



**Politecnico
di Torino**

Master of Science in Computer Engineering

Master Degree Thesis

Security automation for stateful firewalls

Supervisors

prof. Fulvio Valenza

prof. Riccardo Sisto

dott. Daniele Bringhenti

Candidate

Luana PULIGNANO

ACADEMIC YEAR 2023-2024

Summary

Stateful functions are functions that keep some internal state across invocations, allowing them to retain information from previous executions. This is in contrast to stateless functions, which do not preserve any information between calls and always behave in the same way regardless of how many times they are called.

In networking, stateful functions play a crucial role in ensuring proper communication, security, and efficient data transfer. They are used to monitor connections, manage active sessions, and maintain information about the status of network activities. This helps provide continuity, reliability, and control over traffic and resources.

This thesis focuses on stateful firewalls, a key application of stateful functions. Stateful firewalls are crucial for network management because they enhance the control and monitoring of data traffic by maintaining the state of active connections. This capability allows the firewall to make informed decisions about whether to allow or block traffic based on the state of each session. By preserving connection states, stateful firewalls significantly improve system security, as they can more effectively detect and block potential attacks.

This work specifically focuses on enhancing the functionality of VEREFOO, a framework designed to automatically allocate and configure packet filtering firewalls within a service graph, meeting the security and connectivity requirements specified by the user.

As first thing, a series of verification tests were conducted to evaluate the correctness of the Verification Problem within the framework, which had not been fully tested previously. These tests use a Service Graph, representing the network configuration, and a set of Network Security Requirements (NSRs) that the network must adhere to. Various scenarios were tested, all involving stateful firewalls that were already deployed in the network and configured with their respective rules. The objective of these tests was to determine whether the NSRs were met under the given configuration. The test cases were written in XML files and provided to the platform for analysis.

Following these tests, the analysis shifts to the Refinement Problem, where the framework automatically generates and optimally allocates firewalls in the network based on the input Network Security Requirements (NSRs). To model this problem with stateful firewalls, several Boolean logic formulas were defined. These formulas aim to describe the behavior of a stateful firewall, specifying when traffic should be blocked or permitted, and when to consider the connection state in making these decisions.

In the final part of this work, a set of translators for stateful firewalls was implemented in Java. These translators process XML files containing network configurations and the security policies to be applied. The XML content is translated into configuration files used by specific firewall technologies, including Iptables, Ip-Firewall, and Open vSwitch. The resulting configuration files are tailored to each chosen technology and include all the commands necessary to recreate the network configuration described in the XML files. Executing these configuration files will set up a network with the configured security policies in place.

Acknowledgements

First, I want to thank my parents, who allowed me to continue my academic journey and were there for me at every moment, ready to help with any doubt or difficulty.

I also thank the rest of my family, who always believed in me and encouraged me to achieve this goal.

A special thanks to Peppe, for sharing this chapter of life with me. You were always there when I needed you, never leaving me alone. In you, I found someone to celebrate with during the good times, and you gave me support and comfort when things didn't go as I wanted. Thank you for your unconditional presence.

I thank my friends from campus for all the moments we spent together and the laughter we shared. Here, I was able to form meaningful bonds that I hope will last over time. Your presence made me feel at home, even though I was a thousand kilometers away.

Thanks to my classmates, with whom I shared the emotions of this journey. Despite being in different courses, we still managed to build our friendship, even beyond academic life.

Thanks to my friends back home. The distance may have separated us, but you showed me that those who truly care for me will always be there, even if we can't see each other as often as before.

Finally, I thank myself for not giving up, for finding the courage to leave home and change my life. It was difficult at first, but this experience allowed me to broaden my horizons while also realizing how strong my connection to my homeland is. My wish for myself is to maintain the same dedication and perseverance I've shown over these five years, for all the challenges that await me in the future. *Ad maiora semper!*

Contents

List of Figures	8
List of Tables	9
Listings	10
Introduction	12
1 Stateful Functions	15
1.1 Stateful meaning	15
1.1.1 Stateful vs. Stateless	16
1.2 Network stateful functions	16
1.2.1 Stateful Functions in Cloud	18
1.3 Stateful Firewall	20
1.3.1 Generic Structure	20
1.4 Management Tools and Technologies	21
1.4.1 Iptables	22
1.4.2 IpFirewall	23
1.4.3 Open vSwitch	24
2 z3 theorem prover and VEREFOO	27
2.1 z3 theorem prover	27
2.1.1 SAT and SMT problems	29
2.1.2 MaxSMT problem	30
2.2 Network and Security context	31
2.3 VEREFOO Model inputs	33
2.3.1 Service Graph and Allocation Graph	34
2.3.2 Network Security Requirements	35
2.4 VEREFOO Model outputs	36

2.5	VEREFOO Structure	37
2.5.1	Problem formulation	38
2.6	Formal models	38
2.6.1	Traffic model	39
2.6.2	Network Functions model	39
2.6.3	Traffic Flow model	41
2.6.4	Service and Allocation Graph models	41
2.6.5	Network Security Requirements model	43
2.6.6	Firewall configuration	44
3	Thesis objective	48
4	Verification test and Refinement logical model	50
4.1	Stateful Verification test	50
4.1.1	Verification Policy	53
4.2	Refinement problem	61
4.2.1	From stateless to stateful firewalls	62
4.2.2	New constraint formulations	66
5	Translators for stateful firewall technology	69
5.1	Implementation	69
5.1.1	Iptables	70
5.1.2	IpFirewall	72
5.1.3	Open vSwitch	74
5.2	Examples	76
	Conclusions	83
	Bibliography	84

List of Figures

1.1	Network graph with stateful nodes	17
1.2	Stateful firewall	21
1.3	iptables chain example	22
1.4	The components and interfaces of Open vSwitch	25
2.1	z3 architecture	28
2.2	SDN architecture	32
2.3	NFV architecture	33
2.4	Example of Service Graph	34
2.5	Example of Allocation Graph referring to the Service Graph in Figure 2.4	35
2.6	VEREFOO components	37
4.1	Example 1	56
4.2	Example 2	56
4.3	Stateless schema	63
4.4	Stateful schema	64
4.5	Schema for accept conditionally hard constraint	66
5.1	Firewall translator classes schema	70
5.2	Execution of an Iptables script on a Linux machine	81
5.3	Execution of an OVS script on a Linux machine	82

List of Tables

1.1	Stateful Network Functions and their states and computations . . .	20
-----	--	----

Listings

4.1	Example of an XML network graph	51
4.2	Example of an XML stateful firewall containing two rules	53
4.3	Example of an XML list of Property	57
4.4	Complete Example of an XML test case	58
4.5	Additional property	61
5.1	FirewallDeploy enum	70
5.2	iptables preliminary commands	71
5.3	iptables default rule for accepting return traffic	71
5.4	iptables example	72
5.5	ipfw preliminary commands	72
5.6	ipfw default rules	73
5.7	ipfw example	74
5.8	ovs default rules	75
5.9	ovs example	76
5.10	XML input file example	77
5.11	Firewall A iptables translation	78
5.12	Firewall B iptables translation	79
5.13	Firewall A ipfw translation	79
5.14	Firewall B ipfw translation	79
5.15	Firewall A ovs translation	80
5.16	Firewall B ovs translation	80

Introduction

In an era of increasing network complexity and growing cybersecurity threats, the ability to manage and protect network traffic efficiently is more important than ever. One of the foundational concepts in modern network security is the distinction between stateless and stateful functions. Stateless functions treat each request in isolation, with no memory of previous interactions, making decisions based solely on static rules like source and destination IP addresses or port numbers. While this approach may suffice for simple operations, it lacks the flexibility and contextual awareness needed to address the sophisticated threats and dynamic nature of today's network environments.

Stateful functions, on the other hand, maintain information about the state of a connection across multiple invocations, allowing them to make decisions based on the entire history of the interaction. This capability is crucial in various fields, including network security, where it enables technologies like stateful firewalls to monitor and manage data flows more effectively. Unlike stateless firewalls, which operate on predefined, rigid rules, stateful firewalls track the status of active connections and adjust their behavior dynamically based on the context of the communication. This allows for more intelligent traffic filtering, improved efficiency, and enhanced security.

The advantages of stateful firewalls are numerous. By keeping track of connection states, they enable more precise and informed decisions about whether to permit or block traffic. Once a legitimate connection is established, stateful firewalls allow subsequent traffic to pass without reapplying the same rules for every packet, reducing the computational load on the system. This dynamic filtering not only optimizes network resource usage but also improves overall network performance. Furthermore, from a security perspective, stateful firewalls are significantly better equipped to detect and mitigate sophisticated attacks. By monitoring the entire session and not just individual packets, they can identify and block complex threats such as Denial of Service (DoS) attacks, TCP session hijacking, man-in-the-middle attacks, and replay or spoofing attempts.

This thesis focuses on extending and improving the functionality of VEREFOO, a framework designed to automate the allocation and configuration of firewalls within a network's service graph. VEREFOO is particularly valuable in security automation, which aims to manage and enforce security policies with minimal human intervention. Automation in cybersecurity has become critical as it enables organizations to respond more quickly to emerging threats, reduce the risk of human error, and free up security teams to focus on more strategic initiatives. However, while automation enhances efficiency, it must be carefully designed to complement

human oversight and ensure that automated systems are both reliable and adaptable to changing circumstances.

In particular, this thesis explores the integration and enhancement of stateful firewalls within the VEREFOO framework. Prior to this work, VEREFOO had focused primarily on stateless firewalls, and the potential of stateful firewalls had not been fully explored. Given their superior capability to handle complex, state-dependent network traffic, the goal of this research is to expand the framework to better support the advanced features of stateful firewalls. To achieve this, several key objectives were set.

After introducing the problems and objectives addressed in this thesis, the following section provides a description of how the work is structured.

- Chapter 1: This chapter delves into the concept of stateful functions, highlighting the differences between stateful and stateless functions. It provides an overview of various types of network stateful functions, including those increasingly used in cloud systems, which have rapidly expanded in recent years. A particular emphasis is placed on stateful firewalls, the central topic of this thesis, explaining in detail how state management is handled. The chapter concludes with a description of several firewall technologies applied in real-world environments, such as Iptables, IpFirewall, and Open vSwitch. These tools will be utilized in the implementation phase of the project.
- Chapter 2: This chapter introduces the z3 theorem prover, a solver for the MaxSMT problem, which is used within the VEREFOO framework to address the allocation problem. The chapter provides an in-depth explanation of the MaxSMT problem, tracing its evolution from simpler problems to its theoretical formulation. Following this, the network and security context is discussed, with a particular focus on emerging technologies such as Software-Defined Networking (SDN) and Network Functions Virtualization (NFV). The chapter then presents the inputs and outputs of VEREFOO, along with a detailed description of its internal components. Finally, several formal models employed by the platform are explained, including key concepts such as hard and soft constraints.
- Chapter 3: This chapter provides a brief overview of the objectives of this project, explaining the growing importance of stateful firewalls in modern network security. It highlights why the use of stateful firewalls is strongly recommended today, focusing on their ability to detect and prevent attacks by analyzing the state of active connections.
- Chapter 4: This chapter explains two of the three key contributions of this thesis. The first part focuses on verification tests, conducted to demonstrate that the framework correctly verifies whether the Network Security Requirements (NSRs) are met within a given network graph configuration. New types of policies are introduced to incorporate stateful functions into the framework. In the second part, new logical statements are formulated to address the refinement problem for stateful firewalls, detailing how firewalls should handle packets based on connection state. To implement these features, new constraints are defined.

- Chapter 5: This chapter explains the final contribution to the thesis: the translation from medium-level configuration (XML files) to low-level configuration, utilizing the previously presented technologies for real-world applications, namely Iptables, IpFirewall, and Open vSwitch. The chapter provides detailed explanations of how rules are constructed for each tool, presenting all available options. Finally, some translation examples are included to demonstrate the correct functionality of the translation process.
- Conclusions: This final chapter presents the conclusions of this work, summarizing the results achieved and offering suggestions for future research, aimed at expanding the innovations for stateful firewalls introduced in this project.

Chapter 1

Stateful Functions

The use of stateful functions has seen significant growth in recent years due to the numerous advantages they offer in enhancing network operability. These advantages include improved security, better performance, and increased reliability.

The first part of the chapter explains the meaning of the term "stateful" in a general sense. It includes a comparison between stateful and stateless functions to highlight their differences and determine the appropriate use case for each.

Afterwards, we present an introduction to the application of stateful functions in network environments. As networks become more complex and demanding, the ability of stateful functions to maintain context and state information across multiple interactions has become increasingly valuable, making them a critical component in modern network management.

Following this, the chapter delves into the application of stateful functions within the realm of firewalls. It explains how stateful firewalls operate, detailing their ability to track and manage the state of network connections. The discussion covers how stateful firewalls apply filtering rules based on the state and context of each connection, as opposed to stateless firewalls that treat each packet in isolation. The chapter also highlights the advantages of using stateful firewalls, such as enhanced security and more effective control over network traffic.

1.1 Stateful meaning

Stateful functions refer to functions that retain some form of state across multiple invocations, rather than operating solely based on the inputs provided during a single call. The state refers to the stored data or information that a function or an object has at a particular time. When a function is stateful, it can remember what happened during previous calls, influencing its behavior in future calls.

Stateful applications and processes enable users to save, retrieve, and revisit previously established information and processes online. In these applications, the server tracks the state of each user session, retaining information about the user's interactions and past requests. This allows users to return to their sessions repeatedly, with the current interaction potentially influenced by previous ones. Data are

stored in variables that persist between invocations. The state can be mutable, meaning it can change over time. Each function call can modify the state, which will be used in future calls.

1.1.1 Stateful vs. Stateless

In general, stateful systems are more complex than stateless ones because they require the management and preservation of state across interactions, rather than depending solely on the data provided at each individual step.

