simplified fashion where we always mean $\varepsilon\ X\ \equiv\ \textsf{SOME}\ x.\ x \in X$.

[2]configured-SecurityInvariant.defined-offending'

Comparing the two formulas on this page, the main difference is the following: The first one removes all offending flows whereas the second one only removes one arbitrary member of the set of offending flows per invariant.[3] Consequently, if the set of offending flows has more than one member, the second algorithm may result in a better (here: more permissive but still sound) result.

The improved algorithm is implemented as follows.

```
generate-valid-topology2 :: $(\mathcal{G} \Rightarrow \mathbb{B})$ list $\Rightarrow \mathcal{G} \Rightarrow \mathcal{G}$
generate-valid-topology2 [] $G$ $= G$
generate-valid-topology2 $(m_c :: Ms)$ $G =$
        if $m_c$ $G$
        then
            generate-valid-topology2 $Ms$ $G$
        else
            delete-edges (generate-valid-topology2 $Ms$ $G$)
                                        ($\varepsilon$ (set-offending-flows $m_c$ $G$))
```

Analogously to generate-valid-topology, there is an alternative, equivalent definition:

Lemma 4.

$$\text{generate-valid-topology2}\ Ms\ G =$$
$$\text{delete\_edges}\ G \left(\bigcup m_c \in \text{set}\ Ms.\ \textbf{if}\ m_c\ G\ \textbf{then}\ \emptyset\ \textbf{else}\ \varepsilon\ (\text{set-offending-flows}\ m_c\ G)\right)$$

The algorithm is sound[4], as shown by the following Theorem.

Theorem 7 (Soundness of ε Policy Construction)**.** *Let M be a set of configured security invariants. This means, each element of M is of type $\mathcal{G} \Rightarrow \mathbb{B}$, is monotonic, and always has defined offending flows. Then*

$$\forall m_c \in M.\ \ m_c\ \ (\text{generate-valid-topology2}\ M\ G)$$

The new algorithm can compute a superset of the policy which can be computed by the old algorithm.

Lemma 5. *Let edges extract the edges from a graph. Then*

$$\text{edges}\ (\text{generate-valid-topology}\ M\ G) \subseteq \text{edges}\ (\text{generate-valid-topology2}\ M\ G)$$

This justifies the claim that the new algorithm is better: It may compute a more permissive policy (Lemma 5), but the results are still sound (Theorem 7). Consequently, when generate-valid-topology can generate a maximum policy for Φ-structured invariants, the policy computed by generate-valid-topology2 must also be maximal.

[3] Note that set-offending-flows is a set of sets and consequently a member of the set of offending flows is a set of flows.

[4] generate-valid-topology-SOME-sound

Problems of the ε operator As of Isabelle 2016, none of the automated solvers could solve the following example automatically: $(\varepsilon\ \{1, 2, 3\}) = x \longrightarrow x = 1 \vee x = 2 \vee x = 3$. Its proof required one manual step. We have chosen the ε operator because it is the most illustrative way to present the ideas behind the algorithm in this chapter.

However, this operator is complicated to work with. In addition, due to its indefinite choice, the final (deterministic) executable implementation will not use ε.

8.2 Computing One Member of the Set of Offending Flows

To implement generate-valid-topology2, one needs to select one member of the set of offending flows. For this algorithm to be efficient also for non-Φ-structured invariant templates where the size of the set of offending flows can grow exponentially, it is important not to compute the complete set of offending flows. In this section, we present a deterministic algorithm which computes exactly one member of the set of offending flows without constructing the whole set.

A member of the set of offending flows F has to fulfill three properties (cf. Definition 5). Here, we summarize them for a configured security invariant m_c:

1. $\neg\, m_c\ G$
2. $m_c\ (V,\ E \setminus F)$
3. $\forall (s, r) \in F.\ \neg\, m_c\ \ (V,\ (E \setminus F) \cup \{(s, r)\})$

The first property states that, independent of F, the invariant must be violated; otherwise, there are no offending flows. The second property states that after removing the flows F from the policy, the security invariant is no longer violated, i.e., F can 'fix' the policy. Finally, the third property states that every individual flow in F must be responsible for the violation of m_c.

In this section, we present an algorithm, given m_c and G, it will calculate one such F which fulfills all three properties. It does not rely on an efficient implementation for set-offending-flows (as we could assume for Φ-structured invariants). For this section, we will assume that the first property ($\neg\, m_c\ G$) holds. Otherwise, the complete set-offending-flows is trivially the empty set.

The algorithm needs an over-approximation of F to start with. By the term over-approximation, we mean a set such that property two already holds but property three may not hold.

Example. We showed that any well-formed security invariant should be fulfilled for the deny-all policy, i.e., $m_c\ (V, \emptyset)$, cf. Theorem 1. We require this property for any well-formed security invariant. Consequently, for $G = (V, E)$, the complete set of edges E is an over-approximation which fulfills the second property.

We will call the over-approximation to start the algorithm with *fs*. It is written in lower case because *fs* must be a (finite) list. In addition, the set of *fs* must be a subset of the edges of G and *fs* must be distinct. As has been shown by the ε operator, it is enough to compute one arbitrary member of the set of offending flows. In contrast to sets where the order of the elements does not matter, the order in the list *fs* determines which member of the offending flows is computed.

The algorithm takes four parameters. The first parameter is the configured security invariant m_c. The second parameter, *fs* is an over-approximation of the offending flows. The third parameter, *keeps*, corresponds to the flows which will be returned after minimizing. The fourth parameter is the security policy G.

```
minimalize-offending-overapprox
            $:: (\mathcal{G} \Rightarrow \mathbb{B}) \Rightarrow (\mathcal{V} \times \mathcal{V})$ list $\Rightarrow (\mathcal{V} \times \mathcal{V})$ list $\Rightarrow \mathcal{G} \Rightarrow (\mathcal{V} \times \mathcal{V})$ list
minimalize-offending-overapprox _ [] $keeps$ _        $= keeps$
minimalize-offending-overapprox $m_c$ $(f :: fs)$ $keeps$ $G =$
      if $m_c$ (delete-edges $G$ $(fs ::: keeps)$)
      then
            minimalize-offending-verapprox $m_c$ $fs$ $keeps$ $G$
      else
            minimalize-offending-overapprox $m_c$ $fs$ $(f :: keeps)$ $G$
```

Idea of the Algorithm The first and the fourth parameter are fixed and do not change during a run on the algorithm. The algorithm iterates over its second parameter *fs* and stores intermediate results in its third parameter *keeps*. For each flow f in *fs*, it checks whether it is necessary and responsible for the violation of m_c. Therefore, the algorithm checks whether m_c is valid if *fs* together with the *keeps* but without f is removed. If this is the case, the violation of m_c can be fixed without f, consequently, f is not responsible for the violation and can be removed. Otherwise, f is responsible for a violation and is saved in the *keeps* and will be part of the final result.

There are many constraints for *keeps* which can be found in the formalization. They are necessary for the correctness proof. Setting *keeps* = [] fulfills all constraints and is the only way we will ultimately call the algorithm. Lemma 6 proves[5] correctness of the algorithm: If called with the right set of parameters, it returns one member of the set of offending flows.

Lemma 6. *Assume* $\neg\, m_c\ G$, *and* $G = (V, E)$. *Let Es be a distinct list which corresponds to the set* E. *Then*

$$\begin{gathered}\text{set (minimalize-offending-overapprox } m_c\ Es\ []\ G) \\ \in \\ \text{set-offending-flows } m_c\ G\end{gathered}$$

The runtime of minimalize-offending-overapprox is $|E|$ times the runtime of m_c. If used for policy construction, it will call m_c exactly $|V|^2$ times. Hence, if m_c can be computed in polynomial time, minimalize-offending-overapprox is also in polynomial time.

With minimalize-offending-overapprox, an executable, efficient, and deterministic implementation of generate-valid-topology2 is obtained. We will call it generate-valid-topology3.

[5]lemma minimalize-offending-overapprox-gives-some-offending-flow

It is implemented as follows.

```
generate-valid-topology3 :: $(\mathcal{G} \Rightarrow \mathbb{B})$ list $\Rightarrow \mathcal{G} \Rightarrow \mathcal{G}$
generate-valid-topology3 [] $G$ $= G$
generate-valid-topology3 $(m_c :: Ms)$ $G =$
        if $m_c$ $G$
        then
            generate-valid-topology3 $Ms$ $G$
        else
        delete-edges (generate-valid-topology3 $Ms$ $G$)
          (minimalize-offending-overapprox $m_c$ (edges $G$) [] $G$)
```

The same ideas as applied in Theorem 7 can be applied to show that generate-valid-topology3 is sound.[6]

In Section 7.3.4, we have already presented that our improved algorithm can immediately compute a policy, even with the ‘problematic’ NonInterference invariant. It also showed that a user may influence the result of generate-valid-topology3 by reordering the edges: The edges which are listed first are preferred. Therefore, our framework now supports any kinds of invariant templates.

[6]generate-valid-topology-some-sound

Chapter 9

Directed Security Policies: A Stateful Network Implementation

This chapter is an extended version of the following paper [129]:

- Cornelius Diekmann, Lars Hupel, and Georg Carle. *Directed Security Policies: A Stateful Network Implementation.* In Engineering Safety and Security Systems, volume 150 of Electronic Proceedings in Theoretical Computer Science, pages 20-34, Singapore, May 2014. Open Publishing Association.

The following improvements and new contributions were added:

- This work has been evaluated with the cabin data network scenario (Section 9.7).

Statement on author's contributions For the original paper, the author of this thesis provided major contributions for the ideas, requirement specification, formalization, realization, implementation, and proof of the algorithms. He researched related work, evaluated, and conducted the case study. All improvements with regard to the paper are the work of the author of this thesis.

Abstract A security policy describes the communication relationship between networked entities. The security policy defines rules, for example that A can connect to B. In the previous chapters, the policy was represented as a directed graph on the connection level. This policy should be implemented in a network, for example by firewalls, such that A can establish a connection to B and all packets belonging to established connections are allowed. We call this a stateful implementation. This stateful implementation is usually required for a network's functionality, but it introduces the backflow from B to A, which might contradict the security policy. We derive compliance criteria for a policy and its stateful implementation and present a fast algorithm to translate a security policy to a stateful policy.

9.1 Introduction

Large systems with high requirements for security and reliability, such as SCADA or enterprise landscapes, no longer exist in isolation but are internetworked [130]. Uncontrolled information leakage and access control violations may cause severe financial loss – as demonstrated by Stuxnet – and may even harm people if critical infrastructure is attacked. Hence, network security is crucial for system security.

A central task of a network security policy is defining the network's desired connectivity structure and hence decreasing its attack surface against access control breaches and information leakage. A security policy defines, among others, rules determining which host is allowed to communicate with which other hosts. One of the most prominent security mechanisms to enforce a policy are network firewalls. For adequate protection by a firewall, its ruleset is critical [88, 33]. For example, let A and B be sets of networked hosts identified by their IP addresses. Let $A \rightarrow B$ denote a policy rule describing that A is allowed to communicate with B. Several solutions from the fields of formal testing [131] to formal verification [13] can guarantee that a firewall actually implements the policy $A \rightarrow B$. However, to the best of our knowledge, one subtlety between firewall rules and policy rules remains unsolved: For different scenarios, there are diverging means with different protection for translating the connection-level rule $A \rightarrow B$ to network-level firewall rules. We will exemplify this by two scenarios.

Scenario 1 Let A be a workstation in some local network and B represent a hosts in the Internet. The policy rule $A \rightarrow B$ can be justified as follows: The workstation can access the Internet, but the hosts in the Internet cannot access the workstation, i.e., the workstation is protected from attacks from the Internet. This policy can be translated to e.g., the Linux iptables firewall [23] as illustrated in Figure 9.1. The first rule allows A to establish a new connection to B. The second rule allows any communication over established connections in both directions, a very common practice. For example, A can request a website and the answer is transmitted back to A over the established connection. Finally, the last rule drops all other packets. In particular, no one can establish a connection to A; hence A is protected from malicious accesses from the Internet.

```
iptables -A INPUT -s $A -d $B -m conntrack --ctstate NEW -j ACCEPT
iptables -A INPUT -m conntrack --ctstate ESTABLISHED -j ACCEPT
iptables -A INPUT -j DROP
```

Figure 9.1: Stateful implementation of $A \rightarrow B$ in Scenario 1

```
iptables -A INPUT -s $A -d $B -j ACCEPT
iptables -A INPUT -j DROP
```

Figure 9.2: Stateless implementation of $A \rightarrow B$ in Scenario 2

Scenario 2 In a different scenario, the same policy rule $A \rightarrow B$ has to be translated to a completely different set of firewall rules. Assume that A is a smart meter recording electrical energy consumption data, which is in turn sent to the provider's billing gateway B. There, smart meter records of many customers are collected. That data must not flow back to any customer, as this could be a violation of other customers' privacy. For example, under the assumption that B sends packets back to A, a malicious customer could try to infer the energy consumption records of their neighbors with a timing attack. In Germany, the requirement for unidirectional communication of smart meters is even standardized by a federal government agency [132]. The corresponding firewall rules for this scenario can be written down as shown in Figure 9.2. The first rule allows packets from A to B, whereas the second rule discards all other packets. No connection state is established; hence no packets can be sent from B to A.

These two firewall rulesets were created from the same security policy rule $A \to B$. The first implementation treats "$\to$" as "can initiate connections to", whereas the second implementation treats "$\to$" as "can send packets to". The second implementation appears to be simpler and more secure, and the firewall rules are justifiable more easily by the policy. However, this firewall configuration is undesirable in many scenarios as it might affect the desired functionality of the network. For example, surfing the web is not possible as no responses (i. e., websites) can be transferred back to the requesting host.

A decision must be made whether to implement a policy rule $A \to B$ in the stateful (Figure 9.1) or in the stateless fashion (Figure 9.2). The stateful fashion bears the risk of undesired side effects by allowing packet flows that are opposite to the security policy rule. In particular, this could introduce information leakage. On the other hand, the stateless fashion might impair the network's functionality. Hence, stateful flows are preferable for network operation, but are undesirable with regard to security. In this chapter, we tackle this problem by maximizing the number of policy rules that can be made stateful without introducing security issues.

We can see that even if a well-specified security policy exists, its implementation by a firewall configuration remains a manual and hence error-prone task. A 2012 survey [10] of 57 enterprise network administrators confirms that a "majority of administrators stated misconfiguration as the most common cause of failure" [10]. A study [15] conducted by Verizon from 2004 to 2009 and the United States Secret Service during 2008 and 2009 reveals that data leaks are often caused by configuration errors [62].

In this chapter, we answer the following questions:

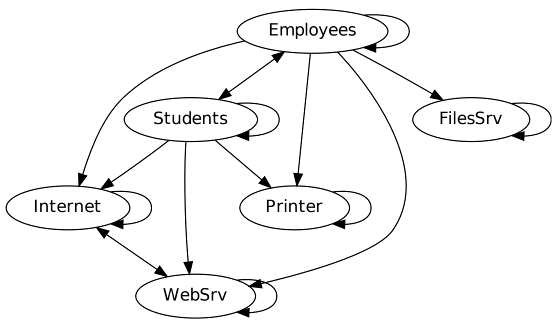
- What conditions can be checked to verify that a stateful policy implementation complies with the directed network security policy rules?
- When can a policy rule $A \to B$ be upgraded to allow a stateful connection between A and B?

Our results apply not only to firewalls but to any network security mechanisms that shape network connectivity.

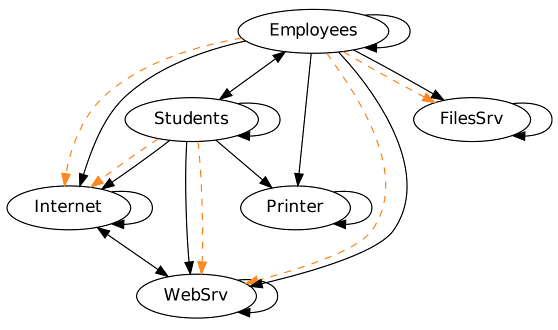
The outline of this chapter is as follows. Section 9.2 presents a guiding example. Section 9.3 formalizes the key concepts of directed policies, security requirements, and stateful policies. Section 9.4 discusses the requirements for a stateful policy to comply with a directed policy. Section 9.5 presents an algorithm to automatically derive a stateful policy. Sections 9.6 and 9.8 evaluate our work: Section 9.6 discusses the computational complexity of the algorithm, and Section 9.8 presents a large real-world case study.

9.2 Example

We introduce a network – from a hypothetical university department – to illustrate the problem with a complete example and outline the solution before we describe its formalization in the next section.



(a) Network security policy



(b) Stateful implementation

Figure 9.3: The network security policy and its stateful implementation

The network (depicted in Figure 9.3) consists of the following participants: the students, the employees, a printer, a file server, a web server, and the Internet. The network security policy rules are depicted in Figure 9.3a as a directed graph. A security policy rule $A \to B$ is denoted by an edge from A to B. We have formalized the security invariants with the help of our security invariant template library. For brevity, we omit the formal configuration details since for this chapter, we will only need the distinction between information flow strategies and access control strategies. For this chapter, we will always write '$\cdot \to \cdot$' to visually denote a policy rule and its direction. The security policy is designed to fulfill the following security invariants:

Access Control Invariants The printer is only accessible by the employees and students; as policy, $\mathit{employees} \to \mathit{printer}$ and $\mathit{students} \to \mathit{printer}$. The file server is only accessible by employees, formally $\mathit{employees} \to \mathit{fileSrv}$. The students and the employees are in a joint subnet that allows collaboration between them but protects against accesses from e.g., the Internet or a compromised web or file server.

Information Flow Invariants The file server stores confidential data that must not leak to untrusted parties. Only the employees have the necessary security level to receive data from the file server. The employees are also trustworthy, i.e., they may declassify and reveal any data received by the file server. The printer is an information sink. Confidential data (such as an exam) might be printed by an employee. No other network participants, in particular no students, are allowed to retrieve any information from the printer that might allow them to draw conclusions about the printed documents. This can be formalized by the policy "$* \to \mathit{printer}$" and "$\mathit{printer} \nrightarrow *$".

Note that Figure 9.3 are screenshots of our tool *topoS*.

Stateful Policy Implementation

Considering Figure 9.3a, it is desirable to allow stateful connections from the employees and students to the Internet and the web server. Figure 9.3b depicts the stateful policy implementation, where the additional dashed edges represent flows that are allowed to be stateful, i.e., answers in the opposite direction are allowed. Only strict stateless unidirectional communication with the printer is necessary. The students and employees can, as already defined by the policy, bidirectionally interact with each other. Hence stateful semantics are not necessary for these flows.

In this chapter, we specify conditions to verify that the stateful policy implementation (e.g., Figure 9.3b) complies with the directed security policy (e.g., Figure 9.3a). We present an efficiently computable condition and formally prove that it implies several complex compliance conditions. Finally, we present an algorithm that automatically computes a stateful policy from the directed policy and the security invariants. We formally prove the algorithm's correctness and that it can always compute a maximal possible set of stateful flows with regard to access control and information flow security strategies.

9.3 Formal Model

In Chapter 5, we introduced our formal model. We presented security invariant templates and how configured security invariants can be derived from them. In this chapter, we only work with configured security invariants. For brevity, we will simply say 'security invariant' for a *configured* security invariant which was derived from a template. We first repeat the core definitions, simplified for configured security invariant templates.

Network Security Policy Rules We represent the network security policy's access rules as directed graph $G = (V, E)$. The type of all graphs is denoted by $\mathcal{G}$. For example, the policy that only consists of the rule that A can send to B, denoted by $A \to B$, is represented by the graph $G = (\{A, B\}, \{(A, B)\})$. An edge in the graph corresponds to a permitted flow in the network. We call this policy a *directed policy.* In Section 9.3.1, we will introduce the notion of a stateful policy.

We consider only syntactically well-formed graphs. A graph is *syntactically well-formed*[1] if all nodes in the edges are also listed in the set of vertices. In addition, since we represent finite networks, we require that V is a finite set. This does not prevent creating nodes that represent collections of arbitrary many hosts, e.g., the node *Internet* in Figure 9.3a represents arbitrarily many hosts.

Network Security Invariants A security invariant m specifies whether a given policy G fulfills its security requirements. As we focus on the network security policy's access rules which specify which hosts are allowed to communicate with which other hosts, we do not take availability or resilience requirements into account. Instead, we deal with only the traditional security invariants that follow the principle "prohibiting more is more or equally secure". We call this principle *monotonicity.* To allow arbitrary network security invariants, almost any total function m of type $\mathcal{G} \Rightarrow \mathbb{B}$ can be used to specify a network security requirement (as long as the proof obligations for the corresponding template imposed by Chapter 5 can be discharged).

We distinguish between the two security strategies that m is set to fulfill: *Information flow security strategies* (IFS) prevent data leakage; *Access control strategies* (ACS) are used to prevent illegal or unauthorized accesses.

[1]FiniteGraph.wf-graph

Definition 12 (Configured Security Invariant)**.** A network security invariant m is a total function $\mathcal{G} \Rightarrow \mathbb{B}$ with a security strategy (either IFS or ACS) satisfying the following conditions:

- If no communication exists in the network, the security invariant must be fulfilled: $m\ (V,\ \emptyset)$
- Monotonicity: $m\ (V, E)\ \wedge\ E' \subseteq E \Longrightarrow m\ (V, E')$

If there is a security violation for m in G, there must be at least one set $F \subseteq E$ such that the security violation can be remedied by removing F from E.[2] We call F offending flows. F is *minimal* if all flows $(s, r) \in F$ contribute to the security violation. For m, the set of all minimal offending flows can be defined. The definition $\mathsf{set\text{-}offending\text{-}flows}$ describes a set of sets, containing all minimal candidates for F.

$$\begin{aligned}&\mathsf{set\text{-}offending\text{-}flows}\ m\ G \equiv\\ &\qquad \big\{F \subseteq E \mid \neg\, m\ G\ \wedge\ m\ (V,\ E \setminus F)\ \ \wedge\ \forall (s,r) \in F.\ \neg\, m\ (V,\ (E \setminus F) \cup \{(s,r)\})\big\}\end{aligned}$$

The offending flows inherit m's monotonicity property.[3]

Lemma 7 (Monotonicity of Offending Flows)**.**

$$E' \subseteq E \Longrightarrow \bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E') \subseteq \bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E)$$

If there is an upper bound for the offending flows, it can be narrowed.[4] This is a key insight which will be used for the efficient implementation of the algorithms in this chapter.

Lemma 8 (Narrowed Upper Bound of Offending Flows)**.** *Let E' be a set of edges. If the offending flows are bounded, i.e., if $\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E) \subseteq X$ holds, then $\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E \setminus E') \subseteq X \setminus E'$.*

Proof. From Lemma 7, we have $\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E \setminus E') \subseteq \bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E)$. This implies that $(\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E \setminus E')) \setminus E' \subseteq (\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E)) \setminus E'$. Since the set of offending flows only returns subsets of the graph's edges, the left hand side can be simplified: $\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E \setminus E') \subseteq (\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E)) \setminus E'$. From the assumption, it follows that $(\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E)) \setminus E' \subseteq X \setminus E'$. We finally obtain

$$\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E \setminus E') \subseteq \Big(\bigcup \mathsf{set\text{-}offending\text{-}flows}\ m\ (V, E)\Big) \setminus E' \subseteq X \setminus E'$$

by transitivity.

□

We now define some helper functions for a set of configured security invariants.

[2]Since $m\ (V,\ \emptyset)$, it is obvious that such a set always exists.
[3]offending-flows-union-mono
[4]Un-set-offending-flows-bound-minus-subseteq

Definition 13 (Configured Security Invariants)**.** We call a finite list of security invariants $M = [m_1, m_2, ..., m_k]$ a network's security invariants. The functions $\mathsf{getIFS}\ M$ (and $\mathsf{getACS}\ M$) return all $m \in M$ with an IFS (and ACS, respectively) security strategy. Additionally, we abbreviate all sets of offending flows for all security invariants with $\mathsf{get\text{-}offending\text{-}flows}\ M\ G = \bigcup_{m \in M} \mathsf{set\text{-}offending\text{-}flows}\ m\ G$. Similarly to $\mathsf{set\text{-}offending\text{-}flows}$, it denotes a set of sets.

9.3.1 Stateful Policy Implementation

We define a stateful policy similarly to a directed policy.

Definition 14 (Stateful Policy)**.** A stateful policy $T = (V,\ E_\tau,\ E_\sigma)$ is a triple consisting of the networked hosts V, the flows E_τ, and the stateful flows $E_\sigma \subseteq E_\tau$.

The meaning of E_σ is that these flows are allowed to be stateful. We consider the stateful flows E_σ as "upgraded" flows, hence $E_\sigma \subseteq E_\tau$. This means that if $(s, r) \in E_\sigma$, flows in the opposite direction, i.e., (r, s) may exist. For a set of edges X, we define the *backflows* of X as $\overleftarrow{X} = \{(r, s) \mid (s, r) \in X\}$. Hence, the semantics of E_σ can be described as that both the flows E_σ and $\overleftarrow{E_\sigma}$ may exist. We define a mapping that translates a stateful policy T to a directed policy G as $\alpha\ T \equiv (V,\ E_\tau \cup E_\sigma \cup \overleftarrow{E_\sigma})$.

> **Example.** The ultimate goal is to translate a directed policy $G = (V,\ E)$ to a stateful implementation $T = (V,\ E_\tau,\ E_\sigma)$ that contains as many stateful flows E_σ as possible without introducing security flaws. The trivial choice is $T_{\text{triv}} = (V,\ E,\ \emptyset)$. It fulfills all security invariants because $\alpha\ T_{\text{triv}} = G$. Since $E_\sigma = \emptyset$, it does not maximize the stateful flows.

Before discussing requirements for the compliance of T and G, we first have to define the requirements for a *syntactically well-formed* stateful security policy.[5] All nodes mentioned in E_τ and E_σ must be listed in V. The flows E_τ must be allowed by the directed policy, hence $E_\tau \subseteq E$, which also implies $E_\sigma \subseteq E$ by transitivity. The nodes in T are equal to the nodes in G. This implies that E_τ and E_σ are finite[6]. In the rest of this chapter, we always assume that T is syntactically well-formed.

From these conditions, we conclude that T and G are similar and T syntactically introduces neither new hosts nor flows. Semantically, however, $\alpha\ T$ adds $\overleftarrow{E_\sigma}$, which might introduce new flows. Hence, the edges of $\alpha\ T$ need not be a subset of G's edges (nor vice versa).

9.4 Requirements for Stateful Policy Implementation

We assume that G is a *valid policy.* In addition to being syntactically well-formed, that means that all security invariants must be fulfilled, i.e., $\forall m \in M.\ m\ G$. We derive requirements to verify that a stateful policy T is a proper stateful implementation of G without introducing security flaws.

[5]wf-stateful-policy, stateful-policy-compliance

[6]wf-stateful-policy.finite-$*$

9.4.1 Requirements for Information Flow Security Compliance

Information leakages are critical and can occur in subtle ways. For example, the widely used transport protocol TCP detects data loss by sending acknowledgment packets. If A establishes a TCP connection to B, then even if B sends no payload, arbitrary information can be transmitted to A, e.g., via timing channels, TCP sequence numbers, or retransmits. Therefore, we treat information flow security requirements carefully: When considering backflows, all information flow security invariants must still be fulfilled.

$$\forall m \in \mathsf{getIFS}\ M.\ \ m\ (\alpha\ T) \tag{9.1}$$

Example. For our simple BLP example on page 25, this means that no TCP connection can be established between hosts of different security levels.

9.4.2 Requirements for Access Control Strategies

In contrast, the requirements for access control invariants can be slightly relaxed: If A accesses B, A might expect an answer from B for its request. If B's answer is transmitted via the connection that A established, B does not access A on its own initiative. Only the expected answer is transmitted back to A. If A's software contains no vulnerability which B could exploit with its answer, no access violation occurs.[7] This behavior is widely deployed in many private and enterprise networks by the standard policy that internal hosts can access the Internet and receive replies, but the Internet cannot initiate connections to internal hosts.

Therefore, we can formulate the requirement for ACS compliance. Access control violations caused by stateful backflows can be tolerated. However, *negative side effects* must not be introduced by permitting these backflows. First, we present an example of a negative side effect. Second, we derive a requirement for verifying the lack of side effects.

Example. We examine a building automation network. Let B be the master controller, A a door locking mechanism, and C a log server that records who enters and who leaves the building. The controller B decides when the door should be opened and what to log. The directed policy is described by $G = (\{A, B, C\},\ \{(B, A), (B, C)\})$. The only security invariant m is that A is not allowed to transitively access C. Let $\rightarrow^*$ denote the transitive closure of $\rightarrow$. Then, m prohibits $A \rightarrow^* C$, but it does not prohibit $C \rightarrow^* A$. In this scenario, that means that the physically accessible locking mechanism must not tamper with the integrity of the log server.

Setting $E_\sigma = \{(B, A)\}$ gives $T = (\{A, B, C\},\ \{(B, A), (B, C)\},\ \{(B, A)\})$, and hence $\alpha\ T = (\{A, B, C\}, \{(B, A), (B, C), (A, B)\})$. This attempt results in a negative side effect. We compute the offending flows for m of $\alpha\ T$ as $\{\{(B, C)\},\ \{(A, B)\}\} = \{\{(B, C)\},\ \overleftarrow{E_\sigma}\}$. Clearly, a violation occurs in $\overleftarrow{E_\sigma}$. Additionally, there is a side effect: the flow from B to C could now cause a violation. Applied to our scenario, this means that in case the locking mechanism sends forged data to the controller, that data could

[7]Note that we make an important assumption here. This assumption is justified as we only work on the network level and do not consider the application level, which is also the correct abstraction for network administrators when configuring network security mechanisms. It also implies that, as always, vulnerable applications with access to the Internet can cause severe damage.

end up in the log. This is a negative side effect. Hence (B, A) cannot securely be made stateful. For completeness, note that because A is just a simple physical actor which only executes B's commands, there is no need for bidirectional communication. On the other hand, (B, C) can be made stateful without side effects.

We formalize the requirement of "no negative side effects" as follows: The violations caused by any subset of the backflows are at most these backflows themselves.

$$\forall X \subseteq \overleftarrow{E_\sigma}.\ \forall F \in \textsf{get-offending-flows}\ (\textsf{getACS}\ M)\ (V,\ E_\tau \cup E_\sigma \cup X).\ F \subseteq X \tag{9.2}$$

In particular, all offending access control violations are at most the stateful backflows. This is directly implied by the previous requirement by choosing X to be $\overleftarrow{E_\sigma}$ (recall the definition of α).

$$\bigcup \textsf{get-offending-flows}\ (\textsf{getACS}\ M)\ (\alpha\ T) \subseteq \overleftarrow{E_\sigma} \tag{9.3}$$

Also, considering all backflows individually, they cause no side effects, i.e., the only violation added is the backflow itself.

$$\begin{aligned}&\forall (r,s) \in \overleftarrow{E_\sigma}.\\ &\quad \bigcup \textsf{get-offending-flows}\ (\textsf{getACS}\ M)\ (V,\ E_\tau \cup E_\sigma \cup \{(r,s)\}) \subseteq \{(r,s)\}\end{aligned} \tag{9.4}$$

It is obvious that (9.2) implies both (9.3) and (9.4).[8] The condition of (9.2) is imposed on all subsets, thus ruling out all possible undesired side effects.

However, translating (9.2) to executable code results in exponential runtime complexity, because it requires iterating over all subsets of $\overleftarrow{E_\sigma}$. This is infeasible for any large set of stateful flows. In this chapter, we contribute a new formula[9], which implies (9.2) and hence (9.3) and (9.4). It has a comparably low computational complexity and thus enables writing executable code for the automated verification of stateful and directed policies.

$$\bigcup \textsf{get-offending-flows}\ (\textsf{getACS}\ M)\ (\alpha\ T)\ \subseteq\ \overleftarrow{E_\sigma} \setminus E_\tau \tag{9.5}$$

Obviously, the runtime complexity of (9.5) is significantly lower than (9.2) (see Section 9.6). The formula also bears great resemblance to (9.3). We explain the intention of (9.5) and prove that it implies (9.2).

Note that $\overleftarrow{E_\sigma} \setminus E_\tau = \overleftarrow{\{(s,r) \in E_\sigma \mid (r,s) \notin E_\tau\}}$ [10], which means that it represents the backflows of all flows that are not already in E_τ. In other words, it represents only the newly added backflows. For example, consider the flows between students and employees in Figure 9.3b: no stateful flows are necessary as bidirectional flows are already allowed by the policy, and the newly added backflows are represented by the dashed edges. Therefore, (9.5) requires that all introduced violations are only due to the newly added backflows. This requirement is sufficient to imply (9.2).[11]

[8] stateful-policy-compliance.compliant-stateful-ACS-only-state-violations-union, stateful-policy-compliance.compliant-stateful-ACS-no-state-singleflow-side-effect

[9] stateful-policy-compliance.compliant-stateful-ACS

[10] backflows-filternew-flows-state

[11] stateful-policy-compliance.compliant-stateful-ACS-no-side-effects

Theorem 8 (Efficient ACS Compliance Criterion)**.** *For ACS, verifying that all introduced violations are only due to the newly added backflows is sufficient to verify the lack of side effects. Formally,* (9.5) $\Longrightarrow$ (9.2)*.*

Proof. We assume (9.5) and show (9.2) for an arbitrary but fixed $X \subseteq \overleftarrow{E_\sigma}$. We need to show that $\forall F \in$ get-offending-flows (getACS M) $(V,\ E_\tau \cup E_\sigma \cup X)$. $F \subseteq X$. We split $\overleftarrow{E_\sigma}$ into $\overleftarrow{E_\sigma} \setminus E_\tau$ and $\overleftarrow{E_\sigma} \setminus (\overleftarrow{E_\sigma} \setminus E_\tau)$. Likewise, we can split X into $X_1 \subseteq \overleftarrow{E_\sigma} \setminus E_\tau$ and $X_2 \subseteq \overleftarrow{E_\sigma} \setminus (\overleftarrow{E_\sigma} \setminus E_\tau)$. Hence, $X_2 \subseteq E_\tau$ and immediately $E_\tau \cup X_2 = E_\tau$. This simplifies the goal as X_2 disappears from the edges:

$$\forall F \in \textsf{get-offending-flows}\ (\textsf{getACS}\ M)\ (V,\ E_\tau \cup E_\sigma \cup X_1).\ F \subseteq X$$

We show an even stricter version of the goal since $X = X_1 \cup X_2$.

$$\forall F \in \textsf{get-offending-flows}\ (\textsf{getACS}\ M)\ (V,\ E_\tau \cup E_\sigma \cup X_1).\ F \subseteq X_1$$

This directly follows[12] by using Lemma 8 and subtracting $(\overleftarrow{E_\sigma} \setminus E_\tau) \setminus X_1$ from (9.5). □

9.5 Automated Stateful Policy Construction

In this section, we present algorithms to calculate a stateful implementation of a directed policy for a given set of security invariants using (9.1) and (9.5).

Instead of a set, the algorithms' last parameter is a list because the order of the elements matters. We use list notation as described in Chapter 3. Since lists can be easily converted to finite sets, we make this conversion implicit for brevity. For example, for a list a, we will write the stateful policy as $(V,\ E,\ a)$, where a is implicitly converted to a finite set.

9.5.1 Information Flow Security Strategies

We start by presenting an algorithm which selects stateful edges in accordance to the IFS security invariants. The algorithm filters a given list of edges for edges which fulfill (9.1). It also takes as input the directed policy G, the security invariants M, and a list of edges as accumulator a.

```
filterIFS $G$ $M$ $a$ []            = $a$
filterIFS $G$ $M$ $a$ $(e :: es)$ = if $\forall m \in$ getIFS $M.\ m\ (\alpha\ (V, E, e :: a))$ then
                                  filterIFS $G$ $M$ $(e :: a)$ $es$
                              else
                                  filterIFS $G$ $M$ $a$ $es$
```

The accumulator, initially empty, returns the result in the end. It is the current set of selected stateful flows. The algorithm is designed such that (9.1) always holds for $T = (V, E, a)$. It simply iterates over all elements e of the input list and checks whether the formula also holds if e is added to a. If so, e is added to the accumulator; otherwise, a is left unchanged.

Depending on the security invariants, multiple results are possible with this filtering criterion. The algorithm deterministically returns one solution. Users can influence the

[12] stateful-policy-compliance.compliant-stateful-ACS-no-side-effects-filternew-helper

choice of edges that they want to be stateful by arranging the input list such that the preferred edges are listed first. If only one arbitrary solution is desired, lists and finite sets are interchangeable.

The algorithm is sound[13] and complete.[14]

Lemma 9 (`filterIFS` Soundness)**.** *If the directed policy $G = (V, E)$ is valid, then for any list $X \subseteq E$, the stateful policy $T = (V, E, \mathsf{filterIFS}\ G\ M\ \texttt{[]}\ X)$ fulfills (9.1).*

Lemma 10 (`filterIFS` Completeness)**.** *For $G = (V, E)$, let $E_\sigma = \mathsf{filterIFS}\ G\ M\ E$. Then, no non-empty subset can be added to E_σ without violating (9.1).*

$$\forall X \subseteq E \setminus E_\sigma,\ X \neq \emptyset.\ \neg\forall m \in \mathsf{getIFS}\ M\ \ (\alpha\ (V, E, E_\sigma \cup X))$$

9.5.2 Access Control Strategies

The algorithm `filterACS` follows the same principles as `filterIFS`.

```
$\mathsf{filterACS}\ G\ M\ a\ \texttt{[]} \quad = \ a$
$\mathsf{filterACS}\ G\ M\ a\ (e :: es) =$
    $\textbf{if}$
        $e \notin \overleftarrow{E}\ \wedge\ (\forall F \in \mathsf{get\text{-}offending\text{-}flows}\ (\mathsf{getACS}\ M)\ (\alpha\ (V, E, e :: a)).\ F \subseteq \overleftarrow{e :: a})$
    $\textbf{then}$
        $\mathsf{filterACS}\ G\ M\ (e :: a)\ es$
    $\textbf{else}$
        $\mathsf{filterACS}\ G\ M\ a\ es$
```

As previously, the order of the elements in the list influences the choice of calculated stateful edges. Edges listed first are preferred. The algorithm is sound[15] and complete.[16]

Lemma 11 (filterACS Soundness)**.** *If the directed policy $G = (V, E)$ is valid, then for any list $X \subseteq E$, the stateful policy $T = (V, E, \mathsf{filterACS}\ G\ M\ \texttt{[]}\ X)$ fulfills (9.5).*

To show that filterACS computes a maximal solution, we must first identify the candidates that filterACS might overlook. Flows that are already bidirectional need not be stateful. As illustrated in the example of Figure 9.3b, no added value is created if stateful connections between students and employees were allowed as no communication restrictions exist between these groups in the first place. Hence only $E \setminus \overleftarrow{E}$ is considered.

Lemma 12 (filterACS Completeness)**.** *For $G = (V, E)$, let $E_\sigma = \mathsf{filterACS}\ G\ M\ E$. Then, no non-empty subsets $X \subseteq E \setminus (E_\sigma \cup \overleftarrow{E})$ can be added to E_σ without violating (9.5).*

$$\begin{aligned}&\forall X \subseteq E \setminus (E_\sigma \cup \overleftarrow{E}),\ X \neq \emptyset.\\ &\qquad \neg\left(\bigcup \mathsf{get\text{-}offending\text{-}flows}\ (\mathsf{getACS}\ M)\ (\alpha\ (V, E, E_\sigma \cup X))\ \subseteq\ \overleftarrow{E_\sigma \cup X} \setminus E\right)\end{aligned}$$

[13]filter-IFS-no-violations-correct
[14]filter-IFS-no-violations-maximal-allsubsets
[15]filter-compliant-stateful-ACS-correct
[16]filter-compliant-stateful-ACS-maximal-allsubsets

9.5.3 IFS and ACS Combined

Finally, we combine the previous section's algorithms to derive algorithms which compute a solution that satisfies all requirements of a stateful policy.

The first algorithm[17] simply chains filterIFS and filterACS.

$$\mathsf{generate1}\ G\ M\ e = (V,\ E,\ \mathsf{filterACS}\ G\ M\ (\mathsf{filterIFS}\ G\ M\ e))$$

The second algorithm[18] takes the intersection of filterIFS and filterACS.

$$\mathsf{generate2}\ G\ M\ e = (V,\ E,\ (\mathsf{filterACS}\ G\ M\ e) \cap (\mathsf{filterIFS}\ G\ M\ e))$$

Both algorithms are sound.[19] It remains unclear whether both are equal in the general case. Furthermore, it is difficult to prove (or disprove) their completeness, because both algorithms work on almost arbitrary functions M. However, we have formal proofs for the completeness of filterACS and filterIFS and the structure of generate1 and generate2 suggest completeness. In our experiments, generate1 and generate2 always calculated the same maximal solution, at least for Φ-structured invariants.

Theorem 9 (generate$\{1, 2\}$ Soundness)**.** *The algorithms* generate1 *and* generate2 *calculate a stateful policy that fulfills both IFS and ACS requirements.*

Example. Recall our running example. We illustrate how Figure 9.3b can be calculated from Figure 9.3a and the security invariants. All ACS invariants impose only local—in contrast to transitive—access restrictions. Therefore, the ACS invariants lack side effects and `filterACS` selects all flows (excluding already bidirectional ones). The invariant that the file server stores confidential data also introduces no restrictions: Both *filesSrv* → *employees* and *employees* → *filesSrv* are allowed and since the employees are trusted, they can further distribute the data. Therefore, filterIFS applied on only this invariant correctly selects all flows. Up to this point, the network's functionality is maximized. However, since the printer is classified as information sink, it must not leak any data. Therefore, filterIFS applied to this invariant selects all but the flows to the printer. Ultimately, both generate algorithms compute[20] the same maximal stateful policy, illustrated in Figure 9.3b. The soundness and completeness of the running example is hence formally proven. The case study in Section 9.8 will focus on performance and feasibility in a large real-world example.

9.6 Computational Complexity

The computational complexity of all presented formulae depends on the computational complexity of the security invariants $m \in M$. As we allow almost any function m as security invariant, the computational complexity can be arbitrarily large. However, most of the security invariants we use in our daily business check a property over all flows in

[17] generate-valid-stateful-policy-IFSACS

[18] generate-valid-stateful-policy-IFSACS-2

[19] generate-valid-stateful-policy-IFSACS-stateful-policy-compliance, generate-valid-stateful-policy-IFSACS-2-stateful-policy-compliance

[20] Impl_List_Playground_ChairNetwork_statefulpolicy_example.thy

the network (Φ-structured). Thus, the computational complexity of m is linear in the number of edges, i.e., $\mathcal{O}(|E|)$. The trivial computational complexity of set-offending-flows is in $\mathcal{O}(2^{|E|} \cdot |E|^2)$, since it iterates over all subsets of E. However, given the Φ-structure of the security invariants we primarily use, we showed[21] that the offending flows for our security invariants are uniquely defined. They can be computed in $\mathcal{O}(|E|)$. The result is a singleton set whose inner set size is also in $\mathcal{O}(|E|)$. We present the computational complexity of our formulae and algorithms in this section for security invariants and offending flows with the mentioned complexity.[22] Our solution is not limited to these security invariants, but the computational complexity increases for more expensive security invariants.

We assume that set inclusion can be computed with the hedge union algorithm [133] in $\mathcal{O}(k_i + k_j)$ for sets of size $k_{\{i,j\}}$. Since E_τ and E_σ are bounded by E, set inclusion is in $\mathcal{O}(|E|)$.

Verifying information flow compliance, i.e., (9.1), can be computed in $\mathcal{O}(|E| \cdot |M|)$. Hence, for a constant number of security invariants, the computational complexity is linear in the number of policy rules.

To verify access control compliance, we first note that (9.2) is in $\mathcal{O}(2^{|E|} \cdot |E| \cdot |M|)$ which is infeasible for a large policy. However, we provide (9.5), which implies (9.2), and can be computed in $\mathcal{O}(|E| \cdot |M|)$. Hence, for a constant number of security invariants, the computational complexity is linear in the number of policy rules.

The filter and generate$\{1, 2\}$ algorithms only add $\mathcal{O}(|E|)$ to the complexity. Hence, for a constant number of security invariants, computing a stateful policy implementation from a directed policy is quadratic in the number of policy rules, which is feasible even for large policies with thousands of rules.

9.7 Case Study Cabin Network Revisited

The security invariants and policy for a cabin data network for the general civil aviation have been presented and analyzed in Section 7.2. The policy and the host attributes have been visualized in Figure 7.3. In this section, we compute the stateful flows for the presented scenario. The stateful flows are visualized as dotted arrows in Figure 9.4.

In the original policy, most of the allowed flows are already bi-directional. The only unidirectional flows are from the cabin core server to the in-flight entertainment system server and the IFE server and the WiFi accessing the SAT uplink to the Internet. The figure shows that those flows may also be stateful. The reason is as follows:

The security invariants only specify one IFS invariant: The Bell-LaPadula (with trust) invariant. The IFEsrv has a lower security level than the cabin core server, so it can send answers to it without an IFS violation. Note that the IFEsrv is trusted, which was required in the original (non-stateful) policy because it needed to receive data from the CC and further distribute it. Answers from the IFEsrv to the CC cause a violation of the ACS Domain Hierarchy invariant, but there is no further negative side effect. The Policy Enforcement Point invariant is not violated by these answers. Consequently, the IFEsrv may send answers to the CC.

[21] BLP-offending-set, CommunicationPartners-offending-set, ...

[22] The computational complexity results are not formalized in Isabelle/HOL, because in its present state, there is no support for reasoning about asymptotic runtime behavior.

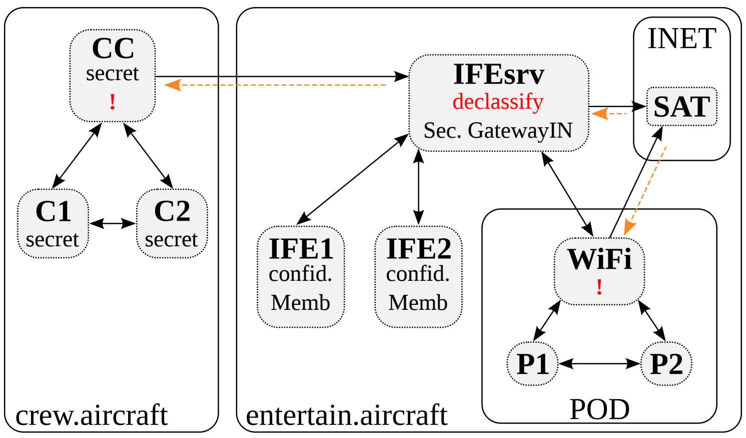


Figure 9.4: Cabin network policy, host attributes, and stateful flows.

For the same reasons, the SAT uplink may send answers to the IFEsrv and the WiFi. All devices have the same security level (though IFEsrv is additionally trusted, it has the lowest security level). The Policy Enforcement Point invariant is not affected by such answers. Only locally-contained violations of the ACS Domain Hierarchy occur. This corresponds exactly to the formalized requirement of accepting expected answers if the connection to the Internet was initialized by the device itself.

The stateful policy shows that the case study can now be implemented in a fully functional network.

9.8 Case Study TUM i8 Firewall

In a study, Wool [7] analyzed 37 firewall rulesets from telecommunications, financial, energy, media, automotive, and many other kinds of organization, collected in 2000 and 2001. The maximum observed ruleset size was 2671, and the average ruleset size was 144. Wool's study "indicates that there are no good high-complexity rulesets" [7]. If in a scenario complicated rulesets are unavoidable, formal verification to assert their correctness is advisable.

In this section, we analyze the firewall ruleset of TUM's Chair of Network Architectures and Services. With approximately 2983 rules as of November 2013, this firewall configuration can be considered representatively large. Almost all rules are stateful, hence the firewall generally allows all established connections and only controls who is allowed to initiate a connection. We publish our complete data set, allowing others to reproduce our results and reuse the raw data for their research.

As there is no written formal security policy for our network, we reverse-engineered the security policy and invariants with the help of our system administrator. The firewall contains rules per IP range that permit the services which are accessible from some IP range. Most rules are similar to rule one in Figure 9.1. We regard the firewall rules about which hosts can initiate a connection as security policy. It is not unusual that the implementation is also the documentation [134, §1]. We verify that the so derived security policy, i.e., which hosts can initiate connections, corresponds to the stateful implementation, i.e., all connections are stateful.

In order to prepare the firewall rules as graph, we used *ITval* [135] to first partition the IP space into classes with equivalent access rights [136] which form the nodes of our policy.

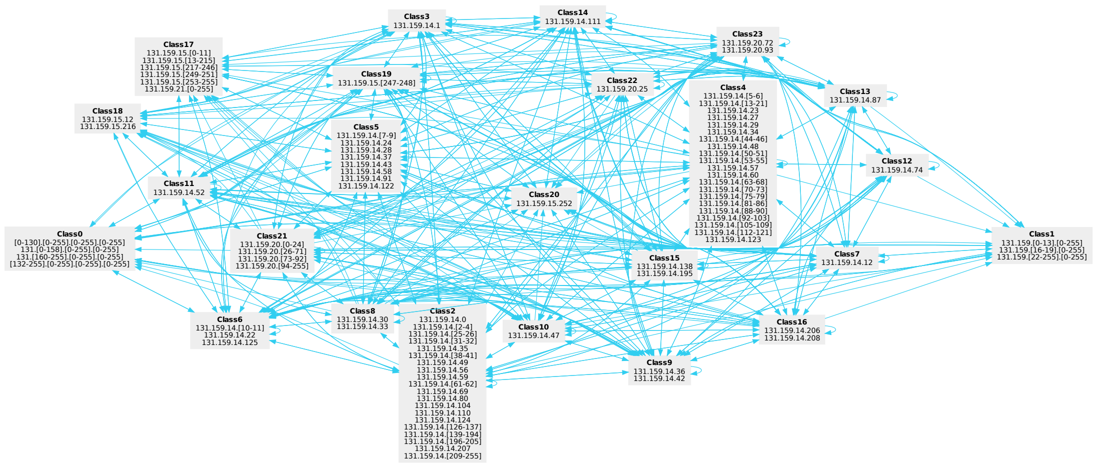


Figure 9.5: SSH landscape of the TUM Chair of Network Architectures and Services

For each of these classes, we selected representatives and queried ITval for “which hosts can this representative connect to” and “which hosts can connect to this representative”. This method is also suggested by Marmorstein [136]. The resulting IP ranges were mapped back to the classes. This generates the edges of the security policy graph.[23] We asserted that these two queries result in the same graph. For brevity, we restrict our attention to the SSH landscape, i.e., TCP port 22. The full data set is publicly available. The SSH landscape results in a security policy with 24 nodes (sets of IP ranges with equal access rights) and 496 edges (permissions to establish SSH connections). The resulting graph is shown in Figure 9.5.

A detailed discussion with our system administrator indicated that the graphical representation of the computed graph contains helpful information. It reveals that the computed policy does not exactly correspond to the firewall’s configuration. At first, we could not clearly identify the cause for this discrepancy. In the following years, when writing our own, fully-verified tool with similar goals, we figured out that ITval contains several bugs which lead to erroneous results; the details will be presented in Part II of this thesis. However, this graph obtained by ITval provides a sufficient approximation of our security policy. Here, we will only use it for a performance evaluation, therefore, its correctness is not crucial but only its size. In fact, the actual, true graph is significantly smaller than the one we use in this chapter. For future work, we planned to generate the graph using the approach by Tongaonkar, Niranjan, and Sekar [137], which, later on, we unfortunately could not reproduce because no code is publicly available. In the long term, we see the need for formally verified means of translating network device configurations, such as firewall rulesets, SDN flow tables, routing tables, and vendor specific access control lists to formally accessible objects, such as graphs. Part II of this thesis will present a tool for this.

After having constructed the security policy, we implemented our security invariants. They state that our IP ranges form a big set of mostly collaborating hosts. As a general rule, internal hosts are protected from accesses from the outside world, but there are many exceptions.

[23]We will reuse this idea and elaborate on it in Chapter 14.

No IFS invariants exist and our ACS invariants cause no side effects. Note that we are evaluating neither the quality of our security policy nor the quality of our security invariants, but the quality of the stateful implementation in this large real-world scenario. As expected, our `generate{1,2}` algorithms identify all unidirectional flows as upgradable to stateful. This shows that the standard practice to declare (almost) all rules as stateful, combined with common simple invariants does not introduce security issues. For our invariants, our algorithms always generate a graph T such that $\alpha\ T = (V,\ E \cup \overleftarrow{E})$. This means that in this scenario, we have a formal justification that all directed policy rules correspond to their stateful implementation, without any security concern. This maximizes the network's functionality without introducing security risks and is thus the optimal solution.

This statement can be generalized to all networks without IFS invariants and without side effects in the ACS invariants. We provide formal proofs for both generate$\{1, 2\}$ algorithms.[24] Due to its simplicity, universality, and convenient implications for everyday use, we state this result explicitly.

Corollary 1. *If there are no information flow security invariants and all access control invariants of a directed policy lack side effects, a security policy can be smoothly implemented as stateful policy, without any security issues concerning state.*

Our algorithms return this result, i.e., α (generate $G\ M\ E) = (V,\ E \cup \overleftarrow{E})$. If there are information flow security invariants or access control invariants with side effects, our algorithms also handle these problems.