A stateless system does not retain any information about previous interactions or transactions. Each request or function call is independent and self-contained, meaning the system does not remember anything about past requests. Stateless operations are often idempotent, meaning that performing the same operation multiple times will yield the same result. For example, fetching a resource should return the same data regardless of how many times the request is made. Since each request is independent, it can be distributed across multiple servers without concerns about shared state, making the system easy to scale and load balance. In contrast, complex interactions that rely on past data cannot be managed by a stateless system and require additional mechanisms to handle state effectively.

On the other hand, stateful systems retain information about past interactions or transactions, allowing the system to remember previous states or data when handling future requests. This capability enables the system to make decisions or perform operations based on historical data. However, managing state introduces additional complexity, as the system must keep track of multiple states and handle transitions between them. Despite this complexity, stateful systems can reduce data transmission overhead and enhance certain system properties.

Therefore, stateful functions are often necessary to meet the requirements of more sophisticated applications that rely on historical data and context.

1.2 Network stateful functions

In networking, stateful functions refer to operations or processes that maintain awareness of the state of network connections or sessions over time. Stateful functions are integral to various network components like firewalls, load balancers, and routing systems, enabling them to make more informed decisions about how to handle traffic.

Stateful devices keep track of the state of network connections, such as TCP connections. When a packet arrives, the stateful device checks if it belongs to an existing connection. If it does, the device knows how to handle it based on the connection's state. If it's a new connection, the device decides whether to allow it and starts tracking it. The Figure 1.1 shows an example of a network graph containing stateful nodes.

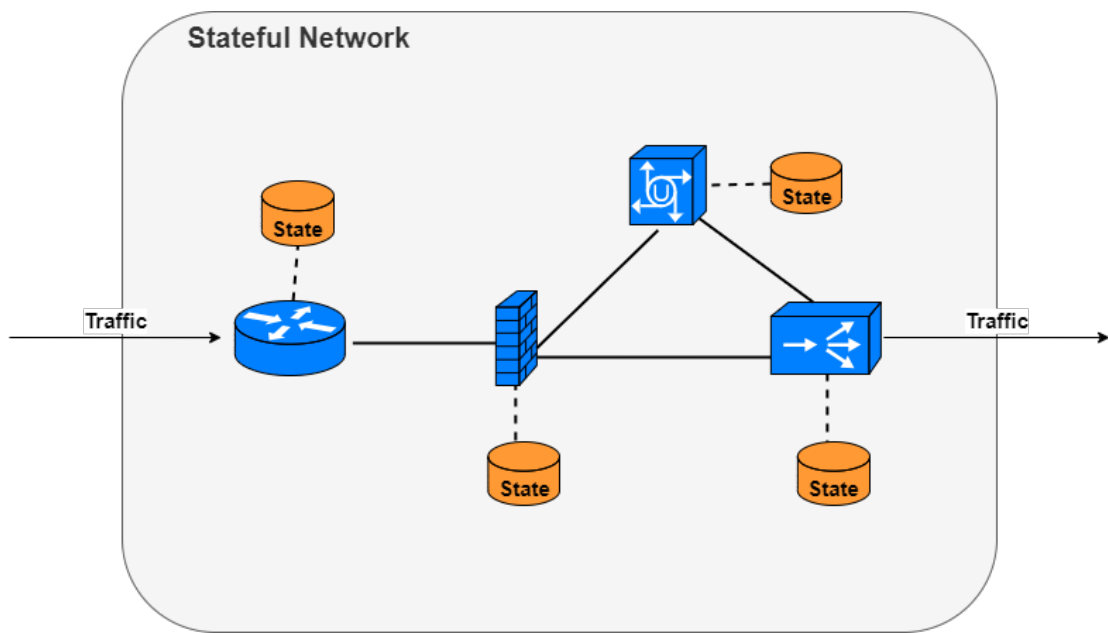


Figure 1.1. Network graph with stateful nodes

Some network elements where stateful functions can be found include the following:

- **Stateful Load Balancers:** They play a crucial role in managing network traffic by distributing incoming requests across multiple servers while maintaining awareness of active sessions. This function is particularly important for applications that require consistent user experiences, such as web applications. By tracking the state of each session, stateful load balancers ensure that all packets related to a specific user interaction are directed to the same server. This capability, often referred to as "sticky sessions," guarantees that once a user logs into an application, their subsequent interactions are handled by the same server, thus preserving the continuity of their session and improving the overall user experience.
- **Network Address Translation:** NAT devices, particularly stateful NAT devices, are essential for managing the translation of internal IP addresses and ports to external ones. These devices maintain a translation table that records each active connection. This table allows them to correctly map outgoing packets to a public IP address and port and ensure that incoming responses are routed back to the appropriate internal device. For example, when a device on a private network initiates a connection to the internet, the stateful NAT device translates the device's internal IP address and port to a public address. When a response returns from the internet, the NAT device uses its state table to direct the response to the correct internal device, ensuring seamless communication.
- **Application-Level Gateways:** ALGs are specialized stateful devices designed to handle specific application protocols such as FTP or SIP, which require

more intricate management than basic packet forwarding. ALGs maintain the state of application-layer connections, allowing them to dynamically open or close ports as needed and to manage complex interactions between clients and servers. For instance, an FTP ALG monitors the state of an FTP session, which includes both control and data connections. It can dynamically open the necessary ports for data transfer, ensuring that the session remains operational and that data is correctly routed.

- Session-aware routers: they use stateful functions to manage network sessions across different interfaces or network segments. These advanced routers ensure that packets belonging to a particular session are consistently routed along the same path through the network. This is crucial in scenarios involving multi-path routing, where maintaining session integrity is vital to prevent out-of-order packet delivery and session disruptions. By tracking the state of sessions, session-aware routers enhance the reliability and performance of network communications, ensuring that all packets of a session follow the intended route and are delivered in sequence.

Firewalls have been intentionally excluded from this list, as they are a central focus of this thesis and will be addressed in a separate section later in the chapter.

1.2.1 Stateful Functions in Cloud

In the evolving landscape of cloud computing, Function-as-a-Service (FaaS) has emerged as a prevalent paradigm, allowing users to develop and deploy discrete functions without worrying about the underlying infrastructure [1]. Traditionally, in FaaS, these functions operate statelessly, meaning they perform computations based on data read from external storage systems and write results back to these systems without retaining any internal state between invocations. This stateless nature enables efficient scaling and parallel execution but does not support state management within the function itself.

Recently, however, there has been a shift towards integrating stateful functions into the serverless model. This integration aims to address the limitations of stateless functions by enabling applications that require consistent state management and high throughput [2].

The introduction of stateful functions brings significant benefits, particularly for use cases where state consistency and transaction coordination are critical. For example, in microservices architectures, maintaining state consistency and managing transactions has traditionally been challenging due to the need for low-latency responses, even under high load. By integrating stateful functions, applications can better handle complex state management scenarios, providing a more robust and reliable solution.

Deploying stateful functions in the cloud, however, presents its own set of operational challenges. Managing state across multiple function instances requires careful consideration of elasticity and failure recovery. Key challenges include:

1. **State Partitioning and Replication:** To manage state efficiently, it should be either partitioned or replicated across active instances of a function. Input events need to be routed to the appropriate partition deterministically. During scale-out, state partitions may need to be further sharded, while scale-in requires merging partitions.
2. **Snapshot Management:** Maintaining a global consistent snapshot of the application’s state is crucial for recovery. This involves capturing snapshots of partitioned state and combining them to ensure consistency at specific points in time. In the event of a failure, a new instance can be instantiated using the latest snapshot, allowing it to reprocess events from the most recent checkpoint.
3. **Integration with Stream Processing:** By integrating stateful functions into a stream processing topology and deploying them on streaming dataflow engines, it is possible to leverage existing research and solutions for more effective state management.

Despite these challenges, the simplicity and cost-effectiveness of Function as a Service (FaaS) make it an appealing choice. To address the need for performant and consistent state management, it is essential to explore designs that combine the benefits of autoscaling and operational efficiency with stateful capabilities. This “stateful” serverless model aims to expand the range of applications and algorithms that can benefit from this type of architectures, showing that serverless computing can support stateful applications while maintaining the ease of use and scalability that characterize the serverless programming model.

As a result, stateful functions are now adopted from many cloud providers in their virtualized environments. Servers use them to manage private address spaces, and offer better QoS and security. Some of the stateful functions supported by public cloud providers are reported in Table 1.1, specifying how the state is managed and detailing the types of computations performed by each function [3]. These functions can be categorized into two main types: some require maintaining state for each individual flow, while others manage state at a coarser level of granularity. Techniques like counters and sketches are used to monitor network usage for diagnostics and billing.

An example of stateful network function adopted in a cloud environment is virtual network peering that enables messages between VMs in different virtual networks (vnets). For it to function correctly, each VM in the involved virtual network must be capable of translating a virtual IP address from any vnet to its corresponding physical address. This translation must be regularly updated whenever VMs are deployed or relocated. Modern implementations of vnet peering store this translation in a stateful layer within the VMs.

Another example is private links. This feature enables VMs to connect with PaaS (Platform as a Service) solutions that have public IP addresses through a more direct route. In this arrangement, a stateful layer within each VM encapsulates outgoing traffic using the VM’s virtual network ID and the PaaS service’s private IP address. On the other hand, a stateful layer at the PaaS service retains

Stateful Function	Network	State at each VM	Computation
Private address spaces		A dictionary that maps customer's private addresses to the provider's physical addresses; one entry per remote endpoint that the VM speaks with.	Lookups, adds and deletes into the mapping dictionary
Stateful ACLs		Per ongoing flow that has passed the ACLs, a hashmap containing the flow's five tuple and the reverse five tuple	Lookups, adds and deletes into the per-flow hash table
Billing		Total bytes, sliced by windows and per billable communicating entity such as a datacenter or a cloud service; also, bursts and peak rates	Multiple counters and sketches
Stateful NATs, load balancers		Per ongoing flow, the new flow to masquerade as.	Lookups, adds and deletes into the rewrite dictionary

Table 1.1. Stateful Network Functions and their states and computations

information about the virtual and physical addresses from which a flow originates during decapsulation and uses this data to correctly encapsulate packets for delivery back to the correct VM.

1.3 Stateful Firewall

In the broader field of applications, stateful functions are notably used in firewalls. Stateful functions in firewalls refer to the capability of a firewall to keep track of the state of network connections and apply filtering rules based on this state. This method enhances security and control compared to stateless filtering, where each packet is treated independently. A stateful firewall monitors the state of active connections and makes filtering decisions based on the state and context of each connection. In contrast, a stateless firewall makes decisions based solely on packet headers, without considering the connection state.

1.3.1 Generic Structure

A stateful firewall maintains a state table (or connection table) that records details about active connections, such as source and destination IP addresses, port

numbers, and the connection state (e.g., established, new). When a packet arrives, the inspection engine checks the state table to determine if the packet is part of an existing connection or a new one, as showed in Figure 1.2. This allows the firewall to make more informed decisions. For a new connection, the firewall examines the packet against predefined rules and, if allowed, creates a new entry in the state table. Subsequent packets that match an entry in the state table are permitted based on the existing connection state, ensuring that only valid packets are allowed through.

The packets that arrive to the firewall can assume several different states:

- New: The initial packet of a connection attempt.
- Established: Packets belonging to a connection that has been established and is ongoing.
- Related: Packets that are related to an existing connection but are not part of the original connection.
- Invalid: Packets that do not match any known connection or have malformed headers.

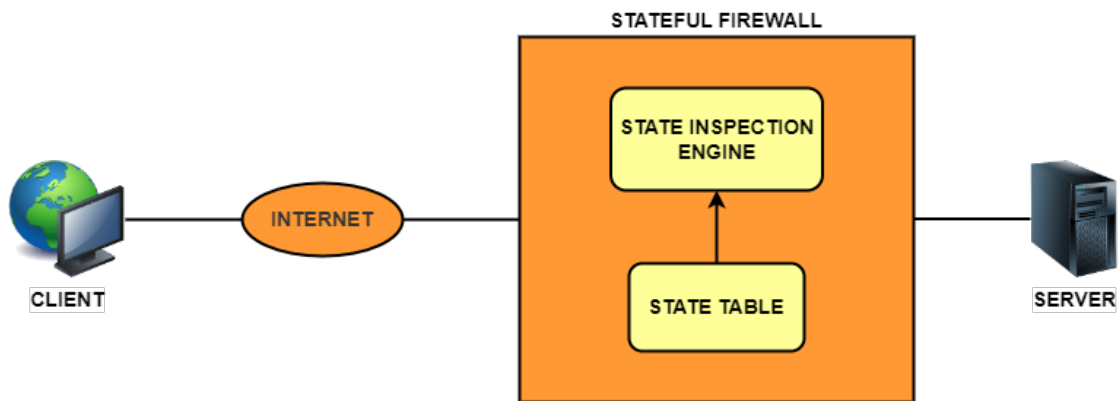


Figure 1.2. Stateful firewall

1.4 Management Tools and Technologies

Stateful firewalls can be managed using various tools designed for network traffic management and firewall configuration. Each tool has its own unique features, so the choice of technology should be based on the specific goals of the configuration. Many of these tools are operated through a Linux terminal, while others provide a web-based GUI for easier management. The following tools are the ones utilized in this thesis work.

1.4.1 Iptables

The primary firewall tool used in Linux is Iptables [4]. An Iptables ruleset is processed by the Linux kernel for each packet, with rules evaluated sequentially. The action is also known as a target and is only applied if the packet meets the criteria specified by the rule. A collection of rules is organized into what is called a chain and it's possible to jump between chains or return to a previous one during processing. The most common actions taken by the Linux kernel are to either ACCEPT or DROP the packet.