All results can be computed interactively on today's standard hardware. The graph preparation, which needs to be done only once, takes several seconds. Our `generate` algorithms take a few seconds. This shows the practical low computational complexity for a large real-world study.

9.9 Related Work

In the research field of firewalls, several successful approaches to ease management [33] and uncovering errors [13] exist. Pozo et al. [134] propose that a network security policy should exist in an informal language. A translation from the informal language to a formalized policy with an information content comparable to the directed policy in this work must be present. The same model for firewall rules and security policy is used. The authors model services, i.e., ports, explicitly but ignore the direction of packets in their firewall model and are hence vulnerable to several attacks, such as spoofing [138]. Constraint Satisfaction Problem (CSP) solving techniques are used to test compliance of the security policy and the firewall ruleset. Using Logic Programming with Priorities (LPP), Bandara et al. [139] build a framework to detect firewall anomalies and generate anomaly-free firewall configurations from a security policy. The authors explicitly point out the need for solving the stateful firewall problem.

Brucker et al. [131, 140] provide a formalization of simple firewall policies in Isabelle/HOL and rewrite rules to simplify them. With this, they introduce HOL-TestGen/FW, a tool to generate test cases for conformance testing of a firewall ruleset, i.e., that the firewall

[24]generate-valid-stateful-policy-IFSACS-noIFS-noACSsideeffects-imp-fullgraph, generate-valid-stateful-policy-IFSACS-2-noIFS-noACSsideeffects-imp-fullgraph

under test implements its ruleset correctly. The authors augment their work [141] with user-friendly high-level policies. This also allows the verification of a network specification with regard to these high-level policies.

Guttman et al. [6, 142] focus on distributed network security mechanisms, such as firewalls, filtering routers, and IPsec gateways. Security goals centered on the path of a packet through the network can be verified against the distributed network security mechanisms configuration.

Using formal methods, network vulnerability analysis reasons about complete networks, including the services and client software running in the network. Using model checking [143] or logic programming [114], network vulnerabilities can be discovered or the absence of vulnerabilities can be shown. One potential drawback of these methods is that the set of vulnerabilities must be known for the analysis, which can be an advantage for postmortem network intrusion analysis, but is also a downside when trying to estimate a network's future vulnerability.

Kazemian et al. [119] present a method for the packet forwarding plane to identify problems such as reachability issues, forwarding loops, and traffic leakage. Considering the individual packet bits, the header space is represented by a ⟨*maximum packet header size in bits*⟩-dimensional space. An efficient algebra on the header space is provided which enables checking of the named use cases.

9.10 Conclusion

Stateful firewall rules are commonly used to enforce network security policies. Due to these state-based rules, flows opposite to the security policy rules might be allowed. On the one hand, we argued that under presence of side effects or information flow invariants, a naive stateful implementation might break security invariants. On the other hand, declaring certain firewall rules to be stateless might impair the functionality of the network. This problem domain has often been overlooked in previous work.

Verifying that a stateful firewall ruleset is compliant with the security policy and its invariants is computationally expensive. In this work, we discovered a linear-time method and contribute algorithms for verifying and also for computing stateful rulesets. We demonstrated that these algorithms are fast enough for reasonably large networks, while provably maintaining soundness and completeness.

Chapter 10

Demonstrating *topoS*: Theorem-Prover-Based Synthesis of Secure Network Configurations

This chapter is an extended version of the following paper [144]:

- Cornelius Diekmann, Andreas Korsten, and Georg Carle. *Demonstrating topoS: Theorem-Prover-Based Synthesis of Secure Network Configurations*. In 2nd International Workshop on Management of SDN and NFV Systems, manSDN/NFV, Barcelona, Spain, November 2015.

The following improvements were added:

- The iptables implementation of our case study has been verified with *fffuu* (cf. Part II).
- Support for microservice architectures built on top of docker has been added.
- The related work section was updated and extended.

Statement on author's contributions For the original paper, the author of this thesis provided major contributions for the ideas, realization, implementation, and proof of the translation process and the *topoS* tool. He researched related work, and conducted the evaluation. Andreas Korsten contributed to the OpenFlow implementation and deployment. A prototypical, incomplete demonstrator of the serialization to OpenVPN has been previously presented in the author's master's thesis [53]. This demonstrator did not consider state, nor information flow security, nor identify the necessary assumptions. For this thesis, we reused the idea of a central OpenVPN router but have re-implemented and re-evaluated the complete setup based on the new translation process. All improvements listed above are the work of the author of this thesis.

Abstract We combine the results of the previous chapters to present the big picture of the translation from high-level security goals to low-level configurations of security mechanism. All results of this Part are combined and we present our tool *topoS* which automatically synthesizes low-level network configurations from high-level security goals. The automation

and a feedback loop help to prevent human errors. Except for a last serialization step, *topoS* is formally verified with Isabelle/HOL, which prevents implementation errors. In a case study, we demonstrate *topoS* by example. The complete transition from high-level security goals to firewall, SDN, and docker configurations is presented.

10.1 Introduction

Network-level access control is a fundamental security mechanism in almost every network. Unfortunately, configuring network-level access control devices still is a challenging, manual, and thus error-prone task [12, 13, 14]. It is a known and unsolved problem for over a decade that "corporate firewalls are often enforcing poorly written rule sets" [7]. Also, "access list conflicts dominate the misconfiguration errors made by administrators" [9]. A recent study confirms that this problem persists as a "majority of administrators stated misconfiguration as the most common cause of failure" [10]. In addition, not only is implementing a policy error-prone, but also developing it is challenging, even for experienced administrators [11].

We demonstrate our tool *topoS*: a constructive, top-down greenfield approach for network security management. *topoS* translates high-level security goals to network security device configurations. The automatic translation steps prevent manual translation errors. Furthermore, *topoS* visualizes the results of all translation steps to help the administrator uncover specification errors. In addition, since all intermediate transformation steps are formally verified, the correctness of *topoS* itself is guaranteed [38]. *topoS* is built on top of the results of this thesis, previously presented to the formal methods community [11, 129], combines these results in a novel way, and transfers the knowledge to the network management community. The automated tool *topoS* is the main technical contribution of this chapter.

We first give a short overview of *topoS* in Section 10.2. Then, in Section 10.3, we present *topoS* in detail with the help of a case study. We discuss limitations and advantages in Section 10.4, present related work in Section 10.5, and conclude in Section 10.6.

10.2 Overview of *topoS*

The security requirements of networks are usually scenario-specific. Our tool *topoS* helps to configure a network according to these needs. It takes as input the high-level security requirements and synthesizes low-level configurations for security device, e.g., netfilter/iptables firewall rules or OpenFlow flow table entries. It operates according to the following four-step process:

A. Formalize high-level security goals
   (a) Categorize security goals
   (b) Add scenario-specific knowledge
   (c) ⋆ Auto-complete information

B. ⋆ Construct security policy

C. ⋆ Construct stateful policy

D. ⋆ Serialize security device configurations

All steps annotated with an asterisk are supported by *topoS*. As the ⋆-steps illustrate, once the security goals are specified, the process is completely automatic. Between the automated steps, manual refinement is possible but requires re-verification. This allows human intervention, while avoiding human error.

We will illustrate the process in a case study: Section 10.3.1 presents the formalization of the security goals, illustrated in Figure 10.2. Figure 10.3 corresponds to the derived security policy. In Figure 10.4, the administrator made some manual changes (which were accepted by the system since the changes do not violate the formalized security goals). Finally, Figure 10.5 corresponds to the stateful policy. The resulting security device configuration will be illustrated for different devices, e. g., Figure 10.6, Figure 10.9 and Figure 10.10.

The automated intermediate ⋆-steps are proven correct for all inputs. The proofs are verified with Isabelle/HOL [37]. Thus, it is guaranteed that *topoS* performs correct transformations [38]. As a side note, since the transformations are proven correct once and for all for all inputs, neither has a user to prove anything manually to use *topoS*, nor is Isabelle/HOL required to run *topoS*.

We did not verify the final step (i. e., serialization of security device configurations) in general since it is merely syntactic rewriting of the result of the previous step (cf. Section 10.3.4). In addition, since our policy only allows positive rules, it is guaranteed to be without any conflicts. Nevertheless, we verified the correctness of the iptables implementation of our case study with *fffuu* (cf. PartII).

We will present the steps A to D in the following section. For the sake of brevity, we only present them by example. Mathematical background has been presented in detail in the previous chapters. In this chapter, we focus on its interoperability and discuss how the underlying assumptions can be fulfilled in a real-world network. Further details, the correctness proofs, and the interplay of the individual steps can be found in the accompanying formalization and implementation of *topoS*.

10.3 *topoS* by Example

In this section, we demonstrate *topoS* with a small case study. The scenario was chosen because it is minimal and comprehensible, but also realistic and contains many important aspects. It runs live and is publicly available.[1]

The case study is schematically illustrated in Figure 10.1. The setup hosts a news aggregation web application, accessible from the Internet (*INET*). It consists of a web application backend server (*WebApp*) and a frontend server (*WebFrnt*). The *WebApp* is connected to a database (*DB*) and actively retrieves data from the Internet. All servers send their logging data to a central, protected log server (*Log*).

We implemented the scenario to utilize several different protocols. A custom backend, the *WebApp* was written in `python`. The *WebFrnt* runs `lighttpd`. It serves static web pages directly and retrieves dynamic websites from the *WebApp* via `FastCGI`. All components send their `syslog` messages via UDP (RFC 5426 [100]) to *Log*.

[1] `http://otoro.net.in.tum.de/goals2config/` and `http://amygdala.ip4.net.in.tum.de/fcgi/`. The configurations are also archived at `https://github.com/diekmann/topoS/blob/master/thy/Network_Security_Policy_Verification/Examples/Distributed_WebApp.SDN_deployed.txt`

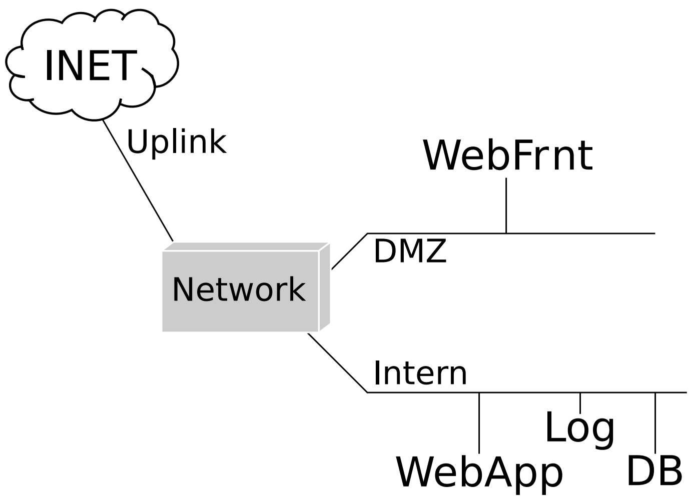


Figure 10.1: Network Schematic

10.3.1 Formalizing Security Goals

The security goals are expressed as security invariants over the network's connectivity structure. An invariant consists of a generic part (the semantics, formalized as security invariant template) and scenario-specific information (formalized as host attributes). The generic part defines the type and general meaning. Our generic invariants currently defined are summarized in Table 6.1.

To construct a scenario-specific invariant, a generic invariant is instantiated with scenario-specific knowledge. This is done by specifying host attributes (cf. Chapter 5). These invariants and the list of entities (*INET*, *WebApp*, *WebFrnt*, *DB*, *Log*) is the only input needed. For this scenario, the following four invariants are expressed, formalized in Figure 10.2.

1. First, as illustrated in Figure 10.1, *DB*, *Log*, and *WebApp* are considered internal hosts. We use the *SubnetsInGW* invariant template (Section 6.14) for this. Internal hosts are labeled with the Member attribute. The *WebFrnt* must be accessible from outside, it is a classical DMZ member. Therefore, we label it as InboundGateway.

2. Next, it is expressed that the logging data must not leave the log server. Therefore, using the *Sink* invariant (Section 6.12), *Log* is classified as information sink.

3. Using the *Bell-LaPadula* invariant (Section 6.3), it is specified that *DB* contains confidential information. Since it sends its log data to the log server, this log server is also assigned the security level confidential. Finally, the *WebApp* is allowed to retrieve data from the *DB* and to publish it to the *WebFrnt*. Therefore, the *WebApp* is trusted and allowed to declassify the data.

4. Finally, an access control list specifies that only *WebApp* may access the *DB*. We use the *Communication Partners* template from Section 6.4.

In this example, several hosts do not have attributes assigned for all invariants. It is sufficient to supply an incomplete host attribute specification, since they are automatically and securely completed by *topoS*. Chapter 5 discusses the details.[2] Once the invariants are

[2]The security of the auto-completion is guaranteed w.r.t. the provided information, i.e., the auto-completion can never lead to an unnoticed security problem, given enough information is provided. For

$$
\begin{aligned}
&\text{SubnetsInGW } \{\mathit{DB} \mapsto \textsf{Member},\\
&\qquad \mathit{Log} \mapsto \textsf{Member},\\
&\qquad \mathit{WebApp} \mapsto \textsf{Member},\\
&\qquad \mathit{WebFrnt} \mapsto \textsf{InboundGateway}\}\\
&\text{Sink } \{\mathit{Log} \mapsto \textsf{Sink}\}\\
&\text{Bell-LaPadula } \{\mathit{DB} \mapsto \textsf{confidential},\\
&\qquad \mathit{Log} \mapsto \textsf{confidential},\\
&\qquad \mathit{WebApp} \mapsto \textsf{declassify (trusted)}\}\\
&\text{Comm. Partners } \{\mathit{DB} \mapsto \textsf{Master} : \mathit{WebApp},\\
&\qquad \mathit{WebApp} \mapsto \textsf{Care}\}
\end{aligned}
$$

Figure 10.2: Security Invariants (Case Study)

specified, their management scales well in the face of changes: When a new host is added to the network, issues are handled by the auto-completion: either, the new host causes a violation, which is consequently uncovered, or it can be added without any further changes. Invariants are composable and modular by design, helping structured representation and archiving of knowledge. In the worst case, inconsistent security invariants may be specified accidentally. This only results in an overly strict security policy being computed, which can be identified in the following step.

It has been shown that a special class of invariants, called Φ-structured, exhibits several nice mathematical properties (cf. Table 6.1). A Φ-structured invariant asserts a predicate for every policy rule. This predicate must only depend on the sender, receiver, and their host attributes. In particular, these invariants and their derived algorithms are very efficiently computable. It is also due to the Φ-structured invariants that a maximum-permissive security policy is uniquely defined.

10.3.2 Constructing the Security Policy

A network's end-to-end connectivity structure, i.e., a global access control matrix, corresponds to the *security policy*. Here, we utilize the textbook definition that a policy consists of the *rules* which ensure that the network is in a secure state. In contrast, the security goals are expressed as *invariants* over the policy and reside on a higher level of abstraction.

Graphically, a policy can be illustrated as a directed graph. The policy of the case study, illustrated in Figure 10.3, was automatically computed from the security invariants.

The algorithm to transform a set of security invariants into a policy starts with the allow-all policy and iteratively removes undesired rules. This is always possible if (and only if, Theorem 1) the invariants hold for the deny-all policy; a static requirement which is only to be proven once for a generic invariant. The algorithm is sound. It is also complete for the invariants utilized in this example (and for Φ-structured invariants in general, Theorem 4).

example, information-leakage is always uncovered, given all confidential data sources are specified. However, if an administrator forgets to label a confidential data source, information leakage can occur. It is trivially possible to design explicit whitelisting invariants which auto-complete to some '*deny*' property. On the downside, this requires lots of manual configuration effort, which is avoided by the invariants utilized in this chapter. Roughly speaking the auto-completion fulfills: "*the more information provided, the more secure the whole system*".

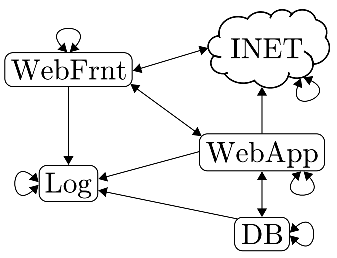


Figure 10.3: Security Policy (computed)

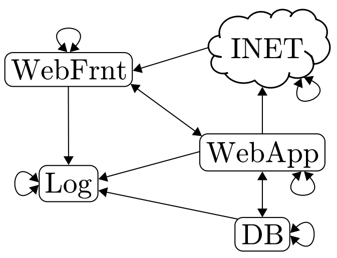


Figure 10.4: Security Policy (manually refined)

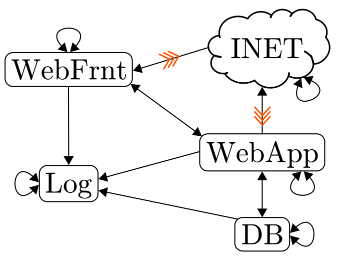


Figure 10.5: Stateful Policy (computed)

In our example, the administrator decides to manually refine the policy: there is no need for the web frontend to connect to the Internet. Therefore, this flow is prohibited. After this manual refinement, the security invariants are re-verified. This refined policy is shown in Figure 10.4.

10.3.3 Constructing the Stateful Policy

The derived policy may appear adequate from a theoretical point of view but has one major problem when it comes to implementation: The *WebApp* can connect to the Internet, but the policy does not specify whether the Internet may answer this request (same for the *WebFrnt* after manual refinement). Obviously, for this scenario, answers should be permitted; otherwise, no one would be able to use the service. In contrast, the Log server uses the `syslog` protocol over UDP (RFC 5426 [100]). This protocol uses a unidirectional UDP channel and it is explicitly specified for security reasons that this is the only way the communication with the log server is permitted.

Therefore, it must be distinguished between stateful and purely unidirectional rules. We extend the security policy to additionally specify whether a flow might be stateful (i.e., answers to requests are allowed). Note that a flow with the stateful attribute might allow packets in the opposite direction of the policy rule and thus potentially violate security invariants. Defining the following two consistency criteria, the stateful attributes can be computed automatically (cf. Chapter 9):

1. No information flow violation must occur.
2. No access control side effects must be introduced.

To compute the stateful policy, not only a single rule but a set S is to be upgraded to stateful rules. However, the interaction of the rules and answer paths of S must not introduce negative implications. Therefore, in particular to verify lack of side effects, all security policies derived from upgrading all *subsets* of S must be verified. A naive approach would require exponential complexity. We proved that this can be done more efficiently, particularly in linear time for Φ-structured invariants (Chapter 9). This insight provides an algorithm for computing the stateful policy from the security policy and the invariants. It is proven sound (Theorem 9) and complete w.r.t. the two criteria individually (Lemma 10, Lemma 12). Multiple solutions for a stateful policy may exist; a user may set preferences.

For the case study, this results in a policy where the Internet can set up connections to the web frontend, likewise, the web backend can set up connections to the Internet. However, the logging channels are purely unidirectional UDP (stateful connections would introduce an information flow violation). We will call this the stateful policy. It is illustrated in Figure 10.5.

10.3.4 Serializing Security Device Configurations

By now, the network of Figure 10.1 was considered a black box. In this section, the stateful policy is serialized to configurations for real network (security) devices. Though the serialization step is merely syntactic rewriting of the stateful policy, care must be taken to correctly transfer the semantics. Therefore, all assumptions of the stateful policy must be taken into account. We first define two types of entities and discuss the assumptions afterwards.

We differentiate between two types of entities.

Policy Entities Entities relevant for the use case and required for specification of functional requirements. E. g. all entities in Figure 10.3.

Network Infrastructure Entities Not required for the description of the high-level function of a system. E. g. Switches, routers, middleboxes. For example, everything in the network box in Figure 10.1.

One policy entity may correspond to several entities in the network. We will call them representatives. For example, when deploying the case study with load-balancing and redundancy, *WebApp* might be a set of backend servers; the symbolic name *WebApp* is translated to the IP range of these servers. Hence, a one-to-many mapping lifts policy identifiers to roles of network representatives.

To serialize security mechanism configurations, *topoS* must fulfill the following three assumptions.

Structure The enforced network connectivity structure must exactly coincide with the policy. This requires that the links are confidential and integrity protected.

Authenticity The policy's entities must match their network representation (e. g., IP/MAC addresses). In particular, no impersonation or spoofing must be possible.

State The stateful connection handling must match the stateful policy's semantics.

10.3.4.1 Reflexive policy rules

When it comes to implementation, we discuss one peculiarity: Reflexive policy rules. A reflexive rule, e. g., $A \to A$, means that the policy entity A can communicate with itself. As can be seen in the case study, all entities can communicate with themselves. Translating reflexive policy rules requires special care. For a one-to-one mapping of policy entities and network representatives, reflexive rules correspond to in-entity communication, which can obviously be ignored. For example, if A maps to 10.0.0.4, in-host communication is out of the scope of network access control. But a policy entity may also correspond to several network representatives. Such a scenario might occur if deployed with load balancing, e. g., *WebApp* may correspond a set of backend servers. In this case, enforcing network access control may be necessary. For example, if A maps to 10.0.0.0/24 and the policy does not permit $A \to A$, the hosts in the 10.0.0.0/24 subnet must not be able to access each other.

For the sake of brevity, we only present a one-to-one mapping between policy entities and their network representatives in this chapter. We present four different possibilities to implement the policy.

10.3.4.2 Central, Directly-Attached Firewall

We assume that all entities are directly connected to a central firewall via an individual cable. An attacker does not have access to the cables. The firewall has an individual interface for each entity and one interface as the uplink to the Internet.

```
-A FORWARD -i $WebFrnt_iface -s $WebFrnt_ipv4 -o $WebFrnt_iface -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i $WebFrnt_iface -s $WebFrnt_ipv4 -o $Log_iface -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $WebFrnt_iface -s $WebFrnt_ipv4 -o $WebApp_iface -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i $DB_iface -s $DB_ipv4 -o $DB_iface -d $DB_ipv4 -j ACCEPT
-A FORWARD -i $DB_iface -s $DB_ipv4 -o $Log_iface -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $DB_iface -s $DB_ipv4 -o $WebApp_iface -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i $Log_iface -s $Log_ipv4 -o $Log_iface -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $WebApp_iface -s $WebApp_ipv4 -o $WebFrnt_iface -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i $WebApp_iface -s $WebApp_ipv4 -o $DB_iface -d $DB_ipv4 -j ACCEPT
-A FORWARD -i $WebApp_iface -s $WebApp_ipv4 -o $Log_iface -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $WebApp_iface -s $WebApp_ipv4 -o $WebApp_iface -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i $WebApp_iface -s $WebApp_ipv4 -o $INET_iface -d $INET_ipv4 -j ACCEPT
-A FORWARD -i $INET_iface -s $INET_ipv4 -o $WebFrnt_iface -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i $INET_iface -s $INET_ipv4 -o $INET_iface -d $INET_ipv4 -j ACCEPT
-I FORWARD -m state --state ESTABLISHED -i $INET_iface -s $INET_ipv4                ↩
           -o $WebApp_iface -d $WebApp_ipv4 -j ACCEPT
-I FORWARD -m state --state ESTABLISHED -i $WebFrnt_iface -s $WebFrnt_ipv4          ↩
           -o $INET_iface -d $INET_ipv4 -j ACCEPT
-P FORWARD DROP
```

Figure 10.6: Central Firewall Rules (can be loaded with `iptables-restore`)

Table 10.1: Variable Assignment

Variable	Value
$*WebFrnt_iface*	`webfrnt`
$*WebFrnt_ipv4*	10.0.0.1
$*Log_iface*	`log`
$*Log_ipv4*	10.0.0.2
$*DB_iface*	`db`
$*DB_ipv4*	10.0.0.3
$*WebApp_iface*	`app`
$*WebApp_ipv4*	10.0.0.4
$*INET_iface*	`inet`
$*INET_ipv4*	0.0.0.0/5, 8.0.0.0/7, 11.0.0.0/8, 12.0.0.0/6, 16.0.0.0/4, 32.0.0.0/3, 64.0.0.0/2, 128.0.0.0/1

topoS generates the firewall template shown in Figure 10.6. To load the rules, we need to set the variables. In our shell, we `export` the interface names and IP addresses of our servers as given in Table 10.1. The value for $*INET_ipv4* deserves some explanation. It corresponds to the set of all IPv4 addresses excluding the 10.0.0.0/8 range. It was computed with our fully verified IP address library [39].

Technically, without additional modules, iptables only allows to match on one source or destination IP range in CIDR notation. However, the `iptables` userland command allows to specify several CIDR ranges as syntactic sugar, as we did for $\$INET\_ipv4$. Therefore, our configuration can be loaded without complaints. Internally, `iptables` translates these rules into several rules which match on one CIDR range at most. For this example, over 100 rules are internally created.

```
webfrnt = [10.0.0.1]
log = [10.0.0.2]
db = [10.0.0.3]
inet = all_but_those_ips [10.0.0.0/8]
app = [10.0.0.4]
```

Figure 10.7: Interface configuration for *fffuu*

To verify that the generated firewall rules provide spoofing protection and to validate our bold setting of $\$INET\_ipv4$, we specify the valid IP ranges per interface. Our tool *fffuu*, which we will use now to verify our configurations, will be presented in detail in Part 11 of this thesis. Figure 10.7 shows this specification with the syntax of our *fffuu* tool. With this, our tool can immediately verify spoofing protection (cf. Chapter 13) of the loaded iptables rules. This fulfills the required authenticity (✓) assumption.

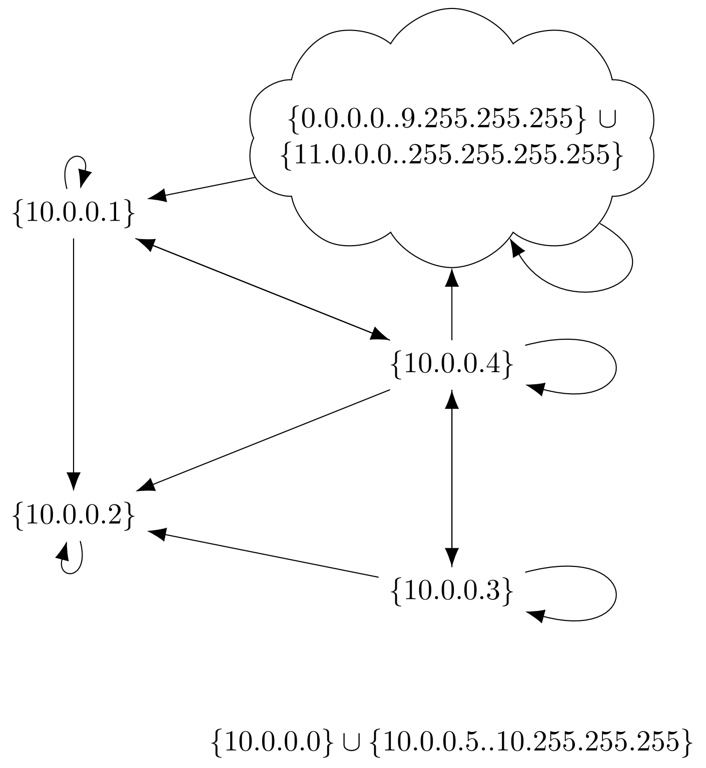


Figure 10.8: Reconstructed Security Policy (Structure of Figure 10.6)

We utilize our tool *fffuu* to compute a service matrix, i. e., given the iptables configuration, we reconstruct the security policy. The result is shown in Figure 10.8. We can see that the computed graph is identical to Figure 10.4. Hence, we have verified that our iptables configuration satisfies the structure (✓) assumption. Analogously to Section 7.3.5, the stateful semantics (✓) can be verified.

10.3.4.3 Firewall & Central VPN Server

All entities connect to a central OpenVPN server which enforces the policy. Entities are bound to their policy name with X.509 certificates. Every entity sets up a layer 3 (`tun`) VPN connection with the server. The server authenticates entities by their certificate and centrally assigns IP addresses. IP spoofing over the tunnel is prevented by the server. This provides authenticity (✓).

```
-A FORWARD -i tun0 -s $WebFrnt_ipv4 -o tun0 -d $Log_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebFrnt_ipv4 -o tun0 -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $DB_ipv4 -o tun0 -d $Log_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $DB_ipv4 -o tun0 -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o tun0 -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o tun0 -d $DB_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o tun0 -d $Log_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o eth0 -d $INET_ipv4 -j ACCEPT
-A FORWARD -i eth0 -s $INET_ipv4 -o tun0 -d $WebFrnt_ipv4 -j ACCEPT
-I FORWARD -m state --state ESTABLISHED                                  ↩
            -i eth0 -s $INET_ipv4 -o tun0 -d $WebApp_ipv4  -j ACCEPT
-I FORWARD -m state --state ESTABLISHED                                  ↩
            -i tun0 -s $WebFrnt_ipv4 -o eth0 -d $INET_ipv4  -j ACCEPT
-P FORWARD DROP
```

Figure 10.9: VPN Server Firewall Rules (can be loaded with `iptables-save`)

Firewalling is applied at the server; the stateful policy is directly translated to iptables rules, shown in Figure 10.9. With this, the stateful semantics (✓) and structure (✓) are enforced.

10.3.4.4 SDN

With complete control over the network, as is the case with data centers, a Software-Defined Network (SDN) may be used to implement the policy. Usually, a data center is a flat layer 2 network [145] and we need to contain layer 2 broadcasting and layer 2 attacks. For this, an entity's switch port[3] must be known.

We deploy all entities as virtual machines on top of the hypervisor xen [146]. All machines are executed on the same host. To interconnect the virtual machines, one Open vSwitch [147], deployed at the hypervisor, provides connectivity.

We install OpenFlow rules which prevent MAC, IP, and ARP spoofing. Figure 10.10 illustrates a template for generating a stateless rule from *src* to *dst*. The rules can be loaded with `ovs-vsctl set-fail-mode` $*switch* `secure && ovs-ofctl add-flows`. The first rule allows ARP requests. Note that we rewrite the layer 2 broadcast addresses directly to the immediate receiver's address. Rule two allows the ARP responses. Both rules ensure that only valid ARP queries and responses are sent and received in the network.[4] The third rule allows IPv4 traffic. For stateful rules, the opposite direction of Figure 10.10, i.e., *src*

[3]virtual hypervisor switch port or physical Top of Rack (ToR) switch port

[4]For the sake of simplicity, this implementation is designed such that it gets along without an SDN controller. This introduces a small hidden information flow channel (structure ✗): the ARP responses. For example in Figure 10.5, *Log* may use a timing channel or the ARP `OPER` field to exfiltrate information. However, the side-channel is easily removed when an SDN controller answers all ARP requests (structure ✓); all necessary information is present.

```
# ARP Request
  in_port=$port_src dl_src=$mac_src dl_dst=ff:ff:ff:ff:ff:ff                    ↩
  arp arp_sha=$mac_src arp_spa=$ip4_src arp_tpa=$ip4_dst                         ↩
  priority=40000 action=mod_dl_dst:$mac_dst,output:$port_dst

# ARP Reply
  dl_src=$mac_dst dl_dst=$mac_src                                                ↩
  arp arp_sha=$mac_dst arp_spa=$ip4_dst arp_tpa=$ip4_src                         ↩
  priority=40000 action=output:$port_src

# IPv4 one-way
  in_port=$port_src dl_src=$mac_src ip nw_src=$ip4_src nw_dst=$ip4_dst           ↩
  priority=40000 action=mod_dl_dst:$mac_dst,output:$port_dst

# if src (respecively dst) is INET,
# replace $ip4_src (respectively $ip4_dst) with * and decrease the priority
```

Figure 10.10: OpenFlow Flow Table Template

and *dst* swapped, is added. Any unmatched packets are dropped. With this set of rules, a mapping of policy identifiers to MAC and IP addresses is enforced. Also, correct address resolution is enforced (authenticity ✓). Without the ARP information leak, the desired connectivity structure (✓) is enforced. The setup does not provide stateful handling (✗) by default. However, a network firewall or SDN firewall app can provide the desired state (✓) handling. As of version 2.5.0, Open vSwitch also supports the `ct` action for connection tracking [148], providing stateful semantics (✓).

10.3.4.5 Microservices, Containers, and Docker

Microservices are on the rise [149]. To build lightweight, efficient, distributed applications, container technology [150] is commonly used. One popular software for container management and deployment is docker [151]. As of September 2016, over 40 thousand questions are tagged with 'docker' on stackoverflow [152].

Docker itself does not provide security out of the box [153, 154]. The security provided by containerization heavily depends on the specific docker and container setup. For example, if a container has the privilege to access raw sockets, the container can spoof arbitrary IP addresses or perform ARP spoofing [155]. Hence, in a generic case, docker does not provide authenticity (✗). Consequently, since a malicious, privileged container can perform ARP spoofing, no guarantees about the structure (✗) and, hence, the stateful semantics (✗) can be given.

Securing docker is out of the scope of this thesis. For this evaluation, we assume that docker is set up securely enough [153] such that the firewall rules which docker generates by default are effective. Hence, we assume that we can refer to the identity of a container by referring to its IP address (assumption: authenticity ✓). Our goal is to enforce the desired connectivity structure with the correct stateful handling.

This case study was built using Docker version 1.12.1, API version: 1.24. We want to deploy our application where each entity is realized as an individual container. For the sake of example, we use the Docker-provided `busybox` container images.[5] We want to use the

[5] `https://hub.docker.com/_/busybox/`

```
*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 10.0.0.0/8 ! -o br-b74b417b331f -j MASQUERADE
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A DOCKER -i br-b74b417b331f -j RETURN
-A DOCKER -i docker0 -j RETURN
COMMIT
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:DOCKER - [0:0]
:DOCKER-ISOLATION - [0:0]
-A FORWARD -j DOCKER-ISOLATION
-A FORWARD -o br-b74b417b331f -j DOCKER
-A FORWARD -o br-b74b417b331f -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i br-b74b417b331f ! -o br-b74b417b331f -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A FORWARD -i br-b74b417b331f -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i docker0 -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i br-b74b417b331f -o docker0 -j DROP
-A DOCKER-ISOLATION -j RETURN
COMMIT
```

Figure 10.11: Default `iptables` firewall used by docker with `mynet`

```
*filter
:INPUT ACCEPT [184:31271]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [179:18898]
:DOCKER - [0:0]
:DOCKER-ISOLATION - [0:0]
:MYNET - [0:0]
-A FORWARD -j DOCKER-ISOLATION
-A FORWARD -j MYNET
-A FORWARD -o br-b74b417b331f -j DOCKER
-A FORWARD -o br-b74b417b331f -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i br-b74b417b331f ! -o br-b74b417b331f -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A FORWARD -i br-b74b417b331f -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i docker0 -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i br-b74b417b331f -o docker0 -j DROP
-A DOCKER-ISOLATION -j RETURN
-A MYNET -m state --state ESTABLISHED                                              ↩
         ! -i br-b74b417b331f -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -m state --state ESTABLISHED                                              ↩
         -i br-b74b417b331f -s 10.0.0.1 ! -o br-b74b417b331f -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.3 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.2 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.3 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 ! -o br-b74b417b331f -j ACCEPT
-A MYNET ! -i br-b74b417b331f -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -j DROP
COMMIT
```

Figure 10.12: Adapted `iptables` firewall for docker (only `filter` table shown).

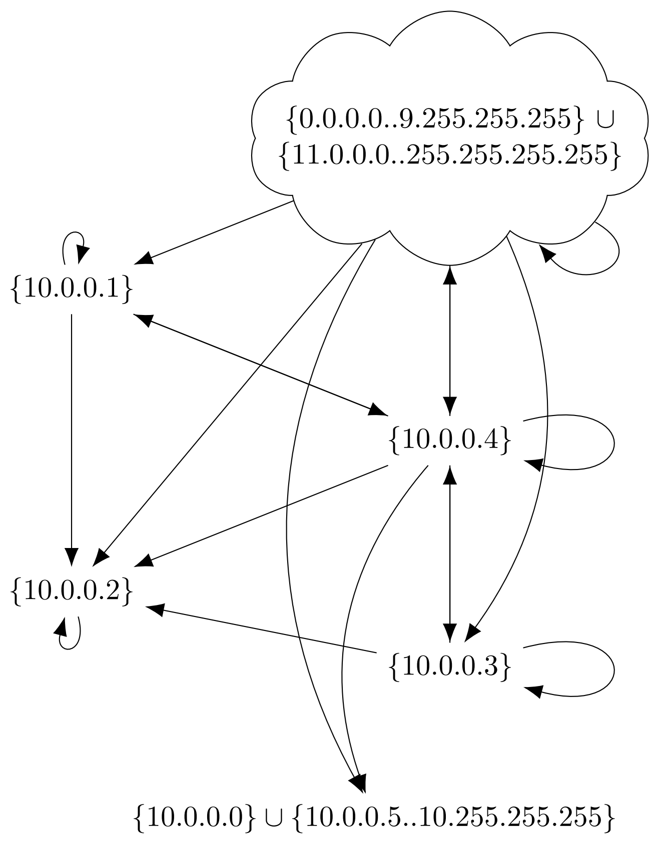


Figure 10.13: Reconstructed Security Policy (Structure of Figure 10.12)

same IPs as in Table 10.1.

The complete application (all containers) should be isolated from other containers on the host. Therefore, we create a new docker network for the desired IP range. We also disable inter-container communication (icc) for this network. The network is called `mynet` and created with the following command.

```
# docker network create -d bridge --subnet=10.0.0.0/8 --gateway=10.42.0.100          ↩
     --opt="com.docker.network.bridge.enable_icc=false" mynet
```

Containers which are started in this network cannot connect to each other but can reach the Internet. Docker generates a default firewall, which is shown in Figure 10.11. The interface `br-b74b417b331f` is the bridge interface which was generated for `mynet`. We now add our custom firewall filtering rules to this setting. The `nat` table remains unchanged. We can almost directly add the rules of Figure 10.6. The `DOCKER-ISOLATION` chain isolates the different docker networks from each other. We add our rules afterwards. The result is shown in Figure 10.12. For simplicity, we approximate the Internet-facing interface by stating that it is not our `mynet`-bridge interface.

The question about the correctness of this ruleset remains. We have empirically tested the connectivity of the deployed containers. Yet, this does not provide a formal guarantee of its correctness.

Similarly to Subsection 10.3.4.2, we utilize our tool *fffuu* to compute a service matrix. The 'overhead' of the additional docker rules does not pose a problem to *fffuu*. The result is shown in Figure 10.13. This graph is almost identical to the graph in Figure 10.8, except for two peculiarities: This time, we did not specify the IP range of the Internet. *WebFrnt* can connect to the Internet, consequently, *WebFrnt* can also connect to the unused IP range starting at 10.0.0.5. Second, we did not configure which containers or ports are publicly accessible. This is a docker configuration option and not part of the firewall setup. Therefore, the firewall setting allows all connection attempts from the Internet, but it depends on the actual docker configuration whether the containers are actually reachable. Hence, this setup—with all its assumptions about a secure and correct setup of docker—can enforce the desired structure (✓). The stateful semantics is provided (✓), which is confirmed by *fffuu* (not visualized in the Figure).

Comparison to Container Links Our presented solution of configuring the iptables firewall of the host system is close to a solution supported natively by docker. Container links provide means to automatically install iptables rules when inter-container communication is desired between two containers [156]. When containers are started with the `--link` option, docker adds iptables rules similar to our setup. With the help of *topoS*, the necessary `--link` parameters could be generated. However, we decided not to follow this direction for the following reasons: Container links are considered legacy by the current version of docker [157]. In addition, the links are always bidirectional, thus no stateful semantics can be provided. Our approach of setting up an individual network is currently the recommended approach [157].

10.4 Discussion

We discuss the limitations and advantages of *topoS*.

10.4.1 Limitations

The process supported by *topoS* currently has two main limitations. First, it is completely static. For example, the mapping of policy entity names to their network representatives is done statically and manually. In general, *naming* is a complex (but orthogonal) issue. This information should usually be managed by a resource and account management system or directory service. Second, only one security device as backend is currently supported. However, related work suggests that this gap can be easily bridged, e.g., by translating to a one big switch abstraction [158, 159]. Many networks additionally employ a variety of heterogeneous, vendor-specific middleboxes. In future work, it might be worth investigating to which extent additional low-level device features (e.g., DHCP, IPv6, timeouts for stateful rules, ...) should be configurable on each abstraction layer.

10.4.2 Advantages

The presented process provides three novel advantages. First, it *bridges several abstraction levels* in a uniform way. The intermediate results are well-specified, which allows manual intervention, visualization, and adding features. Second, the theoretical background is *completely formally verified.* Thus, *topoS* is more than an academic prototype but a highly trustworthy tool. In addition, *topoS*'s library can be reused, extended, exported to several languages, and adapted to fit the needs of other frameworks. Finally, with the formal background, *topoS* is a first step towards high-assurance certification.

Third, *topoS* can scale to large networks w.r.t. theory (*i*), computational complexity (*ii*), and management complexity (*iii*). The theoretical foundation (*i*) scales to arbitrary networks. The computational complexity (*ii*) depends on the type of security invariants. New invariants with arbitrary computational complexity can be developed for *topoS*. However, we found that usually only Φ-structured invariants are needed, which implies the following computational complexity: $\mathrm{O}(|\mathit{invariants}| \cdot |\mathit{entities}|^2)$. An evaluation of *topoS*'s most expensive step has been presented in Section 7.1.1. Finally, (*iii*) the complexity of managing an invariant (with exception for the ACL invariants) is linear in the number entities. Due to the auto-completion, it is actually better than linear. Thus, the management complexity of *topoS* is roughly linear in the number of invariants and entities.

10.5 Related Work

To discuss related work, we first define four management abstraction layers to subsequently classify related work. Each abstraction layer is responsible for an individual problem domain. We illustrate the four layers in Figure 10.14. The layers have well-defined interfaces, thus, it is possible to combine solutions of individual problems. The discussion of related work will exhibit that this work combines perfectly with other related work focusing on different aspects.

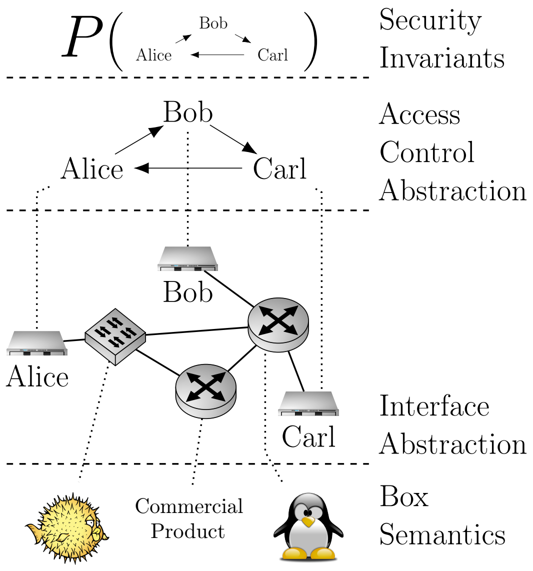


Figure 10.14: Four Layer Abstractions

We propose the following four layers of abstractions.

Security Invariants Defines the high-level security goals. Representable as predicates. For example, Figure 10.2.

Access Control Abstraction Defines the allowed accesses between policy entities. Representable as global access control matrix. For example, Figure 10.3.

Interface Abstraction Defines a model of the complete network topology. Representable as a graph, packets are forwarded between the network entity's interfaces.

Box Semantics Describes the semantics (i.e., behavior) of individual network boxes. Usually, the semantics are vendor-specific (e.g., iptables, Cisco ACLs, Snort IDS).

The main difference between the interface abstraction and the box semantics is that latter describes the behavior of only one network entity, whereas the former describes the interconnection of many, possibly different, network boxes. In the presented case study, both coincide since only one enforcement device is considered.

With regard to Figure 1.1, security invariants are on the abstraction layer of security requirements, the access control abstraction corresponds to the security policy, and the interface abstraction as well as the box semantics correspond to security mechanisms.

In Figure 10.15, we summarize how related work bridges the abstraction layers. Related work may bridge these layers vertically or work horizontally on artifacts at one layer. A direct arrow from the access control abstraction to the box semantics (and vice versa) means that the solution only applies to a single enforcement box. Solutions such as Firmato and Fireman achieve more and are thus listed multiple times.

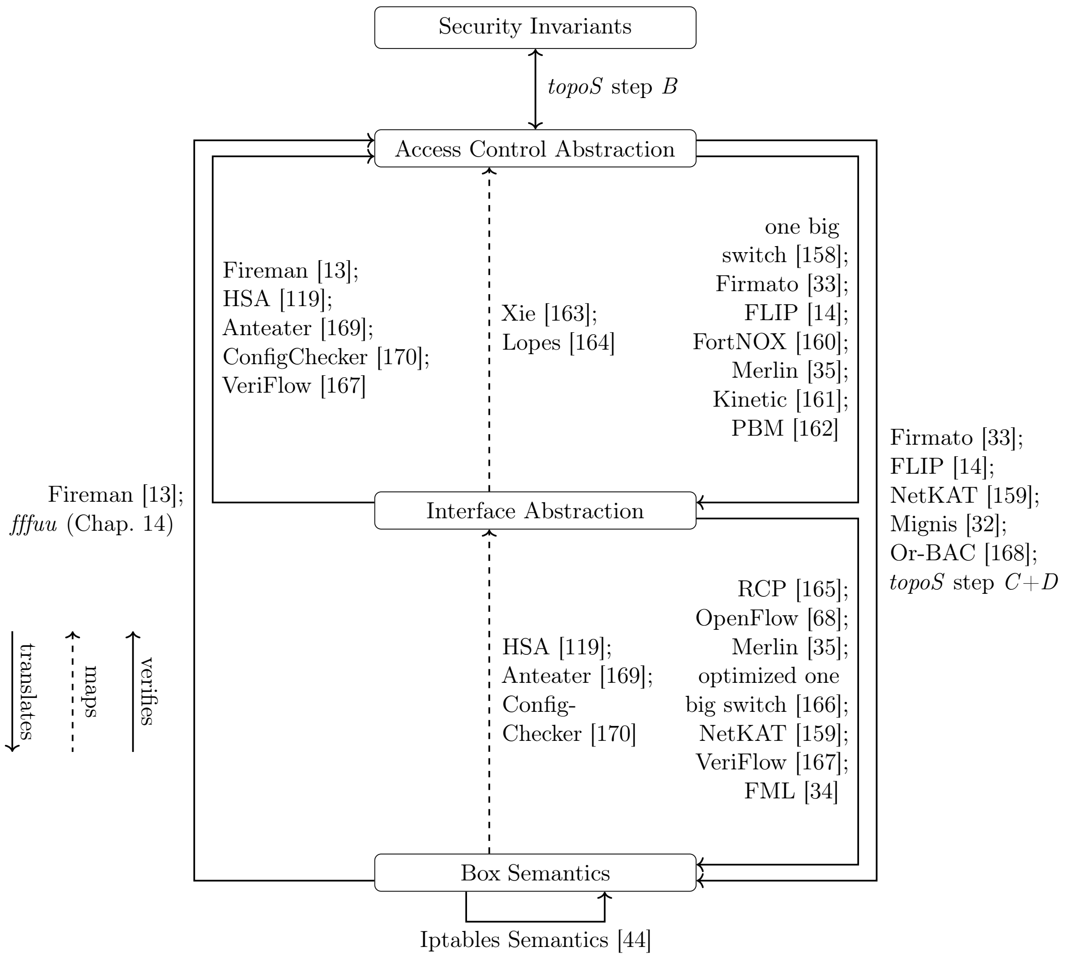

Security Invariants
topoS step B
Access Control Abstraction
Fireman [13]; HSA [119]; Anteater [169]; ConfigChecker [170]; VeriFlow [167]
Xie [163]; Lopes [164]
one big switch [158]; Firmato [33]; FLIP [14]; FortNOX [160]; Merlin [35]; Kinetic [161]; PBM [162]
Firmato [33]; FLIP [14]; NetKAT [159]; Mignis [32]; Or-BAC [168]; topoS step C+D
Fireman [13]; fffuu (Chap. 14)
Interface Abstraction
HSA [119]; Anteater [169]; Config-Checker [170]
RCP [165]; OpenFlow [68]; Merlin [35]; optimized one big switch [166]; NetKAT [159]; VeriFlow [167]; FML [34]
translates
maps
verifies
Box Semantics
Iptables Semantics [44]

Figure 10.15: Four Layer Abstraction in Related Work

Firmato [33] is the work closest related to *topoS*. It defines an entity relationship model to structure network management and compile firewall rules from it, illustrated in Figure 10.16. Firmato focuses on roles, which correspond to policy entities in our model. A role has positive capabilities and is related to other roles, which can be used to derive an access control matrix. Zones, Gateway-Interfaces, and Gateways define the network topology, which corresponds to the interface abstraction. As illustrated in Figure 10.16, the abstraction layers identified in this work can also be identified in Firmato's model. The Host Groups, Role Groups, and Hosts definitions provide a mapping from policy entities to network entities, which is Firmato's approach to the naming problem. With close correspondence in the underlying concepts to Firmato, Cuppens et al. [21] propose a firewall configuration language based on Or-BAC [171]. Similar to Firmato (with more support for negative capabilities) is FLIP [14], which is a high-level language with focus on *service* management (e.g., allow/deny HTTP). Essentially, both FLIP and Firmato enhance the access control abstraction horizontally by including layer four port management and traverse it vertically by serializing to firewall rules.

FML [34] is a flow-based declarative language to define, among others, access control policies in a DATALOG-like language. Comparably to our directed (stateless) policy, FML operates on unidirectional network flows. FML solves the naming problem by assuming that all entities are authenticated with IEEE 802.1X [172].

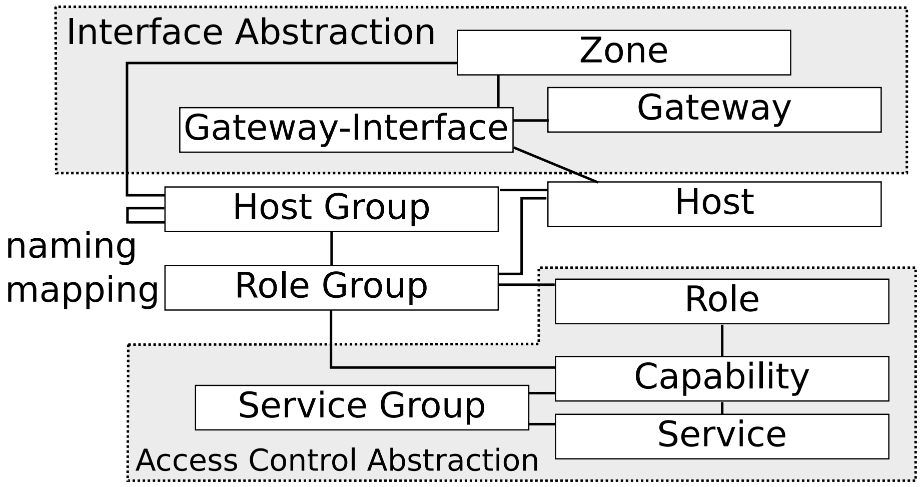


Figure 10.16: Firmato ERM

Policy-based management (PBM) [162] was introduced to simplify network administration. Similar to our work, it proposes different levels of abstraction and describes how to translate between them. Policy-based management defines a generic information model [173, 174] which is not limited to access control, however, we focus our discussion solely on access control and security. In a central policy repository, global policy rules are stored. Policy decision points retrieve these rules and interpret them. Using our terminology, this step translates the access control abstraction to the interface abstraction. A policy decision point forwards decisions to policy enforcement points, implementing the translation from the interface abstraction to box semantics. This last step may be very device-specific [175] and is not the core focus of PBM. While PBM was built on the idea of specifying business-level abstractions in terms of requirements [162], the IETF specified a rule-based policy repository [173, 174], which restricts storing high-level requirements that cannot easily be expressed as rules. In strong contrast to our work, where we focus on specifying higher-level requirements (i.e., security invariants), the IETF Policy Framework working group focused on the specification of lower-level policies [173, §2.1. Policy Scope]. This can also be witnessed in many languages which were developed over the years [176] since, in particular when it comes to security, they usually only provide access control abstractions.

Mignis [32] is a declarative language to manage Netfilter/iptables firewalls. It focuses on packet filtering and NAT. The policy language is restrictive to avoid the usual policy conflicts. In particular, it only allows to write blacklisting-style policies, additionally NAT and filtering cannot arbitrarily be mixed to provide consistent policies.[6] Apart from our work (Part II), it is the only work we are aware of which provides a formal semantics of an iptables firewall. The authors describe a semantics of the packet filtering and NAT behavior of iptables. Their semantics only models the aspects of iptables which are required for their policy language. While it supports NAT (which our semantics of Part II does not), it does not support user-defined chains or arbitrary match conditions.[7] Though supporting NAT and stateful filtering, advanced iptables features such as `NOTRACK` or `connmark` are not considered.

The "One Big Switch" Abstraction [158, 166, 177] allows to manage a network as if it were only one, central big switch. This effectively allows solutions which only support to manage one device to be applied to a complete network, consisting of multiple switches. With regard to Figure 10.15, any solutions which support translating the access control abstraction to box semantics can also be applied to translate from the access control abstraction to the interface abstraction by translating to the "One Big Switch".