For example, consider the firewall rules illustrated in Figure 1.3. The ruleset begins with the INPUT chain. In the first rule, all incoming packets are directed to a user-defined chain called DOS_PROTECT, where rate limiting is applied. If a packet does not exceed certain limits, it passes through this chain and RETURNS to the second rule of the INPUT chain without being DROPPed. In this second rule, the firewall allows all packets belonging to already ESTABLISHED or RELATED connections, which is generally considered good practice. The ESTABLISHED rule often accepts the majority of packets and is typically placed at the beginning of a ruleset for performance reasons. The interesting aspect occurs when the firewall accepts a packet that does not yet belong to an established connection. Once a packet is accepted, subsequent packets from the same connection are treated as ESTABLISHED. The rules that follow are crucial, as they define what types of connections are allowed and which are forbidden. For instance, certain services, identified by their port numbers, are blocked, ensuring that packets with those destination ports cannot establish a connection. At the end the firewall allows all packets from the local network with IP address 192.168.0.0/16, while all the other packets are dropped.

```
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 1.3. iptables chain example

The rulesets of Iptables can be populated by launching a command like the following one:

```
iptables -A TCP -p tcp --dport 22 -j ACCEPT
```

in which it's specified to accept all the TCP packets destined to the port 22. To work with stateful functions, Iptables exploits the functionalities of the *conntrack* module, that manages the states of the connections. The commands used become as the followings

```
iptables -A INPUT -p tcp --dport 22 -m conntrack --ctstate NEW -j ACCEPT
```

This new rule states that all the incoming TCP packets on port 22 that are part of a new connection are allowed and a local states is saved when they are recieved.

```
iptables -A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

This rule permits packets related to an established/related communication for which there is a state expressed in other rules of type `-m conntrack --ctstate NEW` like the first one.

1.4.2 IpFirewall

IpFirewall (IPFW) is a stateful firewall written for FreeBSD, a Unix-like operating system [5]. It presents different components, like the kernel firewall filter rule processor, NAT, stateful inspection and a lot of facilities for accounting, logging, and traffic shaping.

IPFW operates using a list of rules that are evaluated in order. Each rule defines conditions for matching network packets and specifies an action to be taken if a packet matches the rule. Each rule is assigned a number, which determines the order in which it is evaluated. Rules are checked from lowest to highest number, and once a packet matches a rule, the corresponding action is taken, and no further rules are checked, unless the rule specifies to continue. IPFW can filter packets based on whether they are incoming (input), outgoing (output), or being forwarded by the system, for routing scenarios. Rules can match packets based on various criteria like source and destination IP addresses, source and destination ports, protocol, packet flags (such as SYN, ACK in TCP) and interface. The actions that the system can perform on a packet are the subsequent:

- ACCEPT: Allow the packet to continue processing.
- DENY: Drop the packet.
- REJECT: Block the packet and send an error message back to the sender.
- COUNT: Just count the packet but don't alter its flow.

- **DIVERT**: Divert packets to a user-level process, often used with NAT.
- **PIPE**: Pass the packet through a pipe for traffic shaping.

IPFW is integrated with the tool `Dummynet` that provides traffic shaping by using pipes, which control bandwidth, delay, and loss, and queues that manage packet scheduling. Stateful packet inspection is supported by creating dynamic rules that track connections. An example is the following rule

```
ipfw -q add allow tcp from any to any 80 out via tun0 setup keep-state
```

which permits TCP outgoing traffic from any IP address to any IP address with destination port 80 and network interface `tun0` to pass through the firewall. The `setup` option matches packets that are initiating a new TCP connection, that is those with the SYN flag set. The option `keep-state` is responsible of dynamically create a temporary rule that allows packets that are part of this connection to pass through, ensuring that the entire connection (not just the initial packet) is allowed.

1.4.3 Open vSwitch

Open vSwitch (OVS) is an open-source, multilayer virtual switch designed to enable network automation while supporting standard management interfaces and protocols [6]. It is primarily used to provide switching for virtualized environments, including virtual machines and containers like Docker. Open vSwitch is widely used in cloud computing, data center environments, and Software-Defined Networking (SDN) architectures. Open vSwitch is designed to integrate seamlessly with hypervisors like KVM, Xen, and VMware, providing networking capabilities to VMs and containers. It supports OpenFlow, a protocol that allows direct access and manipulation of the forwarding plane of network devices, in accordance with SDN architectures. OVS can perform packet filtering, traffic shaping, and routing, similar to what traditional physical switches and routers do.

In Open vSwitch, two primary components are responsible for directing packet forwarding. The first is `ovs-vswitchd`, a userspace daemon that remains consistent across different operating systems and environments. The second is a datapath kernel module, typically customized for the host operating system to optimize performance.

Figure 1.4 illustrates how these two components collaborate to manage packet forwarding. Initially, the kernel's datapath module receives packets from either a physical NIC or a virtual NIC of a VM. If `ovs-vswitchd` has already provided instructions on how to handle packets of this type, the datapath follows those instructions, which could include actions like transmitting the packet through specific ports, modifying the packet, sampling it, or even dropping it. If the datapath has no prior instructions, it forwards the packet to `ovs-vswitchd` in userspace, where the appropriate handling method is determined. After processing, `ovs-vswitchd` sends the packet back to the datapath with the required actions. Typically, `ovs-vswitchd` also instructs the datapath to cache these actions for more efficient processing of similar packets in the future.

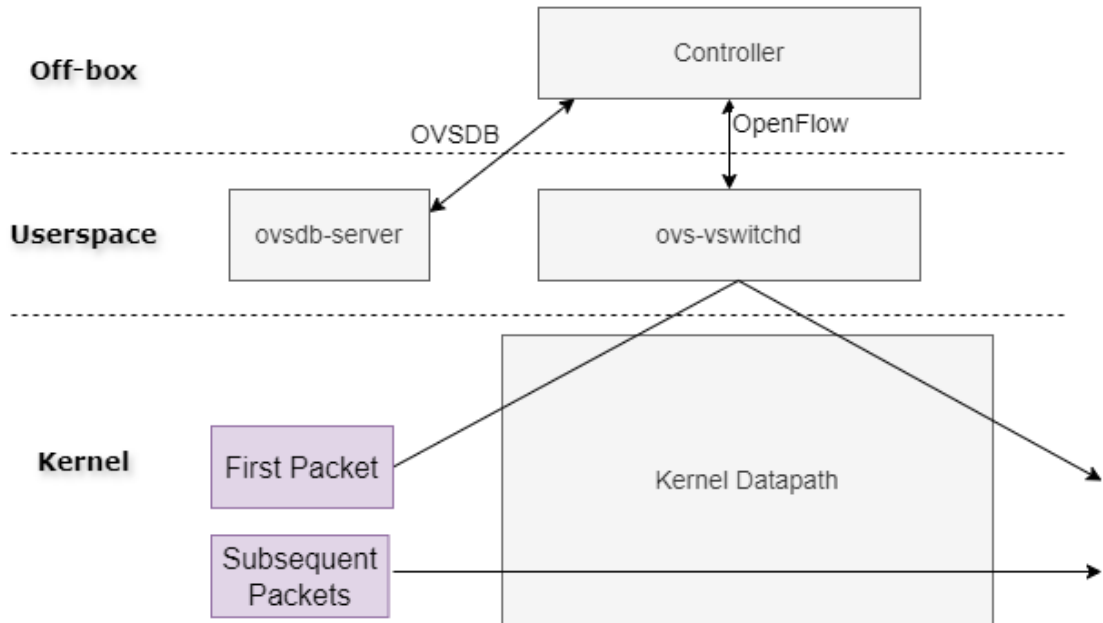


Figure 1.4. The components and interfaces of Open vSwitch

Open vSwitch uses a tuple space search classifier for packet classification in both the kernel and userspace. This approach involves organizing flow tables into hash tables based on the fields they match. When flows with different matching criteria are added, the classifier creates additional hash tables for each unique set of matched fields. During a search, the classifier checks all relevant hash tables. If a match is found in one, that flow is returned; if matches exist in multiple tables, the flow with the highest priority is selected. The classifier dynamically expands to include more hash tables as new flows with different match criteria are added.

OVS can be used with the Connection tracking system to track the state of a connection, supporting both stateless and stateful protocols [7]. The module used for stateful packet inspection is called `conntrack`. An example in which is used this module is the following rule

```
match: in_port(1),tcp,conn_state=-tracked; action: conntrack(zone=10),
normal
```

This specifies that the rule matches TCP packets that are entering the switch on port 1. The `in_port` is a virtual or physical port on the switch. `conn_state` refers to the connection state of the packet as tracked by the `conntrack` module. `tracked` means that it's the first time the packet is being seen by the switch, so it doesn't yet have a connection state associated with it. Each state is preceded by either a "+" for a flag that must be set, or a "-" for a flag that must be unset. `conntrack(zone=10)` assigns the packet to a specific zone, in this case, zone 10.

Zones are a way to isolate different sets of connections, allowing different connection tracking rules or behaviors for different zones. `normal` tells the switch to process the packet according to its usual switching and routing logic, rather than applying specific OpenFlow rules. The described rule is often paired with rules that manage already established connections, like the following

```
match: in_port(2),tcp,conn_state+=established; action: output:1
```

This specifies that the rule matches TCP packets that are entering the switch through port 2. `+established` means the rule matches packets that are part of an already established connection. These packets will be forwarded out of port 1.

Chapter 2

z3 theorem prover and VEREFOO

In recent years, automation has emerged as a critical concept in the realms of networking and cybersecurity. As networks and systems become more complex, the traditional methods of manual configuration and management are increasingly prone to errors and inefficiencies. Automation addresses these challenges by streamlining processes, reducing human intervention, and enhancing overall system reliability. The VEREFOO framework has been developed to address these new security needs while also providing optimal solutions.

One of the framework's problems that has been the focus of this thesis work has been formulated as a MaxSMT problem, discussed in the first part of the chapter. This problem type is addressed by the z3 theorem prover, which provides both a model and a solution.

The second part of the chapter outlines the background problems that led to the emergence of this new model. The remainder of the chapter discusses VEREFOO, detailing its components and the functionalities it offers.

2.1 z3 theorem prover

One of the framework's problems that has been the focus of this thesis work has been formulated as a MaxSMT problem. This problem type is addressed by the z3 theorem prover, which provides both a model and a solution.

z3, developed by Microsoft Research, is a high-performance SMT solver specifically aimed at addressing problems encountered in software verification and analysis [8]. Satisfiability Modulo Theories (SMT) extends Boolean satisfiability (SAT) by incorporating reasoning with additional first-order theories, such as equality, arithmetic, fixed-size bit-vectors, arrays, and quantifiers. An SMT solver is a tool designed to determine the satisfiability or equivalently, the validity of formulas within these theories. SMT solvers are used for different aims, including extended static checking, predicate abstraction, test case generation, and bounded model checking over infinite domains.

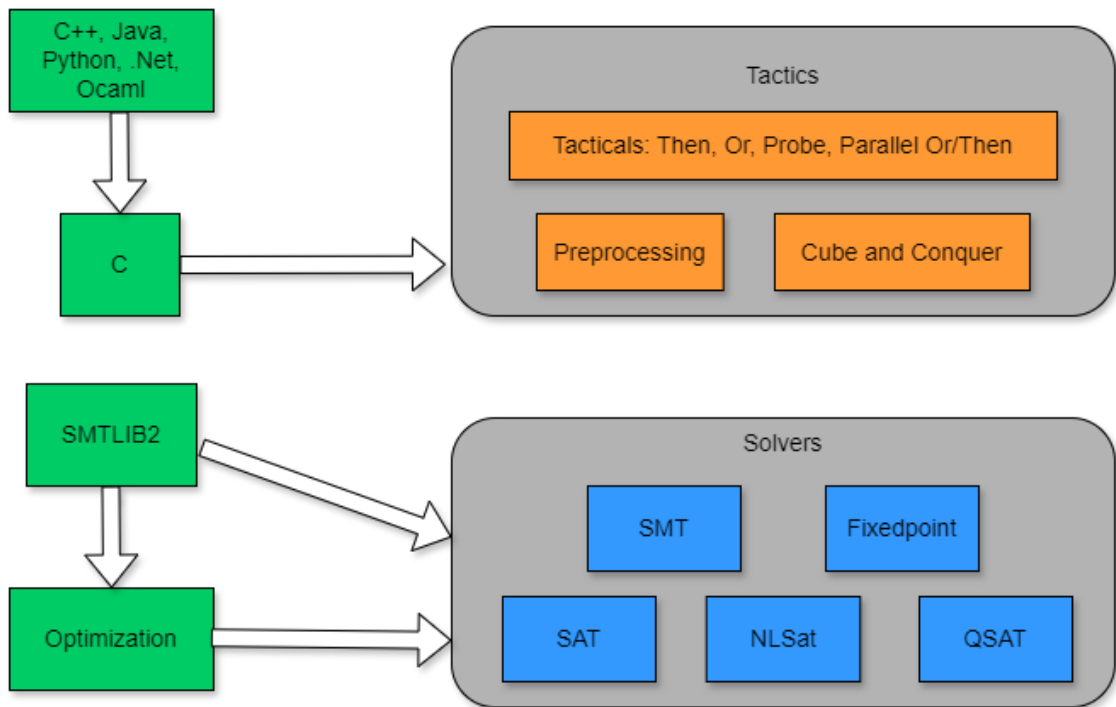


Figure 2.1. z3 architecture

z3 works by converting logical formulas into a format it can process and then applying a series of algorithms and heuristics to determine whether the formula is satisfiable (i.e., there exists an assignment of variables that makes the formula true) or unsatisfiable (no such assignment exists). If the formula is satisfiable, z3 can also produce a model, which is an example of an assignment that satisfies the formula. It provides APIs (Application Programming Interfaces) in various high-level programming languages, C, C++, Java and Python.

z3 is built on a layered architecture that includes:

- **Front-End Parsing:** z3 can accept inputs in various formats like SMT-LIB, a standard language for SMT solvers, and custom formats like Python bindings for more interactive use.
- **Preprocessing:** z3 includes several preprocessing techniques to simplify and normalize the input formulas before they are passed to the core solving engine. This can include formula rewriting, constant propagation, and variable elimination.
- **Core Solving Engine:** The heart of Z3, this engine uses a combination of SAT solving and theory-specific reasoning. SAT solvers handle the propositional logic part, while specialized theory solvers handle constraints related to arithmetic, arrays, bit-vectors, etc.
- **Theory Combination:** z3 uses different techniques to combine results from different theory solvers, allowing it to solve complex formulas that involve multiple theories simultaneously.

- **Conflict Resolution and Learning:** z3 employs techniques from SAT solving like conflict-driven clause learning (CDCL) to efficiently prune the search space and avoid redundant calculations.

To get the solution, one of the possible solvers is used. The solution provided can be optimized by z3 thanks to the use of an optimizer engine called z3Opt. The image [2.1](#) shows the described process.

2.1.1 SAT and SMT problems

The SAT (Boolean Satisfiability) problem is a fundamental problem in computer science and mathematical logic. It involves determining whether there exists an assignment of truth values (true or false) to variables in a Boolean formula such that the formula evaluates to true.

A Boolean formula is composed of variables, logical operators (AND, OR, NOT), and possibly parentheses to group expressions. The problem asks if it's possible to assign the variables in such a way that the entire formula becomes true. If such an assignment exists, the formula is satisfiable; otherwise, it is unsatisfiable. Given the structure of the problem, a SAT solver doesn't need to optimize the solution but it only needs to find one.

SAT is an NP-complete problem, meaning that any problem in the class NP (nondeterministic polynomial time) can be reduced to it. This complexity implies that, while it is easy to verify a solution to a SAT problem, finding that solution is potentially very hard. No polynomial-time algorithm is known for solving all instances of SAT.

The Satisfiability Modulo Theories (SMT) problem is an extension of the Boolean Satisfiability (SAT) problem that involves logical formulas with constraints from various mathematical theories.

SMT extends Boolean logic by incorporating constraints from various theories. These theories represent different types of mathematical or logical domains and provide specialized methods for reasoning about constraints. Common theories include linear and non-linear arithmetic, arrays, bit-vectors, floating point arithmetic and uninterpreted functions. The language obtained is the so-called first-order logic. The formula needs to be satisfied by finding an assignment of values to variables that satisfies both the Boolean part and the theory-specific constraints.

SMT is generally decidable, meaning that there is an algorithm that will correctly determine the satisfiability of any given formula. However, the complexity can vary based on the theories involved. For instance, SMT with quantifiers can be PSPACE-complete, which is a more complex class than NP-complete. The complexity can be higher when dealing with non-linear arithmetic or certain combinations of theories.

z3 is highly optimized for performance and can handle both SAT and complex SMT problems efficiently. It supports incremental solving, which allows users to add constraints dynamically and solve updated problems without restarting from scratch. Also, it offers a wide range of theories and their combinations.

2.1.2 MaxSMT problem

MaxSMT (Maximum Satisfiability Modulo Theories) is an extension of the Satisfiability Modulo Theories (SMT) problem. While SMT seeks to determine whether a formula can be satisfied under given constraints, MaxSMT goes a step further by trying to maximize the number of satisfiable constraints or optimize a weighted sum of satisfiable constraints. It's particularly useful in scenarios where not all constraints can be satisfied simultaneously, and the goal is to find the best possible solution under the circumstances.

MaxSMT problems can be classified into different types based on how the soft constraints are handled, the nature of the optimization goal, and the types of theories involved. Below are some of the most used models:

- **Unweighted MaxSMT:** In an unweighted MaxSMT problem, all soft constraints are treated equally. The goal is to maximize the number of soft constraints that are satisfied. This type is useful in scenarios where there is no preference among the soft constraints, and the objective is simply to satisfy as many as possible.
- **Weighted MaxSMT:** In a weighted MaxSMT problem, each soft constraint is assigned a weight, representing its importance. The goal is to maximize the total weight of the satisfied soft constraints. This type is useful when some soft constraints are more important than others, and you want to prioritize satisfying the more critical ones.
- **Partial MaxSMT:** In a partial MaxSMT problem, the set of constraints is divided into hard and soft constraints. The goal is to satisfy all hard constraints and as many soft constraints as possible, similar to unweighted MaxSMT, but with a clear distinction between mandatory and optional constraints. This type is suitable for situations where some constraints must absolutely be satisfied, and the others are desirable but not mandatory.
- **Weighted Partial MaxSMT:** Weighted Partial MaxSMT is a specific type of MaxSMT problem that combines elements of both weighted MaxSMT and partial MaxSMT. In this type of problem, constraints are divided into two categories: hard constraints that must be satisfied and soft constraints that are weighted. The goal is to satisfy all hard constraints while maximizing the total weight of the satisfied soft constraints.

Among the various contributions of this thesis to the VEREFOO framework, one significant aspect is related to weighted partial MaxSMT. This approach distinguishes between hard and soft constraints. Hard constraints are non-negotiable. Any solution must satisfy all hard constraints. If even one hard constraint cannot be satisfied, the problem is unsatisfiable. Soft constraints are desirable but not mandatory. Each soft constraint is assigned a weight, indicating its importance or priority. The solver's goal is to maximize the total weight of the satisfied soft constraints.

A weighted partial MaxSMT problem is typically formulated as follows:

Given

- A set of hard constraints $\mathcal{H} = \{h_i \mid i = 1, 2, \dots, m\}$ where h_i denotes the i -th hard constraint
- A set of soft constraints $\mathcal{S} = \{(s_i, w_i) \mid i = 1, 2, \dots, n\}$ where s_i denotes the i -th soft constraint and w_i represents its associated weight

The weighted partial MaxSMT problem is expressed as

$$\max \sum_{i=1}^S w_i \cdot s_i$$

subject to $h_j, \forall j \in [1, H]$

To address this type of problem, the z3 solver has been used because it allows for the inclusion of both hard and soft constraints to find the optimal solution.

2.2 Network and Security context

With the rapid advancement of technology, the landscape of networking and software development has undergone significant changes. In particular, new paradigms like Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) have recently been introduced, revolutionizing the way networks are designed and managed. These technologies have brought a new level of flexibility and adaptability to software, enabling it to be more elastic and dynamic in response to changing demands.

SDN decouples the network control plane from the data plane, allowing for centralized management and greater control over network traffic [9]. This separation simplifies the network architecture, making it easier to implement changes, optimize performance (Figure 2.2).

NFV, on the other hand, virtualizes network functions that were traditionally carried out by dedicated hardware devices, such as routers, firewalls, and load balancers [10]. By virtualizing these functions, NFV allows them to run on standard servers, reducing hardware costs and increasing the agility of network operations. An example of an NFV architecture is shown in Figure 2.3.

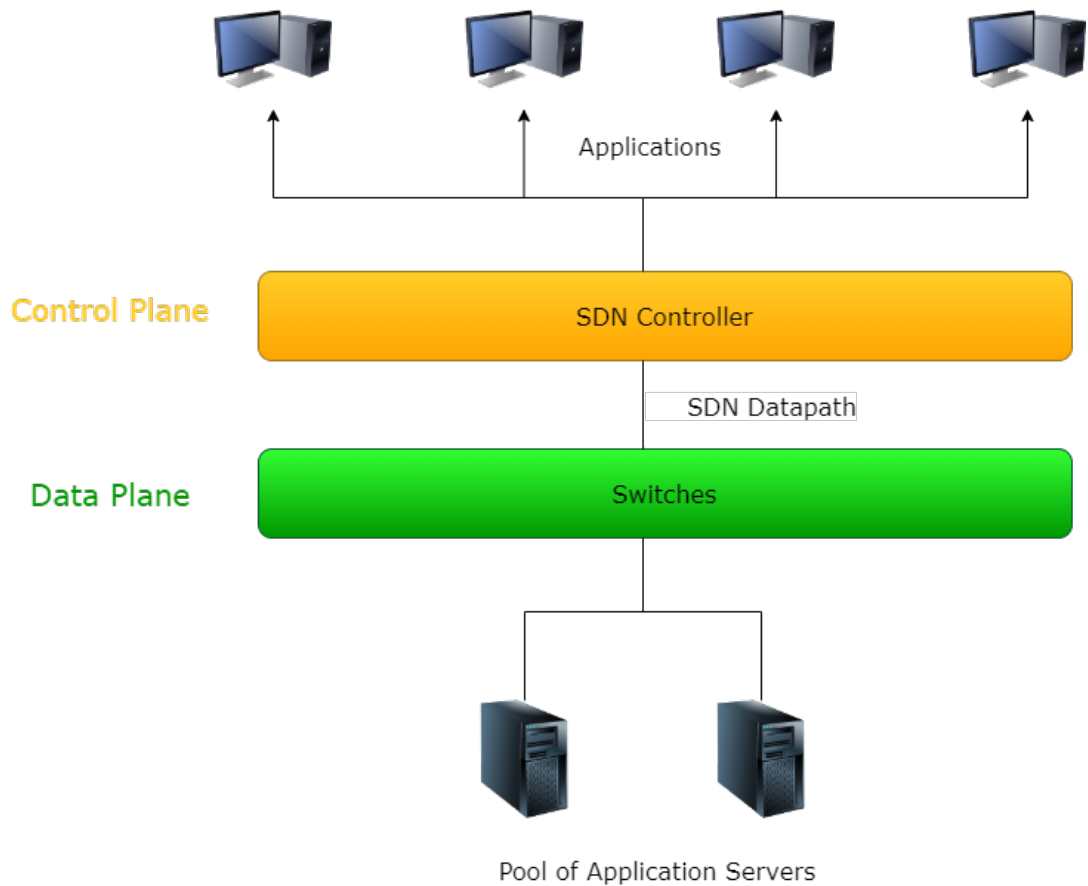


Figure 2.2. SDN architecture

Together, SDN and NFV enable organizations to respond more quickly to evolving business needs and technological developments. They allow for the rapid deployment and scaling of network services, without the need for extensive hardware changes. This dynamic approach not only improves efficiency but also supports the creation of more resilient and adaptive networks that can better handle the complexities of modern digital environments. As a result, these paradigms are playing a crucial role in shaping the future of networking and software development, driving innovation and enabling more agile, scalable, and secure solutions.

Thanks to these new technologies, users are able to dynamically create Service Function Graphs (SG), which represent virtual networks that are independent of the underlying physical infrastructure.

A key challenge is ensuring that the Network Security Requirements (NSRs) are met within these SGs. Traditionally, the management of security has been separate from network management, which can lead to potential errors or suboptimal configurations due to miscommunication or lack of expertise. Additionally, the manual configuration of Network Security Functions (NSFs), such as firewalls, is slow and prone to human error, increasing the risk of vulnerabilities.

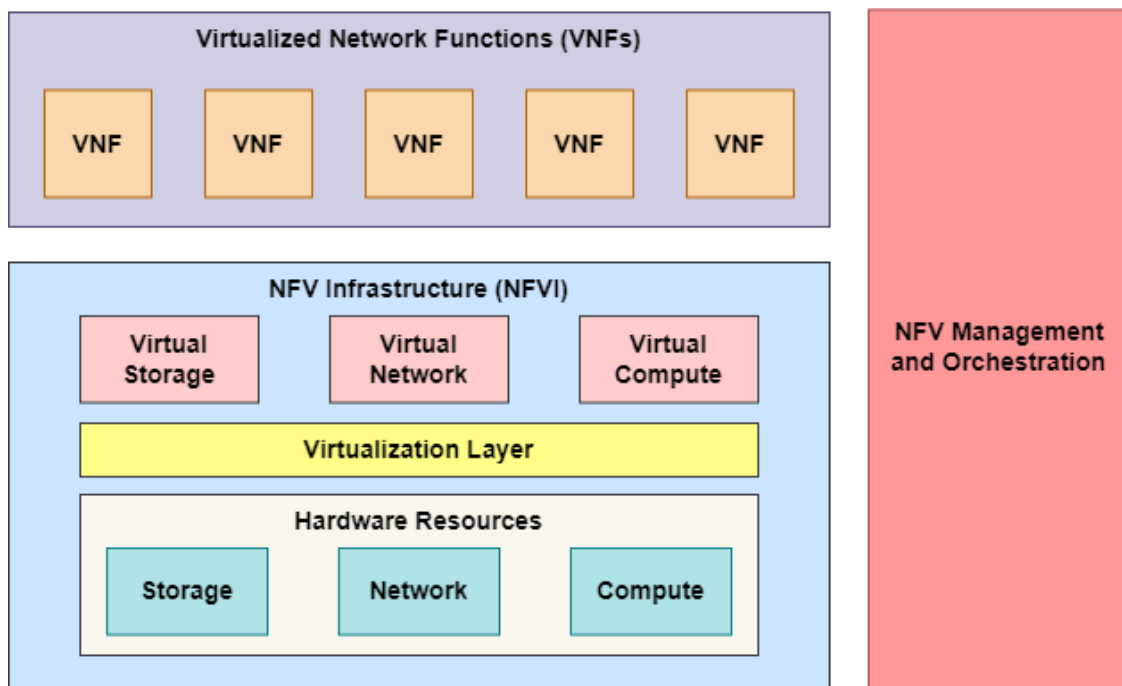


Figure 2.3. NFV architecture

With the rise of virtualized networks and SDN, distributed architectures have become more common, allowing firewalls to be placed at multiple points to enhance security and efficiency. However, manually configuring these complex architectures is challenging and demands automation, alongside formal verification techniques, to ensure correct and optimized solutions.