NetKAT [31, 159] is a SDN programming language with well-defined semantics. It features an efficient compiler for local, global, and virtual programs to flow table entries [159]. Among others, it allows to implement the "One Big Switch" Abstraction [159].

Craven et al. [178] present a generalized (not network-specific) process to translate access control policies, enhanced with several aspects, to enforceable device-specific policies; the implementation requires a model repository of box semantics and their interplay. Pahl delivers a data-centric, network-specific approach for managing and implementing such a repository, further focusing on things [179].

As illustrated in Figure 10.15, Fireman [13] is a counterpart to Firmato. It verifies firewall rules against a global access policy. In addition, Fireman provides verification on the same horizontal layer (i.e., finding shadowed rules or inter-firewall conflicts, which do not affect the resulting end-to-end connectivity but are still most likely an implementation error). Abstracting to its uses, one may call rcc [180] the fireman for BGP. *fffuu* (presented in Part II) is unique as it not only verifies rules, but also translates them back to the access control abstraction.

Header Space Analysis (HSA) [119], Anteater [169], and ConfigChecker [170] verify several horizontal safety properties on the interface abstraction, such as absence of forwarding loops. By analyzing reachability [163, 164, 169, 170, 119], horizontal consistency of the interface abstraction with an access control matrix can also be verified. Verification of incremental changes to the interface abstraction can be done in real-time with VeriFlow [167], which can also prevent installation of violating rules. These models of the interface abstraction have many commonalities: The network boxes in all models are stateless and the network topology is a graph, connecting the entity's interfaces. A function models packet traversal

[6] The required assumptions for the translation impose many restrictions [32, §5].

[7] Mignis supports match conditions but imposes additional assumptions on them to ensure that they are consistent with NAT. For example, the `iprange` module is not allowed. Without inspecting all matching modules manually, there is no generic way to assure whether a filter condition is compatible with Mignis.

at a network box. Verification of incremental changes to the interface abstraction can be done in real-time by NetPlumber [181] and VeriFlow [167]. The latter can also prevent installation of violating rules. These models could be considered as a giant (extended) finite state machine (FSM), where the state of a packet is an (**interface** × **packet**) pair and the network topology and forwarding function represent the state transition function [164, 182]. In contrast to arbitrary state machines, it is believed that those derived from networks are comparatively well-behaved [182]. Anteater [169] differs in that interface information is implicit and packet modification is represented by relations over packet histories.

Most analysis tools make simplifying assumptions about the underlying network boxes. Diekmann et al. [44] (cf. Part II) present simplification of iptables firewalls to make complex real-world firewalls available for tools with simplifying assumptions. Essentially, the authors horizontally simplify the box semantics.

FortNOX [160] horizontally enhances the access control abstraction as it assures that rules by security apps are not overwritten by other apps. Technically, it hooks up at the access control/interface abstraction translation. Kinetic [161, 183] is an SDN language which lifts static policies (as constructed by *topoS*) to dynamic policies. To accomplish this, an administrator can define a simple FSM which dynamically (triggered by network events) switches between static policies. In addition, the FSM can be verified with a model checker.

Features are horizontally added to the interface abstraction: a routing policy allows specifying *paths* of network traffic [166]. Merlin [35, 184] additionally supports bandwidth assignments and network function chaining. Both translate from a global policy to local enforcement and Merlin provides a feature-rich language for interface abstraction policies. Conceptually similar (but with a completely different implementation), RCP [165] allows to logically centralize routing while remaining compatible with existing routers.

Layer two and layer three network architectures with focus on access control have been developed. In the terminology of this thesis, these architectures correspond to (distributed) security mechanisms, located at the interface abstraction. SANE [67] and Ethane [58] focus on centrally-managed enterprise networks. SANE presents a new architecture where each packet needs a Kerberos-like ticket [185] to be forwarded by the network, basically implementing capability-based source routing. In contrast, Ethane is compatible with existing Ethernet networks implementing access control in each switch, programmed in an OpenFlow-like fashion [68]. PBS [186, 187] is an architecture developed for the context of preventing DoS attacks while being compatible with the current Internet. Its use-case requires distributed management, permissions are granted by receivers of a flow. PBS routers are configured with soft-state to allow only authorized flows using in-band signaling based on IPsec [188]. Both SANE [67] and Ethane [58] with their centralized approach towards policy management could serve as a backend for *topoS*. In fact, OpenFlow is a successor of Ethane and we have demonstrated applicability in this chapter.

We discuss further related work in detail in Chapter 19.

10.6 Conclusion

We presented *topoS*, a fully verified tool to manage network-level access control. It was demonstrated by example; nevertheless, the correctness proofs are universally valid and *topoS* is applicable to any larger network. The example demonstrates that a traditional network segmentation into *internal* and *DMZ* cannot cope with complex security goals and the traditional thought model of structure by IP ranges is no longer appropriate. In contrast, *topoS* only requires the high-level security goals and can automatically translate them to low-level configurations, such as firewall rules or SDN flow table entries. During the translation, all intermediate results are well-defined, accessible, and can be visualized. This provides feedback and allows manual refinement of them, including manual optimizations on lower abstraction layers. After manual refinement, re-verification is run to avoid human error. For the first time, the complete, automated, and verified transition from high-level security goals to Firewall, SDN, and Microservice configurations was presented.

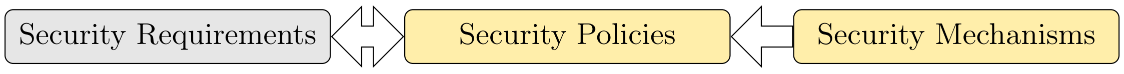
Part II

Understanding Existing Configurations

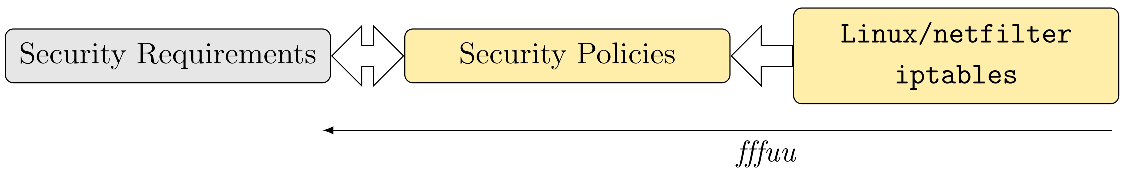
Chapter 11

Overview

Part I focused mainly on the consistency between security requirements and the security policy. It showed how to compute a policy from the requirements and how to verify an existing policy. In this part, we focus on existing configurations of security mechanisms in order to derive the actual policy they are implementing.



We focus on the Linux/netfilter iptables firewall. We present our fully automated tool *fffuu* which can translate a low-level iptables configuration to a policy graph.



This part is structured as follows. First, in Chapter 12, we formalize the filtering behavior of iptables and provide algorithms to transform rulesets to a simpler structure. In Chapter 13, we provide an algorithm to verify spoofing protection of a ruleset. Having excluded IP address spoofing, we can afterwards refer to entities in our policy by their IP addresses. Chapter 14 will then present a method to partition the complete IPv4 and IPv6 address space into classes with equal access rights. This yields the vertices in our policy graph. With this, we compute service matrices which correspond exactly to the required policy graph for a fixed service. We will demonstrate that this solves real-world problems.

Availability

Our Isabelle/HOL theory files with the formalization and the referenced correctness proofs and our tool *fffuu* are available at

`https://github.com/diekmann/Iptables_Semantics` and the AFP [39, 40, 41, 42]

It is the first fully machine-verified iptables analysis tool.

The raw data of the analyzed firewall rulesets can be found at `https://github.com/diekmann/net-network`

To the best of our knowledge, this is the largest, publicly available collection of real-world iptables firewall rulesets.

Chapter 12

Semantics-Preserving Simplification of Real-World Firewall Rule Sets

This chapter is an extended version of the following paper [44]:

- Cornelius Diekmann, Lars Hupel, and Georg Carle. *Semantics-Preserving Simplification of Real-World Firewall Rule Sets*. In 20th International Symposium on Formal Methods, pages 195-212, Oslo, Norway, June 2015. Springer.

The following major improvements and new contributions were added:

- We provide a clarifying discussion of how stateful match conditions can be represented in our (seemingly) stateless model (cf. paragraph 12.3, Model Limitations and Stateful Matchers).
- We prove defindness of our semantics (Theorem 11).
- We add support for the `GOTO` action (which is not discussed in this chapter for brevity but can be found in the accompanying Isabelle formalization).
- We present how our work could be applied to OpenFlow (Section 12.8).

Statement on author's contributions For the original paper, the author of this thesis developed the presented semantics. The author of this thesis provided major contributions for the ideas, formalization, algorithms, and proofs. He researched related work, and conducted the evaluation. All improvements with regard to the paper are the work of the author of this thesis.

Abstract We found that none of the available tools could handle typical, real-world iptables rulesets. This is due to the complex chain model used by iptables, but also due to the vast amount of possible match conditions that occur in real-world firewalls, many of which are not understood by academic and open source tools. We provide a formal big step semantics of iptables filtering behavior. Then, we provide algorithms to transform complex firewall rulesets to a simple list model. Though we do not present our own fully-verified analysis algorithms yet, these transformations already enable existing analysis tools to understand real-world firewall rules.

12.1 Introduction

Firewalls are a fundamental security mechanism for computer networks. Several firewall solutions, ranging from open source [23, 189, 190] to commercial [191, 192], exist. Operating and managing firewalls is challenging as rulesets are usually written manually. While vulnerabilities in the firewall software itself are comparatively rare, it has been known for over a decade [7] that many firewalls enforce poorly written rulesets. However, the prevalent methodology for configuring firewalls has not changed. Consequently, studies regularly report insufficient quality of firewall rulesets [13, 9, 14, 15, 16, 10, 12, 11].

Therefore, several tools [135, 136, 13, 137, 134, 16, 12, 193] have been developed to ease firewall management and reveal configuration errors. However, when we tried to analyze real-world firewalls with the publicly available tools, none of them could handle our firewall rules. We found that the firewall model of the available tools is too simplistic.

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

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

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

Figure 12.1: Linux iptables ruleset of a Synology NAS (network attached storage) device

To demonstrate the problem by example, we decided to use *ITVal* [135] because it natively supports iptables [23], is open source, and supports calls to user-defined chains. However, ITVal's firewall model is representative of the model used by the majority of tools; therefore, the problems described here also apply to a vast range of other tools. Firewall models used in related work are surveyed in Section 12.2. For this example, we use the firewall rules in Figure 12.1, taken from an NAS device. The ruleset reads as follows: First, incoming packets are sent to the user-defined chain `DOS_PROTECT`, where some rate limiting is applied. Afterwards, the firewall allows all packets which belong to already established connections. This is generally considered good practice. Then, some services, identified by their ports, are blocked. Finally, the firewall allows all packets from the local network 192.168.0.0/16 and discards all other packets. We used ITVal to partition the IP space into equivalence classes (i.e., ranges with the same access rights) [136]. The expected result is a set of two IP ranges: the local network 192.168.0.0/16 and the "rest". However, ITVal erroneously only reports one IP range: the universe. Removing the first two rules (in particular the call in the `DOS_PROTECT` chain) lets ITVal compute the expected result.

We identified two main problems which prevent tools from "understanding" real-world firewalls. First, calling and returning from custom chains, due to the possibility of complex nested chain calls. Second, more seriously, most tools do not understand the firewall's match conditions. In the above example, the rate limiting is not understood. The problem of unknown match conditions cannot simply be solved by implementing the rate limiting feature for the respective tool. The major reason is that the underlying algorithm might not be capable of dealing with this special case. Additionally, firewalls, such as iptables, support numerous match conditions and several new ones are added in every release. As of version 1.6.0 (Linux kernel 4.10, early 2017), iptables supports more than 60 match conditions with over 200 individual options. Firewalls, such as iptables, support numerous match conditions and several new ones are added in every release. We expect even more match conditions for nftables [189] in the future since they can be written as simple userspace programs [194]. Therefore, it is virtually impossible to write a tool which understands all possible match conditions.

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

To summarize, we provide the following contributions:

1. A formal semantics of iptables packet filtering (Section 12.3),

2. Chain unfolding: transforming a ruleset in the complex chain model to a ruleset in the simple list model (Section 12.4),

3. An embedded semantics with ternary logic, supporting arbitrary match conditions, introducing a lower/upper closure of accepted packets (Section 12.5), and

4. Normalization and translation of complex logical expressions to an iptables-compatible format, discovering a meta-logical firewall algebra (Section 12.6).

We evaluate applicability on large real-world firewalls in Section 12.7. All proofs are machine-verified with Isabelle [37]. Therefore, the correctness of all obtained results only depends on a small and well-established mathematical kernel and the iptables semantics (Figure 12.2).

12.2 Firewall Models in the Literature and Related Work

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

The literature agrees on the definition of a single firewall rule. It consists of a predicate (the match expression) and an action. If the match expression applies to a packet, the action is performed. Usually, a packet is scrutinized by several rules. Zhang et al. [14] specify a common format for packet filtering rules. The action is either "allow" or "deny", which directly corresponds to the firewall's filtering decision. The ruleset is processed strictly sequentially. Yuan et al. [13] call this the *simple list model*. ITVal also supports calls to user-defined chains as an action. This allows "jumping" within the ruleset without having a final filtering decision yet. This is called the *complex chain model* [13].

In general, a packet header is a bitstring which can be matched against [195]. Zhang et al. [14] support matching on the following packet header fields: IP source and destination address, protocol, and port on layer 4. This model is commonly found in the literature [134, 33, 14, 13, 131, 140]. ITVal extends these match conditions with flags (e.g., `TCP SYN`) and connection states (`INVALID`, `NEW`, `ESTABLISHED`, `RELATED`). The state matching is treated as just another match condition.[1] This model is similar to Margrave's model for IOS [16]. When comparing these features to the simple firewall in Figure 12.1, it becomes obvious that none of these tools supports that firewall.

We are not the first to propose simplifying firewall rulesets to enable subsequent analysis. Brucker et al. [140, 196, 197] provide algorithms to transform firewall rulesets to enable the testing of a firewall. They prove correctness of their transformations in Isabelle/HOL. Unfortunately, their supported firewall model is very limited, even more limited than the model used by the tools presented in the previous paragraph. They support the simple list model and are restricted to match only on networks, ports, and protocols. Instead of natively supporting IP addresses, they support the notion of networks which imposes additional constraints: While it is natural for IP address ranges in a ruleset to overlap (e.g., 10.0.0.0/8 and 10.42.42.0/24), their notion of networks requires that all different networks in a ruleset must be distinct [196, 140]. Their work is only evaluated on three very small real-world rulesets (less than 15 rules each) of the same institute.

We are not aware of any tool which uses a model fundamentally different than those described in the previous paragraphs. Our model enhances existing work in that we use ternary logic to support arbitrary match conditions. To analyze a large iptables firewall, the authors of Margrave [16] translated it to basic Cisco IOS access lists [191] by hand. With our simplification, we can automatically remove all features not understood by basic Cisco

[1]Firewalls can be stateful or stateless. Most firewalls nowadays are stateful, which means the firewall remembers and tracks information of previously seen packets, e.g., the TCP connection a packet belongs to and the state of this connection. ITVal does not track the state of connections. Match conditions on connection states are treated exactly the same as matches on a packet header. In general, focusing on rulesets and not firewall implementation, matching on iptables conntrack states is exactly as matching on any other (stateless) condition. However, internally, not only the packet header is consulted but also the current connection tables. Note that existing firewall analysis tools also largely ignore state [16]. In our semantics, we also model stateless matching.

IOS. This enables translation of any iptables firewall to a basic Cisco access lists which is guaranteed to drop no more packets than the original iptables firewall. This opens up all tools available only for Cisco IOS access lists, e.g., Margrave [16] and Header Space Analysis [119].[2]

12.3 Semantics of iptables

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

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

In one firewall rule, several primitives can be specified. Their logical connective is conjunction, for example `src 192.168.0.0/24` *and* `tcp`. Disjunction is omitted because it is neither needed for the formalization nor supported by iptables; this is consistent with the model by Jeffrey and Samak [200]. Primitives can be combined in an algebra of *match expressions* M_X:

$$\mathit{mexpr} \quad = \quad x \quad \textbf{for } x \in X \quad | \quad \neg\, \mathit{mexpr} \quad | \quad \mathit{mexpr} \wedge \mathit{mexpr} \quad | \quad \mathsf{True}$$

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

Furthermore, we support the following *actions*, modeled closely after iptables: `Accept`, `Reject`, `Drop`, `Log`, `Empty`, `Call` c for a chain c, and `Return`. A *rule* can be defined as a tuple (m, a) for a match expression m and an action a. A list (or sequence) of rules is called a *chain.* For example, the beginning of the `DOS_PROTECT` chain in Figure 12.1 is $[($`icmp` $\wedge$ `icmptype 8 limit: ...`, `Return`$), \ldots]$.

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

The semantics of a firewall w.r.t. to a given packet p, a background ruleset Γ, and a primitive matcher γ can be defined as a relation over the currently active chain and the state before and the state after processing this chain. The semantics is specified in Figure 12.2.[3] On the left side of the turnstile ($\vdash$), the constants Γ,γ,p are written. The expression $\Gamma, \gamma, p \vdash \langle rs,\ t\rangle \Rightarrow t'$ states that starting with state t, after processing the chain rs, the resulting state is t'. For a packet p, our semantics focuses on firewall filtering

[2]Note that the other direction is considered easy [198], because basic Cisco IOS access lists have "no nice features" [199]. Also note that there also are *Advanced* Access Lists.

[3]inductive iptables-bigstep

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

$$\textsc{Drop}\ \frac{\mathsf{match}\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash \big\langle [(m,\ \texttt{Drop})],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{\times}}$$

$$\textsc{Reject}\ \frac{\mathsf{match}\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash \big\langle [(m,\ \texttt{Reject})],\ \textcircled{?} \big\rangle \Rightarrow \textcircled{\times}}$$

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

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

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

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

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

(for any primitive matcher γ and any well-formed ruleset Γ)

Figure 12.2: Big Step semantics for iptables

decisions. Therefore, only the following three states are necessary: The firewall may allow ($\textcircled{\checkmark}$) or deny ($\textcircled{\times}$) the packet, or it may not have come to a decision yet ($\textcircled{?}$).

We will now discuss the most important rules. The Accept rule describes the following: if the packet p matches the match expression m, then the firewall with no filtering decision ($\textcircled{?}$) processes the singleton chain $[(m,\ \texttt{Accept})]$ by switching to the allow state. Both the Drop and Reject rules deny a packet; the difference is only in whether the firewall generates some informational message, which does not influence filtering. The NoMatch rule specifies that if the firewall has not come to a filtering decision yet, it can process any non-matching rule without changing its state. The Decision rule specifies that as soon as the firewall made a filtering decision, it does not change its decision. The Seq rule specifies that if the firewall has not come to a filtering decision and it processes the chain rs_1 which results in state t and starting from t processes the chain rs_2 which results in state t', then both chains can be processed sequentially, ending in state t'. The CallResult rule specifies that if a matching `Call` to a chain named "c" occurs, the resulting state t is the result of processing the chain $\Gamma\ c$. Likewise, the CallReturn rule specifies that if processing a prefix rs_1 of the called chain does not lead to a filtering decision and directly afterwards, a matching `Return` rule occurs, the called chain is processed without result.[4] The LOG rule does not influence the filtering behavior. Similarly, the EMPTY rule does not result in a filtering decision. An EMPTY rule, i.e., a rule without an action, occurs if iptables only updates its internal state, e.g., updating packet counters.

Model Limitations and Stateful Matchers Our primitive matcher is completely stateless: $\gamma :: (X \Rightarrow \mathit{packet} \Rightarrow \mathbb{B})$. However, iptables also allows stateful operations, such as marking a packet and matching on the marking later on.

The documentation of iptables distinguishes between match extensions and target extensions. Ideally, almost all match extensions can be used as if they were stateless. Anything which performs an action should be implemented as target extension, i.e., action. For example, marking a packet with `CONNMARK` is an action. Matching on a `CONNMARK` marking is a match condition. Our semantics does not support the `CONNMARK` action. This is not a problem since usually, new `CONNMARK` markings are not set in the `filter` table. However, it is possible to match on existing markings. Since our primitive matchers and packets are completely generic, this case can be represented within our model: Instead of keeping an internal `CONNMARK` state, an additional "ghost field" must be introduced in the packet model. Since packets are immutable, this field cannot be set by a rule but the packet must be given to the firewall with the final value of the ghost field already set. Hence, an analysis must be carried out with the correct value in the ghost fields when the packet is given to the `filter` table. We admit that this model is very unwieldy in general. However, for one of the most used stateful modules of iptables, namely connection state tracking with `conntrack` and `state`, this model has been proven to be very convenient.[5] We will elaborate on stateful connection tracking (which can be fully supported by our semantics) in Section 14.4.2. When later embedding into the more practical ternary semantics, all further stateful primitives (such as `CONNMARK`) can be considered "unknown" and are correctly abstracted by these semantics.

What if a match extension maintains an internal state and changes its behavior on every invocation? Ideally, due to usability, iptables match extensions should not exhibit this behavior; however, the `recent` module or the `connbytes` exhibit similar behavior. This means, the tautology in Boolean logic "$a \wedge \neg a = \mathsf{False}$" does not hold if a is a module which updates an internal state and its matching behavior after every invocation. Therefore, one might argue that our iptables model can only be applied to stateless match conditions. If we add some state σ and updated state σ' to the match condition, the formula "$a_\sigma \wedge \neg a_{\sigma'}$" now correctly represents stateful match conditions. Therefore, it is only wrong to perform equality operations on stateful match conditions but not to model stateful match conditions with a specific, fixed state. In our implementation, we immediately embed everything in ternary logic and treat all stateful primitives as "unknown"; more precisely, we only perform simplification on primitives which are definitely stateless. This prevents this error and yields "$a \wedge \neg a = \mathsf{Unknown}$", which is a correct model since we do not know about a potential

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

[5]Semantics-Stateful.thy

internal state of some arbitrary match condition a. To additionally convince the reader about the soundness of our approach, it would be possible to adapt the parser to give a unique ID to every primitive which is not known to be stateless. This unique ID represents the internal state of that particular match condition on that particular position in a ruleset. It prevents equality operations between multiple invocations of a stateful match condition. It does not change any of our algorithms since we immediately embed all these primitives in ternary logic and treat them as “unknown”.

For future work, if we want to consider e.g., the `raw` or `mangle` table with its extended set of actions or OpenFlow with its full set of actions, a semantics needs to be designed with a mutable packet model.

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

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

All our proofs are machine-verified with Isabelle. Therefore, once the reader is convinced of the semantics as specified in Figure 12.2, the correctness of all subsequent theorems follows automatically – without any hidden assumptions or limitations.

The rules in Figure 12.2 are designed such that every rule can be inspected individually. However, considering all of them together, it is not immediately clear whether the result depends on the order of their application to a concrete ruleset and packet. Theorem 10 states that the semantics is deterministic, i.e., only one uniquely defined outcome is possible.[6]

Theorem 10 (Determinism).

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

Next, we show that the semantics are actually defined, i.e., there is always a decision for any packet and ruleset.[7] We assume that the ruleset does not have an infinite loop and that all chains which are called exist in the background ruleset. These conditions are checked by the Linux kernel and can thus be assumed. The way we start any analysis (namely [(`True`, `Call` *start-chain*), (`True`, *default-policy*)], where the default policy is either `Accept` or `Drop`) directly implies that we cannot have a top-level `Return`. In addition, we assume that only the actions defined in Figure 12.2 occur in the ruleset; our parser rejects everything else.

Theorem 11 (Defined). *If the caller-callee relation is well-founded and finite (i.e., there are no infinite calling loops) and* Γ *is well-formed (i.e., all chain names which are called are defined) and there is no* `Return` *on top-level and all actions are supported by the semantics, then*

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

[6]iptables-bigstep-deterministic
[7]semantics-bigstep-defined

To also assert *empirically* that we only allow analysis of iptables rulesets which are defined according to our semantics, we always check the preconditions of Theorem 11 at runtime when our tool loads a ruleset. First, we can statically verify that Γ is well-formed by verifying that all chain names which are referenced in an action are defined and that no unsupported actions occur. Next, our tool verifies that there are no infinite loops by unfolding the ruleset (see next section) only a finite but sufficiently large number of times and abort if the ruleset is not in the proper form afterwards. For all real-world firewalls we have analyzed, these conditions were almost never violated. They were only violated for a negligible quantity of firewalls which used very special iptables actions[8] not supported by our semantics or special hand-crafted firewalls which deliberately violate a property and which are also rejected by the Linux kernel.

12.4 Custom Chain Unfolding

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

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

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

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

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

The procedure `pc` can be applied arbitrarily many times and preserves the semantics.[9] It is sound and complete.

Theorem 12 (Soundness and Completeness).

$$\Gamma, \gamma, p \vdash \left\langle \mathsf{pc}^n\ rs,\ t \right\rangle \Rightarrow t' \quad \textit{iff} \quad \Gamma, \gamma, p \vdash \left\langle rs,\ t \right\rangle \Rightarrow t'$$

[8]For example setting `CONNMARK` in the `filter` table where it was not necessary, redirecting packets to userspace with `NFQUEUE` where we do not know how the userspace application handles them, or specialized loggings such as `NFLOG` which is technically equivalent to the `LOG` semantics and could directly be supported.

[9]unfolding-n-sound-complete

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

Figure 12.3: Unfolded Synology Firewall

In each iteration, the algorithm unfolds one level of `Calls`. The algorithm needs to be applied until the result no longer changes. Note that the semantics allows non-terminating rulesets; however, the only rulesets that are interesting for analysis are the ones actually accepted by the Linux kernel.[10] Since it rejects rulesets with loops, both our algorithm and the resulting ruleset are guaranteed to terminate.

Corollary 2. *Every ruleset (with only* `Accept`*,* `Drop`*,* `Reject`*,* `Log`*,* `Empty`*,* `Call`*,* `Return` *actions) accepted by the Linux kernel can be unfolded completely while preserving its filtering behavior.*

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

- Replacing `Reject` actions with `Drop` actions,[11]
- Removing `Empty` and `Log` rules,[12]
- Simplifying match expressions which contain `True` or ¬ `True`.[13]
- For some given primitive matcher, specific optimizations may also be performed,[14] e.g., rewriting `src 0.0.0.0/0` to `True`.

Therefore, after unfolding and optimizing, a chain which only contains `Allow` or `Drop` actions is left. In the subsequent sections, we require this as a precondition. As an example, recall the firewall in Figure 12.1. Its `INPUT` chain after unfolding and optimizing is listed in Figure 12.3. Observe that the computed match expressions are beyond iptable's expressiveness. An algorithm to normalize the rules to an iptables-compatible format will be described in Section 12.6.

12.5 Unknown Primitives

As we argued earlier, it is infeasible to support all possible primitives of a firewall. Suppose a new firewall module is created which provides the `ssh_blacklisted` and `ssh_innocent` primitives. The former applies if an IP address has had too many invalid SSH login attempts in the past; the latter is the opposite of the former. Since we made up these primitives, no existing tool will support them. However, a new version of iptables could implement them or

[10] The relevant check is in `mark_source_chains`, file `source/net/ipv4/netfilter/ip_tables.c` of the Linux kernel version 3.2.
[11] iptables-bigstep-rw-Reject
[12] iptables-bigstep-rm-LogEmpty
[13] unfold-optimize-ruleset-CHAIN
[14] unfold-optimize-ruleset-CHAIN

they can be provided as third-party kernel modules. Therefore, our ruleset transformations must take unknown primitives into account. To achieve this, we lift the primitive matcher γ to ternary logic, adding Unknown as matching outcome. We embed this new "approximate" semantics into the semantics described in the previous sections. Thus, it becomes easier to construct matchers tailored to the primitives supported by a particular tool.

12.5.1 Ternary Matching

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

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

These rules correspond to Kleene's 3-valued logic [201] and are well-suited for firewall semantics: The first equation states that, if one condition matches, the final result only depends on the other condition. The next equation states that a rule cannot match if one of its conditions does not match. Finally, by negating an unknown value, no additional information can be inferred.

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

12.5.2 Closures

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

We introduce the *in-doubt-allow* and *in-doubt-deny* tactics. The first tactic forces a match if the rule's action is `Accept` and a mismatch if it is `Drop`. The second tactic behaves in the opposite manner. Note that an unfolded ruleset is necessary, since no behavior can be specified for `Call` and `Return` actions.[15]

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

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

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

For example, the unfolded firewall of Figure 12.1 contains rules which drop a packet if a limit is exceeded. If this rate limiting is not understood by γ, the *in-doubt-allow* tactic will never apply this rule, while with the *in-doubt-deny* tactic, it is applied universally.

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

If the exact and ternary matcher agree, then the set of all packets allowed by the *in-doubt-deny* tactic is a subset of the packets allowed by the exact semantics, which in turn is a subset of the packets allowed by the *in-doubt-allow* tactic.[16] Therefore, we call all packets accepted by $\Rightarrow_{\text{deny}}$ the *lower closure*, i.e., the semantics which accepts at most the packets that the exact semantics accepts. Likewise, we call all packets accepted by $\Rightarrow_{\text{allow}}$ the *upper closure*, i.e., the semantics which accepts at least the packets that the exact semantics accepts. Every packet which is not in the upper closure is guaranteed to be dropped by the firewall.

Theorem 13 (Lower and Upper Closure of Allowed Packets).

$$\begin{gathered}\big\{p.\ \Gamma,\gamma,p \vdash \langle rs,\ ⓘ\rangle \Rightarrow_{\text{deny}} \oslash\big\}\\ \subseteq\\ \big\{p.\ \Gamma,\gamma,p \vdash \langle rs,\ ⓘ\rangle \Rightarrow \oslash\big\}\\ \subseteq\\ \big\{p.\ \Gamma,\gamma,p \vdash \langle rs,\ ⓘ\rangle \Rightarrow_{\text{allow}} \oslash\big\}\end{gathered}$$

The opposite holds for the set of denied packets.[17]

For the example in Figure 12.1, we computed the closures (without the `RELATED, ESTABLISHED` rule, see Section 12.5.4) and a ternary matcher which only understands IP addresses and layer 4 protocols. The lower closure is the empty set since rate limiting could apply to any packet. The upper closure is the set of packets originating from 192.168.0.0/16.

12.5.3 Removing Unknown Matches

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

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

Hence, the base cases of `pu` are straightforward. However, the case of a negated conjunction of match expressions requires some care. The following equation represents the De

[16] FinalAllowClosure

[17] FinalDenyClosure

Morgan rule, specialized to the in-doubt-allow tactic.

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

A note about the notation: The algorithm explicitly tests for '$\cdot = \mathsf{True}$', since in this context, True is the syntactic base case of a match expression M_X.

The $\neg\,\mathsf{Unknown} = \mathsf{Unknown}$ equation is responsible for the complicated nature of the De Morgan rule. Fortunately, we machine-verified all our algorithms.[18] For example, during our research, we wrote a seemingly simple (but incorrect) version of pu and everybody agreed that the algorithm looks correct. In the early empirical evaluation, with yet unfinished proofs, we did not observe our bug. Only because of the failed correctness proof did we realize that we introduced an equation that only holds in Boolean logic.

Theorem 14 (Soundness and Completeness)**.**

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

Theorem 15. *Algorithm* pu *removes all unknown primitive match expressions.*

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

As an example, we examine the ruleset of the upper closure of Figure 12.1 (without the `RELATED,ESTABLISHED` rule, see Section 12.5.4) for a ternary matcher which only understands IP addresses and layer 4 protocols. The ruleset is simplified to [(`src 192.168.0.0/16`, `Accept`), (`True`, `Drop`)]. ITVal can now directly compute the correct results on this ruleset.

12.5.4 The `RELATED,ESTABLISHED` Rule

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

It is good practice [202] to start a firewall with an `ESTABLISHED` (and sometimes `RELATED`) rule. This also happens in Figure 12.1 after the rate limiting. The `ESTABLISHED` rule usually matches most of the packets [202],[19] which is important for performance; however, when analyzing the filtering behavior of a firewall, it is important to consider how a connection can be brought to this state. Therefore, we remove this rule and only focus on the connection setup.

[18]transform-remove-unknowns-upper, transform-remove-unknowns-lower

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

The `ESTABLISHED` rule essentially allows packet flows in the opposite direction of all subsequent rules (cf. Chapter 9). Unless there are special security requirements (which is not the case in any of our analyzed scenarios), the `ESTABLISHED` rule can be excluded when analyzing the connection setup (Corollary 1).[20] If the `ESTABLISHED` rule is removed and in the subsequent rules, for example, a primitive `state NEW` occurs, our ternary matcher returns `Unknown`. The closure procedures handle these cases automatically, without the need for any additional knowledge.

In Section 14.4.2, we will describe our improvements which will enable support for conntrack state. There will no longer be the need to manually exclude rules. In short, we will fully support matches on conntrack state such as `ESTABLISHED` or `NEW`. The observation and argument of this section remains: for access control analysis, we focus on `NEW` packets.

12.6 Normalization

Ruleset unfolding may result in non-atomic match expressions, e.g., $\neg\,(a \wedge b)$. iptables only supports match expressions in *Negation Normal Form* (NNF).[21] There, a negation may only occur before a primitive, not before compound expressions. For example, $\neg\,(\texttt{src}\ \mathit{ip}) \wedge \texttt{tcp}$ is a valid NNF formula, whereas $\neg\,((\texttt{src}\ \mathit{ip}) \wedge \texttt{tcp})$ is not. The reason is that iptables rules are usually specified on the command line and each primitive is an argument to the `iptables` command, for example `! --src` *ip* `-p tcp`. We normalize match expressions to NNF, using the following observations:

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

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

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

$$\begin{aligned}
\mathsf{n}\ \mathsf{True} &= [\mathsf{True}] \\
\mathsf{n}\ (m_1 \wedge m_2) &= [x \wedge y.\ \ x \leftarrow \mathsf{n}\ m_1,\ y \leftarrow \mathsf{n}\ m_2] \\
\mathsf{n}\ (\neg\,(m_1 \wedge m_2)) &= \mathsf{n}\ (\neg m_1)\ \ ::: \ \ \mathsf{n}\ (\neg m_2) \\
\mathsf{n}\ (\neg\neg m) &= \mathsf{n}\ m \\
\mathsf{n}\ (\neg \mathsf{True}) &= [] \\
\mathsf{n}\ x &= [x] \qquad \text{for } x \in X
\end{aligned}$$

[20]The same can be concluded for reflexive ACLs in Cisco's IOS Firewall [191].

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

$$\mathsf{n}\ (\neg x) \qquad = [\neg x] \qquad \text{for } x \in X$$

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

The unfolded ruleset of Figure 12.3, which consists of 9 rules, can be normalized to a ruleset of 20 rules (due to distributivity). In the worst case, normalization can cause an exponential blowup. Our evaluation shows that this is not a problem in practice, even for large rulesets. This is because rulesets are usually managed manually, which naturally limits their complexity to a level processible by state-of-the-art hardware.

Theorem 16. n *always terminates, all match expressions in the returned list are in NNF, and their conjunction is equivalent to the original expression.*[22]

We show soundness and completeness w.r.t. arbitrary γ, α, and primitives.[23] Hence, it also holds for the Boolean semantics. In general, proofs about the ternary semantics are stronger (the ternary primitive matcher can simulate the Boolean matcher).[24]

Theorem 17 (Soundness and Completeness)**.**

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

After having been normalized by n, the rules can mostly be fed back to iptables. For some specific primitives, iptables imposes additional restrictions, e.g., that at most one primitive of a type may be present in a single rule. For our evaluation, we only need to solve this issue for IP address ranges in CIDR notation [203]. We introduced and verified another transformation which computes intersection of IP address ranges, which returns at most one range. This is sufficient to process all rulesets we encountered during evaluation. In the following chapters, we show how to support more primitives; the evaluation in this chapter only focuses on IP addresses.

12.7 Evaluation

In this section, we demonstrate the applicability of our ruleset preprocessing. Usually, network administrators are not inclined towards publishing their firewall ruleset because of potential negative security implications. For this evaluation, we have obtained approximately 20k real-world rules and the permission to publish them. In addition to the running example in Figure 12.1 (a small real-world firewall), we tested our algorithms on four other real-world firewalls. We put focus on the third ruleset, because it is one of the largest and the most interesting one.

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

[22] normalized-nnf-match-normalize-match
[23] normalize-match-correct
[24] β_{magic}-approximating-bigstep-fun-iff-iptables-bigstep, LukassLemma

Ruleset 1 is taken from a Shorewall [204] firewall, running on a home router, with around 500 rules. We verified that our algorithms correctly unfold, preprocess, and simplify this ruleset. We expected to see, in both the upper and lower closure, that the firewall drops packets from private IP ranges. However, we could not see this in the upper closure and verified that the firewall does indeed not block such packets if their connection is in a certain state. The administrator of the firewall confirmed this issue and is currently investigating it.

Ruleset 2 is taken from a small firewall script found online [205]. Although it only contains about 50 rules, we found that it contains a serious mistake. We assume the author accidentally confused iptables' `-I` (insert at top) and `-A` (append at tail) options. We saw this after unfolding, as the firewall allows nearly all packets at the beginning. Subsequent rules are shadowed and cannot apply. However, these rules come with a documentation of their intended purpose, such as "drop reserved addresses", which highlights the error. We verified the erroneous behavior by installing the firewall on our systems. The author is currently investigating this issue. Thus, our unfolding algorithm alone can provide valuable insights.

Ruleset 3 & 4 are taken from the main firewall of our lab (Chair of Network Architectures and Services). One snapshot was taken 2013 with 2800 rules and one snapshot was taken 2014, containing around 4000 rules. It is obvious that these rulesets have historically grown. About ten years ago, these two rulesets would have been the largest real-world rulesets ever analyzed in academia [7].

We present the analysis results of the 2013 version of the firewall. Details can be found in the additional material. We removed the first three rules. The first rule was the `ESTABLISHED` rule, as discussed in Section 12.5.4. Our focus was put on the second rule when we calculated the lower closure: this rule was responsible for the lower closure being the empty set. Upon closer inspection of this rule, we realized that it was 'dead', i.e., it can never apply. We confirmed this observation by changing the target to a `Log` action on the real firewall and could never see a hit of this rule for months. Due to our analysis, this rule could be removed. The third rule performed SSH rate limiting (a `Drop` rule). We removed this rule because we had a very good understanding of it. Keeping it would not influence correctness of the upper closure, but lead to a smaller lower closure than necessary.

First, we tested the ruleset with the well-maintained Firewall Builder [193]. The original ruleset could not be imported by Firewall Builder due to 22 errors, caused by unknown match expressions. Using the calculated upper closure, Firewall Builder could import this ruleset without any problems.

Next, we tested ITVal's IP space partitioning query [136]. On our original ruleset with 2800 rules, ITVal completed the query with around 3 GB of RAM in around 1 min. Analyzing ITVal's debug output, we found that most of the rules were not understood correctly due to unknown primitives. Thus, the results were spurious. We could verify this as 127.0.0.0/8, obviously dropped by our firewall, was grouped into the same class as the rest of the Internet. In contrast, using the upper and lower closure ruleset, ITVal correctly identifies 127.0.0.0/8 as its own class.

We found another interesting result about the ITVal tool: The (optimized) upper closure ruleset only contains around 1000 rules and the lower closure only around 500 rules. Thus, we expected that ITVal could process these rulesets significantly faster. However, the opposite is the case: ITVal requires more than 10 times the resources (both CPU and RAM, we had to move the analysis to a $>$ 40 GB RAM cluster) to finish the analysis of the closures. We assume that this is due to the fact that ITVal now understands *all* rules. Yet, Chapter 14 will reveal that ITVal still computes wrong results.

12.8 Outlook: Verifying OpenFlow Rules

OpenFlow [206, 207] is a standard for configuring OpenFlow-enabled switches. It is usually referred to in the context of Software-Defined Networking (SDN) and is a hot topic in network management and operations since almost ten years (cf. Section 2.2). In this section, we elaborate on our decision to focus on iptables instead of OpenFlow and describe how our results could also contribute to the verification of low-level OpenFlow rulesets.

This chapter, and in general the complete Part II of this thesis, focuses on the analysis of iptables instead of OpenFlow for several reasons: Despite OpenFlow 1.0 [206] being available for over five years, compared to iptables, OpenFlow is a relatively young and not very wide-spread technology. In contrast, the iptables firewall is wide-spread, real-world approved, supports a large amount of features, and is in productive use for over a decade. There are also decade-old configurations which utilize a vast amount of features, which are no longer fully understood by the administrator [25]. As of July 2016, the popular network of stackoverflow [152] serverfault[25] counts more than a hundred times more questions related to iptables than OpenFlow and the superuser[26] network (also a part of the stackoverflow network) counts even a thousand times more questions related to iptables than OpenFlow. Over the years, iptables has evolved into a system with an enormous amount of (legacy) features. Compared to this, OpenFlow is a relatively young and tidy technology. But we anticipate to see a similar feature-creep over the years, considering e.g., Nicira extensions [208] or attempts to enhance OpenFlow with generic FPGAs to add "exotic funcionality [sic]" [209]. In a broader context, by extending OpenFlow or one of its stateful, more feature-rich, proposed successors [210], many iptables features have been already reimplemented on top of it [211].

Our declared goal was to provide scientific methods to understand challenging configurations (as observed in iptables) and evaluate our methodology on complex, real-world, legacy-grown systems. The insights we obtained can also be applied to OpenFlow. In particular, a large portion of Part II focuses on match conditions, e.g., abstracting over unknowns, optimizing, rewriting, normalizing, or even replacing interfaces by IP addresses. Our work on match condition can be directly reused in the context of OpenFlow.

However, iptables is not OpenFlow. In particular, the OpenFlow standard defines a vast amount of actions which can be performed for a packet. In contrast, iptables `filter`ing primarily uses the two actions `ACCEPT` and `DROP`. This is because a firewall clearly separates filtering from other network functions, such as packet rewriting. OpenFlow implementations tend to easily mixes those. We have shown how to deal with unknown match conditions but unknown actions are an unsolved problem. We have discussed what would be required for a full OpenFlow semantics . In particular, a mutable packet model (cf. paragraph 12.3,

[25]`https://serverfault.com/`
[26]`https://superuser.com/`

Model Limitations and Stateful Matchers) would be necessary, which our methods do not support. However, there is no technical need for OpenFlow switches to mix packet filtering with other operations. For example, the pipelined OpenFlow Router architecture constructed by Nelson et al. [212, cf. §3, Fig. 3] clearly separates packet filtering from packet forwarding and rewriting. In general, using pipeline processing as specified in recent OpenFlow standards [207] might be a step forward to separate filtering from forwarding and rewriting. This may also help compilers which produce OpenFlow rules and suffer from a large blow-up which is induced by a cross product over several tables to join rules for different actions into one table [159]. Such a filtering table implemented by OpenFlow rules without unspecified behavior could be analyzed by our presented methods.

12.9 Conclusion

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

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

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

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

Chapter 13

Certifying Spoofing-Protection of Firewalls

This chapter is an extended version of the following paper [213]:

- Cornelius Diekmann, Lukas Schwaighofer, and Georg Carle. *Certifying Spoofing-Protection of Firewalls.* In 11th International Conference on Network and Service Management, CNSM, Barcelona, Spain, November 2015.

The following improvements were added:

- One new evaluation section (Section 13.7.4) was added to the accepted version of the paper which also made it into the published, camera-ready version.
- Two new examples were added to the evaluation.
- Spoofing protection was integrated into the *fffuu* tool.
- We discuss why the simple firewall model (cf. Chaper 14) is explicitly not used here.

Statement on author's contributions For the original paper, the author of this thesis provided major contributions for the ideas, mathematical formalization, realization, implementation, and proof of the presented algorithm. He researched related work, and conducted the evaluation. All improvements are the work of the author of this thesis.

Abstract In this chapter, we present an algorithm to certify IP spoofing protection of firewall rulesets. It fits the following into the big picture of this thesis: Once the spoofing protection of a firewall configuration is verified, interfaces and IP addresses can be related. Overall, it allows to identify and name entities by their IP address. This will be necessary to construct service matrices in the following chapter.

13.1 Introduction

In firewalls, it is good practice and sometimes an essential security feature to prevent IP address spoofing attacks [214, 215, 216, 33, 135, 138]. The Linux kernel offers reverse path filtering [217], which can conveniently prevent such attacks. However, in many scenarios (e.g., asymmetric routing [202], failover [216], zone-spanning interfaces, or multihoming [215]), reverse path filtering must be disabled or is too coarse-grained (e.g., for filtering special purpose addresses [218]). In these cases, spoofing protection has to be provided by the firewall which is configured by the administrator. Unfortunately, writing firewall rules is prone to human error, a fact known for over a decade [7].

For Linux netfilter/iptables firewalls [23], we present an algorithm to certify spoofing protection of a ruleset. It provides the following contributions; a unique set of features in their combination:

- It is formally and machine-verifiably proven sound with Isabelle/HOL.
- It supports the largest subset of iptables features compared to any other firewall analysis system.
- It was tested on the largest (publicly-available) firewall which was ever analyzed in academia.
- It terminates within a second for thousands of rules.

We chose iptables because it provides one of the most complex firewall semantics widely deployed [13, 22]. Of course, our algorithm is also applicable to similar or less complex (e.g., Cisco PIX) firewall systems.

13.2 Related Work

There are several popular static firewall analysis tools. The Firewall Policy Advisor [219] discovers inconsistencies between pairs of rules (e.g., one rule completely overshadowing the other) in distributed firewall setups. A similar tool is FIREMAN [13]. It can discover inconsistencies within rulesets or between firewalls and verify setups against administrator-defined policies. To represent sets of packets, Binary Decision Diagrams (BDDs) are used. It does not support matching on interfaces in a rule. Since interfaces can be strings of arbitrary length, they may not be ideal for the encoding in BDDs and adding support to FIREMAN might be complicated or deteriorate its performance.

Margrave [16] can be used to query (distributed) firewalls. It is well suited to troubleshoot and to debug firewall configurations or to show the impact of ruleset edits. For certain queries, it can also find scenarios, e.g., it can show concrete packets which violate a security policy. This scenario finding is sound but not complete. Its query language as well as Margrave's internal implementation is based on first-order logic. A similar tool (relying on BDDs), with a different query language and focus on complete networks, is ConfigChecker [220, 170]. ITVal [135] can also be used to query firewalls. It can also describe the firewall in terms of equivalent IP address spaces [136], which does not require the administrator to pose specific queries.

None of these tools can directly verify spoofing protection. Nor are the tools themselves formally verified, which limits confidence in their results. In addition, the tools only support a limited subset of real-world firewalls (cf. Chapter 12). If the firewall under analysis exceeds this feature set, the tools either produce erroneous results or cannot continue.

Jeffrey and Samak [200] analyze the complexity of firewall analysis and conclude that most questions (for variable packet models) are NP-complete. They show that SAT solvers usually outperform BDD-based implementations.

13.3 Mathematical Background

Iptables Semantics To define and prove correctness of our algorithm, one must first agree on the semantics (i.e., behavior) of an iptables firewall. We rely on our semantics specified in Chapter 12. The semantics are well-suited for the tables where spoofing protection is usually implemented: the `filter` table or the `raw` table; given that only the following actions are used: `Accept`, `Drop`, `Reject`, `Log`, calling to and `Return`ing from user-defined chains, as well as the "empty" action.

In the semantics, matching a packet against match conditions (e.g., IP source or destination addresses) is mathematically defined with a "magic oracle" which understands all possible matches. Obviously, semantics with a "magic oracle" cannot be expressed in terms of executable code. Nevertheless, the algorithm we present in this chapter is both executable and proven sound w.r.t. these semantics.

13.4 Spoofing Protection – Mathematically

To define spoofing protection, two data sets are required: The firewall ruleset *rs* and the IP addresses assignment *ipassmt*. *ipassmt* is a mapping from interfaces to an IP address range. Usually, it can be obtained by `ip route`. We will write *ipassmt*[*i*] to get the corresponding IP range of interface *i*. For the following examples, we assume

$$\mathit{ipassmt} = [\texttt{eth0} \mapsto \{192.168.0.0/24\}]$$

Definition 15 (Spoofing Protection)**.** A firewall ruleset provides *spoofing protection* if for all interfaces *i* specified in *ipassmt*, all packets from *i* which are accepted by the ruleset have a source IP address contained in *ipassmt*[*i*].

For example, using pseudo iptables syntax (which omits the chain), the following ruleset implements spoofing protection:

```
-i eth0 ! --src 192.168.0.0/24 -j DROP
-j ACCEPT
```

Spoofing Protection with Unknowns iptables supports numerous match conditions and new ones may be added in future. It is practically infeasible to support all match conditions in a tool (cf. Chapter 12). As we do not want our algorithm to abort if an unknown match occurs, we will refine Definition 15 to incorporate unknown matches. This is motivated by the following examples.

We assume that `--foo` is a match condition which is unknown to us. Therefore, we cannot guarantee that

```
-i eth0 ! --src 192.168.0.0/24 --foo -j DROP
-j ACCEPT
```

implements spoofing protections since `--foo` could prevent certain spoofed packets from being dropped. Also, the following ruleset might neither implement spoofing protection since `--foo` might allow spoofed packets:

```
--foo -j ACCEPT
-i eth0 ! --src 192.168.0.0/24 -j DROP
-j ACCEPT
```

However, the following ruleset definitely implements spoofing protection; Independently of the meaning of `--foo` and `--bar`, it is guaranteed that no spoofed packets are allowed:

```
--foo -j DROP
-i eth0 ! --src 192.168.0.0/24 -j DROP
--bar -j ACCEPT
```

This motivates Definition 16.

Definition 16 (Certifiable Spoofing Protection)**.** A firewall ruleset provides *spoofing protection* if for all interfaces i specified in *ipassmt*, all packets from i which are *potentially* accepted by the ruleset have a source IP address in the IP range of i.

This new definition is stricter than the original one: Definition 16 implies Definition 15[1], which directly follows from Theorem 13. Therefore the new definition is *sound* and can be used to prove that the last example implements spoofing protection.

However, depending on the meaning of `--foo`, some of the previous examples might also implement spoofing protection. This cannot be shown with Definition 16, so the new definition is not *complete*. However, as long as we anticipate unknown matches to occur, it is impossible to obtain completeness.

13.5 Spoofing Protection – Executable

A straight forward spoofing protection proof of a firewall ruleset using Definition 16 would require iterating over all packets, which is obviously infeasible. We present an efficient executable algorithm to certify spoofing protection. It does not rely on BDDs or an external SAT/SMT solver.

We assume the ruleset to be certified is preprocessed. For this, we rely on the semantics-preserving ruleset simplification of Chapter 12 which rewrites a ruleset to a semantically equivalent ruleset where only `Accept` and `Drop` actions occur. A preprocessed ruleset always has an explicit deny-all or allow-all rule at the end; it can never be empty. This rule corresponds to a chain's default policy.

It would also be possible writing the algorithm for the simple firewall model which will be presented in Chapter 14. We chose not to do so since the simple model does not support negated interfaces, which are essential for spoofing protection. The big picture of the thesis is as follows: Once we have verified spoofing protection, we can rewrite matches on interfaces with matches in IP addresses. Thus, we need to check for spoofing protection first and can afterwards translate to a simpler firewall model which does not support matches on (negated) interfaces. These details are presented in the following chapter.

[1]approximating-imp-booloan-semantics-nospoofing

$$\begin{aligned}
&\mathsf{sp}\ []\ A\ D &&= (A \setminus D) \subseteq \bigcup ipassmt[i] \\
&\mathsf{sp}\ ((m, \texttt{Accept}) :: rs)\ A\ D = \\
&\qquad\qquad \mathsf{sp}\ rs\ (A \cup \{ip \mid \exists p \text{ from interface } i \text{ with src address } ip.\ \mathsf{match}\ \gamma\ m\ p\})\ D \\
&\mathsf{sp}\ ((m, \texttt{Drop}) :: rs)\ A\ D &&= \\
&\qquad \mathsf{sp}\ rs\ A\ \ (D \cup (\{ip \mid \forall p \text{ from interface } i \text{ with src address } ip.\ \mathsf{match}\ \gamma\ m\ p\} \setminus A))
\end{aligned}$$

Figure 13.1: Our algorithm to certify spoofing protection.

We call our algorithm sp ("spoofing protection"). It certifies spoofing protection for one interface i in *ipassmt*. Using sp to certify all $i \in ipassmt$ for a ruleset implies spoofing protection according to Definition 16.

We assume the global, static, and fixed parameters of sp are an interface i and the *ipassmt*. Then, sp has the following type signature:

$$rule\ list\ \Rightarrow\ ipaddr\ set\ \Rightarrow\ ipaddr\ set \Rightarrow \mathbb{B}$$

The first parameter (*rule list*) is the preprocessed firewall ruleset. To recapitulate, a rule is a tuple (m, a), where m is the match condition and a the action. The action is either `Accept` or `Drop`. For a packet p, there is a predicate $\mathsf{match}\ \gamma\ m\ p$ which tells whether the packet p matches the match condition m. The second parameter (*ipaddr set*) is the set of potentially allowed source IP addresses for i. The third parameter (*ipaddr set*) is the set of definitely denied source IP addresses for i. The last parameter ($\mathbb{B}$) is a Boolean, which is true if spoofing protection could be certified.

Before we present the algorithm, we first present its correctness theorem (which will be proven later).[2]

Theorem 18 (sp sound)**.** *For any ruleset rs, if*

$$\forall i \in ipassmt.\ \ \mathsf{sp}\ rs\ \{\}\ \{\}$$

then rs provides spoofing protection according to Definition 16.

The algorithm sp, presented in Figure 13.1, is implemented recursively. It iterates over the firewall ruleset.

The base case is for an empty ruleset. Here, A and D are the set of allowed/denied source IP addresses. The firewall provides spoofing protection if the set of potentially allowed sources minus the set of definitely denied sources is a subset of the allowed IP range.

The two recursive calls collect these sets A and D. If the rule is an `Accept` rule, the set A is extended with the set of sources possibly accepted in this rule. If the rule is a `Drop` rule, the set D is extended with the set of sources definitely denied in this rule, excluding any sources which were potentially accepted earlier.

As Theorem 18 already states, sp can be started with any ruleset and the empty set for A and D.

[2]no-spoofing-iface, and ultimately spoofing protection according to Definition 15: no-spoofing-executable-set

We will now describe how the *ipaddr set* operations are implemented. In general, we symbolically represent a set of IP addresses as a set of IP range intervals. Since IP ranges are commonly expressed in CIDR notation (e.g., $a.b.c.d/n$), the interval datatype proves to be very efficient.

Next, the set $\{ip \mid \exists p \text{ from interface } i \text{ with src address } ip.\ \mathsf{match}\ \gamma\ m\ p\}$ requires executable code. Obviously, a straight-forward implementation which tests the existence of any packet is infeasible. We provide an over-approximation for this set, the correctness proof confirms that this approach is sound. First we check that i matches all input interfaces specified in the match expression m. If this is not the case, the set is obviously empty. Otherwise, we collect the intersection of all matches on source IPs in m. If no source IPs are specified in m, then m matches any source IP and we return the universe.

The set $\{ip \mid \forall p \text{ from interface } i \text{ with src address } ip.\ \mathsf{match}\ \gamma\ m\ p\}$ can be computed similarly. However, we need to return an under-approximation here. First, we check that i matches, otherwise the set is empty. Next, we remove all matches on input interfaces and source IPs from m. If the remaining match expression is not unconditionally true, then we return the empty set. Otherwise, we return the intersection of all source IP addresses specified in m, or the universe if m does not restrict source IPs.

Note that after preprocessing we always have an explicit allow-all or deny-all rule at the end of the firewall ruleset. Thus, $A \cup D$ will always hold the universe after consuming the last rule.

13.6 Evaluation – Mathematically

We outline the main idea of the correctness proof. Since sp operates on a fixed interface, we define certifiable spoofing protection for a fixed interface i. Showing Definition 17 for all interfaces is equivalent to Definition 16.

Definition 17.

$$\forall p \in \{p \mid p \text{ from } i \text{ and potentially accepted by the firewall}\}.\ \ \mathsf{src\text{-}ip}\ p \in \bigcup ipassmt[i]$$

The correctness proof of sp is done by induction over the firewall ruleset. Theorem 18 does not lend itself to induction, since it features two empty sets which would generate unusable induction hypotheses. To obtain a strong induction hypothesis, we generalize. The ruleset is split into two parts: rs_1 and rs_2. We assume that the algorithm correctly iterated over rs_1. For this lemma, we use the following notation:

$$A_{\text{exact}} = \{ip \mid \exists p.\ p \text{ from } i \text{ with src } ip \text{ and accepted by } rs_1\}$$
$$D_{\text{exact}} = \{ip \mid \forall p.\ p \text{ from } i \text{ with src } ip \text{ and denied by } rs_1\}$$

Lemma 13. *If* $A_{\text{exact}} \subseteq A$ *and* $D \subseteq D_{\text{exact}}$ *and* $\mathsf{sp}\ rs_2\ A\ D$ *then Definition 17 holds for* $rs_1 ::: rs_2$

Proof. The proof is done by induction over rs_2 for arbitrary rs_1, A, and D.

Base case (i.e., $rs_2 = []$): From $\mathsf{sp}\ []\ A\ D$ we conclude $(A \setminus D) \subseteq \bigcup ipassmt[i]$. Since A is an over-approximation and D an under-approximation: $A_{\text{exact}} \setminus D_{\text{exact}} \subseteq A \setminus D$. Since no IP address can be both accepted and denied we get $A_{\text{exact}} \setminus D_{\text{exact}} = A_{\text{exact}}$. From transitivity we conclude $A_{\text{exact}} \subseteq \bigcup \mathit{ipassmt}[i]$, which implies spoofing protection for that interface according to Definition 17.

The two induction steps (one for `Accept` and one for `Drop` rules) follow from the induction hypothesis. The over- and under- approximations were carefully constructed, such that the subset relations continue to hold. The executable implementations of these sets also respect the subset relation; hence, the induction hypothesis solves these cases. □

Proof of Theorem 18. Lemma 13 can be instantiated where rs_1 is the empty ruleset and A and D are the empty set. For this particular choice, it is easy to see that the preconditions hold. Thus, for any rs, we conclude `sp` rs $\{\}$ $\{\}$ implies Definition 17. Since this holds for arbitrary interfaces, we conclude Definition 16. □

Thus, our algorithm is proven *sound*. This means, if the algorithm certifies a ruleset, then this ruleset is guaranteed to implement spoofing protection.

Note that our algorithm only certifies; debugging a ruleset in case of a certification failure remains manual. To debug, the proof of Lemma 13 suggest to consider the first rule where $(A \setminus D) \subseteq \bigcup \mathit{ipassmt}[i]$ is violated.

The algorithm is *not complete*. This means, there may be rulesets which implement spoofing protection but cannot be certified by the algorithm. This is bought by the approximations and by the support for unknown match conditions. For example, the following ruleset cannot be certified:

```
-i eth0 ! --src 192.168.0.0/24 --foo -j DROP
-i eth0 ! --src 192.168.0.0/24 ! --foo -j DROP
-j ACCEPT
```

This is a reasonable decision for a completely unknown `--foo`, since it might update an internal state and the mathematical equation "`foo` $\vee \neg$ `foo` $=$ `True`" may not hold. However, if `foo` is replaced by the known and stateless match condition `--protocol tcp`, the ruleset can be shown to correctly implement spoofing protection. The algorithm, however, cannot certify it, since it does not track this match condition. However, this is a made-up and bad-practice example, and we never encountered such special cases in any real-world ruleset. The evaluation —in which vast amounts of unknowns occurred— shows that the algorithm certified all rulesets which included spoofing protection and correctly failed only for those rulesets which did not (correctly) implement it. Thus, the incompleteness is primarily a theoretical limitation.

13.7 Evaluation – Empirically

Often, firewalls start with an `ESTABLISHED` rule. A packet can only match this rule if it belongs to a connection which has been accepted by the firewall previously. Hence, the `ESTABLISHED` rule does not contribute to the access control policy for connection setup enforced by the firewall [13]. Likewise, spoofed packets can only be allowed by the `ESTABLISHED` rule if they are allowed by any of the subsequent ACL rules. Therefore, as done in the previous chapter (Section 12.5.4), we either exclude this rule from our analysis or only consider packets of state `NEW`.

We tested the algorithm on several real-word rulesets [25]. Most of them either did not provide spoofing protection or had an obvious spoofing protection and could thus be certified. In this Section, we present the results of certifying four rulesets. The first two rulesets [221, 222] are simple examples and were found as the first two results of a goolge query for "`iptables spoofing`". They primarily serve as example. Afterwards, we will put our main focus in this section on a large and interesting real-world ruleset.

First of all, for all rulesets, our algorithm was extremely fast: Once the ruleset is preprocessed (few seconds) the certification algorithm only takes fractions of seconds for rulesets with several thousand rules. We omit a detailed performance evaluation since these orders of magnitude are sufficient for a static/offline analysis system. Certification runs of our algorithm, in particular for the larger rulesets, were usually faster than reloading the ruleset on the firewall system itself.

13.7.1 Firewall Builder Documentation

```
*filter
:INPUT ACCEPT [16:6016]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [16:6016]
:In_RULE_0 - [0:0]
-A INPUT -s 192.0.2.1/32 -i eth0 -j In_RULE_0
-A INPUT -s 192.168.1.1/32 -i eth0 -j In_RULE_0
-A INPUT -s 192.168.1.0/24 -i eth0 -j In_RULE_0
-A FORWARD -s 192.0.2.1/32 -i eth0 -j In_RULE_0
-A FORWARD -s 192.168.1.1/32 -i eth0 -j In_RULE_0
-A FORWARD -s 192.168.1.0/24 -i eth0 -j In_RULE_0
-A In_RULE_0 -j LOG --log-prefix "RULE 0 -- DENY " --log-level 6
-A In_RULE_0 -j DROP
COMMIT
```

Figure 13.2: Spoofing Protection Ruleset from the Firewall Builder Documentation

Our first example is taken from the Firewall Builder [193] user guide [221, 14.2.6. Anti-spoofing rules]. Their example defines the IP addresses of the firewall as 192.168.1.1 and 192.0.2.1 and network behind as 192.168.1.0/24. Consequently, on the external-facing interface (`eth0`), if any of these IP addresses occurs, it must be spoofed. For spoofing protection, these IP addresses must not occur. Let UNIV denote the universe of all IPv4 addresses, then $\mathit{ipassmt} = [\texttt{eth0} \mapsto \mathsf{UNIV} \setminus \{192.168.1.1,\ 192.0.2.1,\ 192.168.1.0 \mathrel{..} 192.168.1.255\}]$. Our implementation allows to conveniently express this as `eth0 = all_but_those_ips [192.168.1.1, 192.0.2.1, 192.168.1.0/24]`. Aside, our tool warns that the *ipassmt* is not complete, i.e., it does not cover the complete IPv4 address space. This is mainly because the firewall only defines spoofing protection on the external-facing interface and does not consider (in this example) outgoing spoofing from the internal interface. The firewall rules are shown in Figure 13.2 and our tool immediately certifies spoofing protection for `eth0` for both the `FORWARD` and the `INPUT` chain.

Spoofing protection can be considered a rather low-level property of a firewall. Considering the big picture of this thesis, once we have certified spoofing protection, we can abstract over such low-level properties. For example for this ruleset, we may now assume that we have spoofing protection. If we now simplify the ruleset under this assumption, the ruleset is effectively only one rule: Accept all. This high-level view and the respective simplifications which are built on top of spoofing protection will be presented in the following chapter.

13.7.2 Blog Post

```
*filter
:INPUT ACCEPT [77:24937]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [78:25303]
-A INPUT -s 202.54.10.20/32 -i eth1 -j DROP
-A INPUT -s 192.168.1.0/24 -i eth1 -j DROP
-A INPUT -s 0.0.0.0/8 -i eth1 -j DROP
-A INPUT -s 127.0.0.0/8 -i eth1 -j DROP
-A INPUT -s 10.0.0.0/8 -i eth1 -j DROP
-A INPUT -s 172.16.0.0/12 -i eth1 -j DROP
-A INPUT -s 192.168.0.0/16 -i eth1 -j DROP
-A INPUT -s 224.0.0.0/3 -i eth1 -j DROP
-A OUTPUT -s 202.54.10.20/32 -o eth1 -j DROP
-A OUTPUT -s 192.168.1.0/24 -o eth1 -j DROP
-A OUTPUT -s 0.0.0.0/8 -o eth1 -j DROP
-A OUTPUT -s 127.0.0.0/8 -o eth1 -j DROP
-A OUTPUT -s 10.0.0.0/8 -o eth1 -j DROP
-A OUTPUT -s 172.16.0.0/12 -o eth1 -j DROP
-A OUTPUT -s 192.168.0.0/16 -o eth1 -j DROP
-A OUTPUT -s 224.0.0.0/3 -o eth1 -j DROP
COMMIT
```

Figure 13.3: Spoofing Protection Ruleset from "Linux Iptables Avoid IP Spoofing And Bad Addresses Attacks" Blog Post

The next example is taken from a Linux blog which explains spoofing protection with iptables [222]. With 202.54.10.20 beeing the IP address of the machine which should be protected, the author lists the following IP ranges as "bad": 0.0.0.0/8, 127.0.0.0/8, 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, 224.0.0.0/3, and the "LAN IP range" 192.168.1.0/24. Note that the last IP range is already included in the fifth IP range, which causes no problem to our tool. The author defines `eth1` as the Internet-facing interface. As in the previous example, we can define *ipassmt* as `eth0 = all_but_those_ips [`*"bad" IP ranges*`]`. The firewall rules are generated by a shell script, ultimately, the rules as shown in Figure 13.3 are generated. Our tool immediately certifies spoofing protection for the `INPUT` chain.

Additionally, the shell script loads the very same rules with `-o eth1` instead of `-i eth1` for the `OUTPUT` chain. Therefore, we also want to check spoofing protection for the `OUTPUT` chain. For outbound spoofing protection, it must be assured that the machine uses only its assigned IP address, i.e., *ipassmt* = [`eth1` $\mapsto$ {202.54.10.20}]. The check for spoofing protection immediately fails. The `OUTPUT` chain prevents that the machine spoofs with the set of "bad" IP ranges. However, it does not prevent that the machine spoofs its source IP address with some valid IP address which is not defined as "bad". More severe, as the very first rule of the `OUTPUT` chain, the ruleset prevents outgoing packets from the only valid IP address: The address of the machine itself. This means the machine is not able to send any (non-spoofed) packets to the Internet. Reading the comment section of the blog post reveals that three users also noticed that their machine loses outbound Internet connectivity if they run the shell script.

13.7.3 Firewall of Our Lab

We present the certification of a firewall with about 4800 rules, connecting about 20 VLANs. Every VLAN has its own interface. An excerpt of the ruleset and the *ipassmt* is shown in Figure 13.4.

Trying the certification, it immediately fails. Responsible was a work-around rule which should only have existed temporarily but was forgotten. This rule is now on the administrator's "things to do the right way" list and we exclude it for further evaluation.

Certifying spoofing protection for the first VLAN interface succeeds instantly. However, trying to certify all other VLANs fails. The reason is an error in the ruleset. For every VLAN n, the firewall defines three custom chains: `mac_`n, `ranges_`n, and `filter_`n. The `mac_`n chain verifies that for hosts with registered MAC addresses and static IP addresses, nobody (with a different MAC address) steals the IP address. This chain is primarily to avoid manual IP assignment errors. Next, the `ranges_`n chain should prevent outgoing spoofing. Finally, the `filter_`n chain allows packets for certain registered services. The main error was that a spoofed packet from VLAN m could be accepted by `filter_`n before it had to pass the `ranges_`m check. The discovery of this error also discovered that the `mac_`m chains were not working reliably. We verified these findings by sending and receiving such spoofed packet via the real firewall.

Finally, we fixed the firewall by moving all `mac_`n and `ranges_`n chains before any `filter_`n chains. The certification for all but one[3] internal VLAN interfaces succeeds.

Next, the interfaces attached to the Internet are certified. The IP address range was defined as the universe of all IPs, excluding the IPs owned by the institute. Here, certification failed in a first run. Responsible were some ssh rate limiting rules. These rules were originally designed to prevent too many outgoing ssh connections. However, since spoofing protection did not apply to them, an attacker could exploit them for a DOS attack against the internal network: The attacker floods the firewall with ssh TCP SYN packets with spoofed internal addresses. This exhausts the ssh limit of the internal hosts and it is no longer possible for them to establish new ssh connections. This flaw was fixed and certification subsequently succeeds.

The improved and certified firewall ruleset is now in production use.

13.7.4 Remote Firewall

After this, another administrator got interested and wanted to implement spoofing protection for his firewall. To complicate matters, he was in Japan and the firewall in Germany. It was a key requirement that he would not lock himself out. Our tool could certify both: His proposed changes to the firewall correctly enforce spoofing protection and he will not lose ssh access. To provide a *sound* guarantee for the latter, we applied the same idea as in our algorithm, but in reverse: the ruleset is abstracted to a stricter version (i.e., a version that blocks more packets) and we consequently certify that it still allows `NEW` and `ESTABLISHED` ssh packets from the Internet.

[3]This one VLAN where the certification fails is only for internal testing purposes and deliberately features no spoofing protection

13.8 Conclusion

We present an easy-to-use algorithm. It is fast enough to be run on every ruleset update. It discovered real problems in a large, production-use firewall. Both, the theoretical algorithm as well as the executable code are proven sound, hence if the algorithm certifies a firewall, the ruleset is *proven* to implement spoofing protection correctly.

Old rules (without spoofing protection):

```
-A FORWARD -p tcp -m state --state NEW          ↩
              -m tcp --dport 22 ...             ↩
              -m recent --update ...            ↩
              --name ratessh --rsource          ↩
              -j LOG_RECENT_DROP
...
-A FORWARD -s 131.159.14.0/25 -i eth1.96 -j mac_96
-A FORWARD -i eth1.96 -j ranges_96
-A FORWARD -d 131.159.14.0/25 -o eth1.96        ↩
              -j filter_96
-A FORWARD -d 224.0.0.0/4 -o eth1.96 -j filter_96
-A FORWARD -s 131.159.14.128/26 -i eth1.108     ↩
              -j mac_108
-A FORWARD -i eth1.108 -j ranges_108
-A FORWARD -d 131.159.14.128/26 -o eth1.108     ↩
              -j filter_108
-A FORWARD -d 224.0.0.0/4 -o eth1.108 -j filter_108
...
-A mac_96 -s 131.159.14.92/32                   ↩
             -m mac --mac-source ... -j RETURN
-A mac_96 -s 131.159.14.92/32 -j DROP
...
-A ranges_96 -s 131.159.14.0/25 -j RETURN
-A ranges_96 -j LOG_DROP
...
```

Improved rules (with spoofing protection):

```
-A FORWARD -i eth1.110 -j NOTFROMHERE
-A FORWARD -p tcp -m state --state NEW          ↩
              -m tcp --dport 22 ...             ↩
              -m recent --update ...            ↩
              --name ratessh --rsource          ↩
              -j LOG_RECENT_DROP
...
-A FORWARD -s 131.159.14.0/25 -i eth1.96 -j mac_96
-A FORWARD -i eth1.96 -j ranges_96
-A FORWARD -s 131.159.14.128/26 -i eth1.108     ↩
              -j mac_108
-A FORWARD -i eth1.108 -j ranges_108
...
-A FORWARD -d 131.159.14.0/25 -o eth1.96        ↩
              -j filter_96
-A FORWARD -d 224.0.0.0/4 -o eth1.96 -j filter_96
-A FORWARD -d 131.159.14.128/26 -o eth1.108     ↩
              -j filter_108
-A FORWARD -d 224.0.0.0/4 -o eth1.108 -j filter_108
...
-A mac_96 -s 131.159.14.92/32                   ↩
             -m mac --mac-source ... -j RETURN
-A mac_96 -s 131.159.14.92/32 -j DROP
...
-A ranges_96 -s 131.159.14.0/25 -j RETURN
-A ranges_96 -j LOG_DROP
...
-A NOTFROMHERE -s 131.159.14.0/23 -j LOG_DROP
...
```

Specified *ipassmt*:

```
eth0 = [0.0.0.0-255.255.255.255]
eth1.96 = [131.159.14.3/25]
eth1.108 = [131.159.14.129/26]
eth1.109 = [131.159.20.11/24]
eth1.110 = all_but_those_ips [
  131.159.14.0/23,
  131.159.20.0/23,
  192.168.212.0/23,
  188.95.233.0/24,
  188.95.232.192/27,
  188.95.234.0/23,
  192.48.107.0/24,
  188.95.236.0/22,
  185.86.232.0/22
  ]
eth1.116 = [131.159.15.131/26]
eth1.152 = [131.159.15.252/28]
eth1.171 = [131.159.15.2/26]
eth1.173 = [131.159.21.252/24]
eth1.1010 = [131.159.15.227/28]
eth1.1011 = [131.159.14.194/27]
eth1.1012 = [131.159.14.238/28]
eth1.1014 = [131.159.15.217/27]
eth1.1016 = [131.159.15.66/26]
eth1.1017 = [131.159.14.242/28]
eth1.1111 = [192.168.212.4/24]
eth1.97 = [188.95.233.2/24]
eth1.1019 = [188.95.234.2/23]
eth1.1020 = [192.48.107.2/24]
eth1.1023 = [188.95.236.2/22]
eth1.1025 = [185.86.232.2/22]
eth1.1024 = all_but_those_ips [
  131.159.14.0/23,
  131.159.20.0/23,
  192.168.212.0/23,
  188.95.233.0/24,
  188.95.232.192/27,
  188.95.234.0/23,
  192.48.107.0/24,
  188.95.236.0/22,
  185.86.232.0/22
  ]
```

Figure 13.4: Firewall rules without spoofing protection and improved rules

Chapter 14

fffuu: Verified iptables Firewall Analysis

This chapter is an extended version of the following paper [45]:

- Cornelius Diekmann, Julius Michaelis, Maximilian Haslbeck, and Georg Carle, *Verified iptables Firewall Analysis.* In IFIP Networking 2016, Vienna, Austria, May 2016.

The following major improvements were added:

- Performance improvement of about 10x; correctness proven.
- Identified and fixed a bug in the port matching semantics, large rework of the theory.
  - Generic result: Bug either affects any firewall analysis tool or tool is limited in its expressiveness (Section 14.4.3).
- Added ultimate service matrix correctness theorem (Theorem 22).
- Performed the evaluation again several times (performance measurements, identify effects of bug).
- Stand-alone haskell tool *fffuu* (Section 14.7), which gives another few orders of magnitude speedup.
- IPv6 support for the translation, simple firewall, and presented algorithms. Featuring probably the first formally verified IPv6 RFC 5952 [223] pretty printer [39].
- A formalization of longest prefix routing [42] to enable output port rewriting.
- Additional evaluation.

Statement on author's contributions For the original paper, the author of this thesis provided major contributions for the overall idea. He invented and implemented the simple firewall model and provided major contributions for the ideas, realization, implementation, and proof of the translation from the real-world iptables firewall model to the simplified model. For this, he also enhanced the model with state. He researched related work, collected the data, and conducted the evaluation. He is the main author of the Haskell tool *fffuu*.

Maximilian Haslbeck implemented (as a part of his Bachelor's thesis) a first prototype of the IP address space partitioning algorithm as well as the service matrix algorithm and showed its correctness. The author of this thesis later on improved the performance of the algorithm and showed the final service matrix theorem (based on Haslbeck's previous work). Julius Michaelis provided major contributions for the CIDR split algorithm and its correctness proof and contributed to the word interval type which we use to store and operate on IPv4 addresses, which was generalized by the author of this thesis to support arbitrary machine words which is necessary for IPv6 support. Julius Michaelis formalized longest prefix routing and implemented the output port rewriting. All other improvements with regard to the paper are the work of the author of this thesis.

Abstract In this chapter, we present our fully verified iptables firewall ruleset analysis framework. The core analysis capability is the translation of an iptables ruleset to a policy, i.e., we present a method to compute graphs which partition the complete IPv4 and IPv6 address space and show the allowed accesses between partitions for a fixed service. Such a graph can be visualized for manual inspection or verified with the methods of Part I. This allows uncovering scenario-specific firewall errors. Internally, we are working with a simplified firewall model and a core contribution is the translation of complex real-world iptables firewall rules into this model. A real-world evaluation demonstrates the applicability of our tool.

14.1 Introduction

Firewall rulesets are inherently difficult to manage. It is a well-studied but unsolved problem that many rulesets show several configuration errors [7, 8, 44]. Tools were designed to help uncover configuration errors and verify a ruleset. In this thesis, we focus on tools for the static of rulesets. They have the benefit that the analysis can be carried out offline, without any negative effects on the network. In contrast to testing, static analysis can achieve a full coverage (e.g., the results hold for all packets) and thus are able to uncover all errors and give strong guarantees for the absence of certain classes of errors. However, in practice, static ruleset analysis tools fail for various reasons: They do not support the vast amount of firewall features, they require the administrator to learn a complex query language which might be more complex than the firewall language itself, the analysis algorithms do not scale to large firewalls, and the output of the verification tools itself cannot be trusted.

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

- A simple firewall model, designed for mathematical beauty and ease of static analysis (Section 14.3)
- A series of translation steps to translate real-world firewall rulesets into this simple model (Section 14.4)
- Static and automatic firewall analysis methods, based on the simple model, featuring

  - IP address space partitioning (Section 14.5)
  - Minimal service matrices (Section 14.6)
- Full formal and machine-verifiable proof of correctness (Section Availability)
- Evaluation on large real-world data set (Section 14.8)

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

In contrast to work which considers generic firewall configuration errors [7, 8], the goal of our IP address space partitioning and service matrices is to uncover scenario-specific misconfiguration. For example, our tool will not warn that inbound HTTP is allowed to more than 256 IPs, however, it will generate an overview of which IPs may establish and receive HTTP connections. This overview can then be inspected by an administrator. For a large webhoster, having more than 256 IPs where inbound HTTP is allowed may be perfectly fine, whereas in an airplane, even the possibility of a single HTTP connection from the coffee machine to the avionics may raise concern.[1]

We outline related work in Section 14.2. The real-world and simplified firewall models are presented in Section 14.3. We detail on the translation between these models in Section 14.4. Afterwards, we present the IP address space partitioning (Section 14.5) and service matrices (Section 14.6). In Section 14.8, we evaluate our algorithms on a large set of real-world iptables rulesets.

14.2 Related Work

As introduced in Section 12.3, we call the features a firewall can use to match on packets *primitives*. For example, among others, iptables supports the following primitives: src IP address, layer 4 port, inbound interface, conntrack state, entries and limits in the `recent` list, . . . While our previous work in Chapter 12 focused on the high-level semantics of iptables, this chapter focuses on primitives.

Popular tools for static firewall analysis include FIREMAN [13], Capretta et al. [224], and the Firewall Policy Advisor [219]. They support the following primitives: IP addresses, ports, and protocol. This corresponds to (a subset of) our simple firewall model, hence, these tools would not be applicable to most firewalls from our evaluation. The tools focus on detecting conflicts between rules and can consequently not offer service matrices.

The work most similar to our IP address space partitioning is ITVal [135]: It supports a large set of iptables features and can compute an IP address space partition [136]. Unfortunately, ITVal only supports IPv4, is not formally verified, and its implementation has several errors. For example, ITVal produces spurious results if the number of significant bits in IP addresses in CIDR notation [203] is not a multiple of 8. This appears to be a coding error in the translation to ITVal's internal data structure which is easy to fix for

[1]This is a made up example.

non-negated IP address ranges. ITval does not consider logical negations which may occur when `RETURN`ing prematurely from user-defined chains, which leads to wrong interpretation of complement sets. It does not support abstracting over unknown primitives but simply ignores them, which also leads to spurious results. Those are problems of the internal logic and are thus non-trivial to repair. For rulesets with more than 1000 rules, ITVal requires tens of gigabytes of RAM. It does not support IPv6. Finally, ITVal neither proves the soundness nor the minimality of its IP address range partitioning. Nevertheless, ITVal demonstrates the need for and the use of IP address range partitioning and has demonstrated that its implementation works well on rulesets which do not trigger the aforementioned errors. Building on the ideas of ITVal (but with a different algorithm), we overcome all presented issues.

Exodus [212] translates existing device configurations to a simpler model, similar to our translation step. It translates router configurations to a high-level SDN controller program, which is implemented on top of OpenFlow. Exodus supports many Cisco IOS features. The translation problem solved by Exodus is comparable to this paper's problem of translating to a simple firewall model: OpenFlow 1.0 only supports a limited set of features (comparable to our simple firewall) whereas IOS supports a wide range of features (comparable to iptables); A complex language is ultimately translated to a simple language, which is the 'hard' direction.

Complementary to our verification tool, and well-suited for debugging, is Margrave [16]. It can be used to query firewalls and to troubleshoot configurations or to show the impact of ruleset edits. Margrave can find scenarios, i.e., it can show concrete packets which violate a security policy. Our framework does not show such information. Margrave's query language (which a potential user is required to learn to use this tool) is based on first-order logic.

14.3 Firewall Semantics

First, we present a very simple firewall model. This model was designed to feature nice mathematical properties but it is too simplistic to mirror the real world. Afterwards, we will compare it to our model for real-world firewalls of Chapter 12. Section 14.4 will show how rulesets can be translated between these two models. This preprocessing step converts firewall rulesets from the real-world model to the simple model, which greatly simplifies all future static firewall analysis.

14.3.1 Simple Firewall

We will write firewall rules as tuple $(m,\ a)$, where m is a match expression and a is the action the firewall performs if m matches for a packet. The firewall has two possibilities for the filtering decision: it may accept (✓) the packet or deny (✗) the packet. There is also an intermediate state (?) in which the firewall did not come to a filtering decision. Note that iptables firewalls always have a default policy and the ? case cannot occur as final decision.

The simple model is a simple recursive function. The first parameter is the ruleset the firewall iterates over, the second parameter is the packet.

$$\begin{aligned}
&\textsf{simple-fw}\ \ [\,] && p = \ ?\\
&\textsf{simple-fw}\ \ ((m,\ \texttt{Accept}) :: rs) && p = \textbf{if}\ \textsf{match}\ m\ p\ \textbf{then}\ ✓\ \textbf{else}\ \textsf{simple-fw}\ rs\ p
\end{aligned}$$

$$\mathsf{simple\text{-}fw}\ \ ((m,\ \mathtt{Drop}) :: rs)\quad p = \textbf{if}\ \mathsf{match}\ m\ p\ \textbf{then}\ \otimes\ \textbf{else}\ \mathsf{simple\text{-}fw}\ rs\ p$$

A function match tests whether a packet p matches the match condition m.[2] The match condition is an 7-tuple, consisting of the following primitives:

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

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

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

For example, we obtain an empty match (a match that does not apply to any packet) *iff* an end port is greater than the start port.[3] The match which matches any packet is constructed by setting the interfaces to "+", the ips to 0.0.0.0/0, the ports to 0:65535 and the protocol to Any.[4]

We require that all match conditions are well-formed, i.e., it is only allowed to match on ports (other than the universe 0:65535) if the protocol is tcp, udp, or sctp.

With this type of match expression, it is possible to implement a function conj which takes two match expressions m_1 and m_2 and returns exactly one match expression being the conjunction of both.[5]

Theorem 19 (Conjunction of two simple match expressions)**.**

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

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

The conj of two well-formed matches is again well-formed.[6]

The type of match expressions was carefully designed such that the conjunction of *two* match expressions is only *one* match expression. If features were added to the match expression, for example negated interfaces, this would no longer be possible. Of the features most commonly found in our iptables firewall rulesets [25], we only found that it would further be possible to add TCP flags to the match expression without violating the aforementioned conjunction property. Considering common features of firewalls in general [22], it would probably be possible to enhance the ICMP support of our model.

[2]Note that this is not the same function as in Chapter 12 since this simple match function does not require parameter γ. Basically, it already has the primitive matcher hard-coded in it.

[3]empty-match

[4]simple-match-any

[5]simple-match-and-correct

[6]simple-match-and-valid

14.3.2 Semantics of Iptables

We now repeat the important aspects of our model of a real-world iptables firewall from Chapter 12. Most firewall analysis is concerned with the access control rules of a firewall, therefore the model focuses on the `filter` table. This implies, packet modification (e.g., NAT, which must not occur in this table) is not considered in this work. The model supports the following common actions: `Accept`, `Drop`, `Reject`, `Log`, `Goto`, `Call`ing to and `Return`ing from user-defined chains, as well as the “empty” action. The model is defined as an inductive predicate with the following syntax:

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

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

We make use of these algorithms, in particular: An algorithm which unfolds all calls to and returns from user-defined chains and rewriting of further actions. This leaves a ruleset where only the following actions occur: `Accept` and `Drop`.[7] Thus, a large step for translating the real-world model to the simple firewall model is already accomplished. Translating the match expressions remains. The real-world model allows a match expression to be an arbitrary propositional logic expression. However, iptables only accepts match expressions in *negation normal form* (NNF). A Boolean formula is in NNF *iff* all occurring negations are on primitives, i.e., there are no nested negated expressions. For example, iptables can load `-s 10.0.0.0/8 ! -p tcp` but not `! (-s 10.0.0.0/8 -p tcp)`. However, such negated expressions may occur as a result of the unfolding algorithm. We already presented an algorithm to translate a ruleset to a ruleset where all match conditions are in NNF.[8] However, there is an additional constraint imposed by iptables, not solved by the algorithm: A primitive must only occur at most once. This problem will be addressed in this Chapter.

We have implemented a subset of γ, namely for all primitives supported by the simple firewall and some further primitives, detailed in Section 14.4. Chapter 12 provides an algorithm to abstract over all ‘unknown’ primitives which are not understood by our subset implementation of γ. This algorithm leads to an approximation of the ruleset. It can either be an overapproximation which results in a more permissive ruleset, or an underapproximation, which results in a stricter ruleset. For the sake of example, we will only consider the overapproximation in this chapter, the underapproximation is analogous and can be found in our formalization.

Since stateful firewalls usually accept all packets which belong to an `ESTABLISHED` connection, the interesting access control rules in a ruleset only apply to `NEW` packets. We

[7] rewrite-Goto-chain-safe and unfold-optimize-common-matcher-univ-ruleset-CHAIN
[8] NNF normalizing may create additional rules.

only consider `NEW` packets, i.e., `--ctstate NEW` and `--syn` for TCP packets. Our first goal is to translate a ruleset from the real-world model to the simple model. We have proven[9] that the set of new packets accepted by the simple firewall is a superset (overapproximation) of the packets accepted by the real-world model. This is a core contribution and we detail on the translation in the following section.

Theorem 20 (Translation to simple firewall model)**.**

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

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

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

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

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

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

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

14.4 Translating Primitives

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

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

In this section, we present algorithms to transform an arbitrary rs_1 to rs_2 without changing the behavior of the firewall.[10] In the resulting rs_2, all primitives will be normalized such that

[9] new-packets-to-simple-firewall-overapproximation, new-packets-to-simple-firewall-underapproximation

[10] All lemmas and results of the following subsections ultimately yield Theorem 20 and are referenced in its proof.

the translation to the simple-fw is obvious. We continue by describing the normalization of all common primitives found in iptables rulesets.

14.4.1 IPv4 Addresses

According to Nelson [16], "[m]odeling IP addresses efficiently is challenging." First, we present a datatype to efficiently perform set operations on intervals of machine words, e.g., 32 bit integers. We will use this type for IPv4 addresses, but we have generalized to machine words of arbitrary length, e.g., IPv6 addresses or L4 ports. For the sake of brevity, we will present our formalization at the example of IPv4. We call our datatype a word interval (*wi*), and WI *start end* describes the interval with *start* and *end* inclusive. The Union of two *wi*s is defined recursively.

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

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

An IP address in CIDR notation or IP addresses specified by e.g., `-m iprange` can be translated to one WI. We have implemented and proven the common set operations: '$\cup$', '$\{\}$', '$\setminus$', '$\cap$', '$\subseteq$', and '$=$'. These operations are linear in the number of Union-constructors. The result is optimized by merging adjacent and overlapping intervals and removing empty intervals. We can also represent 'UNIV' (the universe of all IP addresses). Since most rulesets use IP addresses in CIDR notation or intervals in general, the *wi* datatype has proven to be very efficient. Recall that the intersection of two intervals, constructed from addresses in CIDR notation, is either empty or the smaller of both intervals.[11]

The datatype *wi* is an internal representation and for the simple firewall, the result needs to be represented in CIDR notation. For this direction, one WI may correspond to several CIDR ranges. We describe an algorithm to split off one CIDR range from an arbitrary word interval r. The output is a CIDR range and r', the remainder after splitting off this CIDR range. split is implemented as follows: Let a be the lowest element in r. If this does not exist, then r corresponds to the empty set and the algorithm terminates. Otherwise, we construct the list of CIDR ranges $[a/0, a/1, ..., a/32]$. The first element in the list which is well-formed (i.e., all bits after the network prefix must be zero) and which is a subset of r is the wanted element. Note that this element always exists. It is subtracted from r to obtain r'. To convert r completely to a list of CIDR ranges, this is applied recursively until it yields no more results. This algorithm is guaranteed to terminate and the resulting list in CIDR notation corresponds to the same set of IP addresses as represented by r.[12] Formally, $\bigcup \mathsf{map\ set}\ (\mathsf{split}\ r) = \mathsf{set}\ r$.

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

With the help of these functions, arbitrary IP address ranges can be translated to the format required by the simple firewall. The following is applied to matches on src and dst IP addresses: First, the IP match expression is translated to a word interval. If the match on an IP range is negated, we compute $\mathsf{UNIV} \setminus wi$. All matches in one rule can be joined

[11] ipcidr-conjunct-correct
[12] cidr-split-prefix

to a single word interval, using the ∩ operation. The resulting word interval is translated to a set of non-negated CIDR ranges. Using the NNF normalization, at most one match on an IP range in CIDR notation remains. We have proven that this process preserves the firewall's filtering behavior.

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

Note that, while pretty printing IPv4 addresses in dotecimal notation (i.e., `<dotnum> ::= <snum> "." <snum> "." <snum> "." <snum>` [225]) is simple, pretty printing IPv6 addresses is non-trivial [223] and our implementation contains the first formally, machine-verified IPv6 pretty printer [39].

14.4.2 Conntrack State

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

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

We have proven that both models are equivalent.[13] The latter model is simpler for analysis purposes since the conntrack state can be considered an ordinary packet field.[14]

In Theorem 20, we are only interested in `NEW` packets. In contrast to previous work, there is no longer the need to manually exclude `ESTABLISHED` rules from a ruleset. The alternative model allows us to consider only `NEW` packets: all state matches can be removed (by being pre-evaluated for an arbitrary `NEW` packet) from the ruleset without changing the filtering behavior of the firewall.

14.4.3 Layer 4 Ports

Translating singleton ports or intervals of ports to the simple firewall is straightforward. A challenge remains for negated port ranges and the `multiport` module. However, the word interval type is also applicable to 16 bit machine words and solves these challenges. For ports, there is no need to translate an interval back to CIDR notation.[15]

Bug in the Original Paper We made a serious mistake [226] when specifying the semantics of matches on ports (which also made it into the camera ready paper [45]).

[13]Semantics_Stateful.thy

[14]This holds because the semantics does not modify a packet during filtering.

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

Fortunately, the mistake only occurs in corner cases and did not affect the published evaluation.[16] The error is now fixed.

Since we have proven the correctness of all our algorithms and checked all assumptions, the bug did not exist in the code. The bug exists in the model. More precisely, we formalized a semantics of ports which does not correspond to reality. We defined the datatype of a source port match as follows:

$$\textbf{datatype}\ \mathit{src\text{-}ports} = \mathsf{SrcPorts}\ \ \text{‘}16\ \mathit{word} \times 16\ \mathit{word}\text{’}$$

This datatype describes a source port match as an interval of 16 bit port numbers. The match semantics for a packet were defined such that the source port of the packet must be in the interval. For example, packet p matches $\mathsf{SrcPorts}\ a\ b$ iff $m.\text{src-port} \in \{a..b\}$. We defined $\mathsf{DstPorts}$ analogously.

With these semantics, we can construct a corner case which describes why these semantics do not correspond to reality. We consider the following firewall.

```
*filter
:FORWARD ACCEPT [0:0]
:CHAIN - [0:0]
-A FORWARD -j CHAIN
-A CHAIN -p tcp -m tcp --sport 22 -j RETURN
-A CHAIN -p udp -m udp --dport 80 -j RETURN
-A CHAIN -j DROP
COMMIT
```

The firewall in `iptables-save` format shows the `filter` table, which consists of the two chains `FORWARD` and `CHAIN`. The `FORWARD` chain is built-in and has a default policy if `ACCEPT` here. Starting at the `FORWARD` chain, any packet which is processed by this firewall is directly sent to the user-defined chain `CHAIN` first. A packet can only `RETURN` if it is a tcp packet with source port 22 or a udp packet destination port 80. All other packets are dropped. Hence, this firewall expresses in a complicated way the following policy: *“Drop everything which is not tcp src port 22 or udp dst port 80”*. This ruleset, though it does not have an obvious use, was artificially constructed to demonstrate our bug. Our tool has ‘simplified’ the ruleset the following:

$$\begin{array}{lcccl}
(\ast,\ast,\ast,\ast,\ast, & 0:21 & , & 0:79 &) \quad \texttt{DROP} \\
(\ast,\ast,\ast,\ast,\ast, & 0:21 & , & 81:65535 &) \quad \texttt{DROP} \\
(\ast,\ast,\ast,\ast,\ast, & 23:65535 & , & 0:79 &) \quad \texttt{DROP} \\
(\ast,\ast,\ast,\ast,\ast, & 23:65535 & , & 81:65535 &) \quad \texttt{DROP} \\
(\ast,\ast,\ast,\ast,\ast, & \ast & , & \ast &) \quad \texttt{ACCEPT}
\end{array}$$

Given our semantics, the simplification is correct. In reality, this simple firewall is wrong for various reasons. First, it is not well-formed, i.e., it tries to match on ports without specifying a protocol. Second, it has mixed up udp and tcp ports.

[16]However, we have seen rulesets in the wild which triggered the bug, hence, it is not purely of academic nature.

The problem lies in our semantics of SrcPorts and DstPorts. Basically, there is no such a thing as 'ports'. Yet, there exist tcp ports, udp ports, sctp ports, ...

We have fixed the issue by including the protocol in the match for a port:

$$\textbf{datatype}\ \textit{src-ports} = \mathsf{SrcPorts}\ \ \text{`8 } \textit{word}\text{'}\ \ \text{`16 } \textit{word} \times 16\ \textit{word}\text{'}$$

The 8 *word* corresponds to the `protocol` field in IPv4 [227], respectively the `Next Header` field in IPv6 [228], identifying protocols by their assigned numbers [229, 230]. It does not allow a wildcard. The semantics defines that the protocol of a packet must be the same as specified in the datatype and that the source port must be in the interval (as in the first definition).

With the corrected semantics, our tool computes the correct and expected result:

```
( * , * , * , * , UDP ,      *      ,    0 : 79   )  DROP
( * , * , * , * , UDP ,      *      , 81 : 65535)  DROP
( * , * , * , * , TCP ,   0 : 21    ,      *      )  DROP
( * , * , * , * , TCP , 23 : 65535 ,      *      )  DROP
( * , * , * , * ,  *  ,      *      ,      *      )  ACCEPT
```

The negation of a match on ports is the interesting corner case to which the presented problems can be reduced to. We will illustrate the issue by a simpler example. Assuming we have one rule which tries to accept every packet which is not udp destination port 80.[17] For simplicity, we assume we have one rule as follows: `! (-p udp --dport 80) -j ACCEPT`. Semantically, to unfold this negation, the rule matches either everything which is not udp or everything which is udp but not destination port 80. It can be expressed with the following two rules: `! -p udp -j ACCEPT` followed by `-p udp ! --dport80 -j ACCEPT`. We use this strategy in our tool to unfold the negation of matches on ports. Note the type dependencies which occur: Negating one rule that matches on ports yields both a rule which matches on protocols and one rule which matches on ports.

This example also shows that any tool which reduces match conditions to a flat bit vector is either buggy (it loses the protocol which belongs to a match on ports) or cannot support complicated negations. This includes tools which reduce firewall analysis to SAT [200] or BDDs [13, 220]. It may probably also affect ITVal [135] which relies on multi-way decision diagrams (MDD). This was also the case for our $\Gamma, \gamma, p \vdash \langle rs,\ s\rangle \Rightarrow t$ semantics with the buggy γ described in this paragraph. Our simple firewall model does not allow complicated negations and we have proven that the match conditions are always well-formed, hence, the presented class of errors cannot occur there.

14.4.4 TCP Flags

Iptables can match on a set of L4 flags. To match on flags, a *mask* selects the corresponding flags and c declares the flags which must be present. For example, the match `--syn` is

[17]Note that this cannot be expressed directly in one rule with iptables. In the example, we used the semantics of `RETURN` to construct a compound negated match expression.

a synonym for $\mathit{mask} = \texttt{SYN}, \texttt{RST}, \texttt{ACK}, \texttt{FIN}$ and $c = \texttt{SYN}$. For a set f of flags in a packet, matching can be formalized as $(f \cap \mathit{mask}) = c$. If c is not a subset of *mask*, the expression cannot match; we call this the empty match. We proved that two matches (mask_1, c_1) and (mask_2, c_2) are equal if and only if $(\textbf{if } c_1 \subseteq \mathit{mask}_1 \wedge c_2 \subseteq \mathit{mask}_2 \textbf{ then } c_1 = c_2 \wedge \mathit{mask}_1 = \mathit{mask}_2 \textbf{ else } (\neg c_1 \subseteq \mathit{mask}_1) \wedge (\neg c_2 \subseteq \mathit{mask}_2))$ holds. We also proved that the conjunction of two matches is exactly $(\textbf{if } c_1 \subseteq \mathit{mask}_1 \wedge c_2 \subseteq \mathit{mask}_2 \wedge \mathit{mask}_1 \cap \mathit{mask}_2 \cap c_1 = \mathit{mask}_1 \cap \mathit{mask}_2 \cap c_2 \textbf{ then } (\mathit{mask}_1 \cup \mathit{mask}_2,\, c_1 \cup c_2) \textbf{ else } \texttt{empty})$. If we assume `--syn` for a packet, we can remove all matches which are equal to `--syn` and add the `--syn` match as conjunction to all other matches on flags and remove empty matches. Some matches on flags may remain, e. g., `URG`, which need to be abstracted over later.

14.4.5 Interfaces

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

For whitelisting scenarios, one might argue, that negated interfaces is bad practice anyway. This is because new (virtual) interfaces might be added to the system at runtime and a match on negated interfaces might now also include these new interfaces. Therefore, it can be argued that negated interfaces correspond to blacklisting, which is not recommended for most firewalls. However, the main reason why negated interfaces are not supported by our model is of technical nature: Let set denote the set of interfaces that match an interface expression. For example, set `eth0` $= \{\texttt{eth0}\}$ and set `eth+` is the set of all interfaces that start with the prefix `eth`. If the match on `eth+` is negated, then it matches all strings in the complement set: $\mathsf{UNIV} \setminus (\mathsf{set}\ \texttt{eth+})$. The simple firewall model requires that a conjunction of two primitives is again at most one primitive. This can obviously not be achieved with such sets. In addition, working with negated interfaces can cause great confusion. Note that the interface match condition '`+`' matches any interfaces. Also note that '`+`' $\in \mathsf{UNIV} \setminus (\mathsf{set}\ \texttt{eth+})$. In the second equation, '`+`' is not a wildcard character but the name of an interface. The confusion introduced by negated interfaces becomes more apparent when one realizes that '`+`' can occur as both wildcard character and normal character. Therefore, it is not possible to construct an interface match condition which matches exactly on the interface '`+`', because a '`+`' at the end of an interface match condition is interpreted as wildcard.[18] While technically, the Linux kernel would allow to match on '`+`' as a normal character [231], the `iptables` userland command does not permit to construct such a match [232].

Correlating with IP Ranges

Later, in Section 14.5, we will compute an IP address space partition. For best clarity, this partition must not be 'polluted' with interface information. Therefore, for the partition, we will assume that no matches on interfaces occur in the ruleset. In this subsection, we describe a method to get rid of both, negated and non-negated interfaces while preserving their relation to IP address ranges.

Input Interfaces are usually assigned an IP range of valid source IPs which are expected to arrive on that interface. Let *ipassmt* be a mapping from interfaces to an IP address range.

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

This information can be obtained by `ip route` and `ip addr`. We will write $\mathit{ipassmt}[i]$ to get the corresponding IP range of interface i. For the following examples, we assume

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

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

Under the assumption that no spoofed packets occur, we will now present two algorithms to relate an input interface i to $\mathit{ipassmt}[i]$. Both approaches are valid for negated and non-negated interfaces. Approach one provides better results but requires stronger assumptions (which can be checked at runtime), whereas approach two is applicable without further assumptions.

Approach One In general, it is considered bad practice [7, 138] to have zone-spanning interfaces. Two interfaces are zone-spanning if they share a common, overlapping IP address range. Mathematically, absence of zone-spanning interfaces means that for any two interfaces in *ipassmt*, their assigned IP range must be disjoint. Our tool emits a warning if *ipassmt* contains zone-spanning interfaces. If absence of zone-spanning interfaces is checked, then all input interfaces can be replaced by their assigned source IP address range. This preserves exactly the behavior of the firewall. The idea is that in this case a bidirectional mapping between input interfaces and source IPs exists. Interestingly, our proof does not need the assumption that *ipassmt* maps to the complete IP universe.

Approach Two Unfortunately, though considered bad practice, we found many zone-spanning interfaces in many real-world rulesets and hence cannot apply the previous algorithm. First, we proved that correctness of the described rewriting algorithm implies lack of zone-spanning interfaces.[19] This leads to the conclusion that it is impossible to perform rewriting without this assumption. Therefore, we present an algorithm which adds the IP range information to the ruleset (without removing the interface match), thus constraining the match on input interfaces to their IP range. The algorithm computes the following: Whenever there is a match on an input interface i, the algorithm looks up

[19] iface-replace-needs-ipassmt-disjoint

the corresponding IP range of that interface and adds `-s` *ipassmt*[*i*] to the rule. To prove correctness of this algorithm, no assumption about zone-spanning interfaces is needed, *ipassmt* may only be defined for a subset of the interfaces, and the range of *ipassmt* may not cover the complete IP universe. Consequently, there is no need for a user to specify *ipassmt*, but having it may yield more accurate results.

Output Port Rewriting Our presented approaches for input interface rewriting can be generalized to also support output interface (`-o`) rewriting. The very core idea is to replace a match on an output interface by the corresponding IP address range which is determined by the system's routing table. To do this, we parse the routing table, map it to a relation (which provides a structure which is independent of its order), and compute the inverse of the relation. This ultimately provides a mapping for each interface and its corresponding IP address range.

This computed mapping is very similar to the *ipassmt*. In fact, we found it to be a helpful debugging tool to compare the inverse routing relation to an *ipassmt*. For convenience, we also provide a function to compute an *ipassmt* from a routing table.

Essentially, computing the inverse routing relation semantically is the same behavior as found in strict reverse path filtering [215]. We have formally proven[20] this observation.

Because a routing table may change frequently, even triggered by external malicious routing advertisements, by default, we refrain from output port rewriting in this work. It is not applied for Table 14.1; however, we additionally show how the results for firewall **D** in Section 14.8 will improve with its help.

14.4.6 Abstracting Over Primitives

Some primitives cannot be translated to the simple model. Chapter 12 already provides the function pu which removes all unknown match conditions. This leads to an approximation and is the main reason for the '$\subseteq$' relation in Theorem 20. We found that we can also rewrite any known primitive *at any time* to an unknown primitive. This can be used to apply additional knowledge during preprocessing. For example, since we understand flags, we know that the following condition is false, hence rules using it can be removed: `--syn` $\wedge$ `--tcp-flags RST,ACK RST`. After this optimization, all remaining flags can be treated as unknowns and abstracted over afterwards. This allows to easily add additional knowledge and optimization strategies for further primitive match conditions without the need to adapt any algorithm which works on the simple firewall model. We proved soundness of this approach: The '$\subseteq$' relation in Theorem 20 is preserved.

14.5 IP Address Space Partition

In the following sections, we will work on rulesets translated to the simple-fw model. In this section, we will compute a partition of the IP address space. Our algorithms works for IP addresses of arbitrary length, in particular IPv4 and IPv6. For the sake of example, we will present only IPv4 here. All IP addresses in the same partition must show the same behavior w.r.t the firewall ruleset. We do not require that the partition is minimal. Therefore, the

[20]Routing/rpf-strict-correct

following would be a valid solution: $\{\{0\}, \{1\}, \ldots, \{255.255.255.255\}\}$. However, we will need the partition as starting point for a further algorithm and a partition of size 2^{32} is too large for this purpose and infeasible for IPv6. In this section, we will present an algorithm to compute a partition which behaves roughly linear in the number of rules for real-world rulesets. First, we motivate the partitioning idea with the following observation.

Lemma 14. *For an arbitrary packet p, we write $p(src \mapsto s)$ to fix the src IP address to s. Let X be the set of all src IP matches specified in rs, i.e., X is a set of CIDR ranges. If*

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

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

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

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

For simplicity, we use finite sets and lists interchangeably. We will write an algorithm part and reuse the common list algorithm from functional programming foldr. For X, the following algorithm computes a partition: foldr part X $\{\mathsf{UNIV}\}$. In addition, it is guaranteed that the union of the resulting partition is equal to the universe. For our scenario, this means that the partitioning covers the complete IPv4 space. The algorithm part is implemented as follows: The first parameter is a set $S \in X$, the second parameter TS is a set of sets and corresponds to the remaining set which will be partitioned. In the first call $TS = \{\mathsf{UNIV}\}$. For a fixed S, part S TS iterates over TS and splits the set such that the precondition of Lemma 14 holds: Written as recursive function: $\mathsf{part}\ S\ (\{T\} \cup TS) = (S \cap T) \cup (T \setminus S) \cup (\mathsf{part}\ (S \setminus T)\ TS)$

The result size of calling part once can be up to two times the size of TS. This means, the partition of a complete firewall ruleset is in $O(2^{|rules|})$. However, the empirical evaluation shows that the resulting size for real-world rulesets is much better. This is because IP address ranges may overlap in a ruleset, but they do not overlap in the worst possible way for all pairs of rules. Consequently, at least one of the sets $S \cap T$ or $T \setminus S$ is usually empty and can be optimized away. For example, for our largest firewall, the number of computed partitions is 10 times smaller than the number of rules. Table 14.1 confirms that the number of partitions is usually less than the number of rules.