Despite the importance of the issue, the automation of network security configuration has been underexplored in the literature. To address these challenges, VEREFOO has been designed to tackle these emerging problems, as described in [11], [12] and [13]. This approach aims to automatically allocate and configure packet filters—traditional firewall technology—within an SG defined by the service designer, ensuring that the specified security requirements are met. The method uses a formal model to guarantee correctness (correctness-by-construction) and optimizes the placement and configuration of firewalls, minimizing the number of firewalls and rules, thus improving performance and reducing resource costs [14].

The proposed solution formulates the problem as a partial weighted Maximum Satisfiability Modulo Theories (MaxSMT) problem, which can be solved automatically. This approach guarantees that the solution satisfies all the hard logical constraints and is optimal according to the defined criteria.

2.3 VEREFOO Model inputs

This section introduces the components used by the framework as input: the Service Graph (SG), that represents the logical topology of the network and the NSRs, that

are the security properties that the network must satisfy. The SG is often reduced in the Allocation Graph(AG), explained later.

2.3.1 Service Graph and Allocation Graph

A *Service Graph* (SG) represents the logical topology of a virtual network, encompassing an interconnected set of service functions and network nodes that together deliver a complete end-to-end network service. Unlike a Service Function Chain (SFC), where functions are arranged linearly, an SG can feature a complex architecture with multiple traffic paths from sources to destinations. The SG is designed by a network service designer, focusing solely on providing networking services to users, whose access points are represented by the SG's endpoints (e.g., clients, servers, subnetworks). Security considerations are not involved at this stage. An example of a Service Graph can be seen in Figure 2.4.

The functions available to the service designer for constructing an SG are known as *Network Functions* (NFs), which provide various functionalities such as web caching and load balancing. Low-level functions like switches and routers, which simply forward incoming packets based on a forwarding or routing table, are not explicitly included in the SG. However, this does not mean they are absent from the real network; rather, the SG abstracts these functions, concentrating on more complex service functions while assuming the low-level functions correctly implement the SG connections.

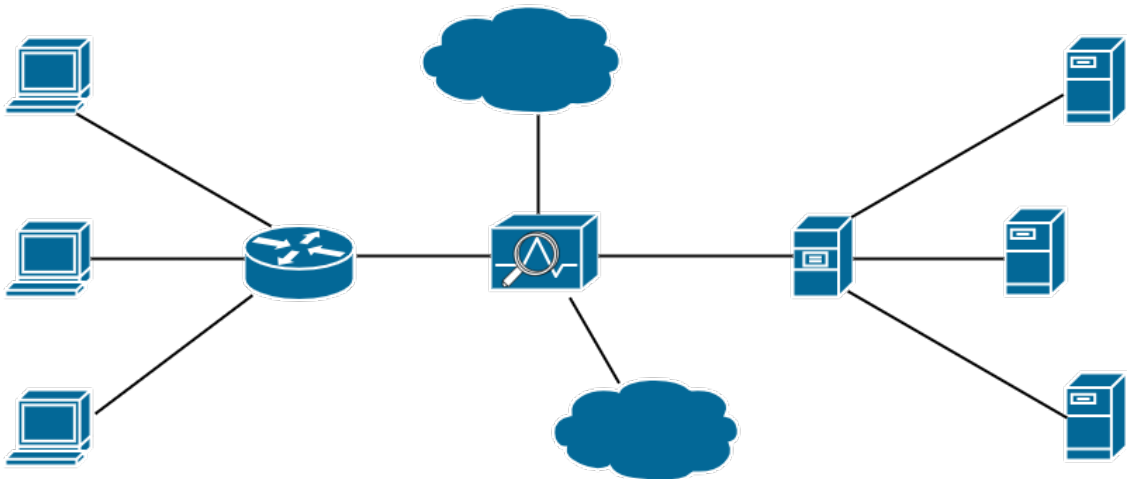


Figure 2.4. Example of Service Graph

The SG designed by the service designer is automatically processed to create an internal representation called the *Allocation Graph* (AG). By default, for each link between network nodes or functions, a placeholder element known as an *Allocation Place* (AP) is generated. The tool then determines whether to place a firewall at each AP to achieve the optimal allocation scheme. Figure 2.5 shows the corresponding Allocation Graph referring to the Service Graph in Figure 2.4.

However, a security-skilled service designer has the flexibility to either enforce the placement of a firewall at a specific AP, preventing its removal by the tool, or prohibit certain APs from being considered as valid firewall positions. This capability enhances the flexibility of the methodology and reduces computation time by narrowing the solution space the tool must search. It is particularly useful in mixed scenarios where firewalls are implemented not only by Virtual Network Functions (VNFs) but also by existing hardware packet filters, which can be modeled as immutable firewalls within the tool.

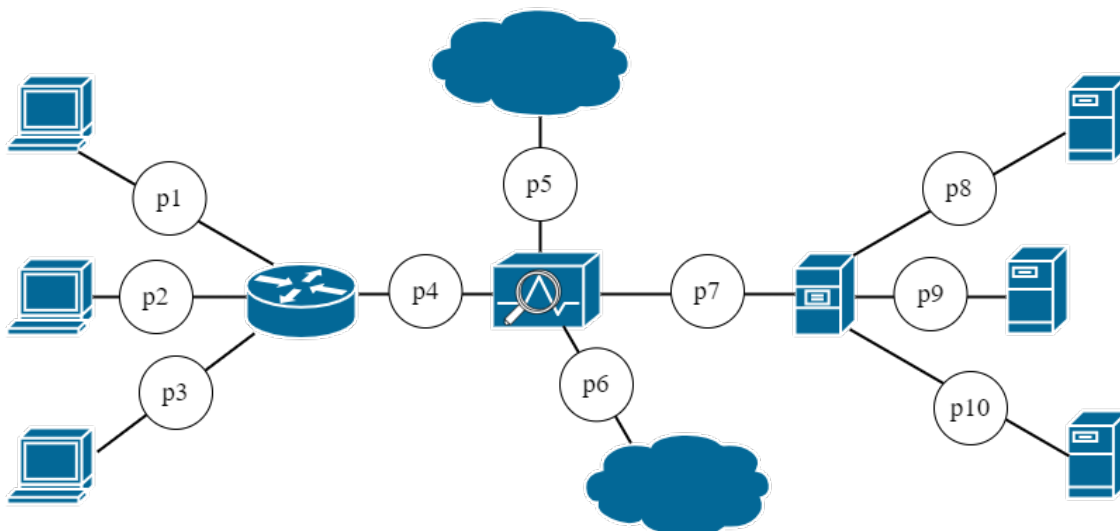


Figure 2.5. Example of Allocation Graph referring to the Service Graph in Figure 2.4

While these manual inputs can be beneficial, they also carry the risk of preventing the tool from finding a solution or leading to a suboptimal one, as potentially acceptable—and even optimal—solutions might be excluded based on user input.

2.3.2 Network Security Requirements

The methodology used primarily addresses the security requirements related to connectivity within the network service, specifically focusing on which traffic flows should be permitted or blocked between any pair of endpoints in the Service Graph (SG). These security constraints are a key input to the framework and are defined by a default behavior that applies to traffic flows for which no specific instructions are provided by the user, and a set of specific Network Security Requirements (NSRs), which indicate whether a particular traffic flow should be allowed (reachability requirement) or blocked by a firewall (isolation requirement).

The service designer has three approaches to define these security constraints. The first two approaches rely on traditional whitelisting and blacklisting methods. In the whitelisting approach, all traffic flows are blocked by default, except for those explicitly allowed by the user through reachability requirements. Conversely, in the blacklisting approach, all traffic flows are permitted by default, except for those explicitly denied through isolation requirements.

The third approach, known as the specific approach, allows the service designer to explicitly define only the requirements of interest, including both isolation and reachability constraints. The system will then compute the optimal solution based solely on this specific set of constraints, automatically determining whether to allow or block unspecified traffic flows. In this approach, it is assumed that the entire set of security requirements is conflict-free. This assumption ensures that no prioritization criteria are needed in the formulation of the security constraints.

2.4 VEREFOO Model outputs

This section outlines the potential outputs of the framework after resolving a security issue using automated technology. The following analysis is referred to the logical level and not the physical one.

In the event of a successful outcome, the framework provides two key outputs, that are the allocation scheme for distributed firewall instances within the Service Graph (SG), and the Filtering Policy (FP) for each allocated firewall. The firewall allocation scheme identifies the specific Allocation Places (APs) where firewall instances should be deployed. Each FP consists of a set of filtering rules configured for the allocated firewalls, expressed in an abstract language that remains independent of any specific firewall configuration languages.

A Filtering Policy includes a default action that can be whitelisting or blacklisting. In the first case the firewall blocks all traffic unless explicitly allowed while in the second the firewall permits all traffic unless explicitly blocked. The FP also includes a set of automatically generated rules, free from anomalies, that state how specific types of traffic flows are managed.

The allocation scheme produced ensures the deployment of the minimum number of firewall required to apply all the Network Security Requirements, thereby optimizing resource usage. Simultaneously, each Filtering Policy is allocated with the minimum number of rules necessary to meet the requirements, reducing memory consumption and enhancing firewall performance.

Once the output solution is generated, it can be automatically deployed into the virtual network using existing technologies. If a new security configuration is needed, such as in response to a new detected attack, new Network Security Requirements can be provided to the tool, that will automatically generate the necessary configuration updates.

However, if no viable solution is found, the framework generates a non-enforceability report. This report helps the user understand why the NSRs could not be applied. A potential motivation for failure could be that the SG does not provide adequate Allocation Places for firewall setting due to user-imposed constraints. In such cases, one strategy for obtaining a working solution is to relax some of the user-defined constraints and rerun the process.

2.5 VEREFOO Structure

VEREFOO handles the configuration of a complex network, represented by a Service Graph (SG) or Allocation Graph (AG), and allocates a set of Network Security Functions in order to meet the specified security policies. The process is automated and delivers an optimized solution. The framework is organized into modules, which are presented below and showed in the picture 2.6.

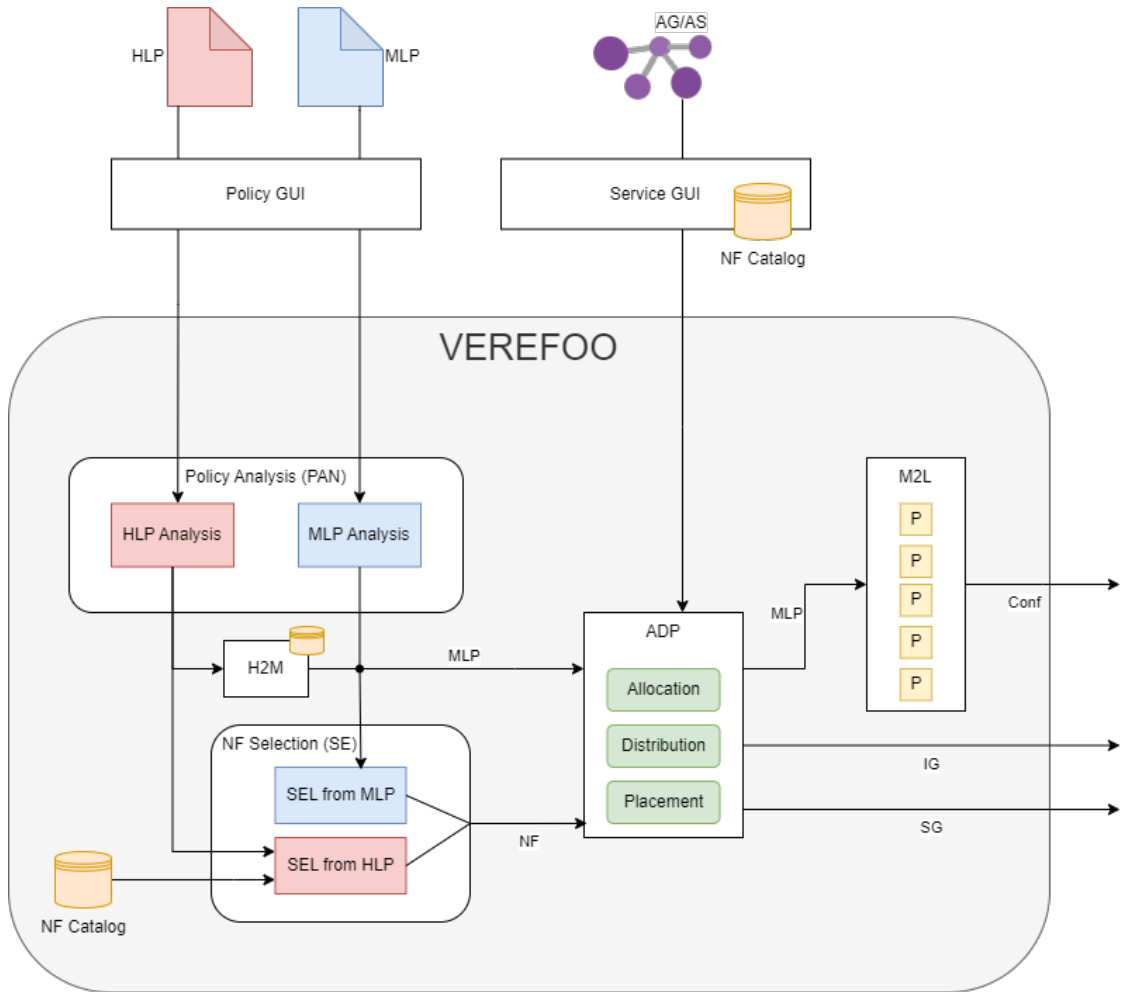


Figure 2.6. VEREFOO components

First, users can define Network Security Requirements (NSRs) using the Policy GUI, with the option to express them as either High-Level Policies (HLP) or Medium-Level Policies (MLP), depending on their experience. HLPs are automatically translated into MLPs by the H2M (High-to-Medium) Module. MLPs contain all the detailed information necessary for configuring the Network Security Functions (NSFs) later.