Our algorithm fulfills the assumption of Lemma 14 for arbitrary X. Because IP addresses occur as source and destination in a ruleset, we use our partitioning algorithm where X is the set of all IPs found in the ruleset. The result is a partition where for any two IPs in the same partition, setting the src or dst of an arbitrary packet to one of the two IPs, the firewall behaves equally. This results in a stronger version of Lemma 14, which holds without any assumption and also holds for both src and dst IPs simultaneously.[21] In addition, the

[21] getParts-samefw

partition covers the complete IPv4 address space.[22]

14.6 Service Matrices

The IP address space partition may not be minimal. That means, two different partitions may exhibit exactly the same behavior. Therefore, for manual firewall verification, these partitions may be misleading. Marmorstein elaborates on this problem [136]. ITVal's solution is to minimize the partition. We suggest to minimize the partition for a fixed service. The evaluation shows that the result is smaller and thus clearer.

A fixed service corresponds to a fixed packet with arbitrary IPs. For example, we can define ssh as TCP, dport 22, arbitrary sport $\geq$ 1024. A service matrix describes the allowed accesses for a specific service over the complete IPv4 address space. It can be visualized as graph; for example, the ruleset of Figure 14.1 is visualized in Figure 14.2. An example of a firewall with several thousands of rules is shown in Figure 14.3. For clarity, this figure uses symbolic names (e.g., *servers*) instead of IP addresses. The raw IP addresses can be found in Figure 14.4. More complicated examples with highly fragmented IP ranges are shown in Figure 17.3 and Figure 17.5; actually they are from the same firewall at a later point in time. All matrices are minimal, i.e., they cannot be compressed any further.

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

$$\begin{aligned}\forall d.\ &\textsf{simple-fw}\ rs\ p(src \mapsto s_1,\ dst \mapsto d) = \\ &\textsf{simple-fw}\ rs\ p(src \mapsto s_2,\ dst \mapsto d)\end{aligned}$$

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

We present a function groupWIs, which computes the minimal partition for a fixed service. For this, the full access control matrix for inbound and outbound connections of each partition member is generated. This can be done by taking arbitrary representatives from each partition as source and destination address and executing simple-fw for the fixed packet with those fixed IPs. The matrix is minimized by merging partitions with equal rights, i.e., equal rows in the matrix. This algorithm is quadratic in the number of partitions. An early evaluation [45] shows that it scales surprisingly well, even for large rulesets, since the number of partitions is usually small.

The algorithm is sound[23], complete[24], and minimal.[25]

Theorem 21 (groupWIs is Sound and Minimal)**.** *For any two IPs in any member of* groupWIs*, the firewall shows the same behavior for a fixed service.*

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

Improving Performance We assume that the ruleset has a default policy; we fall back to our slower algorithm otherwise but any simplified ruleset obtained from well-formed

[22]getParts-complete
[23]build-ip-partition-same-fw
[24]build-ip-partition-complete
[25]build-ip-partition-same-fw-min

iptables has a default policy.[26] The algorithm described above performs two calls (one for source IP and one for destination IP) to simple-fw for each pair of representatives in the partition. The algorithm is significantly slowed down by the quadratic number of calls to simple-fw. Instead of repeatedly executing simple-fw for all representatives as source and destination address, for a fixed service and fixed source address, we can pre-compute the set of all matching destination addresses with one iteration over the ruleset. The same holds for the matching source addresses. This roughly brings down the quadratic number of calls to simple-fw to a linear number of iterations over the ruleset. Note that the asymptotic runtime is still quadratic. We have implemented this improved algorithm and proven that Theorem 21 and Theorem 22 still hold for it. The empirical evaluation shows that this improvement yields a speedup of about 10x, i.e., 1000%.

Final Theorem The function groupWIs computes a minimal partition and the allowed accesses over representatives of the IPv4 address space. However, it does not directly allow drawing graphs as shown in e.g., Figure 14.3, Figure 17.3, or Figure 17.5. To draw a graph, for example with tikz [233], one first needs to print the nodes and print the edges afterwards. The name of the nodes (representatives) should not be printed but th IP range it actually represents. For example, a graph may be defined as follows:

```
\begin{tikzpicture}
  \node (a) at (-4,-4) {$\{131.159.21.0 .. 131.159.21.255\}$};
  \node (b) at (4,-4) {$\{131.159.15.240 .. 131.159.15.255\}$};
  \node (c) at (0,-6) {$\{127.0.0.0 .. 127.255.255.255\}$};
  ...

  \draw (a) to (b);
  \draw (c) to (a);
  \draw (c) to (b);
  \draw (c) to[loop above] (c);
  ...
\end{tikzpicture}
```

In this example, the node names `a`, `b`, and `c` are representatives which semantically correspond to the set of IP addresses on their right. The edges mean that the complete IP ranges may communicate, e.g., `\draw (a) to (b)` means that the complete set 131.159.21.0/24 may establish connections to 131.159.15.240/28. In the final drawing, the identifiers `a`, `b`, and `c` are not shown but only their corresponding IP ranges. We present a final theorem which justifies the correctness of graphs which are drawn according to our method.[27]

Theorem 22 (Service Matrix)**.** *Let V be the nodes of the service matrix. V is a map from representatives to the IP range of the representative. Let E be the edges of the service matrix. Then,*

$$(\exists\ s_{\text{repr}}\ d_{\text{repr}}\ s_{\text{range}}\ d_{\text{range}}.\ \ (s_{\text{repr}},\ d_{\text{repr}}) \in E\ \wedge$$

[26]Since we can easily check at runtime whether a ruleset has a default policy, this fallback only exists that we can write down our theorems without requiring the assumption of a default policy since our faster algorithm (with default policy) and slower algorithm (without default policy) compute the same result. In practice, any ruleset has a default policy and the faster algorithm is always used.

[27]access-matrix

$$V\ s_{\text{repr}} = \mathsf{Some}\ s_{\text{range}}\ \wedge\ s \in s_{\text{range}}\ \wedge$$
$$V\ d_{\text{repr}} = \mathsf{Some}\ d_{\text{range}}\ \wedge\ d \in d_{\text{range}}\big)$$
$$\longleftrightarrow$$
$$\mathsf{simple\text{-}fw}\ rs\ p(src \mapsto s,\ dst \mapsto d)\ = \circledcheck$$

The theorem reads as follows: For a fixed connection, one can look up IP addresses (source s and destination d pairs) in the graph if and only if the firewall accepts this (s, d) IP address pair for the fixed connection.

The part which complicates the formalization is the formulation of 'look up IP addresses [..] in the graph'. To look up source IP address s in the graph, one first locates s as a member in one of the IP ranges, here s_{range}. This IP range is represented by a representative s_{repr}. The same is done to obtain d_{repr}. The theorem now says that $(s_{\text{repr}},\ d_{\text{repr}}) \in E$ iff the firewall allows packets from s to d. The theorem is actually stronger because the if-and-only-if relationship in combination with the existential quantifier also implies that there is always exactly one range in which we can find s and d (which means that our graph always contains a complete and disjoint representation of the IP address space).

14.7 Stand-Alone Haskell Tool *fffuu*

We used Isabelle's code generation features [110, 111] to build a stand-alone tool in Haskell. Since all analysis and transformation algorithms are written in Isabelle, we only needed to add parsers and user interface. Overall, more than 80% of the code is generated by Isabelle, which gives a strong correctness guarantee.

We call our tool *fffuu*, the *f*ancy *f*ormal *f*irewall *u*niversal *u*nderstander.

fffuu requires only one parameter to run, namely, an `iptables-save` dump. This makes it very usable. Optionally, one may pass an *ipassmt*, change the `table` or `chain` which is loaded, pass a routing table for output port rewriting, or select the services for the service matrix.

Example We demonstrate *fffuu* by a small example. We want to infer the intention behind the ruleset shown in Figure 14.1. Though this ruleset was artificially crafted to demonstrate certain corner cases, it is based on actual rules from real-world firewalls [234, 25]. Also note that the interface name `\e[31m☃\e[0m` with utf-8 symbols and shell escapes for color [235] is perfectly valid.

It is hard to guess what the ruleset is implementing. The service matrix will provide clarity. We dump the ruleset into *fffuu*, not requiring any additional parameters or manual steps to compute it. The resulting service matrix (for arbitrary ports) is shown in Figure 14.2. An arrow from one IP range to another IP range indicates that the first range may set up connections with the second.

At the bottom, we see the localhost range of 127.0.0.0/8. The reflexive arrow (localhost to localhost) shows that the firewall does not block its own localhost traffic, which is usually a good sign. However, localhost traffic is usually not interesting for a firewall analysis since this range is usually not routed [218]. We will ignore it from now.

On the top, in the cloud, we see a large set of IP addresses. This corresponds to the Internet. On the left, we see the 131.159.21.0/24 range. It may access the Internet and the

```
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]
:DOS_PROTECT - [0:0]
:GOOD~STUFF - [0:0]
-A FORWARD -j DOS_PROTECT
-A FORWARD -j GOOD~STUFF
-A FORWARD -p tcp -m multiport ! --dports 80,443,6667,6697 -m hashlimit       ↩
    --hashlimit-above 10/sec --hashlimit-burst 20 --hashlimit-mode srcip      ↩
    --hashlimit-name aflood --hashlimit-srcmask 8 -j LOG
-A FORWARD ! -i lo -s 127.0.0.0/8 -j DROP
-A FORWARD -i internal -s 131.159.21.0/24 -j ACCEPT
-A FORWARD -s 131.159.15.240/28 -d 131.159.21.0/24 -j DROP
-A FORWARD -p tcp -d 131.159.15.240/28 -j ACCEPT
-A FORWARD -i ☃ -p tcp -s 131.159.15.240/28 -j ACCEPT
-A GOOD~STUFF -i lo -j ACCEPT
-A GOOD~STUFF -m state --state ESTABLISHED -j ACCEPT
-A GOOD~STUFF -p icmp -m state --state RELATED -j ACCEPT
-A DOS_PROTECT -i eth1 -p icmp -m icmp --icmp-type 8 ... --limit 1/sec -j RETURN
-A DOS_PROTECT -i eth1 -p icmp -m icmp --icmp-type 8 -j DROP
COMMIT
```

Figure 14.1: Example Ruleset

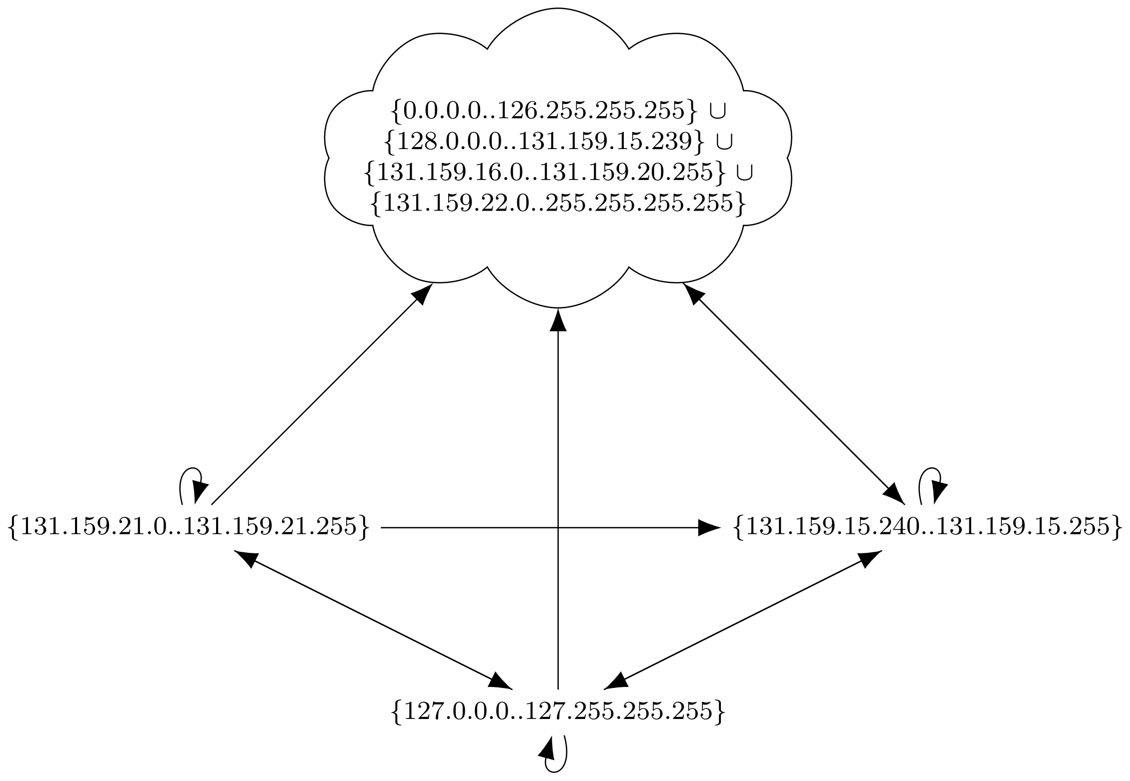


Figure 14.2: Service Matrix of Ruleset in Figure 14.1

131.159.15.240/28 range. On the right, we see the 131.159.15.240/28 range, which may only access the Internet but not the 131.159.21.0/24 range.

Gazing at the figure, we might recognize the overall architecture: The firewall simply implements the textbook version of the Demilitarized Zone (DMZ) architecture. Starting from the original `iptables-save` input, without the help of *fffuu*, this would be extremely hard to uncover and verify.

14.8 Evaluation

We obtained real-world rulesets from over 15 firewalls. Some are central, production-critical devices. They are written by different authors, utilize a vast amount of different features and exhibit different styles and patterns. The fact that we publish the complete rulesets is an important contribution (cf. [7, 8]). To the best of our knowledge, this is the largest, publicly-available collection of real-world iptables rulesets. Note: some administrators wish to remain anonymous so we replaced their public IP addresses with public IP ranges of our institute, preserving all IP subset relationships.

Table 14.1 summarizes the evaluation's results. The first column (Fw) labels the analyzed ruleset. Column two (Rules) contains the number of rules (only the filter table) in the output of `iptables-save`. We work directly and completely on this real-world data. Column three describes the analyzed chain. Depending on the type of firewall, we either analyzed the `FORWARD` (FW) or the `INPUT` (IN) chain. For a host firewall, we analyzed IN; for a network firewall, e. g., on a gateway or router, we analyzed FW. In parentheses, we wrote the number of rules after unfolding the analyzed chain. The unfolding also features some generic, straight-forward optimizations, such as removing rules where the match expression is False. Column four (Simple rules) is the number of rules when translated to the simple firewall. In parentheses, we wrote the number of simple firewall rules when interfaces are removed. This ruleset is used subsequently to compute the partitions and service matrices. In column five (Use), we mark whether the translated simple firewall is useful. We will detail on the metric later. Column six (Parts) lists the number of IP address space partitions. For comparison, we give the number of partitions computed by ITVal in parentheses. In Column seven (ssh) and eight (http), we give the number of partitions for the service matrices for ssh and http. In column nine (Time (ITVal)), for comparison, we put the runtime of the partitioning by ITVal in parentheses in seconds, minutes, or hours. In column ten (Time (this)), we give the overall runtime of our analysis.

When translating to the simple firewall, to accomplish support for arbitrary matching primitives, some approximations need to be performed. For every firewall, the first row states the overapproximation (more permissive), the second row the underapproximation (more strict).

In contrast to previous work, there is no longer the need to manually exclude certain rules from the analysis (cf. Section 12.5.4). For some rulesets, we do not know the interface configuration. For others, there were zone-spanning interfaces. For these reasons, as proven in Section 14.4.5, in the majority of cases, we could not rewrite interfaces. This is one reason for the differences between over- and underapproximation.

We loaded all translated simple firewall rulesets (without interfaces) with `iptables-restore`. This validates that our results are well-formed. We then used ipt-

Fw	Rules	Chain	Simple rules	Use	Parts	ssh	http	Time	Time
		(unfolded)	(no ifaces)		(ITVal)			(ITVal)	(this)
A	2784	FW (2376)	2381 (1920)	✓	246 (1)	13	9	3 h*	172 s
	-	FW (2376)	2837 (581)	✗^{r}	522 (1)	1	1	9 h*	194 s
A	4113	FW (2922)	3114 (2862)	✓	334 (2)	11	11	27 h*	302 s
	-	FW (2922)	3585 (517)	✗^{r}	490 (1)	1	1	8 h	320 s
A	4814	FW (4403)	3574 (3144)	✓	364 (2)	9	12	46 h*	477 s
	-	FW (4403)	5123 (1601)	✗^{r}	1574 (1)	1	1	3 h*	618 s
A	4946	FW (4887)	4004 (3570)	✓	371 (2)	9	12	53 h*	578 s
	-	FW (4887)	5563 (1613)	✗^{r}	1585 (1)	1	1	4 h*	820 s
B	88	FW (40)	110 (106)	✓	50 (4)	4	2	2 s	3 s
	-	FW (40)	183 (75)	✓	40 (1)	1	1	1 s	2 s
C	53	FW (30)	29 (12)	✓	8 (1)	1	1	1 s	1 s
	-	FW (30)	27 (1)	✓	1 (1)	1	1	1 s	1 s
	-	IN (49)	74 (46)	✓	38 (1)	1	1	1 s	1 s
	-	IN (49)	75 (21)	✓	6 (1)	1	1	1 s	1 s
D	373	FW (2649)	3482 (166)	✓	43 (1)	1	1	3 s	22 s
	-	FW (2649)	16592 (1918)	✗	67 (1)	1	1	33 min*	49 s
E	31	IN (24)	57 (27)	✓	4 (3)	1	2	1 s	10 s
	-	IN (24)	61 (45)	✗^{r}	3 (1)	1	1	1 s	1 s
F	263	IN (261)	263 (263)	✓	250 (3)	3	3	2 min	80 s
	-	IN (261)	265 (264)	✓	250 (3)	3	3	3 min	57 s
G	68	IN (28)	20 (20)	✓	8 (5)	1	2	1 s	8 s
	-	IN (28)	19 (19)	✗	8 (2)	2	2	1 s	1 s
H	19	FW (20)	10 (10)	✗	9 (1)	1	1	1 s	8 s
	-	FW (20)	8 (8)	✗^{r}	3 (1)	1	1	1 s	1 s
I	15	FW (5)	4 (4)	✓	4 (4)	4	4	1 s	8 s
	-	FW (5)	4 (4)	✓	4 (4)	4	4	1 s	1 s
J	48	FW (12)	5 (5)	✓	3 (2)	2	2	1 s	6 s
	-	FW (12)	8 (2)	✓	1 (1)	1	1	1 s	1 s
K	21	FW (9)	7 (6)	✓	3 (1)	1	1	1 s	12 s
	-	FW (9)	4 (3)	✓	2 (1)	1	1	1 s	1 s
L	27	IN (16)	19 (19)	✓	17 (3)	2	2	1 s	1 s
	-	IN (16)	18 (18)	✓	17 (3)	2	2	1 s	1 s
M	80	IN (92)	64 (16)	✓	2 (2)	1	2	1 s	6 s
	-	IN (92)	58 (27)	✗	11 (1)	1	1	1 s	1 s
N	34	FW (14)	12 (12)	✓	10 (6)	6	6	2 s	2 s
	-	FW (14)	12 (12)	✓	10 (6)	6	6	2 s	1 s
O	8	IN (7)	9 (9)	✓	3 (3)	1	2	1 s	1 s
	-	IN (7)	8 (8)	✓	3 (3)	1	2	1 s	1 s
P	595	IN (15)	8 (8)	✓	3 (2)	2	2	1 s	6 s
	-	IN (15)	9 (9)	✓	3 (2)	2	2	1 s	6 s
	595	FW (66)	64 (64)	✓	60 (5)	5	4	22 s	6 s
	-	FW (66)	63 (63)	✓	60 (5)	5	4	22 s	7 s
Q	58	IN (59)	65 (65)	✓	21 (1)	1	1	2 s	2 s
	-	IN (59)	62 (62)	✓	21 (2)	2	1	2 s	1 s
R	30	FW (28)	123 (123)	✓	14 (1)	1	6	1 s	1 s
	-	FW (28)	20 (3)	✓	2 (2)	2	1	1 s	1 s

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

Table 14.1: Summary of Evaluation on Real-World Firewalls

ables directly to generate the firewall format required by ITVal (`iptables -L -n`). Our translation to the simple firewall is required because ITVal cannot understand the original complex rulesets and produces flawed results for them.

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

These benchmark settings are extremely 'unfair' for our tool. Indeed, exporting our tool to a standalone Haskell application, replacing some common datastructures with optimized ones from the Haskell std lib, enabling aggressive compiler optimization and parallelization, and running our tool on the Xeon server, the runtime of our tool improves by orders of magnitude. Our stand-alone tool *fffuu* also achieves a better runtime by orders of magnitude. Nevertheless, we chose the 'unfair' setting to demonstrate the feasibility of running fully verified code directly in a theorem prover. In addition, we preserve the property of full verification; even for the results of executable code.[28]

Table 14.1 shows that our tool outperforms ITVal for large firewalls. We added ITVal's memory requirements to the table if they exceeded 20 GB. ITVal requires an infeasible amount of memory for larger rulesets while our tool can finish on commodity hardware. The overall numbers show that the runtime for our tool is sufficient for static, offline analysis, even for large real-word rulesets.

For our daily use and convenience, we use our Haskell tool *fffuu* which adds another order of magnitude of speedup to our numbers of Table 14.1.

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

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

[28]There are methods to improve the performance and provably preserve correctness [236, 237], which are out of the scope of this thesis

case with an $^{\text{r}}$. The table indicates that, usually, at least one approximation per firewall is useful.

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

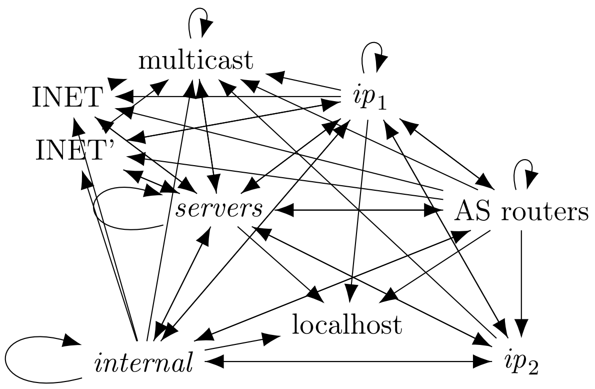


Figure 14.3: TUM i8 ssh Service Matrix

Firewall A This firewall is the core firewall of our lab (Chair of Network Architectures and Services). It has two uplinks, interconnects several VLANs, hence, the firewall matches on more than 20 interfaces. It has around 500 direct users and one transfer network for an AS behind it. The traffic is usually several Mbit/s. The dumps are from Oct 2013, Sep 2014, May 2015, Sep 2015 and the changing number of rules indicates that it is actively managed. The firewall starts with some rate-limiting rules. Therefore, its stricter approximation assumes that the rate-limiting always applies and transforms the ruleset into a deny-all ruleset. The more permissive approximation abstracts over this rate-limiting and provides a very good approximation of the original ruleset. The ssh service matrix is visualized in Figure 14.3 and in Figure 14.4 with the raw IP addresses. The figure can be read as follows: The vast majority of our IP addresses are grouped into *internal* and *servers*. Servers are reachable from the outside, internal hosts are not. *ip*$_1$ and *ip*$_2$ are two individual IP addresses with special exceptions. There is also a group for the backbone routers of the connected AS. INET is the set of IP addresses which does not belong to us, basically the Internet. INET' is another part of the Internet. With the help of the service matrix, the administrator confirmed that the existence of INET' was an error caused by a stale rule. The misconfiguration has been fixed. Figure 14.3 summarizes over 4000 firewall rules and helps to easily visually verify the complex ssh setup of our firewall. The administrator was also interested in the kerberos-adm and ldap service matrices. They helped verifying the complex setup and discovered potential for ruleset cleanup.

We have used the *fffuu* tool further on to analyze our firewall. For example, in Chapter 17, Figure 17.3 and Figure 17.5 were created from a snapshot of June 2016 and depict the service matrix for http. The figures show the raw IP addresses. It can be seen that the 'two INETs' bug has been fixed. Note that the service matrix is minimal, i.e., there is no way to compress it any further. The two figures reveal the intrinsic complexity of this firewall. However, the figures —though complicated— can still be visualized on one page, something which would be impossible for the thousands of rules of the actual ruleset. This demonstrates that our service matrices can give a suitable overview over complicated rulesets.

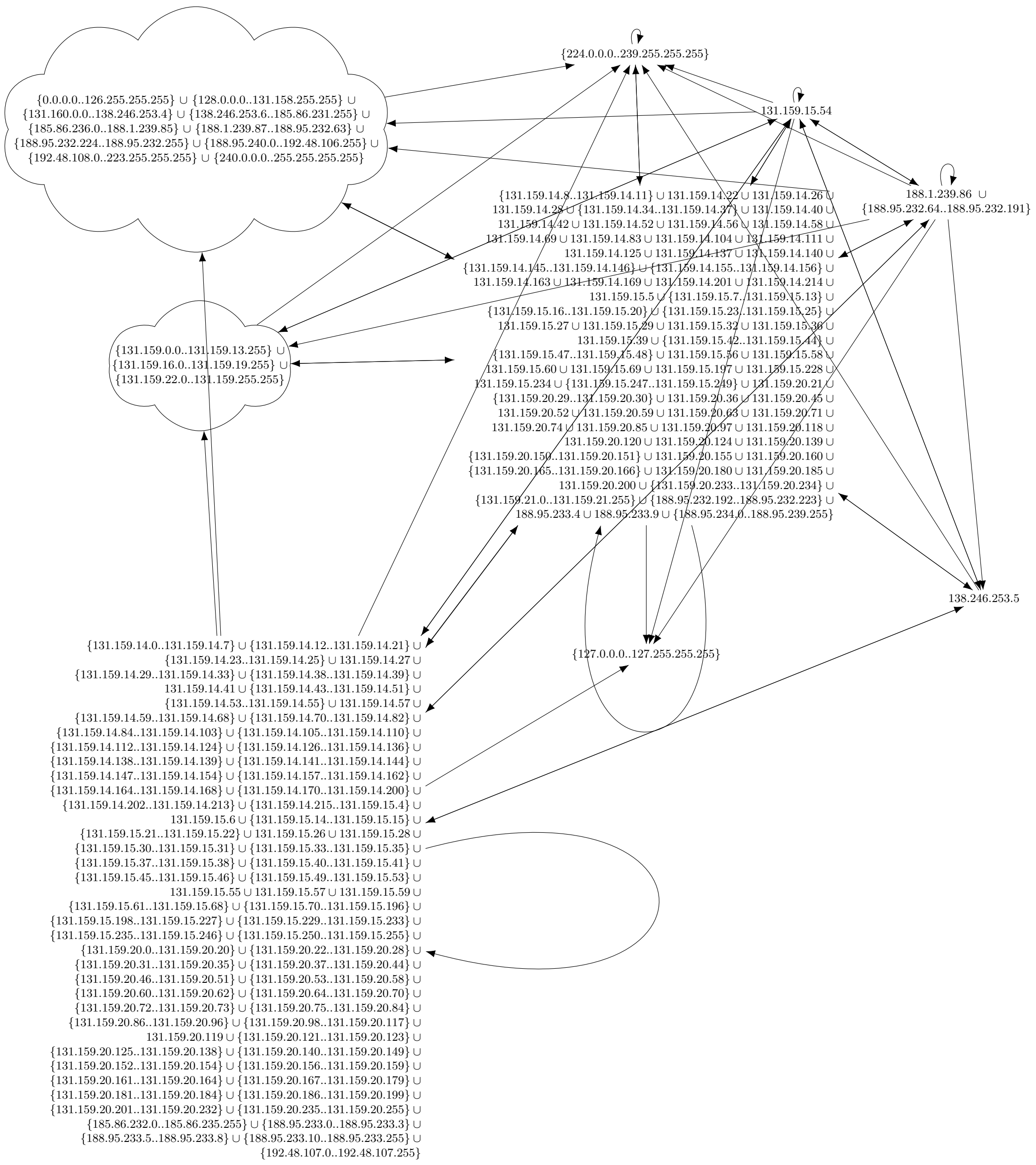


Figure 14.4: TUM i8 ssh Service Matrix (with raw IP addresses)

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

Later on, having obtained more detailed interface and routing configurations, we tried again with input and output port rewriting. The result is not shown in the table but visualized in Figure 14.5. The figure now correctly summarizes the network architecture enforced by the firewall. It shows the general Internet, a debian update server (141.76.2.4), and four internal networks with different access rights.

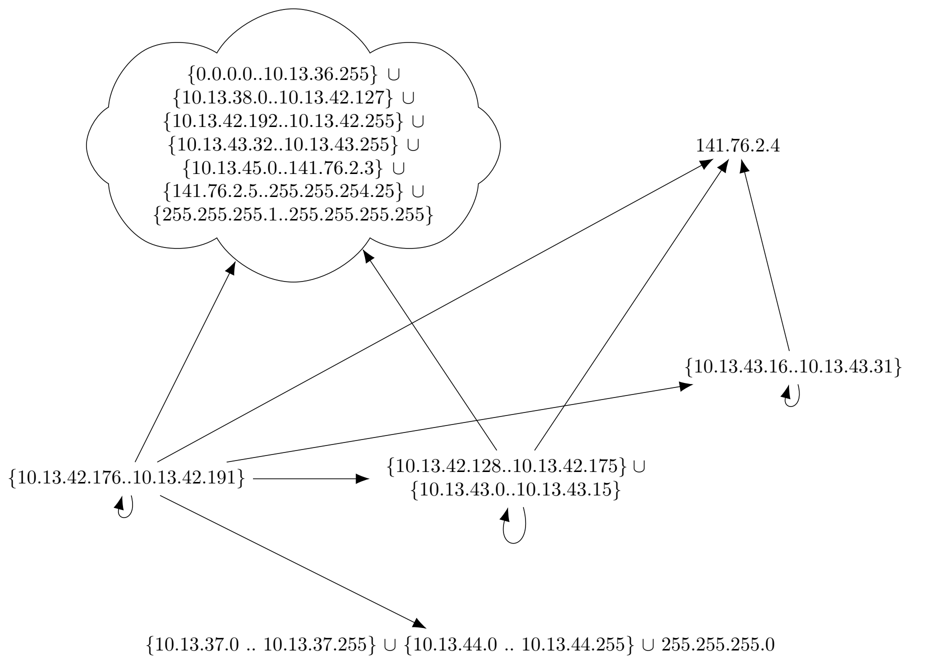


Figure 14.5: Firewall D ssh Service Matrix with input and output port rewriting

Firewall E This ruleset was taken from a NAS device from the introduction (Figure 12.1). The ruleset first performs some rate-limiting, consequently, the underapproximation corresponds to the deny-all ruleset. The table lists a more recent version of the ruleset after a system update. Our ssh service matrix reveals a misconfiguration: ssh was accidentally left enabled after the update. After this discovery, the flaw was fixed. The service matrix for the other services provided by the NAS (not listed in the table) verifies that these services are only accessible from the local network. This finally yields the expected result as motivated in Section 12.1.

Firewall F This firewall is running on a publicly accessible server. The firewall first allows everything for localhost, then blocks IP addresses which have shown malicious behavior in the past and finally allows certain services. Since most rules are devoted to blocking malicious IPs, our IP address space partition roughly grows linear with the number of rules. The service matrices, however, reveal that there are actually only three classes of IP ranges: localhost, the blocked IPs, and all other IPs which are granted access to the services.

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

Firewall H This ruleset from 2003 appears to block Kazaa filesharing traffic during working hours. In addition, a rule drops all packets with the string “X-Kazaa-User”. The more permissive abstraction correctly tells that the firewall may accept all packets for all IPs (if the above conditions do not hold). Hence, the firewall is essentially abstracted to an allow-all ruleset. According to our metric, this information is not useful. However, in this scenario, this information may reveal an error in the ruleset: The firewall explicitly permits certain IP ranges, however, the default policy is `ACCEPT` and includes all these previously explicitly permitted ranges. By inspecting the structure of the firewall, we suppose that the default policy should be `DROP`. This possible misconfiguration was uncovered by the overapproximation. The underapproximation does not understand the string match on “X-Kazaa-User” in the beginning and thus corresponds to the deny-all ruleset. However, a manual inspection of the underapproximation still reveals an interesting error: The ruleset also tries to prevent MAC address spoofing for some hard-coded MAC/IP pairs. However, we could not see any drop rules for spoofed MAC addresses in the underapproximation. Indeed, the ruleset allows non-spoofed packets but forgets to drop the spoofed ones. This firewall demonstrates the worst case for our approximations: one set of accepted packets is the universe, the other is the empty set.[29] However, manual inspection of the simplified ruleset helped reveal several errors. This demonstrates that even if the service matrices do not contain any information, the other output of our tool may still contain interesting information.

Firewall P This is the ruleset of the main firewall of a medium-sized company. The administrator asked us what her ruleset was doing. She did not reveal her intentions to prevent that the analysis might be skewed towards the expected outcome. We calculated the simplified firewall rules and service matrices. Using the underapproximation, we could also give guarantees about the packets which are definitely allowed by the firewall. The administrator critically inspected the output of our tool. Finally, she confirmed that the firewall was working exactly as intended. This demonstrates: Not only finding errors but showing correctness is one of the key strengths of our tool.

After the analysis, the administrator revealed her true intentions. She has previously upgraded the system to iptables. Her users, the company’s employees, got aware of that fact. She received some complaints about connectivity issues and the employees were blaming the firewall. However, the administrator was suspecting that the connectivity issues were triggered by some users who are behaving against the corporate policy, e.g., sharing user

[29]Note that this ruleset is actually severely broken and no better approximation would be possible.

accounts. With the help of our analysis, the administrator could reject all accusations about her firewall config and follow her initial suspicion about misbehaving employees.

A few months later, we received some final feedback that the firewall was perfect and "users are stupid".

Firewall R This ruleset was extracted from a docker host. We discuss this scenario and the history of the ruleset in detail in Chapter 16. For remote management, the ruleset allows unconstrained ssh access for all machines, which can be seen by the fact that the ssh service matrix only shows one partition. In contrast, an advanced setup is enforced for http and the http matrix is visualized in Figure 16.10. Being able to verify the publicly exposed http setup while neglecting the ssh maintenance setup demonstrates the advantage of calculating our access matrices for each service. The lower closure also exhibits one interesting detail: Except for one host which is rate limited, ssh connectivity is guaranteed. Ironically, ITVal segfaults on the original ruleset. With our processing, it terminates successfully but returns a spurious result.

14.9 Conclusion

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

Based on the simplified model, we presented algorithms to partition the IPv4 and IPv6 address space and compute service matrices. This allows summarizing and verifying the firewall in a clear manner.

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

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

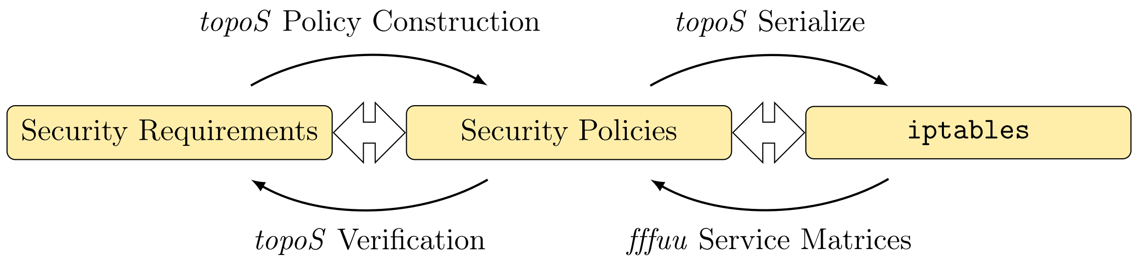
Part III

Applicability & Conclusion

Chapter 15

Overview

In this part, we demonstrate the interplay of the tools we developed in Part I and Part II. As the following figure illustrates, our tools *fffuu* and *topoS* work on common abstractions and allow to translate between different levels of abstraction.



This part is structured as follows. First, we give an overview of the applicability of our tools in Chapter 16 by presenting a story about the dynamic management of docker container networking. Afterwards, in Chapter 17, we demonstrate our tools in a real-world case study, namely a privacy evaluation of the Android MeasrDroid app. Having demonstrated the big picture of the individual results obtained in this thesis, in Chapter 18 we summarize our answers to the scientific questions asked at the beginning. In Chapter 19, we compare this work to the state of the art. We summarize the applicability of our work in Chapter 20 and conclude this thesis in Chapter 21.

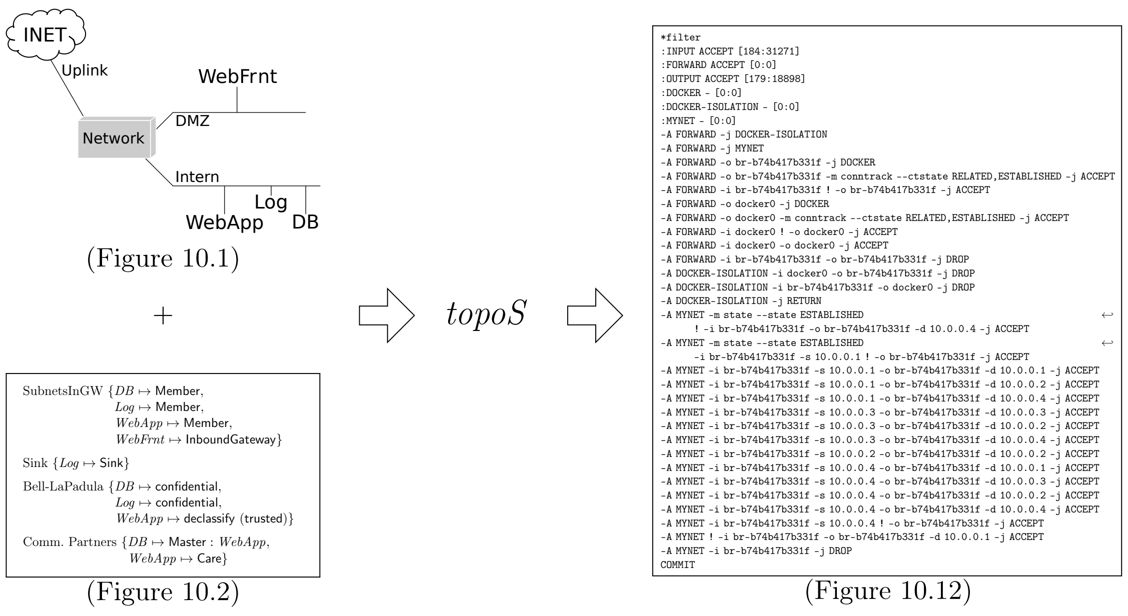
Chapter 16

Demonstrating Dynamic Microservice Management

Abstract In this chapter, we demonstrate the interplay of *topoS* and *fffuu*. We present a fictional story about administrating a dynamically changing docker environment. From the initial design and software engineering through network operations and automation, we show how our tools help.

16.1 Introduction

This chapter demonstrates applicability of our tools in a dynamic context by telling a fictional story about administrator Alice. Alice loves docker and she maintains the distributed web application we have presented in Chapter 10. She knows that it is best security practice when using docker to decrease the attack surface by limiting container networking [238]. Hence, she takes great care about her firewall settings. In Figure 16.1, we summarize how *topoS* was used in Chapter 10 to create the initial firewall rules.



SubnetsInGW $\{DB \mapsto$ Member, $Log \mapsto$ Member, $WebApp \mapsto$ Member, $WebFrnt \mapsto$ InboundGateway$\}$

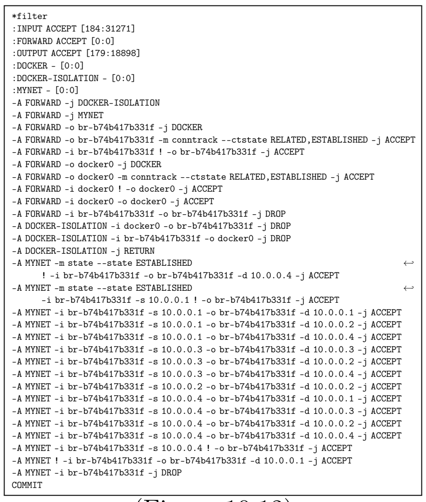
Sink $\{Log \mapsto$ Sink$\}$

Bell-LaPadula $\{DB \mapsto$ confidential, $Log \mapsto$ confidential, $WebApp \mapsto$ declassify (trusted)$\}$

Comm. Partners $\{DB \mapsto$ Master : $WebApp$, $WebApp \mapsto$ Care$\}$

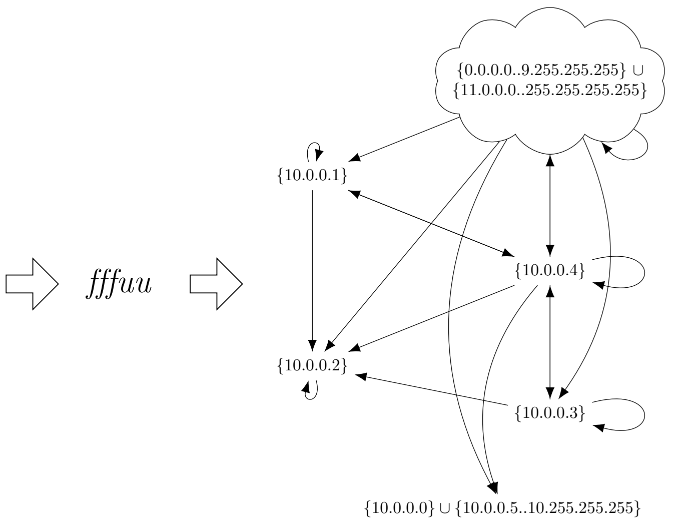
```
*filter
:INPUT ACCEPT [184:31271]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [179:18898]
:DOCKER - [0:0]
:DOCKER-ISOLATION - [0:0]
:MYNET - [0:0]
-A FORWARD -j DOCKER-ISOLATION
-A FORWARD -j MYNET
-A FORWARD -o br-b74b417b331f -j DOCKER
-A FORWARD -o br-b74b417b331f -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i br-b74b417b331f ! -o br-b74b417b331f -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A FORWARD -i br-b74b417b331f -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i docker0 -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i br-b74b417b331f -o docker0 -j DROP
-A DOCKER-ISOLATION -j RETURN
-A MYNET -m state --state ESTABLISHED                                              ↩
      ! -i br-b74b417b331f -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -m state --state ESTABLISHED                                              ↩
      -i br-b74b417b331f -s 10.0.0.1 ! -o br-b74b417b331f -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.3 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.2 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.3 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 ! -o br-b74b417b331f -j ACCEPT
-A MYNET ! -i br-b74b417b331f -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -j DROP
COMMIT
```

Figure 16.1: Input and Output of *topoS*



```
*filter
:INPUT ACCEPT [184:31271]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [179:18898]
:DOCKER - [0:0]
:DOCKER-ISOLATION - [0:0]
:MYNET - [0:0]
-A FORWARD -j DOCKER-ISOLATION
-A FORWARD -j MYNET
-A FORWARD -o br-b74b417b331f -j DOCKER
-A FORWARD -o br-b74b417b331f -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i br-b74b417b331f ! -o br-b74b417b331f -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A FORWARD -i br-b74b417b331f -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i docker0 -o br-b74b417b331f -j DROP
-A DOCKER-ISOLATION -i br-b74b417b331f -o docker0 -j DROP
-A DOCKER-ISOLATION -j RETURN
-A MYNET -m state --state ESTABLISHED                                   ↩
      ! -i br-b74b417b331f -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -m state --state ESTABLISHED                                   ↩
      -i br-b74b417b331f -s 10.0.0.1 ! -o br-b74b417b331f -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.1 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.3 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.3 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.2 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.3 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
-A MYNET -i br-b74b417b331f -s 10.0.0.4 ! -o br-b74b417b331f -j ACCEPT
-A MYNET ! -i br-b74b417b331f -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
-A MYNET -i br-b74b417b331f -j DROP
COMMIT
```

(Figure 10.12)



(Figure 10.13)

Figure 16.2: Input and Output of *fffuu*

Throughout this thesis, we have mainly focused on *static* configuration management. Since our tools are completely automatic, they can also be used in *dynamic* contexts. Alice installs a cronjob which runs `iptables-save` regularly and looks for changes. If a change to the ruleset is discovered, *fffuu* is run to compute an overview of the firewall policy currently enforced. The result is visualized with tikz and emailed to Alice. We visualize this process in Figure 16.2. In this story, the requirements, setup, and configurations are frequently changed and we demonstrate the use of our tools in this dynamic context.

16.2 Network Access Control in Docker

First, Alice questions her initial decision of Chapter 10 to operate custom firewall rules and researches whether docker can provide similar features out of the box.

Since docker version 1.10, it is possible to create custom internal networks [239]. The `--internal` flag protects a network from accesses from the outside. However, by default, all containers in an internal network can access each other. It is possible to configure the network such that containers cannot access each other. Ironically, this feature was broken in our version of docker and containers could still communicate [240]. In addition, there is no possibility to enable fine-grained access control between the containers in a network. A `--link` option exists to connect two containers within a custom network, but it merely sets environment variables, it does not influence the actual IP connectivity.

The docker design philosophy is to decouple the application developer from networking details [241]. Only a coarse-grained network abstraction in terms of different networks is exposed to the application developer. The network IT team —i. e., Alice— should manage the network. Since one compromised container in an internal network can attack all other containers in its network, Alice desires further network-level access control.

Due to the lack of fine-grained access control, Alice sticks to her custom firewall rules in the docker host. In addition, the fact that the fix to the bug mentioned above was initially not considered a security issue by the docker developers confirms Alice's choice of running her own firewall. As it turns out, many administrators are looking for means to fine-tune their docker firewall because of certain shortcomings in docker [242].

16.3 The First Day

At the beginning of this story, the installed iptables ruleset corresponds to the one listed in Figure 10.12. Over time, a ruleset evolves. On her first day, Alice receives a call from a friend at `heise.de` (193.99.144.80), who is complaining that one of her containers is pinging his webserver excessively. Alice knows that the web backend tests connectivity from time to time by sending one echo request to heise. She decides to postpone investigating the core of the problem and installs a short-term mitigation by just rate limiting all connections to heise. We print changes to the ruleset in unified `diff` format. Alice installs the following rules.

```
 :DOCKER-ISOLATION - [0:0]
 :MYNET - [0:0]
 -A FORWARD -j DOCKER-ISOLATION
+-A FORWARD -d 193.99.144.80 -m recent --set --name rateheise --rsource
+-A FORWARD -d 193.99.144.80 -m recent --update --seconds 60 --hitcount 3         ↩
             --name rateheise --rsource -j DROP
 -A FORWARD -j MYNET
 -A FORWARD -o br-b74b417b331f -j DOCKER
 -A FORWARD -o br-b74b417b331f -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

While this change to the ruleset is identified by her cronjob, *fffuu* confirms that the overall access control structure of the firewall has not changed: It still corresponds to Figure 10.13.

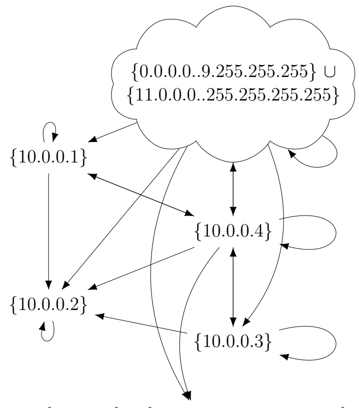
While Alice is updating the ruleset, she remembers from Figure 10.13 that the Internet still has too many direct access rights to her internal containers. This is not a major problem since her docker configuration ultimately prevents unintended accesses from the Internet. Yet, while having opened the ruleset in her editor, Alice decides to fix this issue right away as a second line of defense.

```
 -A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.3 -j ACCEPT
 -A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
 -A MYNET -i br-b74b417b331f -s 10.0.0.4 -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
--A MYNET -i br-b74b417b331f -s 10.0.0.4 ! -o br-b74b417b331f -j ACCEPT
--A MYNET ! -i br-b74b417b331f -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
+-A MYNET -i br-b74b417b331f -s 10.0.0.4 ! -o br-b74b417b331f ! -d 10.0.0.0/8 -j ACCEPT
+-A MYNET ! -i br-b74b417b331f ! -s 10.0.0.0/8 -o br-b74b417b331f -d 10.0.0.1 -j ACCEPT
 -A MYNET -i br-b74b417b331f -j DROP
+-A MYNET -o br-b74b417b331f -j DROP
+-A MYNET -s 10.0.0.0/8 -j DROP
+-A MYNET -d 10.0.0.0/8 -j DROP
 COMMIT
```

After a few seconds, she receives an email with the firewall overview generated by *fffuu*, shown in Figure 16.3. Comparing this figure to Figure 10.13, it can be seen that the Internet is now appropriately constrained.

16.4 A Call from the Web Developer

Continuing the story, Alice receives a call from her web developer. He requests ssh access to all containers. In addition, he requests that the log server (10.0.0.2) may access a status



Copy of Figure 10.13

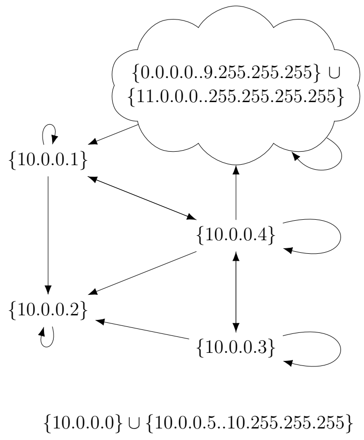


Figure 16.3: Overview computed by *fffuu*

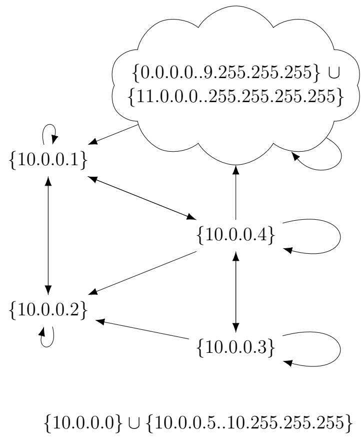


Figure 16.4: Overview computed by *fffuu* after WebDev call

page of the web frontend (10.0.0.1) over HTTP. Both permissions should only be granted temporarily for debugging purposes. Alice sets up the appropriate firewall rules.

```
 -A FORWARD -j DOCKER-ISOLATION
 -A FORWARD -d 193.99.144.80 -m recent --set --name rateheise --rsource
 -A FORWARD -d 193.99.144.80 -m recent --update --seconds 60 --hitcount 3         ↩
            --name rateheise --rsource -j DROP
+-A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
+-A FORWARD -p tcp --dport 22 -j ACCEPT
+-A FORWARD -s 10.0.0.2 -d 10.0.0.1 -p tcp --dport 80 -j ACCEPT
 -A FORWARD -j MYNET
 -A FORWARD -o br-b74b417b331f -j DOCKER
 -A FORWARD -o br-b74b417b331f -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

While the docker container connectivity works as intended, *fffuu* now computes two interesting service matrices.[1] First, it visualizes that there are no longer any restrictions for ssh, i.e., *fffuu* only shows one node. This one node comprises the complete IPv4 address space and may access itself. Second, *fffuu* presents a new HTTP service matrix, shown in Figure 16.4. The only difference is that the log server may access the web frontend. The two service matrices underline that Alice implemented her web developer's request correctly.

16.5 Emergency Response & Scaling Horizontally

While the policy overview computed by *fffuu* is still comprehensible for Alice, the raw firewall ruleset, now comprising 37 rules, is slowly becoming a mess. The ruleset partly contains unused artifacts installed by docker and Alice's hot fixes are cluttered all over it. While Alice is poring over about how she could clean up the rules, she receives an emergency call. Her manager tells her that the webservice was mentioned on reddit and that the web frontend cannot cope with the additional load. Alice is spawning an additional frontend container with IP address 10.0.0.42.

Alice's firewall adheres to best practices and implements whitelisting. Consequently, the new container does not have any connectivity. Alice now is in the urgent situation to get the firewall rules set up which permit connectivity for the second frontend instance. Just

[1]By default, *fffuu* only computes the service matrices for ssh and HTTP.

permitting everything is not an acceptable option for security-aware Alice.

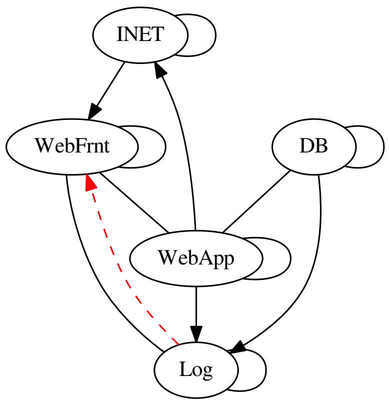


Figure 16.5: Policy Verification reveals a violation (generated by *topoS*)

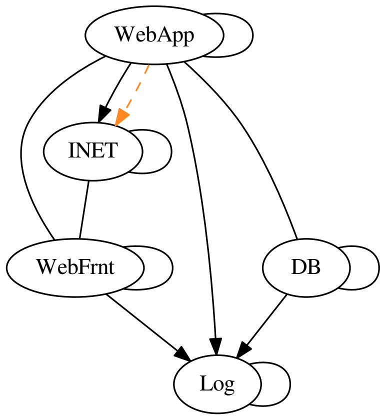


Figure 16.6: Stateful Policy (generated by *topoS*)

Fortunately, Alice remembers that she has specified the security requirements with *topoS* long ago (Figure 10.2). First, Alice loads the policy overview for HTTP computed by *fffuu* (Figure 16.4) into *topoS*. She immediately gets a visualization (Figure 16.5) which tells her that her policy has already diverged from the security requirements: The dashed red arrow, visualized by *topoS*, reveals a flow which is violating the security requirements. It corresponds to the flow installed upon the request of her web developer previously. Based on the security requirements, Alice should probably have rejected the request from her web developer in the very beginning. But this is neither the time to cast the blame nor to discuss security requirements. Given the time pressure and the many odd-looking docker-generated rules in her ruleset, Alice decides that her current ruleset is unsalvageable. Alice is not the first person to notice that the docker daemon may make surprising changes to an iptables configuration [243, 242]. She decides to take over complete control over the ruleset. Therefore, Alice prohibits docker from making any changes to the firewall by setting `--iptables=false`.

```
-A FORWARD -i $dockerbr -s $WebFrnt_ipv4 -o $dockerbr -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebFrnt_ipv4 -o $dockerbr -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebFrnt_ipv4 -o $dockerbr -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebFrnt_ipv4 -o $INET_iface -d $INET_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $DB_ipv4 -o $dockerbr -d $DB_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $DB_ipv4 -o $dockerbr -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $DB_ipv4 -o $dockerbr -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $Log_ipv4 -o $dockerbr -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebApp_ipv4 -o $dockerbr -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebApp_ipv4 -o $dockerbr -d $DB_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebApp_ipv4 -o $dockerbr -d $Log_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebApp_ipv4 -o $dockerbr -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i $dockerbr -s $WebApp_ipv4 -o $INET_iface -d $INET_ipv4 -j ACCEPT
-A FORWARD -i $INET_iface -s $INET_ipv4 -o $dockerbr -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i $INET_iface -s $INET_ipv4 -o $INET_iface -d $INET_ipv4 -j ACCEPT
-I FORWARD -m state --state ESTABLISHED -i $INET_iface                         ↩
           -s $INET_ipv4 -o $dockerbr -d $WebApp_ipv4 -j ACCEPT
```

Figure 16.7: Fresh ruleset generated by *topoS*, considering only the requirements

To start over, Alice asks *topoS* to compute a completely new ruleset for her, based only on the requirements specified before. *topoS* computes the stateful policy shown in Figure 16.6; the dashed orange flow indicates a connection with stateful semantics. The result is serialized to the firewall rules shown in Figure 16.7. All containers are attached to the same docker bridge $\$dockerbr$ = `br-b74b417b331f`. Alice only needs to fill in the IP addresses of the machines and the Internet-facing interface. Alice sticks to the IP addresses of her containers as listed in Table 10.1, except for the web frontend which now has two active containers running. She sets $\$WebFrnt\_ipv4$ = 10.0.0.1,10.0.0.42. This syntax is supported by iptables and iptables will automatically expand this syntax to several rules upon loading. To specify the missing variables, i.e., the interface and IP range of the Internet, Alice simply defines that the Internet is 'everything except her docker subnet'. Therefore, Alice negates her docker interface and internal docker IP range. For example, the second last rule becomes the following:

```
-A FORWARD ! -i br-b74b417b331f ! -s 10.0.0.0/8                              ↩
            ! -o br-b74b417b331f ! -d 10.0.0.0/8 -j ACCEPT
```

Yet, for some rules, the iptables command complains that negation is not allowed with multiple source or destination IP addresses. For example in line four, iptables prohibits the use of `! -d 10.0.0.0/8` in combination with the two source addresses `-s 10.0.0.1,10.0.0.42` specified for $\$WebFrnt\_ipv4$. To work around this iptables limitation, Alice uses the `iprange` module to declare the IP range of the Internet. For example, the fourth rule now becomes

```
-A FORWARD -i br-b74b417b331f -s 10.0.0.1,10.0.0.42                              ↩
           ! -o br-b74b417b331f -m iprange ! --dst-range 10.0.0.0-10.255.255.255 -j ACCEPT
```

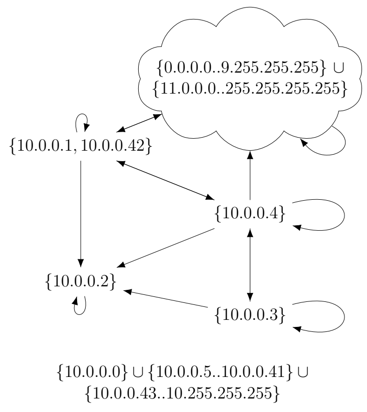


Figure 16.8: Overview of Figure 16.7 computed by *fffuu*

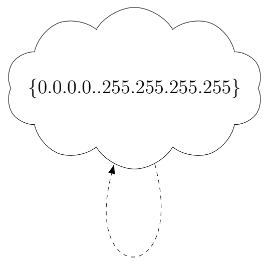


Figure 16.9: Allowed established flows, computed by *fffuu*

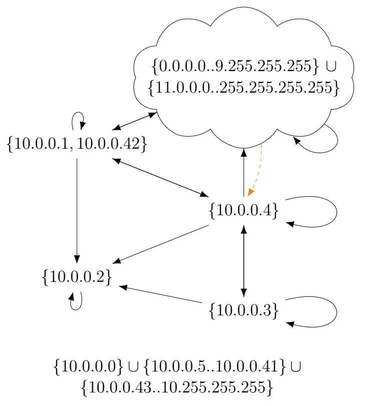


Figure 16.10: HTTP Service Matrix with state (by *fffuu*)

Fortunately, *fffuu* understands all those matching modules. The firewall overview is visualized in Figure 16.8. It is remarkably similar to Figure 16.3, the last, old visualization where the ruleset was still in a good state. The main difference is that the web frontend is now represented by two machines and that it may establish connections to the Internet itself. This has been prohibited by the old policy but it does not contradict any security requirement. A final test confirms that the container connectivity works as expected and the two frontend instances can cope with the load.

16.6 The Logging Information Leak

Looking at her todo list, Alice decides to install some of the old rules again. This time, she designs a clean ruleset and handles all of her temporary rules in a chain she calls `CUSTOM`. After her custom chain, she hands over control to the *topoS*-generated rules. Alice still has not investigated why some container is excessively pinging 193.99.144.80, so she installs the rate limiting again. Alice is more careful about the other temporary rules. *topoS* has shown her that the log server must not communicate with the web frontend, so she is not enabling this rule. However, she does not see a problem with the ssh exception and enables it again. She installs the following rules.

```
 :INPUT ACCEPT [0:0]
 :FORWARD DROP [0:0]
 :OUTPUT ACCEPT [0:0] +:CUSTOM - [0:0]
+-A FORWARD -j CUSTOM
+-A CUSTOM -d 193.99.144.80 -m recent --set --name rateheise --rsource
+-A CUSTOM -d 193.99.144.80 -m recent --update --seconds 60 --hitcount 3      ↩
            --name rateheise --rsource -j DROP
+-A CUSTOM -m state --state ESTABLISHED -j ACCEPT
+-A CUSTOM -p tcp -m tcp --dport 22 -j ACCEPT
 -A FORWARD -i br-b74b417b331f -s 10.0.0.1,10.0.0.42                          ↩
            -o br-b74b417b331f -d 10.0.0.1,10.0.0.42 -j ACCEPT
 -A FORWARD -i br-b74b417b331f -s 10.0.0.1,10.0.0.42                          ↩
            -o br-b74b417b331f -d 10.0.0.2 -j ACCEPT
 -A FORWARD -i br-b74b417b331f -s 10.0.0.1,10.0.0.42                          ↩
            -o br-b74b417b331f -d 10.0.0.4 -j ACCEPT
```

Alice knows that it is a good practice to have a rule which allows all packets belonging to an established connection [202]. She definitely needs an `ESTABLISHED` rule to make ssh work so she just copies it from a guide. Though, Alice wonders why *topoS* did not generate such a rule. Afterwards, she becomes skeptical about her decision and want's to double check. Asking *fffuu* about the potential packet flows once a connection is initiated, *fffuu* confirms that there are currently no limitations at all, not even for HTTP. This is visualized in Figure 16.9. She compares this to the stateful implementation intended by *topoS*, shown in Figure 16.6. The dashed orange line indicates a flow with stateful semantics, i.e., packets may flow in both directions once the connection was initiated by the web app. She realizes that *topoS* takes great care to enforce unidirectional information flow to the log server. This is due to the information sink security invariant specified in the requirements. Alice knows from recent news that a badly protected log server may leak information which may lead to the compromise of all her machines [244]. Therefore, Alice restricts her `ESTABLISHED` rule to ssh. She uses the `multiport` module which conveniently allows to match on source or destination port within one rule. She makes the following final adjustment to her ruleset.

```
 -A CUSTOM
 -A CUSTOM -d 193.99.144.80/32 -m recent --set --name rateheise --rsource
 -A CUSTOM -d 193.99.144.80/32 -m recent --update --seconds 60 --hitcount 3   ↩
            --name rateheise --rsource -j DROP
--A CUSTOM -p tcp -m state --state ESTABLISHED -j ACCEPT
+-A CUSTOM -p tcp -m state --state ESTABLISHED -m multiport --ports 22 -j ACCEPT
 -A CUSTOM -p tcp -m tcp --dport 22 -j ACCEPT
 COMMIT
```

Alice runs one final verification of the implemented policy with *fffuu*, shown in Figure 16.10. This time, she also includes the stateful flows. *fffuu* identifies only one stateful flow, visualized by an orange dashed line. The direction of the stateful flow is the other way round compared to Figure 16.6. This is merely an artifact of the visualization, a stateful flow is essentially bidirectional once it is established. Otherwise, Figure 16.6 and Figure 16.10 show isomorphic graphs. This verifies that Alice's firewall rules are correct.

16.7 Related Docker Work

Tools to improve firewall management for docker hosts exist [245, 246]. Docker-fw [245] is a convenient iptables wrapper with docker-specific features, such as retrieving the IP address from a container name using the Docker API. It currently only supports the default docker bridge, but not custom networks. DFWFW [246] is also a convenient tool to manage the iptables firewall in a docker host. It runs as a daemon and can apply changes dynamically if the docker setup changes, e.g., if new containers are instantiated.

Both tools provide features that could help to make the management process with *topoS* and *fffuu* more convenient. At the moment, *topoS* generates raw iptables rules but leaves the actual IP addresses to be set by the user, e.g., $ *WebFrnt_ipv4* in Figure 16.7. To further automate the setup, *topoS* could generate docker-fw [245] rules which automatically resolve the correct IP address. To further automate firewall management, *topoS* could directly generate DFWFW [246] configurations. This would mean that no manual configuration is required any longer if multiple instances of the same container are spawned.

We have tested *topoS* together with DFWFW. For this, we simply adapted the *topoS* serialization step to generate rules in the DFWFW configuration format. Since DFWFW is also built to primarily support whitelisting, the translation is straightforward. A rule in this format first matches on the docker network (in our scenario, we called the network `mynet`), then it allows to specify the source and destination container, it allows to specify an arbitrary string which will be added to the iptables match expression, and finally the iptables action. The match on the container names permits the use of Perl regular expressions. To allow dynamic spawning of multiple instances of a container, we wrote a regex which matches on the container name and any trailing number, e.g., `webfrnt`, `webfrnt1`, `webfrnt-1`, `webfrnt200`. The beginning of the configuration file looks as follows:

```
{
  "container_to_container": {
  "rules": [
    {
      "network": "mynet",
      "src_container": "Name =~ ^webfrnt-?\\d*$",
      "dst_container": "Name =~ ^webfrnt-?\\d*$",
      "filter": "",
      "action": "ACCEPT"
    },
    {
      "network": "mynet",
      "src_container": "Name =~ ^webfrnt-?\\d*$",
      "dst_container": "Name =~ ^log-?\\d*$",
      "filter": "",
      "action": "ACCEPT"
    },
      ...
```

Disregarding all the line breaks, it is similar to Figure 16.7, but only the first two rules are shown. We tested that all containers have the necessary connectivity with this setting. We also tested that the firewall gets dynamically updated once we instantiate new copies of our containers and that the most obvious attempts to subvert the security policy are successfully blocked. We also verified the generated iptables rules with *fffuu*. It reveals that the overall setting is indeed good, but it also uncovers two open issues: First, Internet connectivity is again unconstrained and the stateless semantics are not enforced correctly, i.e., once a connection with the log server is established, bidirectional communication is permitted. We leave these engineering issues of fine tuning the DFWFW configuration to future work.

16.8 Comparison to Academic State-of-the-Art

We are not aware of any academic related work about network access control management specifically for docker. To the best of our knowledge, our tools are the only ones where applicability for a docker environment has been demonstrated. But docker is merely an application example. We broaden the scope and compare this work to the general state of the art of tools for helping network management and administration. The comparison is outlined in Figure 16.11. The figure is aligned similar to Figure 10.15, but we omit the Interface Abstraction. We add the three security components of Figure 1.1 on the right to visualize the corresponding level of abstraction. The solid single arrows on the left part of the figure visualize work which is able to translate between the corresponding abstractions. The dashed single arrow represents work which is only capable of verification: Given as input an access control policy and security invariants, their conformance can be verified. But one cannot be derived from the other. As discussed Chapter 1, a translation from an access control policy to security invariants is not possible without guessing a policy author's intention.

Our combination of *topoS* and *fffuu* is the only compatible, jointly designed toolset which is able to bridge all levels of abstractions in both directions out of the box. The fictional story reveals that this back-and-forth is useful in several scenarios. The story also reveals single features an administrator may wish for. We now compare *topoS* and *fffuu* with related work specifically for the following use cases, considered in isolation.

Build Networks Based on a Security Requirement Specification Instead of writing a policy or low-level configuration by hand, the fictional story shows that it is useful to generate working network configurations directly from a scenario-specific security requirement specification. This is useful for the initial design and implementation, as well as for starting over in certain scenarios.

VALID [247] allows to express security requirements, but it cannot derive network configurations from them. Zhao et al. [30] also propose a framework which allows to express security requirements. In contrast to VALID, their framework additionally allows to derive working network configurations. The language proposed by Zhao et al. exposes a lot of formalism to the administrator and almost bears more resemblance to programming than it bears to specifying. Compared with this, *topoS* distinguishes between templates and instantiating a template. While defining new templates also bears resemblance to

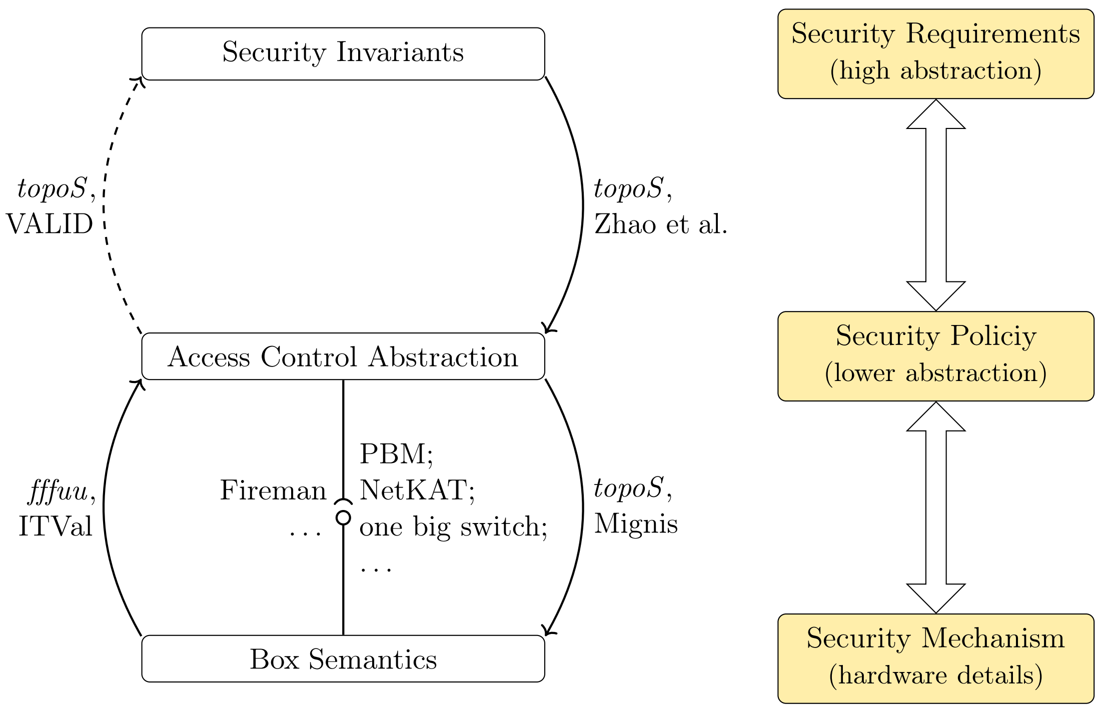

Security Invariants
topoS, VALID
topoS, Zhao et al.
Access Control Abstraction
fffuu, ITVal
Fireman
. . .
PBM; NetKAT; one big switch; . . .
topoS, Mignis
Box Semantics
Security Requirements (high abstraction)
Security Policiy (lower abstraction)
Security Mechanism (hardware details)

Figure 16.11: Overview of Comparison to Related Work

programming and is only intended for expert users, the common operation to define a specification is by instantiating templates, which only requires configurations and exposes very little formalism to the administrator.

Allow Intervention and Low-Level Control for the Administrator The fictional story revealed several occasions where Alice wanted to fine-tune the low-level policy by hand. One example was the temporary, ad-hoc permission for ssh. Another example was the rate limiting to `heise.de`. It may also be imaginable that Alice needs to restructure her ruleset at some point for performance reasons or to support a chain managed by `fail2ban`. Except for the ssh permission, those are true low-level operations which should not be achievable on a higher level of abstraction.[2]

The Mignis [32] firewall configuration language allows to specify filtering policies. It gives the administrator the optional possibility to add arbitrary additional low-level iptables match conditions to the high-level rules. These additional match conditions may introduce soundness issues. The language does not permit the administrator to change the generated iptables rules directly, e.g., reordering, restructuring, or ad-hoc changes without recompiling are not allowed. In contrast, *topoS* permits arbitrary changes to the generated iptables rules since *fffuu* can be used to verify correctness of the changes.

Detect Erosion and Drift of the Implemented Policy vs. the Specified Policy The terms *erosion* and *drift* are usually used for software architectures [127]. However, our fictional story shows that network security policies and the corresponding configurations also easily decay, become unmaintainable, and violations of the original requirements sneak in. In addition, being able to detect differences between a configuration and a specified

[2]tautologically, higher levels of abstraction abstract over low-level details.

policy is an important step towards understanding legacy configurations or verifying manual low-level changes, as discussed in the previous paragraph.

To analyze the current policy enforced by an iptables ruleset, ITVal [135, 136] can be used. Similar to *fffuu*, it allows to partition the complete IPv4 address range into equivalence classes. ITVal computes one set of equivalence classes jointly for all layer 4 ports. In contrast, *fffuu* can only compute them for one selected port. It depends on the scenario which of the two approaches is more suitable: ITVal's overview is very helpful for a first, quick, overview of a firewall. *fffuu*'s service matrices provide better granularity, once one knows which ports one is interested in. ITVal only supports IPv4 while *fffuu* supports IPv4 and IPv6. In addition, ITVal is known to have bugs (Chapter 14) while *fffuu* is formally proven correct. Ironically, ITVal segfaults for some docker rulesets of this chapter while *fffuu* processes them without complaint.

Distributed Enforcement? Our work only focuses on one single, central enforcement device. But having only one central firewall or only one central docker host is not a satisfying scenario. This raises the question whether our work is useless for large installations or whether distributed enforcement is an orthogonal issue. The Interface Abstraction—discussed in Section 10.5 and omitted in Figure 16.11—corresponds to distributed enforcement. Figure 10.15 lists several related work which takes one centralized policy and enforces it in a distributed fashion. Therefore, distributed enforcement is an orthogonal issue and our tools can help to develop, verify, and maintain the centralized policy which is then enforced in a distributed fashion. For example, policy-based management [162] systems or the NetKAT compiler [159] for SDN could be used. We explicitly visualize an interface boundary —(○— in Figure 16.11 to highlight that *topoS* produces an access control matrix, which is understood by many technologies for distributed enforcement. Therefore, *topoS* can be used as a module for access control within another system. In addition, algorithms for distributed firewall analysis (as supported by Fireman [13]) can also benefit from the pre-processing and simplification provided by *fffuu*.

16.9 Conclusion

We showed how our tools *topoS* and *fffuu* can be used to design, manage, and operate docker-based environments. Chapter 10 shows how our toolset helps during the design phase of a setup and this chapter shows how our tools can help during daily operations. We demonstrated several situations in which our tools provide useful feedback, uncover bugs, and even help migrating setups. The duality of *topoS* and *fffuu* in combination with their common policy abstraction makes them a powerful combination and enhances the academic state-of-the-art.

We also enhanced the state-of-the-art of docker container management by combining the dynamic docker firewall framework DFWFW with our static policy management tool *topoS*. While the docker firewall enforces the security policy, *topoS* generates it. This combination lifts *topoS* to dynamic contexts since it allows dynamic spawning and deletion of containers at runtime while still providing strong guarantees about the enforced security requirements. In addition, our tool *fffuu* can verify at runtime the correct operation according to the policy.

Notably, our tools are not limited to docker environments, but also applicable to different scenarios. This becomes explicit as there was not a single change necessary for *fffuu* to support the docker scenarios. To support DFWFW as additional backend, only few lines in the serialization component of *topoS* needed to be adapted.

Chapter 17

Case Study: MeasrDroid Privacy Evaluation and Improvement

This chapter is an extended version of a part of the following paper [93]:

- Marcel von Maltitz, Cornelius Diekmann and Georg Carle, *Taint Analysis for System-Wide Privacy Audits: A Framework and Real-World Case Studies.* In 1st Workshop for Formal Methods on Privacy, Limassol, Cyprus, November 2016. Note: no proceedings published.

Statement on author's contributions The evaluation of MeasrDroid was led by von Maltitz as domain expert on privacy. He provided major contributions for the modeling of MeasrDroid. Both authors, the author of this thesis and von Maltitz, contributed equally to the audit of the real-world MeasrDroid system. Most of the work for the audit was performed by the tools *topoS* and *fffuu* which are developed by the author of this thesis.

Abstract In this chapter, we put all the results of this thesis together, demonstrated by a real-world case study. We audit privacy requirements of the smartphone measurement system MeasrDroid. First, we use *topoS* to model the MeasrDroid architecture and to formalize and verify the privacy requirements. Next, using *fffuu*, we audit the real implementation of MeasrDroid enforced by a network firewall. We uncover previously unknown bugs, use *topoS* to automatically compute a correct firewall ruleset, and deploy the improved ruleset to the real system. Our audit bridges the gap from a high-level security requirement analysis to complex low-level firewall rules. To the best of our knowledge, this is the first time that such an audit has been performed completely with the assurance level provided by the theorem prover Isabelle/HOL.

17.1 Requirements and their Formalization

MeasrDroid [248] is a system for collecting smartphone sensor data for research purposes. The collected data may be privacy-critical. MeasrDroid's architecture is illustrated in Figure 17.1. The figure illustrates three user smartphones. They send their sensor readings over the Internet to *UploadDroid*. Ultimately, the data is stored and analyzed by *CollectDroid*, a trusted machine. To decrease the attack surface of this machine, it is not reachable

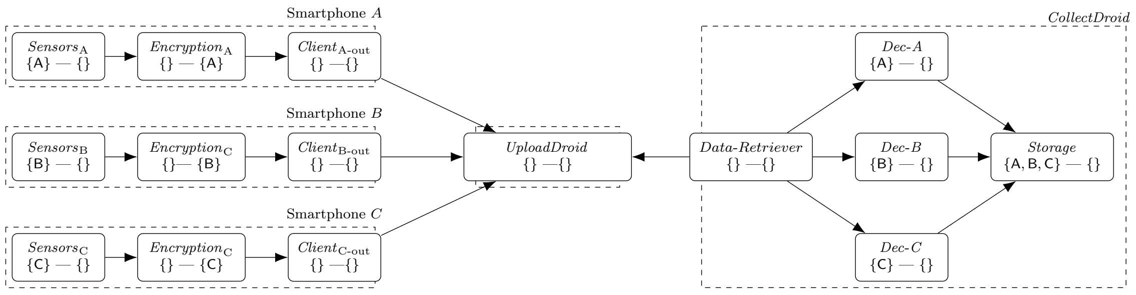


Figure 17.1: MeasrDroid Architecture

over the Internet. Instead, the smartphones push the data to a server called *UploadDroid*. *CollectDroid* in turn regularly polls that server for new information. Since *UploadDroid* is particularly exposed, a compromise of this machine must not lead to a privacy violation of the users. Therefore, *UploadDroid* must be completely uncritical. This is achieved by having the smartphones encrypt the data such that only *CollectDroid* can decrypt again. Consequently, *UploadDroid* only sees encrypted, uncritical data.

We model the system in *topoS* and formalized the privacy requirements. Figure 17.1 provides an overview of the model. The figure shows the architecture and the taint labels (cf. Section 6.16) we assign to each entity. Each smartphone generates individual user data, labeled A, B, and C. The encryption components make the data unreadable for any unauthorized entity, effectively untainting the privacy-critical information. Once the data is decrypted again, the taint labels are restored. The *Storage* is the most critical component as it aggregates all user data. It can be seen that *UploadDroid* is completely uncritical because it does not have any labels at all. In addition to the Tainting invariant, the dotted lines visualize the system boundaries (cf. Section 6.17).

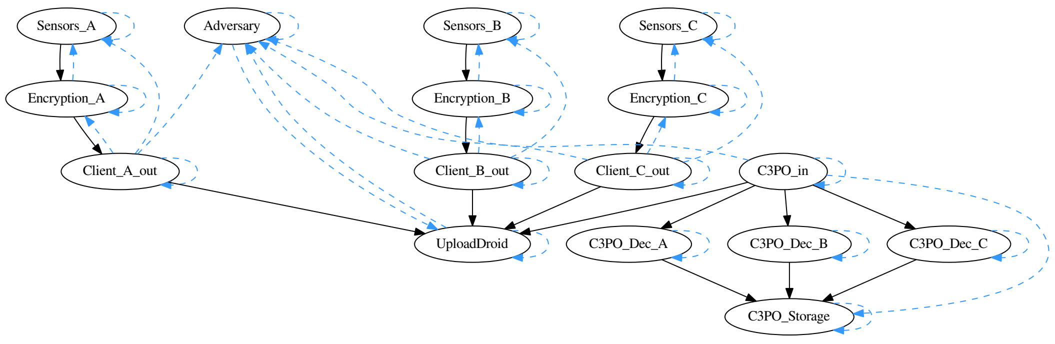


Figure 17.2: Analysis of the MeasrDroid architecture specification (generated by *topoS*)

We evaluate this system specification shown in Figure 17.1.[1] First of all, *topoS* verifies that all security invariants are fulfilled for the presented architecture. Adding an additional adversary node to the architecture, we use *topoS*' policy construction algorithm to visualize which interaction between the entities would be allowed. The visualization generated by *topoS* is shown in Figure 17.2. It is the original unmodified output of *topoS* we obtain during our analysis. In our Isabelle formalization, we give the entities names closer to their

[1] `Examples/Tainting/MeasrDroid.thy`

corresponding name in the actual implementation Most notably, *CollectDroid* is called *C3PO* and the *Data-Retriever* of *CollectDroid* is called *C3PO_in*. We find the following flows that are allowed but not considered in our policy (architecture):

- All reflexive flows, i.e. every component can interact with itself. This is acceptable.
- Within each smartphone, internally, arbitrary communication is possible. We cannot prevent this at a user's smartphone.
- Every smartphone could send data to the adversary. It is important that this is generally allowed since we do not want to put any restrictions on the Internet connectivity of a smartphone. For example, this allows the smartphone user to surf facebook, which is not a trusted component in our system. The collected data is encrypted once it leaves the smartphone (via MeasrDroid). Therefore, sensor data is not leaked.
- *C3PO_in* could send data to the adversary. At this point, the data is still encrypted. It would be possible to add an additional security invariant to make sure that *C3PO_in* only connects to *UploadDroid*.
- An adversary could send data to *UploadDroid*. Since the system shall be accessible to any smartphone connected to the Internet without authentication, we cannot prevent that a malicious user might send fake data.
- The *UploadDroid* could send data to the adversary. This does not undermine the security concept because *UploadDroid* only stores encrypted data. In fact, the security assumptions were from the very beginning that *UploadDroid* can get compromised.
- *CollectDroid* (C3PO) could directly save data in its database without decrypting. This is acceptable and might potentially be used in a future version for backups.

The evaluation of the architecture provides the following high-level conclusions with respect to the overall privacy requirements: Data minimization can already be performed by the client application on the smartphone. This improves control by the data subject, the user. Furthermore, distributed collection better realizes unlinkability directly from the beginning. With regard to asset identification, every smartphone is only critical for the corresponding individual while *CollectDroid* is critical for all users. *UploadDroid* is considered completely uncritical. *CollectDroid* only has an active boundary; intrusion has to be considered as an attack vector and protection using a firewall to restrict incoming connections is vital.

17.2 Auditing the Real MeasrDroid

MeasrDroid is deployed and in productive use since 2013. The previous sections presented a theoretical evaluation of its architecture. This evaluation was not available at the time MeasrDroid was developed. In this section, we evaluate the real MeasrDroid implementation with regard to our findings of the previous sections.

First, we collected all physical and virtual machines which are associated with MeasrDroid. We found the following machines:

droid0 Virtual machine

- IPv4: 131.159.15.16
- IPv6: 2001:4ca0:2001:13:216:3eff:fea7:6ad5
- Name in the model: not present
- Purpose: DNS server, not relevant for MeasrDroid's architecture

droid1 Virtual machine

- IPv4: 131.159.15.42
- IPv6: 2001:4ca0:2001:13:216:3eff:fe03:34f8
- Name in the model: *UploadDroid*
- Purpose: Receive data via http/https

c3po Physical, powerful machine

- IPv4: 131.159.15.52
- IPv6: 2001:4ca0:2001:13:2e0:81ff:fee0:f02e
- Name in the model: *CollectDroid*
- Purpose: Trusted collection and storage

The relevant machines are *UploadDroid* at 131.159.15.42 and *CollectDroid* at 131.159.15.52. We find that the machines do not have a firewall set up. All rely on the central firewall of our lab.

This central firewall may be the largest, real-world, publicly available iptables firewall in the world and handles many different machines and networks. MeasrDroid is only a tiny fragment of it. We obtain a snapshot from June 2016 and make it publicly available [25]. The firewall is managed by several users and consists of over 5500 IPv4 rules.

MeasrDroid relies on the protocols http (port 80), https (port 443), and ssh (port 22). The architecture does not specify any port numbers, but our tool *fffuu* needs a port number to compute a service matrix. For a full audit, the following analysis should be carried out for all port numbers used by MeasrDroid. For conciseness, we focus our audit on port 80. Notably, our theoretical analysis, in particular the model shown in Figure 17.1, has abstracted over concrete port number at all times.

The structure of the MeasrDroid architecture (cf. Figure 17.1) should be recognizable in the rules of our central firewall. In particular, *CollectDroid* should not be reachable from the Internet, *UploadDroid* should be reachable from the Internet, and *CollectDroid* should be able to pull data from *UploadDroid*. This information may be hidden somewhere in the more than 5500 IPv4 and over 6000 IPv6 firewall rules. We use our tool *fffuu* to extract the access control structure of the firewall. The result is visualized in Figure 17.3 for IPv4. The IPv6 structure is shown in Figure 17.5. These figures may first appear highly confusing, which is due to the sheer intrinsic complexity of the access control policy enforced by the firewall. We have highlighted three entities in both figures. Because the structure and the results are similar for IPv4 and IPv6 and due to its long addresses, the IPv6 graph is even worse readable than the IPv4 graph. Thus, we continue our analysis only with IPv4. First, at the top, the IP range enclosed in a cloud corresponds to the IP range that is not used by our department, i.e., the Internet. The large block on the left corresponds to most internal machines that are not globally accessible. The IP address we mark in bold red within this block belongs to *CollectDroid*. Therefore, by inspecting the arrows, we have formally

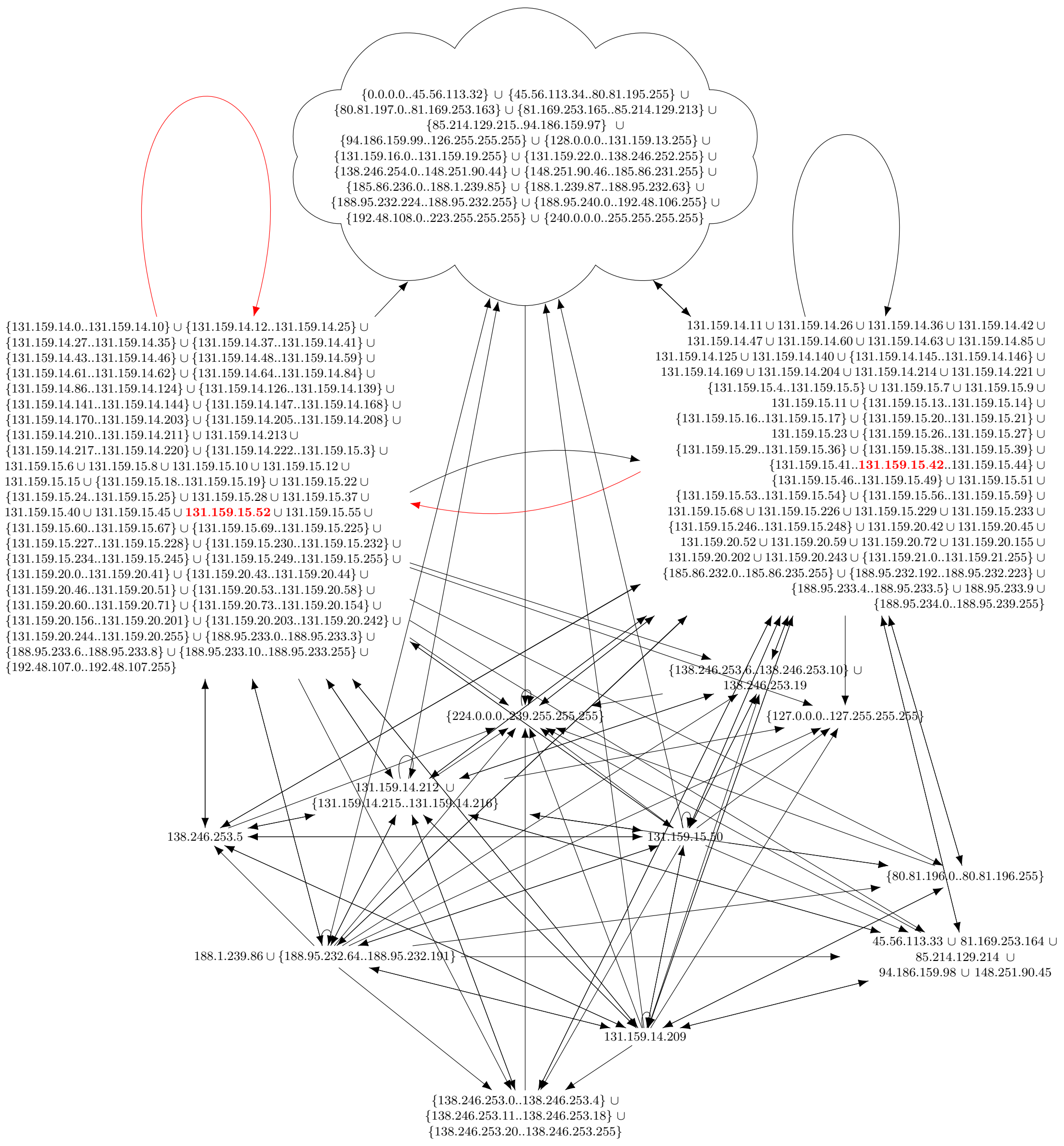


Figure 17.3: MeasrDroid: Main firewall – IPv4 http connectivity matrix

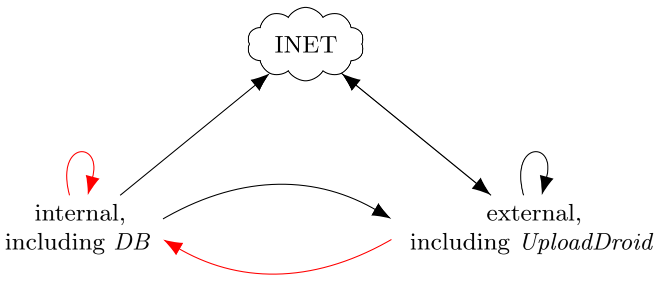


Figure 17.4: MeasrDroid: Main firewall – simplified connectivity matrix

verified our first auditing goal: *CollectDroid* is not directly accessible from the Internet. The other large IP block on the right belongs to machines that are globally accessible. The IP address we marked in bold red within this block belongs to *UploadDroid*. Therefore, we have verified our second auditing goal: *UploadDroid* should be reachable from the Internet. In general, it is pleasant to see that the two machines are in different access groups. Finally, we see that the class of IP addresses including *CollectDroid* can access *UploadDroid*. Hence, we have verified our third auditing goal.

For the sake of example, simplicity, and presentiveness, we disregard that most machines at the bottom of Figure 17.3 could attack *CollectDroid*.[2] Therefore, the huge access control structure at the bottom of Figure 17.3 is not related to MeasrDroid and can be ignored. We extract only the relevant (and simplified) parts in Figure 17.4. In the previous paragraph, we only presented the successful parts of the audit. Our audit also reveals many problems related to MeasrDroid, visualized by red arrows. The problems can be clearly recognized in Figure 17.4:

- *UploadDroid* can connect to *CollectDroid*. This is a clear violation of the architecture. We have empirically verified this highly severe problem by logging into *UploadDroid* and connecting to *CollectDroid*.
- In general, most internal machines may access *CollectDroid*, which violates the architecture.
- There are no restrictions for *UploadDroid* with regard to outgoing connections. In theory, it should only passively retrieve data and never initiate connections by itself (disregarding system updates).
- We uncover a special IP address with special access rights towards *CollectDroid* (only shown in the full figure). We find an abandoned server that has no current relevance for the MeasrDroid system. As a consequence, the access rights are revoked.

Therefore, our audit could verify some core assertions about the actual implementation. In addition, our audit could uncover and confirm serious bugs in the implementation. These bugs were unknown prior to our audit and we could only uncover them with the help of our tools.

17.3 Automatically Fixing Bugs

To fix the problems our audit uncovered, we decide to install additional firewall rules at *CollectDroid*. *topoS* can automatically generate the rules for us. We detail the *topoS*-supported firewall configuration in the next section.

A Firewall for C3PO *topoS* can generate a global firewall for the complete MeasrDroid architecture. We filter the output for rules which affect *CollectDroid*. Note that *topoS* generates a fully functional *stateful* firewall for us. For our architecture, *topoS* generates the two simple rules shown in Figure 17.6.

[2]Our method is also applicable to the complete scenario; this would only decrease clarity without contributing any new insights. We acknowledge the sheer complexity of this real-world setup with all its side-constraints.