The Policy Analysis (PAN) module represents a preliminary phase, where it receives the NSRs as input and checks for errors and conflicts. It ensures that the

NSRs are conflict-free and returns a minimal set of constraints to be satisfied. If conflicts cannot be automatically resolved, the module provides a report detailing the issues. Next, the NF Selection Module (SE) reviews the expressed NSRs and selects the necessary NSFs from a pre-built NF catalog, which includes all available functions for the system.

The central component of the architecture is the Allocation, Distribution, and Placement (ADP) Module. It computes the final Service Graph with the selected NSFs, using the medium-level NSRs, the list of chosen NSFs, and the original Service Graph, which is transformed into the corresponding Allocation Graph. The ADP module employs a partial weighted MaxSMT solver, z3Opt, to find the optimal allocation of NSFs within the network. The solver treats security requirements as non-negotiable hard constraints, while other specifications are treated as weighted, optional soft constraints to achieve the most optimal solution.

Finally, the M2L (Medium-to-Low) Module translates the medium-level policy rules generated by the solver into low-level language. This translation is tailored to the specific implementation of each network function that needs to be configured.

2.5.1 Problem formulation

To achieve the correct solution for the firewall allocation and auto-configuration problem, a partial weighted MaxSMT problem is employed. This formulation is essential to meet the three main goals of the proposed approach: full automation, optimization, and formal correctness. The first one is guaranteed since a MaxSMT problem can be solved without human intervention, apart from the initial input specification. Optimization is accomplished by representing the objectives as soft constraints, while formal correctness is ensured by defining the correctness requirements as hard constraints. The used approach not only enhances confidence in the solution's accuracy but also eliminates the need for post-verification, as the solution is considered formally correct if all components are modeled accurately.

It is essential that these models capture all relevant information influencing the solution's correctness, including both the security requirements and the network's forwarding behavior where these requirements must be applied. Additionally, to maintain scalability, the number and complexity of constraints in the MaxSMT problem must be minimized. Therefore, effective modeling of the problem components during MaxSMT formulation is a critical step.

2.6 Formal models

This section presents the formal models defined and used in VEREFOO to describe network elements, including Allocation and Service Graph, traffic, firewalls, and the predicates and functions employed.

2.6.1 Traffic model

The traffic t , also called class of packet, is represented as a predicate based on the values of the TCP/IP 5-tuple packet fields [15]. In particular, t is represented as a disjunction of predicates $q_{t,1} \vee q_{t,2} \vee \dots \vee q_{t,n}$, where each $q_{t,i}$ is defined over the 5-tuple fields. A packet is part of class t if and only if its 5-tuple satisfies at least one $q_{t,i}$. To maintain simplicity while ensuring a flexible model, each $q_{t,i}$ is assumed to be the conjunction of five predicates, one for each field in the 5-tuple. For convenience, each $q_{t,i}$ is written as:

$$q_{t,i} = (IP_{Src}, IP_{Dst}, p_{Src}, p_{Dst}, t_{Prt})$$

where IP_{Src} , IP_{Dst} , p_{Src} , p_{Dst} , and t_{Prt} are the five predicates.

In the case of IPv4 addresses, it's supposed that IP_{Src} and IP_{Dst} are conjunctions of four predicates, one for each byte of the IP address. Each of these four predicates can represent either a single integer or a range of values between 0 and 255. The predicates for IP_{Src} or IP_{Dst} are written using dotted-decimal notation $ip_1 : ip_2 : ip_3 : ip_4$, where ip_i is either a single decimal number or a range of values, denoted $[ip_{i,l}, ip_{i,h}]$. The wildcard '*' is used to express the range $[0, 255]$. If ip_i represents a range, the predicates following it must be '*', in order to correctly represent network IP addresses. For instance, $IP_{Src} = 127 : 100 : 9 : *$ stands for the predicate $x_1 = 127 \wedge x_2 = 100 \wedge x_3 = 9$, where x_i is the variable corresponding to the i -th byte of the source IP address packet field. This predicate matches all IP addresses within 130.192.5.0/24.

The predicates for source and destination ports $sPort$ and $dPort$ can specify a single integer value or a range in the bounds of 0 to 65535. The notation with the wildcard '*', used for each byte of an IP address is also used for port numbers, to indicate the range $[0, 65535]$. For example, "90" refers to the predicate $x = 90$, and "[600, 700]" refers to $x \leq 700 \wedge x \geq 600$, where x represents the port field.

The predicate t_{Prt} is used for the transport-layer protocol and it can identify a single value or a group of values from a finite set of possible protocols, in most of the case a set containing "TCP" and "UDP" as protocols. In this case, the wildcard '*' is used to represent all the protocols admitted.

The symbol t_0 represents the empty set of packets. In case $t_0 = \text{false}$, it indicates absence traffic. Let Q represent the set of all predicates $q_{t,i}$ that can be defined using the notation just described, and let T denote the set of all disjunctions of such predicates, which corresponds to the set of all packet classes t that can be expressed by this model. It can be shown that T is closed under conjunction, disjunction, and negation. For two traffic predicates $t_1, t_2 \in T$, t_1 is considered a sub-traffic of t_2 , denoted as $t_1 \subseteq t_2$, if t_1 represents a subset of the packets that t_2 represents, meaning $t_1 \Rightarrow t_2$.

2.6.2 Network Functions model

Each Virtual Network Function (VNF) in the Service Graph (SG) processes incoming traffic and generates corresponding output traffic. Its behavior is determined

by its configuration and it is abstractly represented using two functions. The first one is for the forwarding behavior, so it decides which input packets are dropped by the VNF and which are allowed through. The other one is for the transformation behavior, that states how packets are modified as output for each class of input traffic.

The forwarding behavior of a VNF at node $n_i \in N_A$ is modeled by the predicate $\text{deny}_i : T \rightarrow \mathbb{B}$, which returns true for incoming traffic $t \in T$ if node n_i drops all packets represented by t . The traffic I_d^i represents the set of packets that are dropped, meaning $\text{deny}_i(t)$ is true if and only if $t \subseteq I_d^i$. On the other hand, I_a^i represents the complement of I_d^i , indicating the packets that are allowed through the VNF (i.e., not dropped). Thus, $I_d^i \cup I_a^i = \text{true}$ and $I_d^i \cap I_a^i = \text{false}$. This definition will be expanded in the following chapters, where Stateful Firewalls will be discussed in greater detail.

The transformation behavior of the VNF at node $n_i \in N_A$ is captured by the function $T_i : T \rightarrow T$, called the transformer, which maps input traffic to the corresponding output traffic.

Even though the transformer function T_i alone may be sufficient, it is important to keep it separate from $\text{deny}_i(t)$, as the former represents the transformation behavior, while the latter deals with the forwarding behavior. For instance, $T_i(t) = t_0$ applies to all t where $\text{deny}_i(t)$ is true. However, a transformer handles traffic modifications independently of whether packets are dropped. For example, a firewall can be modeled as a VNF where $T_i(t) = t$, which is the identity function, as the firewall does not alter forwarded traffic, while the $\text{deny}_i(t)$ predicate indicates whether packets are dropped according to firewall rules. This separation enables traffic transformations to be calculated independently of firewall configurations, which are governed solely by the deny_i predicates.

For many VNFs, T_i is the identity function, and the following constraint applies to the deny predicate because they do not block packets:

$$\text{deny}_i(t) = \text{false}.$$

This model usually suits traffic monitoring functions, which examine and send on packets without alteration, as well as load balancers, which send on packets without modification but to destinations determined dynamically. Since the decision on packet forwarding is not predictable, a load balancer is shaped as a function that can potentially send any packet to any destination.

There are also VNFs that do not use an identity transformer. One example is NAT, which can perform two different transformations. The first transformation modifies the source IP address of outgoing packets, typically used to allow devices on a private network to access the public internet by replacing their internal private IP with the NAT device's public IP address. The second transformation modifies the destination IP address of incoming packets, which is useful when services inside the private network (such as a web server) need to be accessed externally. In other cases, the NAT leaves the packet unchanged.

2.6.3 Traffic Flow model

To define the transformation behavior of an Allocation Graph (AG) a set of traffic flows, denoted as F , is required. Each flow $f \in F$ stands for a class of packets created by a source endpoint $n_s \in NA$, directed towards a destination endpoint $n_d \in NA$, and routed through an ordered sequence of intermediate nodes $n_a, n_b, \dots \in NA$. These intermediate nodes send the packets at each hop, potentially altering them, as is done by a NAT, or dropping them. Formally, a flow is represented as a list $[n_s, t_{s,a}, n_a, t_{a,b}, n_b, \dots, n_k, t_{k,d}, n_d]$, where $t_{i,j}$ stands for the traffic sent from n_i to n_j . Each $t_{i,j}$ results from the alteration of the previous traffic in the flow by node n_i .

Additionally, each $t_{i,j}$ is homogeneous for node n_j , meaning all packets represented by $t_{i,j}$ are treated identically by n_j , either being fully dropped or fully forwarded. If node n_j enforces transformations, such as a NAT, all packets must belong to the same class.

To complement this definition, three additional functions feature AG flows:

1. $\pi : F \rightarrow (NA)^*$, which maps a flow to the ordered sequence of network nodes traversed by the flow, including the destination but excluding the source.
2. $\tau : F \times NA \rightarrow T$, which maps a flow and a node to the ingress traffic for that node belonging to the flow. If the flow f does not cross node n , then $\tau(f, n) = t_0$.
3. $\nu : NA \times F \rightarrow NA \cup \{n_0\}$, which maps a node n and a flow f to the next node in the flow after n . If n is not in f or is the last node, the function returns n_0 , meaning no node.

If two flows f_1 and f_2 traverse the same list of nodes and, for each of those nodes, the ingress traffic of f_1 is a subset of the ingress traffic of f_2 , then f_1 is called a subflow of f_2 , denoted as $f_1 \subseteq f_2$.

2.6.4 Service and Allocation Graph models

A Service Graph (SG) is represented as a directed graph, denoted by $G_S = (N_S, L_S)$, where N_S is the set of vertices symbolizing the network nodes in the SG, and L_S is the set of edges that depict the directed connections between these nodes. Specifically, the vertex set N_S consists of two disjoint subsets: E_S and S_S . The subset E_S includes end points, which can correspond to terminals, physical servers, or sub-networks within the underlying infrastructure where virtual instances of functions are deployed. On the other hand, S_S consists of service functions—basic Network Functions (NFs) that do not provide security against cyberattacks but are used to establish end-to-end services.

Each element in N_S is uniquely identified by a non-negative integer index k . Let n_k indicate the element of N_S corresponding to index k . In turn, each edge in L_S is represented by a pair of non-negative integers $l_{i,j} \in L_S$, where $i \neq j$, indicating a directed connection from node n_i to node n_j . We define $\text{index}(n_k) = k$.

Additionally, each node $n_k \in N_S$ is associated with either a single IP address, an IP address range, or, more generally, a set of IP addresses. Let I be the set of all IP addresses, and $\alpha : N_S \rightarrow 2^I$ be a function that assigns each node $n \in N_S$ a set of IP addresses.

An Allocation Graph (AG) is also a directed graph, denoted as $G_A = (N_A, L_A)$, where N_A is the set of vertices representing network nodes, and L_A is the set of links interconnecting them. The primary distinction between the AG and the Service Graph (SG) lies in the composition of the vertex set. Particularly, in the AG, the set N_A is divided into three disjoint subsets: $N_A = E_A \cup S_A \cup A_A$. Here, E_A and S_A represent endpoints and middleboxes, respectively, while A_A represents the set of Allocation Places (APs) where firewall instances may be deployed.

When the AG is automatically generated from the SG, it incorporates additional requirements regarding firewall placement provided by the service designer. In this process, the endpoints and service functions remain unchanged, so $E_A = E_S$ and $S_A = S_S$. Since the IP address assignments also remain the same from the SG to the AG, the mapping function α is extended to apply to N_A without any modifications.

The process of allocating firewalls in an AG is based on Boolean logic and specific constraints derived from the SG. The AG is modeled by adding APs between network nodes when necessary. The decision to allocate a firewall at a particular node is governed by a predicate $\text{allocated}(n)$, which indicates whether the node n has been assigned a firewall.

In this model, two Boolean predicates are critical: $\text{forbidden}(l_{i,j})$ and $\text{forced}(l_{i,j})$, which apply to each edge $l_{i,j}$ between nodes n_i and n_j . These predicates specify whether the creation of an AP on that edge is prohibited or required, respectively. The key constraint ensures that both predicates cannot be true simultaneously for the same edge:

$$\forall l_{i,j} \in L_S. \neg(\text{forbidden}(l_{i,j}) \wedge \text{forced}(l_{i,j}))$$

This means that an edge cannot both prohibit and require the allocation of a firewall.

The allocation process follows two rules:

1. 1. If an AP is not prohibited on an edge (i.e., $\text{forbidden}(l_{i,j}) = \text{false}$), an AP a_h is created between nodes n_i and n_j , replacing the original edge with two new edges. This adds the AP to the AG:

$$\forall l_{i,j} \in L_S. (\neg \text{forbidden}(l_{i,j})) \implies (a_h \in A_A \wedge l_{i,h} \in L_A \wedge l_{h,j} \in L_A)$$

2. 2. If an AP is prohibited on an edge (i.e., $\text{forbidden}(l_{i,j}) = \text{true}$), the edge remains unchanged in the AG:

$$\forall l_{i,j} \in L_S. (\text{forbidden}(l_{i,j})) \implies (l_{i,j} \in L_A)$$

Finally, if a firewall is explicitly required on an edge (i.e., $\text{forced}(l_{i,j}) = \text{true}$), the predicate ensures that the AP a_h created on that edge must have a firewall allocated:

$$\forall l_{i,j} \in L_S. (\text{forced}(l_{i,j})) \implies \text{allocated}(a_h)$$

These rules provide a structured approach to determining where firewalls are placed in the network, ensuring that the allocation meets the service designer's requirements while respecting the constraints on AP creation.