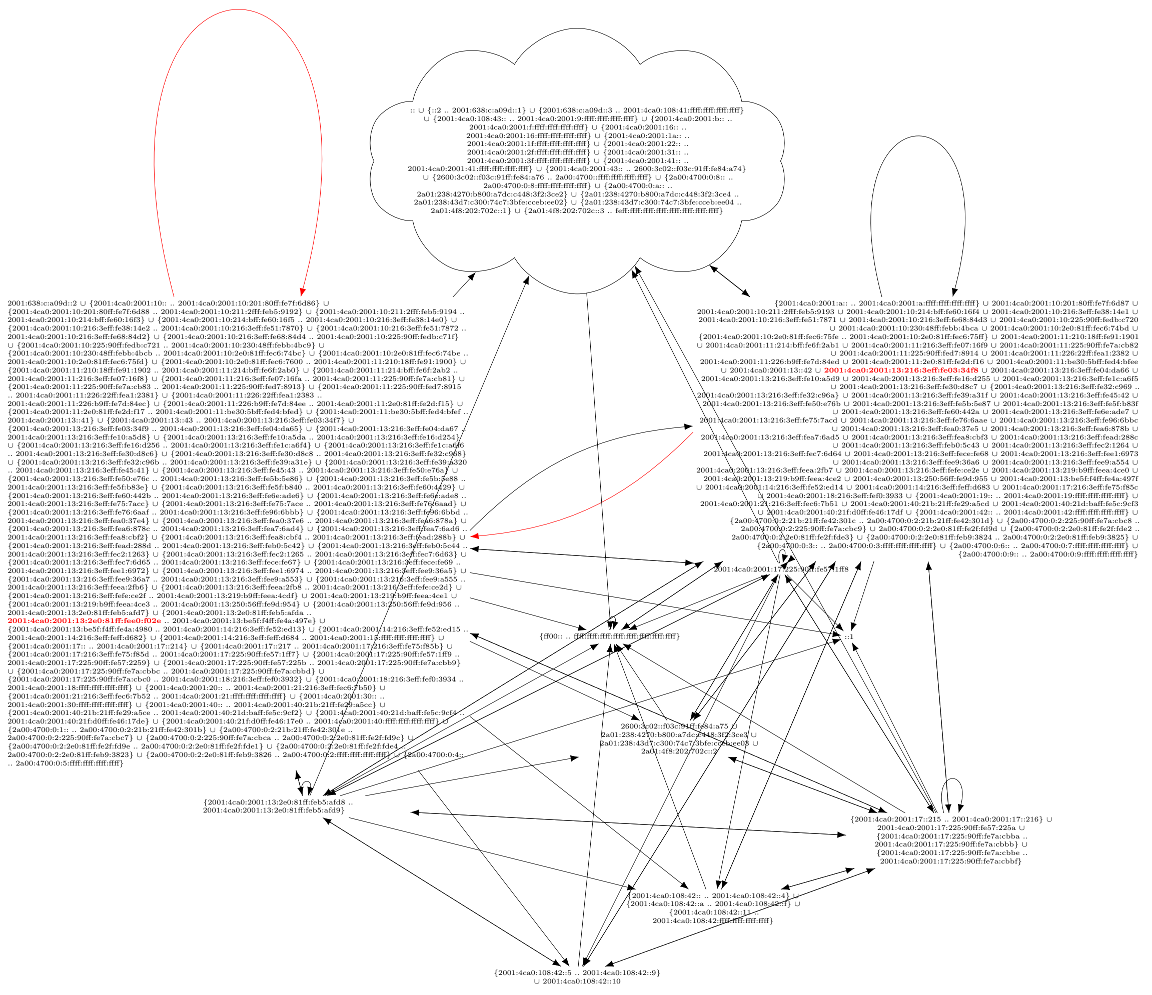

:: ∪ {::2 .. 2001:638:c:a09d::1} ∪ {2001:638:c:a09d::3 .. 2001:4ca0:108:41:ffff:ffff:ffff:ffff}
∪ {2001:4ca0:108:43:: .. 2001:4ca0:2001:9:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:b:: ..
2001:4ca0:2001:f:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:16:: ..
2001:4ca0:2001:16:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:1a:: ..
2001:4ca0:2001:1f:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:22:: ..
2001:4ca0:2001:2f:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:31:: ..
2001:4ca0:2001:3f:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:41:: ..
2001:4ca0:2001:41:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:43:: .. 2600:3c02::f03c:91ff:fe84:a74}
∪ {2600:3c02::f03c:91ff:fe84:a76 .. 2a00:4700::ffff:ffff:ffff:ffff} ∪ {2a00:4700:0:8:: ..
2a00:4700:0:8:ffff:ffff:ffff:ffff} ∪ {2a00:4700:0:a:: ..
2a01:238:4270:b800:a7dc:c448:3f2:3ce2} ∪ {2a01:238:4270:b800:a7dc:c448:3f2:3ce4 ..
2a01:238:43d7:c300:74c7:3bfe:cceb:ee02} ∪ {2a01:238:43d7:c300:74c7:3bfe:cceb:ee04 ..
2a01:4f8:202:702c::1} ∪ {2a01:4f8:202:702c::3 .. feff:ffff:ffff:ffff:ffff:ffff:ffff:ffff}
2001:638:c:a09d::2 ∪ {2001:4ca0:2001:10:: .. 2001:4ca0:2001:10:201:80ff:fe7f:6d86} ∪
{2001:4ca0:2001:10:201:80ff:fe7f:6d88 .. 2001:4ca0:2001:10:211:2fff:feb5:9192} ∪ {2001:4ca0:2001:10:211:2fff:feb5:9194 ..
2001:4ca0:2001:10:214:bff:fe60:16f3} ∪ {2001:4ca0:2001:10:214:bff:fe60:16f5 .. 2001:4ca0:2001:10:216:3eff:fe38:14e0} ∪
{2001:4ca0:2001:10:216:3eff:fe38:14e2 .. 2001:4ca0:2001:10:216:3eff:fe51:7870} ∪ {2001:4ca0:2001:10:216:3eff:fe51:7872 ..
2001:4ca0:2001:10:216:3eff:fe68:84d2} ∪ {2001:4ca0:2001:10:216:3eff:fe68:84d4 .. 2001:4ca0:2001:10:225:90ff:fedb:c71f}
∪ {2001:4ca0:2001:10:225:90ff:fedb:c721 .. 2001:4ca0:2001:10:230:48ff:febb:4bc9} ∪
{2001:4ca0:2001:10:230:48ff:febb:4bcb .. 2001:4ca0:2001:10:2e0:81ff:fec6:74bc} ∪ {2001:4ca0:2001:10:2e0:81ff:fec6:74be ..
2001:4ca0:2001:10:2e0:81ff:fec6:75fd} ∪ {2001:4ca0:2001:10:2e0:81ff:fec6:7600 .. 2001:4ca0:2001:11:210:18ff:fe91:1900} ∪
{2001:4ca0:2001:11:210:18ff:fe91:1902 .. 2001:4ca0:2001:11:214:bff:fe6f:2ab0} ∪ {2001:4ca0:2001:11:214:bff:fe6f:2ab2 ..
2001:4ca0:2001:11:216:3eff:fe07:16f8} ∪ {2001:4ca0:2001:11:216:3eff:fe07:16fa .. 2001:4ca0:2001:11:225:90ff:fe7a:cb81} ∪
{2001:4ca0:2001:11:225:90ff:fe7a:cb83 .. 2001:4ca0:2001:11:225:90ff:fed7:8913} ∪ {2001:4ca0:2001:11:225:90ff:fed7:8915
.. 2001:4ca0:2001:11:226:22ff:fea1:2381} ∪ {2001:4ca0:2001:11:226:22ff:fea1:2383 ..
2001:4ca0:2001:11:226:b9ff:fe7d:84ec} ∪ {2001:4ca0:2001:11:226:b9ff:fe7d:84ee .. 2001:4ca0:2001:11:2e0:81ff:fe2d:f15} ∪
{2001:4ca0:2001:11:2e0:81ff:fe2d:f17 .. 2001:4ca0:2001:11:be30:5bff:fed4:bfed} ∪ {2001:4ca0:2001:11:be30:5bff:fed4:bfef ..
2001:4ca0:2001:13::41} ∪ {2001:4ca0:2001:13::43 .. 2001:4ca0:2001:13:216:3eff:fe03:34f7} ∪
{2001:4ca0:2001:13:216:3eff:fe03:34f9 .. 2001:4ca0:2001:13:216:3eff:fe04:da65} ∪ {2001:4ca0:2001:13:216:3eff:fe04:da67 ..
2001:4ca0:2001:13:216:3eff:fe10:a5d8} ∪ {2001:4ca0:2001:13:216:3eff:fe10:a5da .. 2001:4ca0:2001:13:216:3eff:fe16:d254}
∪ {2001:4ca0:2001:13:216:3eff:fe16:d256 .. 2001:4ca0:2001:13:216:3eff:fe1c:a6f4} ∪ {2001:4ca0:2001:13:216:3eff:fe1c:a6f6
.. 2001:4ca0:2001:13:216:3eff:fe30:d8c6} ∪ {2001:4ca0:2001:13:216:3eff:fe30:d8c8 .. 2001:4ca0:2001:13:216:3eff:fe32:c968}
∪ {2001:4ca0:2001:13:216:3eff:fe32:c96b .. 2001:4ca0:2001:13:216:3eff:fe39:a31e} ∪ {2001:4ca0:2001:13:216:3eff:fe39:a320
.. 2001:4ca0:2001:13:216:3eff:fe45:41} ∪ {2001:4ca0:2001:13:216:3eff:fe45:43 .. 2001:4ca0:2001:13:216:3eff:fe50:e76a} ∪
{2001:4ca0:2001:13:216:3eff:fe50:e76c .. 2001:4ca0:2001:13:216:3eff:fe5b:5e86} ∪ {2001:4ca0:2001:13:216:3eff:fe5b:5e88 ..
2001:4ca0:2001:13:216:3eff:fe5f:b83e} ∪ {2001:4ca0:2001:13:216:3eff:fe5f:b840 .. 2001:4ca0:2001:13:216:3eff:fe60:4429} ∪
{2001:4ca0:2001:13:216:3eff:fe60:442b .. 2001:4ca0:2001:13:216:3eff:fe6e:ade6} ∪ {2001:4ca0:2001:13:216:3eff:fe6e:ade8 ..
2001:4ca0:2001:13:216:3eff:fe75:7acc} ∪ {2001:4ca0:2001:13:216:3eff:fe75:7ace .. 2001:4ca0:2001:13:216:3eff:fe76:6aad} ∪
{2001:4ca0:2001:13:216:3eff:fe76:6aaf .. 2001:4ca0:2001:13:216:3eff:fe96:6bbb} ∪ {2001:4ca0:2001:13:216:3eff:fe96:6bbd ..
2001:4ca0:2001:13:216:3eff:fea0:37e4} ∪ {2001:4ca0:2001:13:216:3eff:fea0:37e6 .. 2001:4ca0:2001:13:216:3eff:fea6:878a} ∪
{2001:4ca0:2001:13:216:3eff:fea6:878c .. 2001:4ca0:2001:13:216:3eff:fea7:6ad4} ∪ {2001:4ca0:2001:13:216:3eff:fea7:6ad6 ..
2001:4ca0:2001:13:216:3eff:fea8:cbf2} ∪ {2001:4ca0:2001:13:216:3eff:fea8:cbf4 .. 2001:4ca0:2001:13:216:3eff:fead:288b} ∪
{2001:4ca0:2001:13:216:3eff:fead:288d .. 2001:4ca0:2001:13:216:3eff:feb0:5c42} ∪ {2001:4ca0:2001:13:216:3eff:feb0:5c44 ..
2001:4ca0:2001:13:216:3eff:fec2:1263} ∪ {2001:4ca0:2001:13:216:3eff:fec2:1265 .. 2001:4ca0:2001:13:216:3eff:fec7:6d63} ∪
{2001:4ca0:2001:13:216:3eff:fec7:6d65 .. 2001:4ca0:2001:13:216:3eff:fece:fe67} ∪ {2001:4ca0:2001:13:216:3eff:fece:fe69 ..
2001:4ca0:2001:13:216:3eff:fee1:6972} ∪ {2001:4ca0:2001:13:216:3eff:fee1:6974 .. 2001:4ca0:2001:13:216:3eff:fee9:36a5} ∪
{2001:4ca0:2001:13:216:3eff:fee9:36a7 .. 2001:4ca0:2001:13:216:3eff:fee9:a553} ∪ {2001:4ca0:2001:13:216:3eff:fee9:a555 ..
2001:4ca0:2001:13:216:3eff:feea:2fb6} ∪ {2001:4ca0:2001:13:216:3eff:feea:2fb8 .. 2001:4ca0:2001:13:216:3eff:fefe:ce2d} ∪
{2001:4ca0:2001:13:216:3eff:fefe:ce2f .. 2001:4ca0:2001:13:219:b9ff:feea:4cdf} ∪ 2001:4ca0:2001:13:219:b9ff:feea:4ce1 ∪
{2001:4ca0:2001:13:219:b9ff:feea:4ce3 .. 2001:4ca0:2001:13:250:56ff:fe9d:954} ∪ {2001:4ca0:2001:13:250:56ff:fe9d:956 ..
2001:4ca0:2001:13:2e0:81ff:feb5:afd7} ∪ {2001:4ca0:2001:13:2e0:81ff:feb5:afda ..
2001:4ca0:2001:13:2e0:81ff:fee0:f02e .. 2001:4ca0:2001:13:be5f:f4ff:fe4a:497e} ∪
{2001:4ca0:2001:13:be5f:f4ff:fe4a:4980 .. 2001:4ca0:2001:14:216:3eff:fe52:ed13} ∪ {2001:4ca0:2001:14:216:3eff:fe52:ed15 ..
2001:4ca0:2001:14:216:3eff:feff:d682} ∪ {2001:4ca0:2001:14:216:3eff:feff:d684 .. 2001:4ca0:2001:15:ffff:ffff:ffff:ffff} ∪
{2001:4ca0:2001:17:: .. 2001:4ca0:2001:17::214} ∪ {2001:4ca0:2001:17::217 .. 2001:4ca0:2001:17:216:3eff:fe75:f85b} ∪
{2001:4ca0:2001:17:216:3eff:fe75:f85d .. 2001:4ca0:2001:17:225:90ff:fe57:1ff7} ∪ {2001:4ca0:2001:17:225:90ff:fe57:1ff9 ..
2001:4ca0:2001:17:225:90ff:fe57:2259} ∪ {2001:4ca0:2001:17:225:90ff:fe57:225b .. 2001:4ca0:2001:17:225:90ff:fe7a:cbb9}
∪ {2001:4ca0:2001:17:225:90ff:fe7a:cbbc .. 2001:4ca0:2001:17:225:90ff:fe7a:cbbd} ∪
{2001:4ca0:2001:17:225:90ff:fe7a:cbc0 .. 2001:4ca0:2001:18:216:3eff:fef0:3932} ∪ {2001:4ca0:2001:18:216:3eff:fef0:3934 ..
2001:4ca0:2001:18:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:20:: .. 2001:4ca0:2001:21:216:3eff:fec6:7b50} ∪
{2001:4ca0:2001:21:216:3eff:fec6:7b52 .. 2001:4ca0:2001:21:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:30:: ..
2001:4ca0:2001:30:ffff:ffff:ffff:ffff} ∪ {2001:4ca0:2001:40:: .. 2001:4ca0:2001:40:21b:21ff:fe29:a5cc} ∪
{2001:4ca0:2001:40:21b:21ff:fe29:a5ce .. 2001:4ca0:2001:40:21d:baff:fe5c:9cf2} ∪ {2001:4ca0:2001:40:21d:baff:fe5c:9cf4 ..
2001:4ca0:2001:40:21f:d0ff:fe46:17de} ∪ {2001:4ca0:2001:40:21f:d0ff:fe46:17e0 .. 2001:4ca0:2001:40:ffff:ffff:ffff:ffff} ∪
{2a00:4700:0:1:: .. 2a00:4700:0:2:21b:21ff:fe42:301b} ∪ {2a00:4700:0:2:21b:21ff:fe42:301e ..
2a00:4700:0:2:225:90ff:fe7a:cbc7} ∪ {2a00:4700:0:2:225:90ff:fe7a:cbca .. 2a00:4700:0:2:2e0:81ff:fe2f:fd9c} ∪
{2a00:4700:0:2:2e0:81ff:fe2f:fd9e .. 2a00:4700:0:2:2e0:81ff:fe2f:fde1} ∪ {2a00:4700:0:2:2e0:81ff:fe2f:fde4 ..
2a00:4700:0:2:2e0:81ff:feb9:3823} ∪ {2a00:4700:0:2:2e0:81ff:feb9:3826 .. 2a00:4700:0:2:ffff:ffff:ffff:ffff} ∪ {2a00:4700:0:4::
.. 2a00:4700:0:5:ffff:ffff:ffff:ffff}
{2001:4ca0:2001:a:: .. 2001:4ca0:2001:a:ffff:ffff:ffff:ffff} ∪ 2001:4ca0:2001:10:201:80ff:fe7f:6d87 ∪
2001:4ca0:2001:10:211:2fff:feb5:9193 ∪ 2001:4ca0:2001:10:214:bff:fe60:16f4 ∪ 2001:4ca0:2001:10:216:3eff:fe38:14e1 ∪
2001:4ca0:2001:10:216:3eff:fe51:7871 ∪ 2001:4ca0:2001:10:216:3eff:fe68:84d3 ∪ 2001:4ca0:2001:10:225:90ff:fedb:c720
∪ 2001:4ca0:2001:10:230:48ff:febb:4bca ∪ 2001:4ca0:2001:10:2e0:81ff:fec6:74bd ∪
{2001:4ca0:2001:10:2e0:81ff:fec6:75fe .. 2001:4ca0:2001:10:2e0:81ff:fec6:75ff} ∪ 2001:4ca0:2001:11:210:18ff:fe91:1901
∪ 2001:4ca0:2001:11:214:bff:fe6f:2ab1 ∪ 2001:4ca0:2001:11:216:3eff:fe07:16f9 ∪ 2001:4ca0:2001:11:225:90ff:fe7a:cb82
∪ 2001:4ca0:2001:11:225:90ff:fed7:8914 ∪ 2001:4ca0:2001:11:226:22ff:fea1:2382 ∪
2001:4ca0:2001:11:226:b9ff:fe7d:84ed ∪ 2001:4ca0:2001:11:2e0:81ff:fe2d:f16 ∪ 2001:4ca0:2001:11:be30:5bff:fed4:bfee
∪ 2001:4ca0:2001:13::42 ∪ **2001:4ca0:2001:13:216:3eff:fe03:34f8** ∪ 2001:4ca0:2001:13:216:3eff:fe04:da66 ∪
2001:4ca0:2001:13:216:3eff:fe10:a5d9 ∪ 2001:4ca0:2001:13:216:3eff:fe16:d255 ∪ 2001:4ca0:2001:13:216:3eff:fe1c:a6f5
∪ 2001:4ca0:2001:13:216:3eff:fe30:d8c7 ∪ {2001:4ca0:2001:13:216:3eff:fe32:c969 ..
2001:4ca0:2001:13:216:3eff:fe32:c96a} ∪ 2001:4ca0:2001:13:216:3eff:fe39:a31f ∪ 2001:4ca0:2001:13:216:3eff:fe45:42 ∪
2001:4ca0:2001:13:216:3eff:fe50:e76b ∪ 2001:4ca0:2001:13:216:3eff:fe5b:5e87 ∪ 2001:4ca0:2001:13:216:3eff:fe5f:b83f
∪ 2001:4ca0:2001:13:216:3eff:fe60:442a ∪ 2001:4ca0:2001:13:216:3eff:fe6e:ade7 ∪
2001:4ca0:2001:13:216:3eff:fe75:7acd ∪ 2001:4ca0:2001:13:216:3eff:fe76:6aae ∪ 2001:4ca0:2001:13:216:3eff:fe96:6bbc
∪ 2001:4ca0:2001:13:216:3eff:fea0:37e5 ∪ 2001:4ca0:2001:13:216:3eff:fea6:878b ∪
2001:4ca0:2001:13:216:3eff:fea7:6ad5 ∪ 2001:4ca0:2001:13:216:3eff:fea8:cbf3 ∪ 2001:4ca0:2001:13:216:3eff:fead:288c
∪ 2001:4ca0:2001:13:216:3eff:feb0:5c43 ∪ 2001:4ca0:2001:13:216:3eff:fec2:1264 ∪
2001:4ca0:2001:13:216:3eff:fec7:6d64 ∪ 2001:4ca0:2001:13:216:3eff:fece:fe68 ∪ 2001:4ca0:2001:13:216:3eff:fee1:6973
∪ 2001:4ca0:2001:13:216:3eff:fee9:36a6 ∪ 2001:4ca0:2001:13:216:3eff:fee9:a554 ∪
2001:4ca0:2001:13:216:3eff:feea:2fb7 ∪ 2001:4ca0:2001:13:216:3eff:fefe:ce2e ∪ 2001:4ca0:2001:13:219:b9ff:feea:4ce0 ∪
2001:4ca0:2001:13:219:b9ff:feea:4ce2 ∪ 2001:4ca0:2001:13:250:56ff:fe9d:955 ∪ 2001:4ca0:2001:13:be5f:f4ff:fe4a:497f
∪ 2001:4ca0:2001:14:216:3eff:fe52:ed14 ∪ 2001:4ca0:2001:14:216:3eff:feff:d683 ∪ 2001:4ca0:2001:17:216:3eff:fe75:f85c
∪ 2001:4ca0:2001:18:216:3eff:fef0:3933 ∪ {2001:4ca0:2001:19:: .. 2001:4ca0:2001:19:ffff:ffff:ffff:ffff} ∪
2001:4ca0:2001:21:216:3eff:fec6:7b51 ∪ 2001:4ca0:2001:40:21b:21ff:fe29:a5cd ∪ 2001:4ca0:2001:40:21d:baff:fe5c:9cf3
∪ 2001:4ca0:2001:40:21f:d0ff:fe46:17df ∪ {2001:4ca0:2001:42:: .. 2001:4ca0:2001:42:ffff:ffff:ffff:ffff} ∪
{2a00:4700:0:2:21b:21ff:fe42:301c .. 2a00:4700:0:2:21b:21ff:fe42:301d} ∪ {2a00:4700:0:2:225:90ff:fe7a:cbc8 ..
2a00:4700:0:2:225:90ff:fe7a:cbc9} ∪ 2a00:4700:0:2:2e0:81ff:fe2f:fd9d ∪ {2a00:4700:0:2:2e0:81ff:fe2f:fde2 ..
2a00:4700:0:2:2e0:81ff:fe2f:fde3} ∪ {2a00:4700:0:2:2e0:81ff:feb9:3824 .. 2a00:4700:0:2:2e0:81ff:feb9:3825} ∪
{2a00:4700:0:3:: .. 2a00:4700:0:3:ffff:ffff:ffff:ffff} ∪ {2a00:4700:0:6:: .. 2a00:4700:0:7:ffff:ffff:ffff:ffff} ∪
{2a00:4700:0:9:: .. 2a00:4700:0:9:ffff:ffff:ffff:ffff}
2001:4ca0:2001:17:225:90ff:fe57:1ff8
{ff00:: .. ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff}
::1
2600:3c02::f03c:91ff:fe84:a75 ∪
2a01:238:4270:b800:a7dc:c448:3f2:3ce3 ∪
2a01:238:43d7:c300:74c7:3bfe:cceb:ee03 ∪
2a01:4f8:202:702c::2
{2001:4ca0:2001:13:2e0:81ff:feb5:afd8 ..
2001:4ca0:2001:13:2e0:81ff:feb5:afd9}
{2001:4ca0:2001:17::215 .. 2001:4ca0:2001:17::216} ∪
2001:4ca0:2001:17:225:90ff:fe57:225a ∪
{2001:4ca0:2001:17:225:90ff:fe7a:cbba ..
2001:4ca0:2001:17:225:90ff:fe7a:cbbb} ∪
{2001:4ca0:2001:17:225:90ff:fe7a:cbbe ..
2001:4ca0:2001:17:225:90ff:fe7a:cbbf}
{2001:4ca0:108:42:: .. 2001:4ca0:108:42::4} ∪
{2001:4ca0:108:42::a .. 2001:4ca0:108:42::f} ∪
{2001:4ca0:108:42::11 ..
2001:4ca0:108:42:ffff:ffff:ffff:ffff}
{2001:4ca0:108:42::5 .. 2001:4ca0:108:42::9}
∪ 2001:4ca0:108:42::10

Figure 17.5: MeasrDroid: Main firewall – IPv6 http connectivity matrix

```
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]
-A OUTPUT -s 131.159.15.52 -d 131.159.15.42 -j ACCEPT
-A INPUT -m state --state ESTABLISHED -s 131.159.15.42 -d 131.159.15.52 -j ACCEPT
COMMIT
```

Figure 17.6: Automatically generated `iptables` rules

The first rule allows *CollectDroid* to connect to the *UploadDroid*. The second rule allows the *UploadDroid* to answer to such existing connections.

```
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]
# topoS generated:
# C3PO -> UploadDroid
-A OUTPUT -s 131.159.15.52 -d 131.159.15.42 -j ACCEPT
# UploadDroid -> C3PO (answer)
-A INPUT -m state --state ESTABLISHED -s 131.159.15.42 -d 131.159.15.52 -j ACCEPT
# custom additional rules
-A INPUT -i lo -j ACCEPT
-A OUTPUT -o lo -j ACCEPT
-A OUTPUT -p icmp -j ACCEPT
-A INPUT -p icmp -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -s 131.159.20.190/24 -p tcp -m tcp --dport 22 -j ACCEPT
-A OUTPUT -m state --state ESTABLISHED -p tcp -m tcp --sport 22 -j ACCEPT
# DHCP
-A INPUT -p udp --dport 67:68 --sport 67:68 -j ACCEPT
-A OUTPUT -p udp --dport 67:68 --sport 67:68 -j ACCEPT
# ntp
-A OUTPUT -p udp --dport 123 -j ACCEPT
-A INPUT -p udp --sport 123 -j ACCEPT
# DNS
-A OUTPUT -p udp --dport 53 -j ACCEPT
-A INPUT -p udp --sport 53 -m state --state ESTABLISHED  -j ACCEPT
# further output, policy could be improved.
# Notice mails to admin, system updates, ...
-A OUTPUT -p tcp -j LOG
-A OUTPUT -p tcp -j ACCEPT
-A INPUT -p tcp -m state --state ESTABLISHED -j LOG
-A INPUT -p tcp -m state --state ESTABLISHED -j ACCEPT
COMMIT
```

Figure 17.7: Manually tuned `iptables` rules

Afterwards, we manually extend the rules to further allow some core network services such as ICMP, DHCP, DNS, NTP, and SSH for remote management. In addition, we allow further outgoing TCP connections from *CollectDroid* (for example for system updates) but log those packets. The modified ruleset is illustrated in Figure 17.7.

We analyze the modified firewall with *fffuu* to ensure that it still conforms to the overall policy. *fffuu* immediately verifies that *CollectDroid* is no longer reachable from any machine (excluding localhost connection) over HTTP. We further verify that we do not lock ourselves out from ssh access from our internal network. After this verification, the firewall is deployed to the real *CollectDroid* machine. Similarly, we implement and deploy an IPv6 firewall.

17.4 Conclusion

We used *topoS* to model the system architecture of MeasrDroid and to formalize privacy and security requirements. Using the built-in verification capabilities of *topoS*, we verified that our formalization corresponds to our intention and that its design guarantees the desired privacy properties. Using *fffuu* to analyze whether the existing firewall correctly enforces the modeled policy, we uncovered previously unknown errors. In turn, we used *topoS* to generate a correct firewall for us, applied some manual low-level improvements, and used our tools again to verify that these improvements do not violate the requirements.

Chapter 18

Achieved Scientific Results

Q: What is the importance of specifications and verifications to industry today and how do you think they will evolve over time? For example, do you believe that more automated tools will help improve the state of the art?

A: There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding that complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought.

Interview with L. Lamport, (2002) [249].

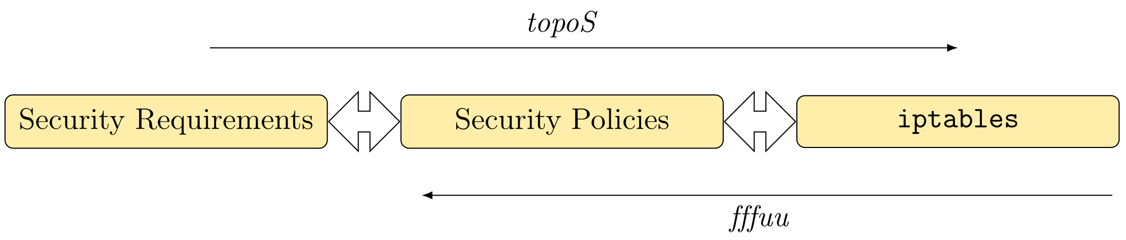
18.1 Answers to the Research Questions

In Section 1.2, we asked two guiding research questions and identified four non-functional requirements that a solution must fulfill. In this section, we summarize our answers and elaborate on how we have fulfilled the non-functional requirements.

Fulfilling the Non-Functional Requirements

NF1 *Can we provide automated tools for the solutions to **Q1** and **Q2**?*

We contribute our fully automated tools *topoS* and *fffuu*. Given the input to the tools i.e., configured security invariants for *topoS* or an iptables firewall for *fffuu*, the tools can operate completely automatic, not requiring any manual proof at runtime.



NF2 *Can the correctness of the tools be justified?*

We have formally and machine-checked proven the correctness of our theory with Isabelle/HOL. The core reasoning logic of both tools is generated by Isabelle, which guarantees partial correctness. In this context *partial* correctness [110, 111] means that the computed results are correct. It is not formally guaranteed that there may always be results, e.g., non-termination or memory limits may abort a computation. Our empirical evaluation shows that the generated code both terminates and runs efficiently on commodity hardware. Consequently, disregarding errors in the user interface or parser, results computed by our tools are formally guaranteed to be correct.

NF3 *Is over-formalism exposed to the administrator?*

Our tool *topoS* requires a specification of the security requirements. However, a user is not required to encode this specification with complicated formulas. Our attribute-based approach means that a user can simply assign attributes to entities. We have demonstrated with the Scala implementation of our tool that a simple set of JSON configuration files is sufficient. The secure auto completion further minimizes manual specification effort. Our tool *fffuu* exposes even less formalism to its users: By default, it does not require any manual input from its users and outputs visualizable results.

NF4 *Are the solutions to **Q1** and **Q2** compatible?*

Our methodology works on well-defined intermediate results and gives the administrator full control over them at all levels of abstraction. We represent policies as graphs, which is the common abstraction level where both directions presented in this thesis (synthesizing new policies vs. verifying existing policies) meet. The two directions are compatible with each other and an administrator may freely choose to which extent she wants to use our methodology and to which extent she wants to remain in full control. It is even possible to use our toolset in a full circle, illustrated in Figure 18.1: Starting at an arbitrary step, one can (*i*) infer the policy of a legacy configuration and (*ii*) verify the policy w.r.t. high-level security requirements. Afterwards, one can (*iii*) abolish the old configuration and synthesize a new, clearer configuration out of the security requirements. Next, one can (*iv*) apply custom low-level optimizations to the synthesized policy or merge it with parts of the legacy configuration again. Afterwards, one can (*v*) verify that it still complies with the specified high-level requirements by abstracting the low-level policy to a policy graph. This corresponds again to steps (*i*) and (*ii*), which closes the circle.

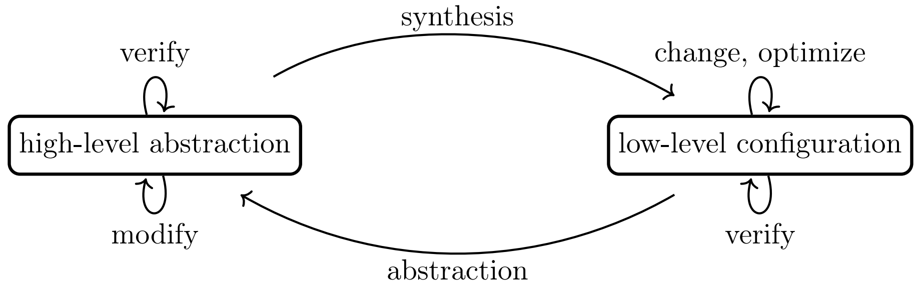


Figure 18.1: Tool-Supported Full Circle

Answers to Question 1

"How can we design secure networks from scratch?"

Q1.1 *How can the security requirements be specified?*

We contribute a theory and generic theoretical framework which allows specifying security requirements as predicates over a policy. Our formalization of security invariants is composable by construction and thus allows to modularize the security requirement specification. It exposes low manual configuration overhead by the secure auto completion and our library of ready-to-use templates. We present a generic method to describe offending flows and proved efficient implementations for certain invariants. This enables easy debugging of a requirements specification and provides visual feedback.

Q1.2 *How can a security policy be derived from the requirements?*

The formalization of security invariants is carefully designed in many iterations such that both automatic derivation of a policy and automatic verification of policies are supported. We further present enhancements that improve their performance. Our evaluations show that the method scales well.

Q1.3 *How can a policy be deployed to real network security mechanisms?*

We augment our graph representation of a policy to incorporate the notion of stateful connections. This extended graph can be directly serialized to several security mechanisms. We identify the assumptions that a real-world setup must fulfill in order to be compatible with the theory. By deploying scenarios to actual networks, we demonstrate applicability of our theory for firewalls, OpenFlow-enabled switches, and containers. For the iptables firewall, we have proven the correctness of the final results.

Answers to Question 2

"How can we analyze and verify existing configurations?"

Q2.1 *What are the semantics of a security mechanism?*

We contribute a formal semantics of iptables packet filtering. To the best of our knowledge, it is the first and only formal semantics for a complicated firewall system which is complete with regard to the matching features and the only formal semantics that supports control flow with user-defined chains.

Q2.2 *How does an entity in a security mechanism configuration correspond to an entity in a policy?*

First, we present an algorithm which verifies that a firewall ruleset implements spoofing protection. This allows to refer to entities by their IP addresses. Afterwards, we present an algorithm to group sets of IP addresses by their access rights, effectively computing the vertices of the graph representation.

Q2.3 *How can a high-level policy be derived from a low-level security mechanism configuration?*

First, we contribute an algorithm which transforms a complicated iptables ruleset to a simpler firewall model, abstracting over all unnecessary low-level details. Despite abstracting over low-level details, we have proven soundness of our approach, but not necessarily completeness due to the loss of low-level details. Any sound approach which converts a complex language into a simpler less expressive language is necessarily incomplete.

Our approach enables to compute an overview of the connectivity permitted by a firewall. This enables sound security evaluation: the actual firewall definitely does not allow more than our simplified firewall, but due to the potential incompleteness, the actual firewall could drop even more.[1] Vice versa, our approach also allows reasoning about the set of packets that a firewall definitely accepts. Despite the incompleteness, the usefulness of our approximation has been successfully evaluated in many real-world rulesets.

Second, we contribute an algorithm which partitions the complete IPv4 and IPv6 address space into equivalence classes with equal access rights for a fixed service. This yields a graph which represents the policy enforced by a firewall, effectively computing a minimal access control matrix. This graph can be visualized and further verified with the answers we contributed for **Q1**.

Q2.4 *Can a derived high-level policy be verified w.r.t. a given set of security requirements?*

As we already mentioned when asking the questions, question **Q1.2** is strengthened such that an answer to it must already entail an answer to this question. Indeed, our theory directly supports this and our answer to **NF1** shows that our methodology supports the full circle.

18.2 Tackling Complexity in Network Security Management

In Section 2.2, we discussed the complexity challenges in software engineering and how they can be interpreted in the context of network security management. We distinguished between *accidental* difficulties and *essential* difficulties. In this section, we sum up how we have approached these difficulties in this thesis.

A large set of related work in the field of configuration management only tackles accidental complexity. Many attempts have been made before to hide low-level (accidentally complex) configuration aspects with a high-level language [32, 14, 168, 34, 58]. But this cannot solve the core problem of essential difficulties.

Essential difficulties cannot be removed. However, in this thesis, we use the following key ideas to tackle them as good as possible:

- *complexity*: Intrinsic complexity cannot be removed. But we can isolate and modularize certain aspects to only focus on one problem at once. For example, in Chapter 5, we break down the task of specifying security requirements into several isolated tasks of specifying well-defined security invariants. Our design ensures composability of these invariants. In Chapter 6 we provide a library to further isolate the generic part of security invariants (template) from the scenario-specific configuration (host attribute

[1]w.r.t. our monotonicity assumption, dropping more does not decrease security

mapping). In Chapter 9, we isolate stateless access control from its stateful network implementation. In Chapter 14, we completely isolate low-level firewall primitives and access control.

- *conformity*: Part II is completely devoted to the understanding and verification existing iptables configurations.
- *changeability*: We provide a formal proof for all our algorithms. The proof, by its very nature, reveals all implicit assumption and is valid under any worst-case assumptions, thus covering all unforeseeable cases.
- *invisibility*: We provide clear, intermediate artifacts. All results of our algorithms can be presented and visualized as graph.

With regard to this view, the structure of this thesis can be understood as follows.

Part I Breaking down the *complexity* of network security management into several isolated aspects and creating *visible* intermediate results.

Part II Ensuring *conformity* with existing, legacy configurations. Separating low-level accidental complexity from inherent complexity.

In 1987, it was envisioned that "one of the most promising of the current technological efforts, and one that attacks the essence, not the accidents, of [network security configuration], is the development of approaches and tools for rapid prototyping of systems as prototyping is part of the iterative specification of requirements." [71][2] Now, 30 years later, our iterative process illustrated in Figure 18.1, finally enables this process and provides formally verified tools for it.

[2] We have strongly adapted the quote for our context. The original quote said "software problem" instead of "network security configuration".

Chapter 19

Comparison to State-of-the-Art

In this chapter, we compare the contributions of this thesis in the field of *static network configuration* to the state of the art. In particular, we consider work that provides answers to the question *"How would I want to design my network?"* and, with focus on security in terms of managing access control, work that provides answers to the question *"Which machines should be allowed to speak to each other?"*.

19.1 List of Criteria

We compare languages, tools, and frameworks for static network configuration and focus on security. We set up a list of criteria based on the finest features found in related work. With regard to **NF1**, we consider only work where executable code is available or an executable implementation is described.

We start with two quality criteria. Our non-functional requirement **NF2** asked *"can the correctness of the tools be justified?"*. To motivate the importance of this question, it was recently found [159] that even the well-engineered SDX setup [250] had errors because of errors in the underlying tool. Guha et al. [121] provide further examples where "tools make simplifying assumptions that are routinely violated by real hardware". To formally justify overall correctness of a tool, at least two artifacts are needed. We describe these artifacts in the following two paragraphs.

Formal Semantics First, a formal semantics must describe a model and the behavior of the targeted network mechanisms to be able to reason about a configuration. Ideally, for a tool which compiles a high-level language to low-level configurations, two semantics should be provided: one semantics of the targeted network elements (e.g., a semantics of OpenFlow switches or the iptables firewall) and one semantics of the high-level management language. NetCore provides a detailed semantics of the operations and essential features of OpenFlow [121]. Both NetKAT [31] and NetCore [28] provide clear, explicit, formal semantics of their language. The presence of a formalized semantics makes assumptions explicit and allows formal reasoning.

Formal Verification Second, when using a tool to help with security-related issues, a lot of trust is placed in the tool. The question about the correctness of the tool w.r.t. the formal semantics needs to be answered. The authors of the Mignis [32] tool provide a proof of correctness of their theory. However, the proof only exists on paper and it remains unclear whether the actual implementation is also correct. Remarkably, the correctness of the NetCore toolset [121] is proven correct with the Coq [122] proof assistant. This provides a machine-verified proof of correctness and constitutes our criterion of formal verification.

Those two criteria, namely the availability of a formal semantics and machine-verified formal correctness proofs, ensure a high quality of a tool and also demonstrate a generic scientifically sound theory.

High-Level Language Next, we survey related work with regard to our first question **Q1** *"How can we design secure networks from scratch?"* In particular, **Q1.1** "*How can the security requirements be specified?*" raises the question at which level of abstraction the security requirements should be expressed. With regard to Figure 1.1, security requirements reside on a higher level of abstraction than policies. However, most related work discussed in this chapter only considers the abstraction level of policies, which leaves the formalization of security requirements as a policy as an error-prone manual step. Notably, Zhao et al. [30] try to tackle this exact problem. They state that "[i]t is increasingly important to develop policy refinement that automates high level requirements into [policy] implementation in policy-based system management." [30]. With regard to Figure 10.14, Zhao et al. propose to start at the Security Invariants while most other work starts at the Access Control Abstraction. We call a language that directly supports specification as security invariants a *High-Level Language.*

Built-In Verification Yet, given a specification in a high-level language or as (high-level) policy, the question whether a specification expresses the intended meaning remains. Therefore, a management language should be built with automated verification in mind. This means that the language should be limited in the expressiveness to allow formal reasoning but still be as expressible as necessary for real-world application [251]. Both Flowlog and NetKAT are based on this principle.

Stateful Connection Semantics Though we primarily focus on static configuration languages, stateful connection semantics must not be overlooked. For example, a simple policy such as 'Internet hosts can only send packets to an internal host if the internal host initiated the connection' may be found in almost every firewall. McClurg et al. [252] note that most languages that have been proposed so far lack this critical feature. Also, Adão et al. [32] note that a lot of firewall work is limited to stateless firewalls and therefore barely applicable to real-world. Both focus on explicit support for stateful connections.

Legacy Support "[A]dministrators quite likely depend upon the behavior of their existing configurations" [212]. Therefore, it is important to investigate how existing configurations can be ported to a new, proposed management language. The Exodus system [212] focuses on migrating existing, legacy configurations from Cisco devices to the Flowlog language. Our criterion of legacy support requires that legacy configurations can be migrated to a proposed language. Afterwards, for this criterion, the legacy configuration and devices can be abandoned; our criterion does not require coexistence with legacy infrastructure. It is even more challenging to build a hybrid approach which allows to operate legacy and new infrastructure in a hybrid way [70].

Low Level Access Our second main research question **Q2** asks *"How can we analyze and verify existing configurations?"*. The previous criterion of legacy support implies a feature to import an existing configuration and, probably, analyze it afterwards. However, once an existing configuration is imported, there is no way of going back. Existing configurations may contain certain low-level statements that are no longer available in the high-level language. Since a high-level language should usually abstract over low-level details, this is a desirable property. Yet, these low-level features are sometimes necessary, as can be seen in several examples of iptables [25, 26, 27]. To cope with this issue, the Mignis [32] firewall configuration language explicitly allows its user to add arbitrary low-level iptables match conditions to the high-level rules. This also gives the user low-level access. It adds the problem that once a user may perform low-level modifications, there arises the need to verify that these low-level modifications comply with high-level specifications.

19.2 Evaluation of Related Work

We summarize our comparison of related work in Table 19.1. First, we summarize the evaluation criteria applied in said table. Afterwards, we detail on the individual related work. Finally, we summarize how our tools achieve the criteria in Section 19.4.

F. Sem (Formal Semantics) A checkmark is awarded if a formal semantics of the targeted network elements or the language exists. Two checkmarks are awarded if the semantics is formalized and at least type checked by a theorem prover.

F. Veri (Formal Verification) A checkmark is awarded if the compilation or translation process of a tool is formally proven. We award two checkmarks if the proof is machine-checked.

HLL (High-Level Language) A checkmark is awarded if the language allows to start with a specification of high-level security requirements (in contrast to languages which start with a policy).

Veri Spec (Built-In Verification) One checkmark is awarded if the language comes with built with built-in verification. Two checkmarks are awarded if some automatic verification without the need for a manual specification is possible.

ct (Stateful Connection Semantics) A checkmark is awarded if the tool supports stateful connection semantics. The abbreviation **ct** is derived from the abbreviation for conntrack.

Leg Sup (Legacy Support) A checkmark is awarded if a solution provides legacy-support, e.g., reading iptables or other existing configurations and making them available to the high-level abstraction.

LL Access (Low Level Access) One checkmark is awarded if the language allows the administrator to perform low-level configuration tuning. Two checkmarks are awarded if it allows to check whether the low-level changes are sound.

Table 19.1: Comparison to related work.

	F. Sem	F. Veri	HLL	Veri Spec	ct	Leg Sup	LL Access
Flowlog & co	✗	✗	✗	✓✓	✓	✓	✗
NetCore	✓✓	✓✓	✗	✓	✗	✗	✗
NetKAT family	✓✓	✓	✗	✓	✗	✗	✗
VALID	✓	✗	✓	✗	✗	✗	✗
Zhao et al.	✗	✗	✓	✗	✗	✗	✗
Mignis	✓	✓	✗	✗	✓	✗	✓
topoS + *fffuu*	✓✓	✓✓	✓	✓✓	✓	✓	✓✓

19.3 Discussion of Related Work in Detail

Flowlog & co Nelson presents the programming language Flowlog [29] for SDNs, which was preliminary presented in a workshop paper [251]. Flowlog is an event-driven language with a syntax similar to SQL. It allows to write dynamic SDN controller programs; strictly speaking, it is not a static configuration language. However, Flowlog tries to compile a large amount of a program to static OpenFlow rules, handling only the bare minimum at the controller. Flowlog supports arbitrary state in a program via mutable tables. For example, it is easy to construct a stateful firewall with it [253]. With the same abstraction, Flowlog allows to call external programs.

Flowlog generates OpenFlow rules indirectly by generating a NetCore (cf. next paragraph) specification, which is in turn compiled to OpenFlow. The system does not allow an administrator to modify the generated NetCore or OpenFlow rules since this could introduce errors w.r.t. the Flowlog specification. It is not possible to assess whether an arbitrary NetCore or OpenFLow ruleset corresponds to a given Flowlog specification. The Flowlog "compiler has only been tested, not proven correct." [29, §7]

Flowlog was especially designed with built-in verification and analysis in mind [254, 251]. Using the alloy analyzer [255], Flowlog programs can be verified or counterexamples can be found. The translation from a Flowlog program to an alloy specification is automatic. Of course, external programs cannot be translated, thus a user must describe them axiomatically. A user can manually formalize a verification goal as alloy statement. Because alloy can only perform bounded verification, a lack of a counterexample does not automatically imply that the verification goal is met. Some experience is required by the user to argue about the verification bounds and the resulting completeness in certain cases. Later on, Chimp [256] was designed which automatically compares the behavior of two versions of a Flowlog program. Chimp does not require any manual specification of verification goals and is thus very usable. Due to the clever construction of the Flowlog language, a user must only provide few bounds about the size of the analyzed network and Chimp can automatically generate bounds for which the analysis is both sound and complete (for most cases, cf. [256, §5]).

Flowlog focuses on network programming. It does not support to specify security requirements directly. Though, an advanced user might formalize them as alloy specification and verify her Flowlog implementation against it. Given only a specification of security requirements, the system does not support to derive a Flowlog program automatically. Therefore, with regard to our criterion definition, Flowlog is not a high-level language.

Legacy network configurations can be translated to Flowlog. The Exodus system [212] processes a set of Cisco IOS configurations. It supports a remarkable subset of all features. The translation is not formally verified, but all assumptions and supported features are clearly stated. The static parts of the evaluation's configurations are statically verified (with HSA [119]) and dynamic parts are verified using Flowlog's built-in verification capabilities.

NetCore NetCore [28] is a stateless forwarding policy language with formal semantics. It was designed from the very beginning to feature formal semantics which allows to build verification tools on top [257]. It contributes an OpenFlow syntax and semantics [121] and "the first machine-verified SDN controller" [121]. High-level properties of NetCore programs can be proven [258]. It was also attempted to extend this verification to networks with several switches [259]. All the verification was carried out with the Coq [122] proof assistant.

NetCore pioneered in its domain but ultimately, it was superseded by NetKAT (see next paragraph) because its semantics is not sound for KAT [31]. Flowlog still uses NetCore as backend. We focus this short summary only on NetCore's features which are formally, machine-checked verified. In contrast, while NetKAT and Flowlog provide more features, they are not formally verified.

NetKAT NetKAT provides a semantics of network packet forwarding [31]. The semantics is based on Kleene Algebra with Tests (KAT), which allows sound and complete reasoning about statements expressed in NetKAT. A network topology can be encoded in NetKAT. In addition, forwarding, filtering, and packet modification policies can also be encoded with NetKAT. Consequently, NetKAT can be considered a static network programming language with well-defined semantics. An explicit low-level hardware semantics – such as Featherweight OpenFlow [121] – is not provided, but the behavior of OpenFlow flow table switch processing can be directly expressed with NetKAT.

A concrete NetKAT statement (network description or network program) can be verified. Verification is done by checking equivalence of two statements. For example, one can verify the equivalence of an implemented NetKAT forwarding policy and the high-level NetKAT policy "A can reach B". To verify that certain packets are dropped, one can check that the implemented policy for a specific packet is equivalent to the drop-all policy [31]. A fast decision procedure exists for these kinds of queries [260]. Unfortunately, a user needs to specify queries manually, which bears the chance of replicating a bug in both a NetKAT program and a NetKAT specification. Pre-defined queries for common verification problems exist e.g., all-pair connectivity and loop freedom. In general, equivalence of NetKAT statements is PSPACE complete.

It is unclear whether existing, complex, legacy network configurations can be automatically translated to NetKAT. For the evaluation of the NetKAT decision procedure [260], router configurations of the Stanford backbone[1] have been translated to NetKAT statements. It is unclear which features are supported by the translation in general since this tool seems not to be present in the repository [261]. However, a parser from JSON to NetKAT exists [262]. The supported features are similar to OpenFlow 1.0, which we do not consider as legacy support.

[1]Archived at `https://github.com/diekmann/net-network/tree/master/hassell-public_Stanford/Stanford_backbone`

NetKAT features an efficient compiler for local, global, and virtual programs to flow table entries [159]. A local program describes the behavior of individual switches. A global program allows to express network-wide behavior, such as paths through the network. A virtual program allows to express a behavior based on a virtual topology e.g., considering the network as one big switch.

NetKAT has been extended to support stateful network programs and event-driven programming [252]. A stateful NetKAT program can be considered a graph where the nodes correspond to static NetKAT programs and the edges correspond to network events, such as a packet arriving at a switch. Upon receipt of an event, the network configuration transitions from one static NetKAT configuration to the next. The NetKAT decision procedure [260] and existing compiler [159] cannot be applied to a stateful NetKAT program, but for each state a stateful NetKAT program can be transformed into a stateless NetKAT program where aforementioned tools are applicable. Therefore, the verification of a stateful NetKAT program requires enumerating all states. Compilation is also based on enumerating all states and reusing the existing compiler for the stateless NetKAT programs. This overall compiler setup pre-compiles OpenFlow rules for all states in advance. While this programming language may be suitable for certain scenarios, it is not well-suited for implementing connection tracking in a stateful firewall. The paper [252] shows an example of a stateful firewall which allows host A to send packets to host B, but only allows host B to send packets to host A if host A has sent a packet first. This link layer filtering behavior does not correspond to the actual connection tracking performed by stateful firewalls. A common stateful firewall performs connection tracking, i.e., tracking the IP-5-tuple tuple of source/destination IP addresses, protocol, and source/destination ports. Enumerating all possible states (here, all possible values of an IP-5-tuple) or pre-compiling all possible rules is infeasible for a state of such size.

A decision procedure for KA (Kleene Algebra) and KAT (Kleene Algebra with Tests) has been formally verified with the Coq proof assistant [263, 264]. Also, the semantics of NetKAT has been formalized in Coq [265]. However, KAT tools cannot be directly applied to NetKAT because "NetKAT extends KAT with network-specific primitives" [260]. Consequently, the NetKAT toolset itself (compiler, decision procedure) is not formally verified [265].

VALID Bleikertz and Groß state the need to express security requirements in a high-level language and verify policies against them. They propose VALID [247] which is a specification language to express security invariants in cloud infrastructures. VALID has aspects in common with *topoS*, for example, it requires a network's connectivity and information flow structure in the format of a graph as input and allows to specify predicates over it. The language is formally defined in the AVISPA Intermediate Format (IF). It can verify that a given network topology conforms to specified high-level goals, but it cannot translate these goals to a network topology nor give feedback about the meaning of the specified goals. VALID is merely a language to express security requirements, hence, a VALID specification cannot be deployed to a network.

Zhao et al. Zhao et al. [30] present a policy refinement framework for network services. They state the need to express security requirements in high-level terms and present a high-level logic language to encode them. For this language, an automated translation (*refinement*) procedure to low-level policies is presented. The low-level policies can be directly enforced by security mechanisms.

The logic-based, abstract policy language roughly corresponds to first-order logic with set theory, orderings, and relations over an UML class diagram. The policy language allows to express almost unrestricted statements. As an example, given an UML class diagram which models all natural numbers as $\mathit{class}(\mathbb{N})$, statements about all natural numbers are possible. It is thinkable that number theory and even Gödel's incompleteness theorem can be modeled in the language, which demonstrates its expressiveness. If it were possible to express enumerable but infinite UML objects, e.g., $\mathit{obj}(\mathsf{zero}, \mathbb{N})$, $\mathit{obj}(\mathsf{one}, \mathbb{N})$, $\mathit{obj}(\mathsf{two}, \mathbb{N})$, etc., arbitrary computations could be possible. The policy language by itself is expressive enough to make statements about any natural number. However, a finite UML definition for the policy domain needs to be given. Ultimately, this means that only statements about a finite amount of natural numbers can be given to the refinement process and the refinement process itself is decidable. We believe that the finiteness of the UML diagram is the only factor which prevents this mighty system from being Turing complete. The downside for a user of such a language may be that it is hard to specify and hard to read. The system does not provide a reader with feedback about whether the encoded high-level security requirement actually corresponds to the policy author's intention.

In contrast, our tool *topoS* comes with a library of pre-specified logic formulas (security invariant templates). A policy author only needs to assign attributes but is not required to manually encode requirements as formulas. In addition, *topoS* gives immediate feedback to a policy author by visualizing the resulting access control policy as graph. This enables a policy author to verify her requirement specification or perform a *"What-if?"* analysis.

While the language is well thought out, the prototype implementation and refinement process hint at some small problems. The refinement process contains a non-deterministic step, which means that several different refinements might fulfill a high-level policy rule. Under certain circumstances, this may result in soundness issues: Such a refined rule may only cover one subset of subjects that are allowed/denied access, but where the high-level policy intended to allow/deny also access for a different set of subjects. This remark is rather theoretical, as we were unable to empirically construct such an example for the prototype implementation. The current prototype implementation only supports refinement of one single policy rule. Since rules may conflict or overlap, it is not possible to just translate a set of rules one by one individually. This effectively limits policies for the prototype implementation to one rule (without introducing conflict resolution or soundness issues due to the ordering of ACLs). The translation to a Routing+Firewall architecture (called ROFL) may result in further soundness issues since Bloom filters [266] are used to encode permitted subjects and a false-positive may result in granting access to a subject that should not have access. Testing the prototype implementation, we found a policy which produces a refinement which is not expected, but we could not figure out the reason of the problem.

The author of this thesis wants to thank Hang Zhao for providing the source code of the prototype implementation on request and for a constructive email conversation.

Mignis Mignis [32] presents a formal semantics of the filtering behavior and NAT behavior for the netfilter/iptables firewall. The semantics supports the actions `ACCEPT`, `DROP`, `SNAT`, and `DNAT`; it does not consider user-defined chains. The match conditions for matching on source/destination IP addresses and source/destination ports are modeled explicitly. In addition, an arbitrary predicate ϕ over the packet and the connection tracking state is modeled to represent arbitrary further match conditions. This predicate ϕ is always assumed to be well-behaved. Unfortunately, the semantics only exists on paper. Common questions whether the semantics is always defined or determinism remain unanswered. The Mignis tool is written in python [267].

The main focus of the Mignis tool is to present a new policy language for firewall management. Hence, the semantics mainly models the subset of iptables that is also supported by the Mignis language. The predicate ϕ is intended to give the user low-level access to add additional matching features to iptables rules which are generated by Mignis.

The policy language is declarative and rules are position-independent. The language supports `DROP`, `ACCEPT`, `DNAT`, and `SNAT` rules. `DROP` rules always take precedence over `ACCEPT` rules, which avoids traditional firewall rule conflicts by definition. For each rule, the matching IP addresses and ports can be specified. In addition, for each rule an arbitrary predicate ϕ can be added to give the administrator additional low-level control over the ruleset.

The Mignis tool decides the overall control flow of the generated iptables rules. This prevents an administrator from reordering rules – a common performance optimization which is obstructed by Mignis. An administrator can only influence match conditions for individual rules via the ϕ predicates. Since a user cannot influence the control flow, this reduces the effective matching possibilities originally provided by iptables: With one iptables rule, it is not possible to write a rule that negates several primitive match conditions. For example, one cannot write directly `! (-p tcp --dport 80)`, which should match everything that is not tcp port 80.[2] However, calling a user-defined chain and `RETURN`ing allows to logically construct exactly the desired match condition. This pattern is observed quite often in real-world rulesets [25]. Therefore, the low-level control admitted by Mignis is rather limited.

The user-defined predicate ϕ must be well-behaved. For example, it must behave exactly the same for a packet before and after NATing. The Mignis implementation [267] checks that no ϕ contains strings such as `--sport` or `--source-port`. However, this check can be too restrictive as, for example, Mignis also rejects the comment `-m comment " --sports "`. More severely, without having an explicit whitelist which enumerates all 'good' match features, it is impossible to check that a user-specified predicate is actually well-behaved. For example, the match extensions `bpf`, `string`, `u32` can all be used to match on ports and IP addresses, and `iprange`, `addrtype`, `realm`, `recent`, and `set` can be used to match on IP addresses. On the one hand, if Mignis allows arbitrary predicates ϕ, it sacrifices soundness. On the other hand, if Mignis only allows an explicit whitelist for ϕ, it limits its expressiveness. In the current implementation "mignis cannot provide any correctness guarantee about the whole configuration if [arbitrary match extensions are used]" [268].[3]

A simple query language to verify Mignis policies is work in progress [267].

[2]For this specific example, it is possible encode the desired behavior manually by two rules: `-p tcp` and `-p tcp ! -dport 80`. This manual approach is error prone and does not scale for larger, complicated expressions.

[3]Original quote: "You can write any filter you want after |, but mignis cannot provide any correctness guarantee about the whole configuration if you do that." [268]

19.4 Evaluation of *topoS* and *fffuu* w.r.t. the Criteria

In this section, we summarize how the results of this thesis are evaluated with regard to the presented criteria.

Formal Semantics We provide both, a low-level semantics of iptables and formalization of our high-level language as security invariants. Both are formalized in the Isabelle/HOL theorem prover. In addition, important properties of the semantics are shown, e. g., determinism of the iptables semantics or definedness of the offending flows.

Formal Verification Our tools *topoS* and *fffuu* are formally machine-verified by Isabelle/HOL.

High-Level Language Our approach allows to start with a high-level language to encode security requirements. For user-friendliness, a security requirement can be formalized by instantiating pre-defined security invariant templates with scenario-specific information. In addition, any higher-order logic formula which fulfills the proof obligations for a security invariant template[4] can be added to the language. Ultimately, requirements are expressed as a set of security invariants over a policy. We showed that, given only the invariants, a policy can be automatically computed. This makes *topoS* a true high-level language according to our criterion.

Built-In Verification Our tool *topoS* also provides built-in verification. For example, it is possible to compare formalized requirements with a manually specified policy. Once a specification of the requirements exists, the generic policy construction algorithm allows to provide automatic feedback about the requirements. This does not require any additional input. Among others, it can compute the allowed flows resulting from the requirements, or conversely, can visualize everything the requirements prohibit. The semantical difference between two requirement specifications can be automatically compared by comparing the policies constructed from these requirements. Our tool *fffuu* gives feedback about an iptables ruleset without any additional input: it can visualize the overall policy as a partition over the complete IP address space.

Stateful Connection Semantics Both our tools *topoS* and *fffuu* support stateful connection semantics.

Legacy Support Legacy support for iptables firewalls is provided by *fffuu*, which translates an iptables ruleset to a policy. The resulting policy can, in turn, be loaded by *topoS*.

Low Level Access The other way round, any iptables ruleset generated by *topoS* can be manually low-level tuned, and *fffuu* can compute whether the policy has changed. If it has changed, *topoS* in turn can verify whether the changes introduce a violation with regard to the security requirements.

[4]Definition 4, the invariant must be true for the deny-all policy (Theorem 1), and Definition 8

Chapter 20

Summary of Applicability and Application

Beware of "the real world". A speaker's appeal to it is always an invitation not to challenge his tacit assumptions. When a speaker refers to the real world, a useful counterploy is pointing out that he only refers to his perception of it.

E. W. Dijkstra, A bagatelle for the left hand (1982) [269].

Applicability and Application

In Chapter 1, we asked the question

"How can we provide means to help the administrator to configure secure networks or verify the security of existing network configurations?"

We have split the problem statement into various sub problems and presented our answers in Chapter 18. In this chapter, we summarize how our results apply to various domains.

20.1 Generic Policy Management and Reasoning

In this thesis, we developed a generic theory for policy management, reasoning, and verification. Our theory is independent of enforcing mechanisms and can thus be applied in various domains, not only traditional network management. We demonstrated the applicability in the following areas:

Designing Privacy Policies As shown in Chapter 17, our framework and tool *topoS* can be used to construct and evaluate policies with regard to privacy.

Auditing After having constructed a privacy policy, our *fffuu* tool can be used to perform an audit of the real implementation of the system. We discovered and fixed previously unknown errors. The results are also presented in Chapter 17.

Microservice Management As shown in Chapter 10 and Chapter 16, our *topoS* framework can be used to construct an access policy for microservices. The ruleset can be deployed on a docker host.

Hypervisor Management We showed that the same policy we have constructed in Chapter 10 can also be enforced by Open vSwitch. Thus, instead of only relying on the weak isolation provided by docker, the complete scenario can be deployed where each service is running in its own virtual machine on top of xen. This provides stronger isolation and we have shown how that our approach can be easily used to manage the internal network of a hyperviser such as xen.

Cyber Physical Systems In Section 7.2, we showed how our tool *topoS* can help in a real-world scenario —in collaboration with Airbus Group— to evaluate a cabin data network for the general civil aviation. Section 7.3 further exemplifies the applicability for a different cyber physical system.

Software Architectures Section 7.4 shows a striking parallel between our work and the field of software architectures. Viewing an architecture as policy, our method of policy verification is also applicable to software architectures. We have successfully used our method to verify the software architecture of a Munich startup (unfortunately, the results are not public).

20.2 Iptables Firewall Analysis

The theoretical aspects of our work are "substantial" [46]. Among others, we contribute a generic method to abstract over unknown match conditions which allowed us to build the first tool which can understand *any* iptables match condition. For the first time, a fully machine-verified analysis tool, and in general, a tool to analyze non-trivial firewalls is presented. Our tool is probably also a stepping stone to bridge gaps between the formal methods community and the computer networking community.

Throughout this thesis, we demonstrated real-world applicability of our tool. We also use it internally in our lab [270]. In addition, we have successfully solved real problems on serverfault [234, 271, 272, 273, 274, 274, 275], debugged real-word problems (Chapter 14), found real-world errors (Chapter 17), and earned several stars on github [276]. The practical applicability has also been demonstrated to hundreds of hackers [48, 49, 50, 51].

20.3 Software-Defined Networking

Software-Defined Networking (SDN) is a hot topic in network management and operations since almost ten years (cf. Section 2.2). For its driving position in research, we discuss the results of this thesis in the context of SDN in more detail.

SDN is often considered to be equivalent to the technology OpenFlow [206, 207]. But SDN is more than just OpenFlow. In the beginning, OpenFlow was just an "an initial attempt" [68, §1], constraint by the hardware "vendors' need for closed platforms" [68, §1]. Yet, OpenFlow is an available technology and it is widely used in research [31, 159] and practice [69, 65, 70, 277]. In the next two subsections, we classify the contributions of this thesis with regard to the two point of views: "SDN-is-OpenFlow" and "SDN as paradigm".

20.3.1 Contributions to the OpenFlow-Centric SDN Point of View

Creating New OpenFlow Rules In Part I of this thesis, we show how to generate OpenFlow rules, given only a high-level specification of security requirements. For the first time, the complete, automated, and verified translation from high-level security goals to an annotated graph that could be translated to OpenFlow rules was shown.

The formal guarantees only cover the transition from the security goals to the graph structure. The final syntactic rewriting step to OpenFlow is not formally verified. The advantage of this generic approach is that it allows to replace OpenFlow by different backend technologies. For example, we also generated (and verified) iptables firewall rules.

To make our demonstrator easily accessible, reproducible, and to focus on the core contributions of our research, we chose to set up our demonstrator with only one central OpenFlow-enabled switch. However, there is no technical limitation in our approach that requires this specific setup. It would also be possible to serialize the output of *topoS* to a SDN programing language [159, 35, 184], which then compiles to a distributed set of OpenFlow-enabled switches. We discuss related work in Section 10.5 and illustrate how *topoS* could interact with related work in Figure 10.15.

Verifying OpenFlow Rules In Section 12.8, we discuss how our insights and library for firewall ruleset analysis can also contribute to the analysis of OpenFlow rules.

Translating Existing Technologies to OpenFlow Based on the results of this thesis, the first fully machine-verified translation of iptables to OpenFlow rules has been demonstrated [43].

20.3.2 Contributions to SDN as Abstract Paradigm

The core paradigm of SDN is the separation of the control plane from the data plane and centralization of control [278]. In this thesis, we put the separation of control plane from data plane to the extreme by only considering static configurations.

Creating New SDNs In Part I, we propose methods to design new networks. Our graph model corresponds to the SDN global network view [4]. Therefore, our methodology is well-suited to implement an access control module in a network operating system [120, 177].

Since our methods allow the static, offline verification of a policy, it can guarantee within the limits of the model that there are no runtime security problems. Our control plane (i.e., policy) can be completely separated from the data plane components (e.g., OpenFlow rules, iptables rules).

We demonstrated how *topoS* can generate iptables rules for a central OpenVPN controller. This architecture of a star topology enforced by a central VPN sever with access policies implemented in software is also a SDN by definition. This generalized view was well appreciated by the workshop participants [144] and demonstrates the general applicability of our method. The complete Part I of this thesis explores network policy management independently of the underlying data plane.

Chapter 21

Conclusion

Beware of bugs in the above code; I have only proved it correct, not tried it.

D. Knuth, Letter to Emde Boas (1977) [279].

In this thesis, we explored the application of formal methods to make our networks more secure. In order to ease the daily life of network administrators and to prevent configuration errors, we presented two fully automated tools. While a user of our tools needs not to worry about any formalism, the output of our tools provides a very high level of confidence due to the formal verification. While we found that many theory-heavy works are not applicable to real-word data, attributable to the feature creep of most practical configuration languages, we have demonstrated large-scale real-world applicability of our tools and theory.

During this thesis, we roamed between two communities, hopefully narrowing the gaps between the two. In the Isabelle community, we started and actively populated the networks section in the archive of formal proofs. In the networks community, we demonstrated the application of rigorous formal methods to real-world problems and data. Our formal approach allows to reduce complex problems to a simple structure, allowing to translate between different levels of abstraction. We provide not only academic prototypical tools, but a large library of ready-to-use formally verified components for simplifying network access control.

Acknowledgements

This thesis would not have been possible without the support of many people. Foremost, I want to thank all people who put great effort into this research over many years, constantly supported me, co-authored, and provided many important contributions. Thank you Lars Hupel, Julius Michaelis, Andreas Korsten, and Lukas Schwaighofer.

During this research, many papers have been published. Now, I would like to thank all of my co-authors: Thank you Marcel von Maltitz, Stephan-A. Posselt, Heiko Niedermayer, Oliver Hanka, Holger Kinkelin, and Michael Schlatt. In addition, I want to thank all of the people who have not been listed before but who commented on the early drafts of our papers and research and gave valuable feedback: Thank you Lars Noschinski, Manuel Eberl, Lothar Braun, Fabian Immler, Benjamin Hof, and Jasmin Blanchette. We thank all the (anonymous) administrators who donated their firewall configs to our research.

I would like express my gratitude to Prof. Dr.-Ing. Georg Carle for supervising this thesis, and for making this research possible altogether.

Finally, I am also very grateful to my parents Annette and Hermann Diekmann who have supported me and always believed in me.

Bibliography

[1] E. W. Dijkstra, "On the nature of Computing Science," Aug. 1984, EWD896. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD896.PDF

[2] M. Burgess, *Principles of Network and System Administration*, 2nd, Ed. Wiley, Feb. 2004, ISBN 978-0-470-86807-2.

[3] S. M. Bellovin and R. Bush, "Configuration Management and Security," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 268–274, Apr. 2009.

[4] S. Shenker, M. Casado, T. Koponen, and N. McKeown, "The Future of Networking, and the Past of Protocols," Open Networking Summit, Oct. 2011.

[5] N. McKeown, "How SDNs will tame networks," Hot Interconnects, Aug. 2012, keynote.

[6] J. D. Guttman and A. L. Herzog, "Rigorous Automated Network Security Management," *International Journal of Information Security*, vol. 4, no. 1, pp. 29–48, Feb. 2005.

[7] A. Wool, "A Quantitative Study of Firewall Configuration Errors," *IEEE Computer*, vol. 37, no. 6, pp. 62–67, Jun. 2004.

[8] A. Wool, "Trends in Firewall Configuration Errors: Measuring the Holes in Swiss Cheese," *IEEE Internet Computing*, vol. 14, no. 4, pp. 58–65, Jul. 2010.

[9] H. Hamed and E. Al-Shaer, "Taxonomy of Conflicts in Network Security Policies," *IEEE Communications Magazine*, vol. 44, no. 3, pp. 134–141, Mar. 2006.

[10] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, Oct. 2012.

[11] C. Diekmann, S.-A. Posselt, H. Niedermayer, H. Kinkelin, O. Hanka, and G. Carle, "Verifying Security Policies using Host Attributes," in *Formal Techniques for Distributed Objects, Components, and Systems: 34th IFIP WG 6.1 International Conference (FORTE).* Berlin, Germany: Springer Berlin Heidelberg, Jun. 2014, pp. 133–148.

[12] F. Mansmann, T. Göbel, and W. Cheswick, "Visual Analysis of Complex Firewall Configurations," in *9th International Symposium on Visualization for Cyber Security*, ser. VizSec '12. ACM, Oct. 2012, pp. 1–8.

[13] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, “FIREMAN: A Toolkit for FIREwall Modeling and ANalysis,” in *IEEE Symposium on Security and Privacy*, May 2006, pp. 199–213.

[14] B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher, “Specifications of a High-level Conflict-free Firewall Policy Language for Multi-domain Networks,” in *12th ACM symposium on Access control models and technologies*, ser. SACMAT’07. ACM, Jun. 2007, pp. 185–194.

[15] Verizon Business RISK team and United States Secret Service, “2010 Data Breach Investigations Report,” 2010. [Online]. Available: http://www.verizonenterprise.com/resources/reports/rp_2010-DBIR-combined-reports_en_xg.pdf

[16] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, “The Margrave Tool for Firewall Analysis,” in *24th USENIX Large Installation System Administration Conference (LISA).* San Jose, CA: USENIX Association, Nov. 2010.

[17] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, “A Survey on Network Troubleshooting,” Technical Report Stanford/TR12-HPNG-061012, Jun. 2012.

[18] *IT-Grundschutz-Kataloge*, BSI Std., Rev. 15. EL Stand 2016. [Online]. Available: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/itgrundschutzkataloge_node.html

[19] Netcordia, Inc., “Network Downtime, the Configuration Errors,” whitepaper, 2009, currently unavailable; archived mirror. [Online]. Available: https://web.archive.org/web/20160717225153/http://www.pmi.it/file/whitepaper/000140.pdf

[20] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, “Why do Internet services fail, and what can be done about it?” in *4th USENIX Symposium on Internet Technologies and Systems*, ser. USITS’03, vol. 67, Seattle, WA. USENIX Association, Mar. 2003.

[21] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège, *Formal Aspects in Security and Trust: IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST), World Computer Congress.* Boston, MA: Springer US, 2005, ch. A Formal Approach to Specify and Deploy a Network Security Policy, pp. 203–218.

[22] S. Pozo, R. Ceballos, and R. M. Gasca, “Model-Based Development of firewall rule sets: Diagnosing model inconsistencies,” *Information and Software Technology*, vol. 51, no. 5, pp. 894–915, May 2009.

[23] The netfilter.org project, “netfilter/iptables project.” [Online]. Available: http://www.netfilter.org/

[24] “man (8) iptables-extensions,” version 1.4.21.

[25] “Analyzed firewall rulesets (raw data),” accompanying material. [Online]. Available: https://github.com/diekmann/net-network

[26] M. Majkowski, “BPF – the forgotten bytecode,” CloudFlare blog, May 2014, retrieved May 2016. [Online]. Available: https://blog.cloudflare.com/bpf-the-forgotten-bytecode/

[27] “Over 4000 questions tagged with ‘iptables’,” serverfault, Jun. 2016. [Online]. Available: http://serverfault.com/questions/tagged/iptables

[28] C. Monsanto, N. Foster, R. Harrison, and D. Walker, “A Compiler and Run-time System for Network Programming Languages,” in *39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL ’12. ACM, Jan. 2012, pp. 217–230.

[29] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, “Tierless Programming and Reasoning for Software-Defined Networks,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI’14. Seattle, WA: USENIX Association, Apr. 2014, pp. 519–531.

[30] H. Zhao, J. Lobo, A. Roy, and S. M. Bellovin, “Policy Refinement of Network Services for MANETs,” in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*, Dublin, Ireland, May 2011.

[31] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, “NetKAT: Semantic Foundations for Networks,” in *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’14. San Diego, California: ACM, Jan. 2014, pp. 113–126.

[32] P. Adão, C. Bozzato, G. Dei Rossi, R. Focardi, and F. L. Luccio, “Mignis: A Semantic Based Tool for Firewall Configuration,” in *27th Computer Security Foundations Symposium*, ser. CSF. IEEE, Jul. 2014, pp. 351–365.

[33] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, “Firmato: A Novel Firewall Management Toolkit,” in *Symposium on Security and Privacy*. IEEE, May 1999, pp. 17–31.

[34] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, “Practical Declarative Network Management,” in *1st ACM workshop on Research on enterprise networking*, ser. WREN’09. ACM, Aug. 2009, pp. 1–10.

[35] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, “Merlin: A Language for Provisioning Network Resources,” *CoRR*, vol. abs/1407.1199, 2014. [Online]. Available: http://arxiv.org/abs/1407.1199

[36] P. Bera, S. Ghosh, and P. Dasgupta, “Policy Based Security Analysis in Enterprise Networks: A Formal Approach,” *IEEE Transactions on Network and Service Management*, vol. 7, no. 4, pp. 231–243, Dec. 2010.

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

[38] C. Diekmann, “Network Security Policy Verification,” *Archive of Formal Proofs*, Jul. 2016, formal proof development. [Online]. Available: https://www.isa-afp.org/entries/Network_Security_Policy_Verification.shtml

[39] C. Diekmann, J. Michaelis, and L. Hupel, “IP Addresses,” *Archive of Formal Proofs*, Jun. 2016, formal proof development. [Online]. Available: http://isa-afp.org/entries/IP_Addresses.shtml

[40] C. Diekmann, J. Michaelis, and M. Haslbeck, “Simple Firewall,” *Archive of Formal Proofs*, Aug. 2016, formal proof development. [Online]. Available: http://isa-afp.org/entries/Simple_Firewall.shtml

[41] C. Diekmann and L. Hupel, “Iptables Semantics,” *Archive of Formal Proofs*, Sep. 2016, formal proof development. [Online]. Available: http://isa-afp.org/entries/Iptables_Semantics.shtml

[42] J. Michaelis and C. Diekmann, “Routing,” *Archive of Formal Proofs*, Aug. 2016, formal proof development. [Online]. Available: http://isa-afp.org/entries/Routing.shtml

[43] J. Michaelis and C. Diekmann, “LOFT – Verified Migration of Linux Firewalls to SDN,” *Archive of Formal Proofs*, Oct. 2016, formal proof development. [Online]. Available: http://isa-afp.org/entries/LOFT.shtml

[44] C. Diekmann, L. Hupel, and G. Carle, “Semantics-Preserving Simplification of Real-World Firewall Rule Sets,” in *Formal Methods*, Jun. 2015.

[45] C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle, “Verified iptables Firewall Analysis,” in *IFIP Networking 2016*, Vienna, Austria, May 2016.

[46] L. Paulson, “new in the AFP: Iptables_Semantics,” isabelle users mailing list, Sep. 2016. [Online]. Available: https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2016-September/msg00023.html

[47] G. Klein, “AFP: LOFT,” isabelle users mailing list, Oct. 2016. [Online]. Available: https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2016-October/msg00055.html

[48] C. Diekmann, “Verified Firewall Ruleset Verification – Math, Functional Programming, Theorem Proving, and an Introduction to Isabelle/HOL,” 32. Chaos Communication Congress (32C3), Dec. 2015. [Online]. Available: https://media.ccc.de/v/32c3-7195-verified_firewall_ruleset_verification

[49] C. Diekmann, “Verified Firewall Ruleset Verification 1/2,” elfte Treffen des Curry Clubs Augsburg, Feb. 2016. [Online]. Available: http://curry-club-augsburg.de/posts/2016-01-05-elftes-treffen.html

[50] C. Diekmann, “Verified Firewall Ruleset Verification 2/2,” zwölfte Treffen des Curry Clubs Augsburg, Mar. 2016. [Online]. Available: http://curry-club-augsburg.de/posts/2016-02-26-zwoelftes-treffen.html

[51] C. Diekmann, “Make Firewalls Verified Again,” achtzehnte Treffen des Curry Clubs Augsburg, Sep. 2016. [Online]. Available: http://curry-club-augsburg.de/posts/2016-06-29-achtzehntes-treffen.html

[52] M. Bishop, "What Is Computer Security?" *IEEE Security & Privacy*, vol. 1, no. 1, pp. 67–69, Feb. 2003.

[53] C. Diekmann, "Security Requirement Modeling as Configuration Management for Scenario-Specific Networks," Master's thesis, Technische Universität München, May 2013.

[54] R. Hertzog and R. Mas, *The Debian Administrator's Handbook – Debian Jessie from Discovery to Mastery*, 1st ed. debian-handbook.info, Oct. 2015, ISBN 979-10-91414-05-0.

[55] Arbor Networks, "Worldwide infrastructure security report," Sep. 2007, archived at http://web.archive.org/web/20160728134338/http://www.lateledipenelope.it/public/WISP_US_12sept07.pdf.

[56] Arbor Networks, "Worldwide infrastructure security report," 2016, archived at https://web.archive.org/web/20160417124116/https://www.arbornetworks.com/images/documents/WISR2016_EN_Web.pdf.

[57] R. Potharaju and N. Jain, "Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters," in *13th SIGCOMM Conference on Internet Measurement (IMC)*. ACM, Oct. 2013.

[58] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *Conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '07. Kyoto, Japan: ACM, Aug. 2007, pp. 1–12.

[59] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, P. Rao, D. Rosenbluth, A. V. Surendran, and J. Martin, D.M., "Automatic Management of Network Security Policy," in *DARPA Information Survivability Conference & Exposition II. DISCEX*, vol. 2, Aug. 2001, pp. 12–26.

[60] S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich, "Report on the NSF Workshop on Formal Methods for Security," Aug. 2016. [Online]. Available: https://arxiv.org/abs/1608.00678v1

[61] N. Bjørner, N. Foster, P. B. Godfrey, and P. Zave, "Formal Foundations for Networking (Dagstuhl Seminar 15071)," *Dagstuhl Reports*, vol. 5, no. 2, pp. 44–63, 2015. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/5044

[62] J. Kirk, "Verizon: Data breaches often caused by configuration errors," networkworld, Jul. 2010, retrieved Aug 2016. [Online]. Available: http://www.computerworld.com/article/2519657/network-security/verizon--data-breaches-often-caused-by-configuration-errors.html

[63] Verizon and Contributing Organizations, "2016 Data Breach Investigations Report," 2016, retrieved Aug 2016. [Online]. Available: http://www.verizonenterprise.com/verizon-insights-lab/dbir/

[64] A. Peters, “Faulty Access Controls Led to Morgan Stanley Data Breach: FTC,” americanbanker.com, Aug. 2015, retrieved Aug 2016. [Online]. Available: http://www.americanbanker.com/news/bank-technology/faulty-access-controls-led-to-morgan-stanley-data-breach-ftc-1076098-1.html

[65] A. Gladisch and F.-J. Westphal, “Carrier Landscape for SDN – Next Level of Telco Industrialization?” 44. Treffen der VDE/ITG-Fachgruppe 5.2.4, Mobile Network (Function) Virtualization and Software Defined Networking, Nov. 2013, deutsche Telekom Innovation Labs. [Online]. Available: http://www.ikr.uni-stuttgart.de/Content/itg/fg524/Meetings/2013-11-15-Muenchen/08_ITG524_Muenchen_Westphal.pdf

[66] P. Peresini and D. Kostic, “Is the Network Turing-Complete?” EPFL, Tech. Rep. EPFL-REPORT-187131, Jun. 2013, after a personal communication with the author of this thesis in Sep. 2015, the report was updated.

[67] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, “SANE: A Protection Architecture for Enterprise Networks,” in *15th USENIX Security Symposium*, vol. 15, no. 10. Vancouver, B.C., Canada: USENIX Association, Jul. 2006.

[68] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling Innovation in Campus Networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[69] U. Hoelzle, “OpenFlow @ Google,” Open Networking Summit, Apr. 2012, keynote.

[70] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, “B4: Experience with a Globally-deployed Software Defined WAN,” in *ACM SIGCOMM.* Hong Kong, China: ACM, Aug. 2013, pp. 3–14.

[71] F. P. Brooks, “No Silver Bullet – Essence and Accidents of Software Engineering,” *IEEE computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[72] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications.* Addison-Wesley Professional, Apr. 2007, no. Third Edition.

[73] M. McNickle, “Nick Feamster: OpenFlow SDN for access control is just the beginning,” techtarget.com, Nov. 2013. [Online]. Available: http://searchsdn.techtarget.com/news/2240208949/Nick-Feamster-OpenFlow-SDN-for-access-control-is-just-the-beginning

[74] T. Benson, A. Akella, and D. Maltz, “Unraveling the Complexity of Network Management,” in *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI’09. Berkeley, CA, USA: USENIX Association, Apr. 2009, pp. 335–348.

[75] Common Criteria, “Part 3: Security assurance components,” *Common Criteria for Information Technology Security Evaluation*, vol. CCMB-2012-09-003, no. Version 3.1

Revision 4, Sep. 2012. [Online]. Available: http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R4.pdf

[76] G. Klein, T. Nipkow, and L. Paulson, Eds., *Archive of Formal Proofs*, 2016. [Online]. Available: https://www.isa-afp.org/

[77] E. Yuan and J. Tong, "Attributed Based Access Control (ABAC) for Web Services," in *IEEE International Conference on Web Services*, ser. ICWS, Jul. 2005.

[78] P. Samarati and S. C. de Vimercati, *Foundations of Security Analysis and Design: Tutorial Lectures.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, ch. Access Control: Policies, Models, and Mechanisms, pp. 137–196.

[79] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian, "Flexible Support for Multiple Access Control Policies," *ACM Transactions on Database Systems (TODS)*, vol. 26, no. 2, pp. 214–260, Jun. 2001.

[80] M. D. McIlroy and J. A. Reeds, "Multilevel Security in the UNIX Tradition," *Software: Practice and Experience*, vol. 22, no. 8, pp. 673–694, Aug. 1992.

[81] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From General Purpose to a Proof of Information Flow Enforcement," in *IEEE Symposium on Security and Privacy*, May 2013, pp. 415–429.

[82] D. E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[83] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations," MITRE Corporation, Bedford, MA, MTR-2547 Vol. I, Mar. 1973.

[84] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: A Mathematical Model," MITRE Corporation, Bedford, MA, MTR-2547 Vol. II, May 1973.

[85] D. E. Bell, "Secure Computer Systems: A Refinement of the Mathematical Model," MITRE Corporation, Bedford, MA, MTR-2547 Vol. III, Dec. 1973.

[86] D. E. Bell and L. J. LaPadula, "Secure computer systems: Unified Exposition and MULTICS Interpretation," MITRE Corporation, Bedford, MA, MTR-2997, Jul. 1975.

[87] D. E. Bell, "Looking Back at the Bell-La Padula Model," in *21st Annual Computer Security Applications Conference*, Dec. 2005, pp. 337–351.

[88] M. Bishop, *Computer Security: Art and Science.* Addison-Wesley, 2003, ISBN 0-201-44099-7.

[89] C. Eckert, *IT-Sicherheit: Konzepte-Verfahren-Protokolle*, 8th ed. Oldenbourg Verlag, 2013, ISBN 3486721380.

[90] S. M. Bellovin, "A Look Back at "Security Problems in the TCP/IP Protocol Suite"," in *Annual Computer Security Applications Conference*, Dec. 2004, invited paper.

[91] L. Gong and X. Qian, "The Complexity and Composability of Secure Interoperation," in *IEEE Computer Society Symposium on Research in Security and Privacy*, May 1994, pp. 190–200.

[92] D. McCullough, "A Hookup Theorem for Multilevel Security," *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 563–568, Jun. 1990.

[93] M. von Maltitz, C. Diekmann, and G. Carle, "Taint Analysis for System-Wide Privacy Audits: A Framework and Real-World Case Studies," 1st Workshop for Formal Methods on Privacy, Nov. 2016, workshop without proceedings.

[94] C. Sternagel and R. Thiemann, "Executable Transitive Closures of Finite Relations," *Archive of Formal Proofs*, Mar. 2011, formal proof development. [Online]. Available: https://www.isa-afp.org/entries/Transitive-Closure.shtml

[95] R. S. Sandhu, E. J. Coynek, H. L. Feinsteink, and C. E. Youmank, "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[96] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *IEEE Symposium on Security and Privacy*, Apr. 1982, pp. 11–20.

[97] J. Rushby, "Noninterference, Transitivity, and Channel-Control Security Policies," Tech. Rep., Dec. 1992. [Online]. Available: http://www.csl.sri.com/papers/csl-92-2/

[98] N. Broberg and D. Sands, "Flow-Sensitive Semantics for Dynamic Information Flow Policies," in *Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS '09. ACM, Jun. 2009, pp. 101–112.

[99] A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 129–142, Oct. 1997.

[100] A. Okmianski, "Transmission of Syslog Messages over UDP," RFC 5426 (Proposed Standard), Internet Engineering Task Force, Mar. 2009.

[101] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy.* IEEE, May 2010, pp. 317–331.

[102] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Operating Systems Design and Implementation*, ser. OSDI'10. Vancouver, BC, Canada: USENIX Association, Oct. 2010, pp. 393–407.

[103] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 5:1–5:29, Jun. 2014.

[104] E. Tromer and R. Schuster, “DroidDisintegrator: Intra-Application Information Flow Control in Android Apps (extended version),” in *11th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’16. Xi’an, China: ACM, May 2016, pp. 401–412, author’s extended version referenced. [Online]. Available: http://www.cs.tau.ac.il/~tromer/disintegrator/disintegrator.pdf

[105] M. Rost and A. Pfitzmann, “Datenschutz-Schutzziele – revisited,” *Datenschutz und Datensicherheit DuD*, vol. 33, no. 6, pp. 353–358, Jul. 2009.

[106] K. Bock and M. Rost, “Privacy By Design und die Neuen Schutzziele,” *DuD*, vol. 35, no. 1, pp. 30–35, Mar. 2011.

[107] G. Danezis, J. Domingo-Ferrer, M. Hansen, J.-H. Hoepman, D. L. Metayer, R. Tirtea, and S. Schiffner, “Privacy and Data Protection by Design – from policy to engineering,” ENISA, Tech. Rep., Jan. 2015.

[108] AK Technik der Konferenz der unabhängigen Datenschutzbehörden des Bundes und der Länder, “Das Standard-Datenschutzmodell,” Konferenz der unabhängigen Datenschutzbehörden des Bundes und der Länder, Darmstadt, Tech. Rep. V.0.9, Oct. 2015. [Online]. Available: https://www.datenschutzzentrum.de/artikel/954-Handbuch-zum-Standard-Datenschutzmodell.html

[109] M. von Maltitz, C. Diekmann, and G. Carle, “Taint Analysis for System-Wide Privacy Audits: A Framework and Real-World Case Studies,” *ArXiv e-prints*, vol. abs/1608.04671, Aug. 2016. [Online]. Available: https://arxiv.org/abs/1608.04671

[110] F. Haftmann and L. Bulwahn, *Code generation from Isabelle/HOL theories*, Feb. 2016.

[111] F. Haftmann and T. Nipkow, “Code generation via higher-order rewrite systems,” in *Functional and Logic Programming.* Springer, 2010, pp. 103–117.

[112] L. C. Paulson, *ML for the Working Programmer*, 2nd ed. Cambridge University Press, Jun. 1996, ISBN 978-0521565431.

[113] E. R. Gansner, E. Koutsofios, and S. C. North, “Drawing Graphs with *dot*,” Jan. 2015. [Online]. Available: http://www.graphviz.org/pdf/dotguide.pdf

[114] X. Ou, S. Govindavajhala, and A. W. Appel, “MulVAL: A Logic-based Network Security Analyzer,” in *14th USENIX Security Symposium.* Baltimore, MD: USENIX Association, Jul. 2005, pp. 113–128.

[115] *Aircraft Data Network, Part 5, Network Domain Characteristics and Interconnection*, ARINC Specification 664P5, Apr. 2005.

[116] C. Diekmann, O. Hanka, S.-A. Posselt, and M. Schlatt, “Imaginary Aircraft Cabin Data Network (Toy Example),” Jul. 2013. [Online]. Available: http://www.net.in.tum.de/pub/diekmann/cabin_data_network.pdf

[117] M. Johnson, J. Karat, C. M. Karat, and K. Grueneberg, “Usable Policy Template Authoring for Iterative Policy Refinement,” in *Symposium on Policies for Distributed Systems and Networks (POLICY).* IEEE, Jul. 2010, pp. 18–21.

[118] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu, "Formally Verifiable Networking," in *8th ACM Workshop on Hot Topics in Networks*, Oct. 2009.

[119] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'12. San Jose, CA: USENIX Association, Apr. 2012, pp. 113–126.

[120] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[121] A. Guha, M. Reitblatt, and N. Foster, "Machine-Verified Network Controllers," in *34th ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '13. Seattle, Washington, USA: ACM, Jun. 2013, pp. 483–494.

[122] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2004, last updated 2016. [Online]. Available: http://coq.inria.fr/

[123] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *Policies for Distributed Systems and Networks*, ser. LNCS. Springer Berlin Heidelberg, 2001, vol. 1995, pp. 18–38.

[124] R. Craven, J. Lobo, J. Ma, A. Russo, E. Lupu, and A. Bandara, "Expressive Policy Analysis with Enhanced System Dynamicity," in *4th International Symposium on Information, Computer, and Communications Security*, ser. ASIACCS '09. Sydney, Australia: ACM, Mar. 2009, pp. 239–250.

[125] M. Feilkas, D. Ratiu, and E. Jürgens, "The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study," in *17th International Conference on Program Comprehension (ICPC)*, May 2009, pp. 188–197.

[126] G. C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-level Models," *SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 18–28, Oct. 1995.

[127] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[128] M. Beller and E. Jürgens:, "How Strict is Your Architecture?" in *14th Workshop on Software Reengineering*, Bad Honnef (Germany), 2012.

[129] C. Diekmann, L. Hupel, and G. Carle, "Directed Security Policies: A Stateful Network Implementation," in *Engineering Safety and Security Systems (ESSS)*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 150. Singapore: Open Publishing Association, May 2014, pp. 20–34.

[130] A. H. R. Hansen, "Protecting critical infrastructure," *ASA Institute for Risk & Innovation*, pp. 1–12, Jun. 2012. [Online]. Available: http://anniesearle.com/web-services/Documents/ResearchNotes/ASA_ResearchNote_ProtectingCriticalInfrastructure_June2012.pdf

[131] A. D. Brucker, L. Brügger, and B. Wolff, "Model-based Firewall Conformance Testing," in *Testing of Software and Communicating Systems.* Springer, 2008, pp. 103–118.

[132] *Technische Richtlinie BSI TR-03109-1 – Anforderungen an die Interoperabilität der Kommunikationseinheit eines intelligenten Messsystems*, 1st ed., Bundesamt für Sicherheit in der Informationstechnik, Mar. 2013, https://www.bsi.bund.de.

[133] S. Adams, "Implementing Sets Efficiently in a Functional Language," University of Southampton, Tech. Rep. CSTR 92-10, 1992. [Online]. Available: http://groups.csail.mit.edu/mac/users/adams/BB/

[134] S. Pozo, R. Ceballos, and R. M. Gasca, "CSP-Based Firewall Rule Set Diagnosis using Security Policies," in *2nd International Conference on Availability, Reliability and Security (ARES).* Los Alamitos, CA, USA: IEEE, Apr. 2007, pp. 723–729.

[135] R. M. Marmorstein and P. Kearns, "A Tool for Automated iptables Firewall Analysis," in *USENIX Annual Technical Conference, FREENIX Track.* USENIX Association, Apr. 2005, pp. 71–81.

[136] R. Marmorstein and P. Kearns, "Firewall Analysis with Policy-based Host Classification," in *20th USENIX Large Installation System Administration Conference (LISA)*, vol. 6. Washington, D.C.: USENIX Association, Dec. 2006.

[137] A. Tongaonkar, N. Inamdar, and R. Sekar, "Inferring Higher Level Policies from Firewall Rules," in *21st USENIX Large Installation System Administration Conference (LISA)*, vol. 7. Dallas, TX: USENIX Association, Nov. 2007, pp. 1–10.

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

[139] A. K. Bandara, A. C. Kakas, E. C. Lupu, and A. Russo, "Using Argumentation Logic for Firewall Configuration Management," in *IFIP/IEEE International Symposium on Integrated Network Management*, Jun. 2009, pp. 180–187.

[140] A. D. Brucker, L. Brügger, and B. Wolff, "Formal Firewall Conformance Testing: An Application of Test and Proof Techniques," *Software Testing, Verification & Reliability (STVR)*, vol. 25, no. 1, pp. 34–71, 2015. [Online]. Available: https://www.brucker.ch/bibliography/abstract/brucker.ea-formal-fw-testing-2014

[141] A. D. Brucker, L. Brügger, and B. Wolff, "HOL-TestGen/FW: An Environment for Specification-based Firewall Conformance Testing," in *International Colloquium on Theoretical Aspects of Computing -- ICTAC 2013*, ser. Lecture Notes in Computer Science, Z. Liu, J. Woodcock, and H. Zhu, Eds. Springer Berlin Heidelberg, 2013, vol. 8049, pp. 112–121.

[142] J. D. Guttman, "Filtering Postures: Local Enforcement for Global Policies," in *IEEE Symposium on Security and Privacy*, Washington, DC, USA, May 1997.

[143] R. Ritchey and P. Ammann, "Using Model Checking to Analyze Network Vulnerabilities," in *IEEE Symposium on Security and Privacy*, May 2000, pp. 156–165.

[144] C. Diekmann, A. Korsten, and G. Carle, "Demonstrating topoS: Theorem-prover-based synthesis of secure network configurations," in *11th International Conference on Network and Service Management (CNSM)*, Barcelona, Spain, Nov. 2015, pp. 366–371.

[145] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data Center Network Virtualization: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, pp. 909–928, Sep. 2013.

[146] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[147] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending Networking into the Virtualization Layer," in *Workshop on Hot Topics in Networks (HotNets)*. ACM, Oct. 2009. [Online]. Available: http://openvswitch.org/papers/hotnets2009.pdf

[148] Open vSwitch, "Release Notes v2.5.0," Feb. 2016. [Online]. Available: http://openvswitch.org/releases/NEWS-2.5.0

[149] P. Calçado, "How we ended up with microservices," Sep. 2015, retrieved Sep 2016. [Online]. Available: http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html

[150] "LinuxContainers.org – Infrastructure for container projects." Sep. 2016. [Online]. Available: https://linuxcontainers.org/

[151] Docker, Inc., "docker," 2016. [Online]. Available: https://www.docker.com/

[152] "Stack Overflow." [Online]. Available: http://stackoverflow.com/

[153] Docker Inc., "Docker security," Sep. 2016. [Online]. Available: https://docs.docker.com/engine/security/security/

[154] J. Hertz, "Abusing Privileged and Unprivileged Linux Containers," NCC Group, 2016. [Online]. Available: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/june/container_whitepaperpdf/

[155] T. Bui, "Analysis of Docker Security," *CoRR*, vol. abs/1501.02967, Jan. 2015. [Online]. Available: http://arxiv.org/abs/1501.02967

[156] Docker, Inc., "Network Configuration," version v1.5. [Online]. Available: https://docs.docker.com/v1.5/articles/networking/

[157] Docker Inc., "Legacy container links," Sep. 2016. [Online]. Available: https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/

[158] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software Defined Networks," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'13. Lombard, IL: USENIX Association, Apr. 2013, pp. 1–13.

[159] S. Smolka, S. A. Eliopoulos, N. Foster, and A. Guha, "A Fast Compiler for NetKAT," in *International Conference on Functional Programming (ICFP).* ACM, Sep. 2015, pp. 328–341.

[160] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A Security Enforcement Kernel for OpenFlow Networks," in *First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. Helsinki, Finland: ACM, Aug. 2012, pp. 121–126.

[161] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable Dynamic Network Control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'15. Oakland, CA: USENIX Association, May 2015, pp. 59–72.

[162] D. C. Verma, "Simplifying Network Administration Using Policy-Based Management," *IEEE Network*, vol. 16, no. 2, pp. 20–26, Mar. 2002.

[163] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford, "On Static Reachability Analysis of IP Networks," in *24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 3. IEEE, Mar. 2005, pp. 2170–2183.

[164] N. P. Lopes, N. Bjørner, P. Godefroid, and G. Varghese, "Network Verification in the Light of Program Verification," Tech. Rep., Sep. 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=201589

[165] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and Implementation of a Routing Control Platform," in *2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'05. Boston, MA: USENIX Association, May 2005, pp. 15–28.

[166] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "One Big Switch" Abstraction in Software-defined Networks," in *9th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT. ACM, Dec. 2013, pp. 13–24.

[167] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'13. Lombard, IL: USENIX Association, Apr. 2013, pp. 15–27.

[168] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège, "A Formal Approach to Specify and Deploy a Network Security Policy," in *Formal Aspects of Security and Trust (FAST).* Springer US, Aug. 2004, pp. 203–218.

[169] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the Data Plane with Anteater," in *ACM SIGCOMM*, Toronto, Ontario, Canada, Aug. 2011, pp. 290–301.

[170] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, "Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security," in *International Conference on Network Protocols (ICNP).* IEEE, Oct. 2009, pp. 123–132.

[171] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin, "Organization Based Access Control," in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY)*, Jun. 2003, pp. 120–131.

[172] *802.1X – Port based Network Access Control*, IEEE Std. 802.1X-2001, Jun. 2001.

[173] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy Core Information Model – Version 1 Specification," RFC 3060 (Proposed Standard), Internet Engineering Task Force, Feb. 2001, updated by RFC 3460.

[174] B. Moore, "Policy Core Information Model (PCIM) Extensions," RFC 3460 (Proposed Standard), Internet Engineering Task Force, Jan. 2003.

[175] S. Hinrichs, "Policy-Based Management: Bridging the Gap," in *15th Computer Security Applications Conference (ACSAC).* IEEE, Dec. 1999, pp. 209–218.

[176] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477–489, Jan. 2012.

[177] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," *;login: The USENIX Magazine*, vol. 38, no. 5, pp. 40–47, Oct. 2013.

[178] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Policy Refinement: Decomposition and Operationalization for Dynamic Domains," in *7th Conference on Network and Service Management (CNSM)*, Oct. 2011, pp. 1–9.

[179] M.-O. Pahl, "Data-Centric Service-Oriented Management of Things," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Ottawa, Canada, May 2015.

[180] N. Feamster and H. Balakrishnan, "Detecting BGP Configuration Faults with Static Analysis," in *2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'05. Boston, MA: USENIX Association, May 2005, pp. 43–56.

[181] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'13. Lombard, IL: USENIX Association, Apr. 2013, pp. 99–111.

[182] S. Zhang, S. Malik, and R. McGeer, "Verification of Computer Switching Networks: An Overview," in *Automated Technology for Verification and Analysis*, ser. LNCS. Springer, 2012, pp. 1–16.

[183] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, “Kinetic: Verifiable Dynamic Control,” 2014, retrieved Sep 2016. [Online]. Available: http://resonance.noise.gatech.edu/

[184] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, “Merlin: A Language for Provisioning Network Resources,” in *10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT. Sydney, Australia: ACM, Dec. 2014, pp. 213–226.

[185] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, “The Kerberos Network Authentication Service (V5),” RFC 4120 (Proposed Standard), Internet Engineering Task Force, Jul. 2005, updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806, 7751.

[186] S. G. Hong and H. Schulzrinne, “PBS: Signaling Architecture for Network Traffic Authorization,” *IEEE Communications Magazine*, vol. 51, no. 7, pp. 89–96, Jul. 2013.

[187] S. G. Hong, H. Schulzrinne, and S. Weiland, “Signaling Architecture for Network Traffic Authorization,” in *Global Telecommunications Conference (GLOBECOM)*. IEEE, Dec. 2010, pp. 1–6.

[188] S. Kent and K. Seo, “Security Architecture for the Internet Protocol,” RFC 4301 (Proposed Standard), Internet Engineering Task Force, Dec. 2005, updated by RFC 6040.

[189] The netfilter.org project, “netfilter/nftables project.” [Online]. Available: http://www.netfilter.org/

[190] “PF: The OpenBSD packet filter.” [Online]. Available: http://www.openbsd.org/faq/pf/

[191] “Cisco IOS Firewall – Configuring IP Access Lists,” Document ID: 23602, Dec. 2007. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/security/ios-firewall/23602-confaccesslists.html

[192] Hewlett Packard, “IP firewall configuration guide,” 2005. [Online]. Available: ftp://ftp.hp.com/pub/networking/software/ProCurve-SR-IP-Firewall-Config-Guide.pdf

[193] NetCitadel, Inc., “FirewallBuilder,” ver. 5.1. [Online]. Available: http://www.fwbuilder.org

[194] E. Leblond, “Why you will love nftables,” Jan. 2014, https://home.regit.org/2014/01/why-you-will-love-nftables/.

[195] S. Zhang, A. Mahmoud, S. Malik, and S. Narain, “Verification and Synthesis of Firewalls Using SAT and QBF,” in *Network Protocols (ICNP)*, Oct. 2012, pp. 1–6.

[196] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff, “Verified firewall policy transformations for test case generation,” in *3rd International Conference on Software Testing, Verification and Validation*. IEEE, Apr. 2010, pp. 345–354.

[197] A. D. Brucker, L. Brügger, and B. Wolff, “Formal Network Models and Their Application to Firewall Policies,” *Archive of Formal Proofs*, Jan. 2017, formal proof development. [Online]. Available: http://isa-afp.org/entries/UPF_Firewall.shtml

[198] B. Renard, “cisco-acl-to-iptables,” 2013, retrieved Sep 2014. [Online]. Available: http://git.zionetrix.net/?a=summary&p=cisco-acl-to-iptables

[199] M. Gartenmeister, “Iptables vs. Cisco PIX,” Apr. 2005. [Online]. Available: http://lists.netfilter.org/pipermail/netfilter/2005-April/059714.html

[200] A. Jeffrey and T. Samak, “Model Checking Firewall Policy Configurations,” in *Policies for Distributed Systems and Networks.* IEEE, Jul. 2009, pp. 60–67.

[201] S. C. Kleene, *Introduction to Metamathematics*, ser. Bibliotheca Mathematica, D. Bruijn, V. Dantzig, and D. Groot, Eds. Amsterdam: North-Holland, 1952, ISBN 978-0923891572.

[202] J. Engelhardt, “Towards the perfect ruleset,” May 2011. [Online]. Available: http://inai.de/documents/Perfect_Ruleset.pdf

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

[204] T. M. Eastep, “iptables made ease – shorewall,” 2014. [Online]. Available: http://shorewall.net/

[205] “IPTables Example Config,” retrieved Sep 2014. [Online]. Available: http://networking.ringofsaturn.com/Unix/iptables.php

[206] B. Pfaff, B. Heller, D. Talayco, D. Erickson, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Pettit, K. Yap, M. Casado, M. Kobayashi, N. McKeown, P. Balland, R. Price, R. Sherwood, and Y. Yiakoumis, “OpenFlow Switch Specification v1.0.0,” Dec. 2009. [Online]. Available: http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf

[207] A. Nygren, B. Pfaff, B. Lantz, B. Heller, C. Barker, C. Beckmann, D. Cohn, D. Malek, D. Talayco, D. Erickson, D. McDysan, D. Ward, E. Crabbe, F. Schneider, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Tonsing, J. Pettit, K. Yap, L. Poutievski, L. Dunbar, L. Vicisano, M. Casado, M. Takahashi, M. Kobayashi, M. Orr, N. Yadav, N. McKeown, N. dHeureuse, P. Balland, R. Madabushi, R. Ramanathan, R. Price, R. Sherwood, S. Das, S. Gandham, S. Curtis, S. Natarajan, T. Mizrahi, T. Yabe, W. Ding, Y. Yiakoumis, Y. Moses, and Z. L. Kis, “OpenFlow Switch Specification v1.5.1,” Apr. 2015, ONF TS-025. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf

[208] Nicira, Inc., “Nicira extensions,” openvswitch/ovs repository, Jun. 2016, revision fb8f22c186b89cd36059c37908f940a1aa5e1569. [Online]. Available: https://github.com/openvswitch/ovs/blob/master/include/openflow/nicira-ext.h

[209] S. Byma, N. Tarafdar, T. Xu, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Expanding OpenFlow Capabilities with Virtualized Reconfigurable Hardware," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. Monterey, California: ACM, Feb. 2015, pp. 94–97.

[210] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, Apr. 2014.

[211] L. Petrucci, N. Bonelli, M. Bonola, G. Procissi, C. Cascone, D. Sanvito, S. Pontarelli, G. Bianchi, and R. Bifulco, "Towards a Stateful Forwarding Abstraction to Implement Scalable Network Functions in Software and Hardware," *ArXiv e-prints*, Nov. 2016.

[212] T. Nelson, A. D. Ferguson, D. Yu, R. Fonseca, and S. Krishnamurthi, "Exodus: Toward Automatic Migration of Enterprise Network Configurations to SDNs," in *1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15, no. 13. Santa Clara, California: ACM, Jun. 2015, pp. 13:1–13:7.

[213] C. Diekmann, L. Schwaighofer, and G. Carle, "Certifying Spoofing-Protection of Firewalls," in *11th International Conference on Network and Service Management (CNSM)*, Barcelona, Spain, Nov. 2015, pp. 168–172.

[214] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing," RFC 2827 (Best Current Practice), Internet Engineering Task Force, May 2000, updated by RFC 3704.

[215] F. Baker and P. Savola, "Ingress Filtering for Multihomed Networks," RFC 3704 (Best Current Practice), Internet Engineering Task Force, Mar. 2004.

[216] T. J. Wagner, "Disabling reverse-path filtering in complex networks," Jul. 2009, retrieved Mai 2015. [Online]. Available: https://www.tolaris.com/2009/07/13/disabling-reverse-path-filtering-in-complex-networks/

[217] Linux Kernel Sources, "ip-sysctl," Kernel 4.0 Documentation, 2015. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

[218] M. Cotton, L. Vegoda, R. Bonica, and B. Haberman, "Special-Purpose IP Address Registries," RFC 6890 (Best Current Practice), Internet Engineering Task Force, Apr. 2013.

[219] E. S. Al-Shaer and H. H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," in *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 4, Mar. 2004, pp. 2605–2616.

[220] E. Al-Shaer and M. Alsaleh, "ConfigChecker: A Tool for Comprehensive Security Configuration Analytics," in *Configuration Analytics and Automation (SAFECONFIG)*, Oct. 2011, pp. 1–2.

[221] NetCitadel, Inc., "Firewall Builder 5 User's Guide," 2011. [Online]. Available: http://www.fwbuilder.org/4.0/docs/users_guide5/UsersGuide5.pdf

[222] V. Gite, "Linux Iptables Avoid IP Spoofing And Bad Addresses Attacks," nixCraft blog, Jun. 2005. [Online]. Available: http://www.cyberciti.biz/tips/linux-iptables-8-how-to-avoid-spoofing-and-bad-addresses-attack.html

[223] S. Kawamura and M. Kawashima, "A Recommendation for IPv6 Address Text Representation," RFC 5952 (Proposed Standard), Internet Engineering Task Force, Aug. 2010.

[224] V. Capretta, B. Stepien, A. Felty, and S. Matwin, "Formal Correctness of Conflict Detection for Firewalls," in *Workshop on Formal Methods in Security Engineering.* ACM, Nov. 2007, pp. 22–30.

[225] S. Sluizer and J. Postel, "Mail Transfer Protocol," RFC 780, Internet Engineering Task Force, May 1981, obsoleted by RFC 788.

[226] diekmann/Iptables_Semantics, "Issue #113 – Port numbers belong to a specific protocol," github, Jul. 2016. [Online]. Available: https://github.com/diekmann/Iptables_Semantics/issues/113

[227] J. Postel, "Internet Protocol," RFC 791 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1349, 2474, 6864.

[228] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460 (Draft Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112.

[229] J. Reynolds and J. Postel, "Assigned Numbers," RFC 1700 (Historic), Internet Engineering Task Force, Oct. 1994, obsoleted by RFC 3232.

[230] J. Reynolds, "Assigned Numbers: RFC 1700 is Replaced by an On-line Database," RFC 3232 (Informational), Internet Engineering Task Force, Jan. 2002.

[231] Linux Kernel Sources, "Linux/include/linux/netfilter/x_tables.h," Kernel 4.6. [Online]. Available: http://lxr.free-electrons.com/source/include/linux/netfilter/x_tables.h?v=4.6#L343

[232] netfilter coreteam, "libxtables/xtables.c." [Online]. Available: https://git.netfilter.org/iptables/tree/libxtables/xtables.c?h=v1.6.0#n518

[233] T. Tantau and C. Feuersaenger, "The TikZ and pgf packages," 2016, pgfversion 3.0.1a.

[234] CrazyCat, "iptables multiport and negation," Server Fault Question, Aug. 2016. [Online]. Available: http://serverfault.com/questions/793631/iptables-multiport-and-negation/

[235] E. Moy, S. Gildea, and T. Dickey, "XTerm Control Sequences," 2016. [Online]. Available: http://invisible-island.net/xterm/ctlseqs/ctlseqs.html

[236] P. Lammich, *Automatic Data Refinement.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 84–99.

[237] P. Lammich and T. Tuerk, *Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 166–182.

[238] A. Mouat, *Docker Security – Using Containers Safely in Production*, 1st ed. O'Reilly, 2015. [Online]. Available: https://www.openshift.com/promotions/docker-security.html

[239] Docker Core Engineering, "Docker 1.10: New Compose File, Improved Security, Networking And Much More!" blog, Feb. 2016, retrieved Dec 2016. [Online]. Available: https://blog.docker.com/2016/02/docker-1-10/

[240] C. Diekmann, "Issue #29108 – Networking/Security: Custom net with both --internal and --icc=false does not block icc," github, Dec. 2016. [Online]. Available: https://github.com/docker/docker/issues/29108

[241] J. Radhakrishnan, "Docker Networking Design Philosophy," blog, Mar. 2016, retrieved Dec 2016. [Online]. Available: https://blog.docker.com/2016/03/docker-networking-design-philosophy/

[242] B. Meyer, "Issue #22054 – Docker Network bypasses Firewall, no option to disable," github issue and discussion, Apr. 2016, retrieved Dec 2016. [Online]. Available: https://github.com/docker/docker/issues/22054

[243] V. Petersson, "The dangers of UFW + Docker," blog, Nov. 2014, retrieved Nov 2016. [Online]. Available: http://blog.viktorpetersson.com/post/101707677489/the-dangers-of-ufw-docker

[244] I. Duffy, "Azure bug bounty Pwning Red Hat Enterprise Linux," blog, Nov. 2016, retrieved Nov 2016. [Online]. Available: http://ianduffy.ie/blog/2016/11/26/azure-bug-bounty-pwning-red-hat-enterprise-linux/

[245] gdm85, "Docker-fw," github, Mar. 2016, commit d335396. [Online]. Available: https://github.com/gdm85/docker-fw

[246] I. Rad, "DFWFW – Docker Firewall Framework," github, Aug. 2016, commit 94d1036. [Online]. Available: https://github.com/irsl/dfwfw

[247] S. Bleikertz and T. Groß, "A Virtualization Assurance Language for Isolation and Deployment," in *International Symposium on Policies for Distributed Systems and Networks (POLICY).* IEEE, Jun. 2011, pp. 33–40.

[248] Chair of Network Architectures and Services, TUM, "MeasrDroid," http://www.droid.net.in.tum.de/, https://play.google.com/store/apps/details?id=de.tum.in.net.measrdroid.gui.stats.

[249] D. Milojicic, "A Discussion with Leslie Lamport," Aug. 2002. [Online]. Available: http://research.microsoft.com/en-us/um/people/lamport/pubs/ds-interview.pdf

[250] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A Software Defined Internet Exchange," in *ACM SIGCOMM.* Chicago, Illinois: ACM, Aug. 2014, pp. 551–562.

[251] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A Balance of Power: Expressive, Analyzable Controller Programming," in *Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. Hong Kong, China: ACM, Aug. 2013, pp. 79–84.

[252] J. McClurg, H. Hojjat, N. Foster, and P. Černý, "Event-driven Network Programming," in *37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. Santa Barbara, CA: ACM, Jun. 2016, pp. 369–385.

[253] T. Nelson, "FlowLog," github, Apr. 2015, stateful firewall, FlowLog/interpreter/examples/getting-started/sfw.flg, commit 8be5051. [Online]. Available: https://github.com/tnelson/FlowLog

[254] T. Nelson, "First-Order Models for Configuration Analysis," etd-042513-142414, Worcester Polytechnic Institute, Apr. 2013. [Online]. Available: http://www.wpi.edu/Pubs/ETD/Available/etd-042513-142414/

[255] D. Jackson, A. Milicevic, E. Torlak, E. Kang, J. Near, J. Edwards, R. Seater, D. Rayside, G. Dennis, F. Chang, I. Shlyakhter, M. Taghdiri, M. Vaziri, S. Khurshid, and M. Sridharan, "alloy: a language & tool for relational models." [Online]. Available: http://alloy.mit.edu/alloy/

[256] T. Nelson, A. D. Ferguson, and S. Krishnamurthi, *Static Differential Program Analysis for Software-Defined Networks.* Springer International Publishing, 2015, pp. 395–413.

[257] A. Guha, M. Reitblatt, and N. Foster, "Formal Foundations for Software Defined Networks," *Open Networking Summit (ONS) Research Track*, 2013. [Online]. Available: http://www.cs.cornell.edu/~jnfoster/papers/frenetic-verified-controllers-ons.pdf

[258] G. Stewart, *Computational Verification of Network Programs in Coq.* Springer International Publishing, Dec. 2013, pp. 33–49.

[259] H. Date and N. Yoshiura, *Computational Verification of Network Programs for Several OpenFlow Switches in Coq.* Springer International Publishing, Jul. 2016, pp. 223–238.

[260] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A Coalgebraic Decision Procedure for NetKAT," in *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. Mumbai, India: ACM, Jan. 2015, pp. 343–355.

[261] frenetic-lang/netkat-automata, "A Coalgebraic Decision Procedure for NetKAT," github, May 2016, revision d89b2ba8c20a66fda6172edaec5d4714590f7f4b. [Online]. Available: https://github.com/frenetic-lang/netkat-automata

[262] frenetic-lang/frenetic, "frenetic/lib/Frenetic_NetKAT_Json.mli," github, Jul. 2016, revision 8a8d8630cc45dd0904a538efc60b5bec8deed936. [Online]. Available: https://github.com/frenetic-lang/frenetic/

[263] T. Braibant and D. Pous, *An Efficient Coq Tactic for Deciding Kleene Algebras.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 163–178.

[264] D. Pous, *Kleene Algebra with Tests and Coq Tools for while Programs.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 180–196.

[265] frenetic-lang/netkat, "Issue #2 – Current State?" github, Sep. 2016. [Online]. Available: https://web.archive.org/web/20160910140202/https://github.com/frenetic-lang/netkat/issues/2

[266] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[267] secgroup/Mignis, "Mignis is a semantic based tool for firewall configuration." github, Mar. 2014, revision 3448282d5b8e057ebb745bc75af994eae45ff793. [Online]. Available: https://github.com/secgroup/Mignis

[268] secgroup/Mignis, "Issue #2 – _check_filters not called on all user-added matching features," github, Sep. 2016. [Online]. Available: https://web.archive.org/web/20160914073740/https://github.com/secgroup/Mignis/issues/2

[269] E. W. Dijkstra, "A bagatelle for the left hand," Jan. 1982, EWD800. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD800.PDF

[270] M. Dorfhuber and C. Rudolf, "INSALATA – IT NetworkS AnaLysis And deploymenT Application," github. [Online]. Available: https://github.com/tumi8/INSALATA

[271] rakemanyohneth, "What does -P INPUT ACCEPT mean with regards to iptables?" Server Fault Question, Mar. 2016. [Online]. Available: http://serverfault.com/questions/766198/what-does-p-input-accept-mean-with-regards-to-iptables/770971#770971

[272] excanoe, "What is the destination of the spoofed source ip packet in terms of the netfilter chains?" Server Fault Question, Aug. 2016. [Online]. Available: http://serverfault.com/questions/800229/what-is-the-destination-of-the-spoofed-source-ip-packet-in-terms-of-the-netfilte/800434#800434

[273] hsivonen, "Why are these ip6tables rules blocking ssh over IPv6 when the iptables version allows it over IPv4," Server Fault Question, Aug. 2016. [Online]. Available: http://serverfault.com/questions/796630/why-are-these-ip6tables-rules-blocking-ssh-over-ipv6-when-the-iptables-version-a/796729#796729

[274] M. Ragoso, "Iptables rules which only whitelist dns and drop everything else," Server Fault Question, Aug.

2016. [Online]. Available: http://serverfault.com/questions/799212/iptables-rules-which-only-whitelist-dns-and-drop-everything-else/799273#799273

[275] R. Koch, "Will using ACCEPT then DROP for a specific port/ip couple allow the ip but nothing else on that port?" Server Fault Question, Aug. 2016. [Online]. Available: http://serverfault.com/questions/795506/will-using-accept-then-drop-for-a-specific-port-ip-couple-allow-the-ip-but-nothi/795524#795524

[276] C. Diekmann, "Iptables_Semantics – Verified iptables Firewall Ruleset Analysis," github. [Online]. Available: https://github.com/diekmann/Iptables_Semantics

[277] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering – How Google Runs Production Systems*. O'Reilly Media, Mar. 2016, ISBN 978-1-4919-2909-4.

[278] N. McKeown, "Forwarding Plane Correctness," in *Summer School on Formal Methods and Networks*, 2013. [Online]. Available: http://www.cs.cornell.edu/conferences/formalnetworks/

[279] D. E. Knuth, "The correspondence between Donald E. Knuth and Peter van Emde Boas on priority deques during the spring of 1977." [Online]. Available: https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf

ISBN 978-3-937201-57-3

ISBN 978-3-937201-57-3
DOI 10.2313/NET-2017-07-2

ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)