2.6.5 Network Security Requirements model

The Network Security Requirements (NSRs) for a Service Graph (SG) involve a default behavior D , which can be one of the already explained in section 2.3.2. Additionally, there is a set of specific NSRs, denoted as R_s . Each element $r \in R_s$ is expressed in a medium-level language as a pair $r = (C, a)$, where C is the condition and a is the action to be applied to the flows that meet the condition.

The condition C is a predicate similar to those used to define packet classes, written as $C = (IPSrc, IPDst, pSrc, pDst, tPrt)$. Here, $IPSrc$ and $pSrc$ define the traffic source, while $IPDst$, $pDst$, and $tPrt$ specify the destination and protocol involved in the requirement. A flow $f = [e_s, t_{s,a}, \dots, t_{k,d}, e_d]$ fulfills C if the following conditions are met:

1. The source and destination endpoints e_s and e_d have IP addresses that match $IPSrc$ and $IPDst$, ($a(e_s) \subseteq C.IPSrc$ and $a(e_d) \subseteq C.IPDst$).
2. The source traffic matches $IPSrc$ and $pSrc$, ($t_{s,a} \subseteq (C.IPSrc, *, C.pSrc, *, *)$).
3. The destination traffic matches $IPDst$, $pDst$, and $tPrt$, ($t_{k,d} \subseteq (*, C.IPDst, *, C.pDst, C.tPrt)$).

Let $F_r \subseteq F$ represent the set of flows that fulfill $r.C$. Consequently, all subflows of any flow in F_r are also in F_r .

The action a belongs to the set $A_T = \{\text{DENY}, \text{ALLOW}\}$. If $r.a = \text{DENY}$, the requirement is called an isolation requirement, meaning that all flows fulfilling $r.C$ are dropped, so the packets does not reach their destination. If $r.a = \text{ALLOW}$, it is a reachability requirement, meaning at least one flow fulfilling $r.C$ is allowed to reach the destination.

An isolation requirement r can be formally written as:

$$\forall f \in F_r. \exists i. (n_i \in p(f) \wedge \text{allocated}(n_i) \wedge \text{deny}_i(t(f, n_i)))$$

A reachability requirement r can be formally written as:

$$\exists f \in F_r. \forall i. (n_i \in p(f) \wedge \text{allocated}(n_i) \implies \neg \text{deny}_i(t(f, n_i)))$$

As will be discussed in Chapter 4, the definitions of isolation and reachability requirements currently used by VEREFOO will be reused and extended in the new formulations for stateful firewalls.

The requirements formulated here are passed as hard constraints in the MaxSMT problem used to configure the firewall, meaning they must always be respected.

The set R_D is defined to constitute the default behavior when D is either blacklisting or whitelisting. For every correct combination of 5-tuple elements not covered by any requirement in R_s , there is an element in R_D , where $r.C$ matches that combination and $r.a = \text{ALLOW}$ if $D = \text{blacklisting}$, or $r.a = \text{DENY}$ if $D = \text{whitelisting}$. The complete set of requirements is given by $R = R_s \cup R_D$.

2.6.6 Firewall configuration

To correctly configure the firewalls, the MaxSMT problem plays a crucial role. Specifically, both hard constraints, as presented in the previous section, and soft constraints must be defined to properly structure the problem.

Soft constraints are essential for determining the optimal allocation and configuration of firewalls. These constraints incorporate free variables that allow the solver to explore different possibilities. The soft clauses are designed with two primary optimization objectives in mind: minimizing the number of allocated firewalls and minimizing the number of rules assigned to each firewall. Since the free variables are shared across all clauses, the solution that satisfies both goals simultaneously is considered optimal. However, weights are assigned to prioritize minimizing firewalls since is more relevant from a prestational point of view.

To achieve the first objective, since a firewall can be allocated at any $a_h \in A_A$, a set of soft clauses is introduced to express a preference for not allocating firewalls at each a_h . This is represented by the following expression, where $Soft(f, c)$ denotes a soft clause with formula f and weight c :

$$\forall a_h \in A_A. Soft(allocated(a_h) = false, c_h)$$

This structure helps the solver prioritize solutions that reduce the number of firewalls while ensuring the overall optimization objectives are met.

To achieve the second optimization goal (minimizing the number of rules for each firewall), each firewall placed at $a_h \in A_A$ is characterized by a default action, d_h , and a set of specific rules, U_h . Each rule $u \in U_h$ is defined by a condition C , which includes parameters such as source/destination IP addresses, ports, and protocol types, and an action a , which can either be DENY or ALLOW. This is similar to how Network Service Requirements (NSRs) are defined, though the relationship between NSRs and firewall rules is not one-to-one; a single firewall rule can address multiple NSRs, and a single NSR might require multiple firewall rules across different nodes.

Given the potentially large number of firewall rules, it is important to limit the number of soft clauses required to minimize the rules, ensuring a balance between scalability and accuracy. To achieve this, the default action d_h for each firewall is determined before solving the MaxSMT problem, aiming to minimize the number of rules needed to satisfy the NSRs.

The default action is chosen based on the NSRs that impact the AP a_h . If more reachability requirements (which allow traffic) pass through a_h than isolation requirements (which block traffic), the default action d_h is set to ALLOW. Otherwise, it is set to DENY. This pre-selection reduces the need for specific rules, thus minimizing the total number of rules required.

This approach is formalized by the predicate $wlst : A_A \rightarrow \mathbb{B}$, which is true if the default action of a firewall at a_h is DENY. Additionally, the predicate $enforces : A_T \times R \rightarrow \mathbb{B}$ indicates whether the default action d_h satisfies a given requirement r , i.e., if $d_h = r.a$. This method ensures an efficient firewall configuration by reducing unnecessary rules and focusing on optimal default actions.

To optimize firewall configuration, it is necessary to determine the set of rules that might be useful for each firewall allocated at an allocation place (AP) $a_h \in A_A$. These rules are referred to as placeholder rules and form the set P_h , where each rule $p_i = (C, a)$ is similar to the real firewall rules in U_h . A placeholder rule p_i is included in P_h if it might be needed to satisfy security requirements for the flows passing through a_h and if the default action d_h assigned to the firewall is insufficient to meet those requirements.

The set Q_h represents all the possible packet (5-tuples) classes for which specific rules might be needed in the firewall located at a_h . A 5-tuple is included in Q_h if two conditions are met:

1. The traffic flow f passes through a_h ($a_h \in p(f)$).
2. The default action d_h of the firewall in a_h does not satisfy the security requirement r ($enforces(d_h, r)$ is false).

Thus, Q_h is the smallest set of 5-tuples that satisfy these conditions:

$$\forall r \in R, \forall f \in F_r^M. (a_h \in p(f) \wedge \neg enforces(d_h, r)) \Rightarrow (\forall q \in t(f, a_h). q \in Q_h)$$

This ensures that Q_h contains only the packet classes for which specific firewall rules are necessary. If a packet $q \in Q_h$ meets these conditions, a corresponding placeholder rule p_i is defined in P_h with an action opposite to d_h . The conditions for each p_i are expressed using free variables, whose values are not fixed in advance but are automatically determined by the solver. These values are calculated in accordance with the hard constraints defined in the MaxSMT problem, ensuring that the solution respects the given constraints while optimizing the firewall configuration.

Placeholder rules will only be included in U_h if the solver determines they are necessary to achieve the optimal solution. This decision is represented by the predicate $configured : P_h \rightarrow B$, which is true if a placeholder rule is configured in the firewall. To minimize the number of rules, a soft constraint is introduced, favoring a solution where as few rules as possible are configured, possibly no one:

$$\forall p_i \in P_h. Soft(\neg configured(p_i), c_{h,i})$$

If at least one placeholder rule $p_i \in P_h$ is configured, a firewall instance must be placed at a_h , as indicated by the following hard constraint:

$$(\exists p_i \in P_h. configured(p_i)) \Rightarrow allocated(a_h)$$

Finally, since the primary goal is to minimize the number of firewalls allocated, the weight of the soft constraint for firewall allocation $allocated(a_h)$ must be greater than the sum of the weights associated with all placeholder rules:

$$\sum_{i:p_i \in P_h} c_{h,i} < c_h$$

A second category of soft clauses is introduced to emphasize the preference for using wildcards in each component of filtering rules. Wildcards are beneficial not only for minimizing the number of placeholder rules during the pre-processing phase but also for reducing the number of rules in the solution of the MaxSMT problem. The following statements specify the use of wildcards for each of the four components of IP addresses in dotted quad notation and also for transport-level ports and protocols:

$$\forall i, \forall j, \pi_i \in \Pi_k : \forall j \in \{1, 2, 3, 4\}. \text{Soft}(\pi_i : \text{IPSrc}_j = *; c_{kij1})$$

$$\forall i, \forall j, \pi_i \in \Pi_k : \forall j \in \{1, 2, 3, 4\}. \text{Soft}(\pi_i : \text{IPDst}_j = *; c_{kij2})$$

$$\forall p_i \in P_k. \text{Soft}(p_i.\text{pSrc} = [0, 65535], c_{ki3})$$

$$\forall p_i \in P_k. \text{Soft}(p_i.\text{pDst} = [0, 65535], c_{ki4})$$

$$\forall p_i \in P_k. \text{Soft}(p_i.\text{tProto} = *, c_{ki5})$$

The weight given to the configuration of the placeholder rule must exceed the total weight of the wildcard-based rules. If the placeholder rule is unnecessary and thus not configured, there is no point in attempting to apply wildcards for that rule. Therefore, the following constraint must be satisfied for every placeholder rule:

$$\sum_{j=1}^4 (c_{kij1} + c_{kij2}) < c_{ki}$$

The generated set of statements is subsequently handled by the MaxSMT solver. If partial satisfiability is attained, the solver returns the optimal firewall allocation and configuration. However, if partial satisfiability is not achieved, a non-enforceability report is produced. This report may highlight a lack of APs caused by additional user-defined constraints that hinder their creation. The insights from the report can guide adjustments for the next iteration of the tool, once the inputs have been updated accordingly.

To finalize the configuration of each firewall's filtering policy (FP), additional hard clauses are necessary beyond those previously introduced. These clauses ensure that the firewall's configuration is consistent with its allocation and traffic policies. For each access point a_h , the potential input traffic set T_h must be considered. For each traffic $t \in T_h$, two hard constraints are defined: one for when the firewall must drop t and another for when it must allow t .

To formulate these constraints, the predicates $\text{matchAll} : P_h \times Q \rightarrow B$ and $\text{matchNone} : P_h \times Q \rightarrow B$ are introduced. Given a placeholder rule $p_i \in P_h$ and a 5-tuple $q \in Q$:

- The matchAll predicate returns true if the rule conditions completely include the values of the 5-tuple fields, i.e., if $\text{matchAll}(p_i, q) \iff q \subseteq p_i : C$.

- The `matchNone` predicate returns true if the packet classes expressed by the rule conditions and by the 5-tuple fields are disjoint, i.e., if $matchNone(p_i, q) \iff \neg(q \cap p_i : C)$.

Then, for each $t \in T_h$, the two hard clauses are defined as follows:

1. When the firewall must drop traffic t :

$$allocated(a_h) \wedge deny_h(t) \Rightarrow (a) \vee (b)$$

where:

$$(a) = wlst(a_h) \wedge \forall q \in t. (\forall p_i \in P_h. \neg configured(p_i) \wedge matchNone(p_i, q))$$

$$(b) = \neg wlst(a_h) \wedge \exists p_i \in P_h. (configured(p_i) \wedge matchAll(p_i, q))$$

In this formula, the right side states that either condition (a) or condition (b) must hold. Condition (a) requires the firewall to be in whitelisting mode and for all 5-tuple fields of traffic t to match none of the configured placeholder rules. Condition (b) states that if the firewall is in blacklisting mode, at least one configured rule must match all fields of the traffic t , explicitly blocking it.

2. When the firewall must allow traffic t :

$$allocated(a_h) \wedge \neg deny_h(t) \Rightarrow (a) \vee (b)$$

where:

$$(a) = wlst(a_h) \wedge \exists p_i \in P_h. (configured(p_i) \wedge matchAll(p_i, q))$$

$$(b) = \neg wlst(a_h) \wedge \forall q \in t. (\forall p_i \in P_h. \neg configured(p_i) \wedge matchNone(p_i, q))$$

Here, Condition (a) states that if the firewall is in whitelisting mode, there must be at least one configured rule that matches all fields of traffic t . Condition (b) indicates that if the firewall is in blacklisting mode, none of the configured rules can block the traffic t , allowing it through.

These formulas ensure that the firewall behaves correctly according to its configuration regarding whether to drop or allow specific traffic flows.

Chapter 3

Thesis objective

Security automation involves the use of software and technology to manage security tasks with minimal human intervention, enhancing both efficiency and response times in dealing with cyber threats. It plays a critical role in modern cybersecurity by automating repetitive or complex tasks.

By automating many of the repetitive and time-consuming tasks in cybersecurity, organizations reduce the risk of human error and can respond to threats faster, often within milliseconds. This reduces the potential for damage and data loss, while also allowing security teams to focus on more critical or strategic initiatives. Automating policy enforcement, warning control, prioritization, and incident management will significantly enhance business efficiency and reduce costs [16].

It improves threat detection, speeds up response times, simplifies compliance, and reduces the burden on security teams, allowing for a more scalable and consistent security posture across the board. However, implementing and managing automation requires careful planning to ensure it complements human oversight without over-reliance on automated processes.

This problem has been addressed in the VEREFOO framework, which aims to enhance security automation within network systems. In particular, this thesis focuses on expanding the functionalities of VEREFOO, with a specific emphasis on stateful firewalls, a feature that had not been explored in detail prior to this work.

A stateful firewall is a critical component of modern network security due to its ability to monitor and manage the state of active connections, making it far more effective than stateless firewalls in protecting networks. Unlike stateless firewalls, which make decisions based solely on static rules such as source/destination IP addresses or port numbers, stateful firewalls track the entire context of a connection, including its state (whether it's established, related, or new).

These firewalls enable dynamic packet filtering, reducing the load on network resources and improving efficiency. Once a legitimate connection is established, stateful firewalls allow traffic to flow without constantly checking the rules for each individual packet, saving time and resources.

From a security standpoint, stateful firewalls are more effective at recognizing abnormal behavior and blocking unauthorized access attempts or malicious traffic, making them highly effective in defending against Denial of Service (DoS) attacks.

By monitoring the entire session, they can prevent complex attacks like TCP session hijacking or man-in-the-middle attacks. Additionally, stateful firewalls are essential in preventing spoofing and replay attacks, as they track the sequence and timing of packets.

The objectives of this thesis are to present a method for implementing security automation, with a specific focus on stateful firewalls, in order to highlight the importance of integrating stateful firewalls into modern systems to enhance security and accelerate processes.

This work aims to extend the VEREFOO platform with new functionalities, following an initial phase dedicated to verifying the correctness of the existing features. Specifically, this phase involves conducting tests to demonstrate the accurate functioning of the Verification Problem.

In the subsequent phase, a logical model has been proposed to represent Stateful Firewalls within the framework. Finally, a translator has been implemented to adapt network configurations specified as input into configuration files for various platforms, including Iptables, Open vSwitch, and IpFirewall.

Chapter 4

Verification test and Refinement logical model

This chapter presents the effective contribution of this thesis work. The central element of this work is represented by stateful firewalls, and all the next works are incentred on this topic. Furthermore all the completed tasks aim to extend the functionalities of the VEREFOO framework, described in the previous chapter.

As first thing, verification tests were done, in order to show the correctness of the verification part of the framework. Thus, a group of test case have been written and executed to show if the presented network configuration with the contained policy matches the Network Security Requirement given in input.

The second part of this chapter presents a formal model, proposed to define an auto-configuration algorithm for the refinement problem with the presence of stateful firewalls. In particular the model has been defined through the use of boolean logic formulas.

4.1 Stateful Verification test

In this part of the work, a group of test case has been submitted to the algorithm, to show its correctness in the verification process. In particular, this process differs from the one of auto-configuration in which the position of the firewalls and the policy to insert inside the firewalls are decided by the algorithm. In this case, instead, the firewalls are already allocated and they contains the filtering policy written.

The scope of the verification part is to check if the Network Security Policy (NSPs) given as input are compatible with the structure of the network presented. In doing so, VEREFOO formulates this issue as a MaxSMT problem managed by z3, whose functions are exploited in the Java code of the algorithm. The framework takes the specification in input through an XML file while the output is showed in the console. In particular, the framework returns SAT if the NSPs can be satisfied by the presented configuration and UNSAT if they don't meet the network disposition.

The XML file contains the graph describing the network composition. Every element is inserted as a *node* element and it can be of many functional types: Stateful Firewall, Load Balancer, NAT, WebClient, WebServer and others. An IP address is associated to every node, used also to express the neighbourhood with other nodes.

In 4.1 is showed an example of an XML file describing a network graph with its elements inside.

```

<graphs>
  <graph id="0">
    <node functional_type="WEBCIENT" name="10.0.0.1">
      <neighbour name="30.0.0.2"/>
      <configuration description="A simple description" name="confA">
        <webclient nameWebServer="20.0.0.1"/>
      </configuration>
    </node>
    <node functional_type="WEBCIENT" name="10.0.0.2">
      <neighbour name="30.0.0.2"/>
      <configuration description="A simple description" name="confB">
        <webclient nameWebServer="20.0.0.1"/>
      </configuration>
    </node>
    <node functional_type="TRAFFIC_MONITOR" name="30.0.0.2">
      <neighbour name="10.0.0.1" />
      <neighbour name="10.0.0.2" />
      <neighbour name="30.0.0.1" />
    </node>
    <node functional_type="STATEFUL_FIREWALL" name="30.0.0.1">
      <neighbour name="30.0.0.2"/>
      <neighbour name="20.0.0.1" />
      <configuration description="A simple description" name="conf1">
        <stateful_firewall defaultAction="DENY">
          </stateful_firewall>
        </configuration>
      </node>
    <node functional_type="WEBSERVER" name="20.0.0.1">
      <neighbour name="30.0.0.1"/>
      <configuration description="A simple description" name="confB">
        <webserver>
          <name>b</name>
        </webserver>
      </configuration>
    </node>
  </graph>
</graphs>

```

Listing 4.1. Example of an XML network graph

The relevant type is the Stateful Firewall, in which are inserted the rules used to block or allow the connections. In the firewall is always expressed a default action, used when there aren't specific rules that match the packets that arrives to the firewall. In VEREFOO, three types of actions are possible, as specified in Chapter 2: blacklisting, whitelisting and specific, but, in this framework, for stateful firewalls only the whitelisting mode can be used. As a consequence, the

default action applied is always *DENY*.

A substantial difference between stateless and stateful firewalls in VEREFOO is the type of actions that they support. Indeed, in a stateless firewall only two actions are permitted:

- *ALLOW*, used to consent the passage of a specific packet or class of packets.
- *DENY*, used to block the passage of a specific packet or class of packets.

Instead, a stateful firewall offers these actions and a new one, that is *ALLOW_COND*. This action is the one that permits to work with a state because it consents or blocks the traffic conditionally. In particular, given a source *A* and a destination *B* on which a traffic *t* is built on, a rule of *ALLOW_COND* applied on *t* means that

- the traffic from the source *A* to the destination *B* is always permitted.
- the traffic from the destination *B* to the source *A* is permitted conditionally.

Therefore, this action implies two operations at the same time. Particularly, the *ALLOW_COND* action corresponds to the *ALLOW* for the first part, so if a stateless rule is needed, both *ALLOW_COND* and *ALLOW* can be used, but remembering that the first option enables a return connection.

To model the concept of time, that should be used for a stateful connection, the command *ALLOW_COND* includes the first connection (from *A* to *B*) that is always permitted and the second (from *B* to *A*) that is permitted as a response to the first one. This means that, in temporal order, the latter must come after the former. Otherwise, in case the first request would come from *B* to *A*, the connection would be denied by the firewall because it's not a response to *A*. So, the construction of this command permits to introduce the concept of time for stateful firewalls.

What is explained above is reported also in the XML files used for testing. In particular, a rule is inserted in the firewall as a *element*. Each *element* contains the following:

- *action*, that can be *ALLOW* or *ALLOW_COND*
- *source*, the source IP address. It can be a single IP address or a subnet expressed with the use of -1.
- *destination*, the destination IP address. It can be a single IP address or a subnet expressed with the use of -1.
- *protocol*, that is the protocol used. It can be TCP, UDP or ANY that means both.
- *src_port*, the source port. It can be a single port number or an interval. The wildcard '*' means all the port numbers.

- *dst_port*, the destination port. It can be a single port number or an interval. The wildcard '*' means all the port numbers.

In Listing 4.2 is reported a stateful firewall containing two rules. The first contains an *ALLOW_COND* rule with 10.0.0.1 as source IP address and 20.3.0.0/16 as destination IP address of the subnet. The second is an *ALLOW* rule having 10.0.0.2 as source IP address and 20.6.2.0/24 as destination IP address of the subnet.

```

<node functional_type="STATEFUL_FIREWALL" name="30.0.0.1">
  <neighbour name="30.0.0.2"/>
  <neighbour name="20.0.0.1" />
  <configuration description="A simple description" name="conf1">
    <stateful_firewall defaultAction="DENY">
      <elements>
        <action>ALLOW_COND</action>
        <source>10.0.0.1</source>
        <destination>20.3.-1.-1</destination>
        <protocol>TCP</protocol>
        <src_port>10</src_port>
        <dst_port>*</dst_port>
      </elements>
      <elements>
        <action>ALLOW</action>
        <source>10.0.0.2</source>
        <destination>20.6.2.-1</destination>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>300-400</dst_port>
      </elements>
    </stateful_firewall>
  </configuration>
</node>

```

Listing 4.2. Example of an XML stateful firewall containing two rules

4.1.1 Verification Policy

In the context of firewalls, a Network Security Policy (NSP) outlines the rules and protocols that govern how a firewall controls inbound and outbound traffic. As the primary defense in network security, firewalls implement these policies to either allow or deny data packets, aligning with the organization's security goals. A robust firewall policy plays a crucial role in safeguarding the network from unauthorized access, malicious traffic, and potential security breaches.

Unlike the stateless case, where only two types of NSPs are considered, the Verification Problem with Stateful Firewalls takes into account four types of Network Security Policies (NSPs):

- Isolation
- Simple Reachability

- Strong Reachability
- Conditional Reachability

Each of them has a formal definition in First-Order Logic. To introduce this notation, we need to describe some predicates and functions, as they are used to define our policies.

- $allocated(n) : N \rightarrow \mathbb{B}$. Predicate that returns true if a firewall is allocated in the allocation node $n \in N$.
- $deny(t) : T \rightarrow \mathbb{B}$. Predicate that returns true for ingress traffic $t \in T$ if the node drops all packets belonging to t while it returns false if there is the possibility that it allows the ingress traffic t .
- $cond_permit(t) : T \rightarrow \mathbb{B}$. Predicate that returns true if the possibility that $t \in T$ is allowed by $n \in N$ is conditioned by the existence of a state while it returns false if either the traffic t is blocked or it may be allowed independently from the existence of a state.
- $\pi(f) : F \rightarrow (N)^*$. Function that maps a flow $f \in F$ to the ordered list of network nodes that are crossed by that flow, including the destination, but not the source.
- $\tau(f, n) : F \times N \rightarrow T$. Function which maps a flow $f \in F$ and a node $n \in N$ to the ingress traffic of that node belonging to that flow. In case flow f does not cross node n , the function returns t_0 , that is the empty set.

Isolation Policy

$$\forall f \in Fr. \exists i. (n_i \in \pi(f) \wedge allocated(n_i) \wedge deny_i(\tau(f, n_i)))$$

For all the possible flows satisfying the requirement conditions, it must exist at least a function allocated in the flow path that blocks the flow traffic it receives.

This definition is independent from the presence of stateful functions.

Simple Reachability Policy

$$\exists f \in Fr. \forall i. (n_i \in \pi(f) \wedge allocated(n_i) \rightarrow \neg deny_i(\tau(f, n_i)))$$

There exists at least a flow satisfying the requirement conditions where, for all flow path nodes, if a function is there allocated, there is the possibility that it allows the flow traffic it receives.

This definition is independent from the presence of stateful functions.

Strong Reachability Policy

$$\begin{aligned} \exists f \in Fr. \forall i. (n_i \in \pi(f) \wedge \text{allocated}(n_i) \rightarrow \neg \text{deny}_i(\tau(f, n_i)) \\ \wedge \neg \text{cond_permit}_i(\tau(f, n_i))) \end{aligned}$$

It must exist a possible flow satisfying the requirement conditions, such that, for all the nodes of the flow path, if there is a function allocated, then it allows the traffic (independently from the state):

1. If the node has a stateless function allocated, then it allows the flow traffic;
2. If the node has a stateful function allocated, then it allows the flow traffic independently from the state.

Conditional Reachability Policy

$$\exists f \in Fr ((1) \wedge (2) \wedge (\forall f \in Fr ((1) \wedge (2) \vee (3))))$$

$$(1) = \forall i (n_i \in \pi(f) \wedge \text{allocated}(n_i) \rightarrow \neg \text{deny}_i(\tau(f, n_i)))$$

$$(2) = \exists i (n_i \in \pi(f) \wedge \text{allocated}(n_i) \wedge \text{cond_permit}_i(\tau(f, n_i)))$$

$$(3) = \exists i (n_i \in \pi(f) \wedge \text{allocated}(n_i) \wedge \text{deny}_i(\tau(f, n_i)))$$

For all the possible flows satisfying the requirement conditions, two conditions must be true:

1. For all the nodes of the flow path, if there is a function allocated, then there is the possibility that it allows the traffic ($\neg \text{deny}_i(\tau(f, n_i))$);
2. There exists at least a node of the path which allows that traffic only if there is a state ($\text{cond_permit}_i(\tau(f, n_i))$).

The deny_i and cond_permit_i predicates have different meanings:

- $\text{deny}_i(p)$ is true if n_i blocks p , while it is false if there is the possibility that it allows p ;
- $\text{cond_permit}_i(p)$ is true if the possibility that p is allowed by n_i is conditioned by the existence of a state, while it is false if either the traffic is blocked or it may be allowed independently from the existence of a state.

To better understand how these types of policies work, the following examples are provided, illustrating whether the presented policies are satisfied or not, along with the reasons behind it.

In the first example, there are two nodes separated by a stateful firewall. In the second configuration, the setup is similar, but a third node is introduced, connecting the first two.

The stateful firewall f_i allows traffic from e_1 to e_2 , and it saves the state (so, there is the possibility that it allows the return traffic from e_2 to e_1) and it has "deny" as the default action.

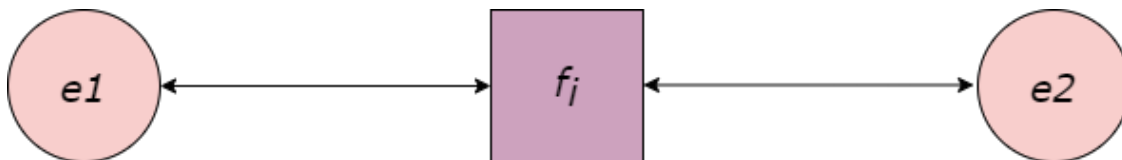


Figure 4.1. Example 1

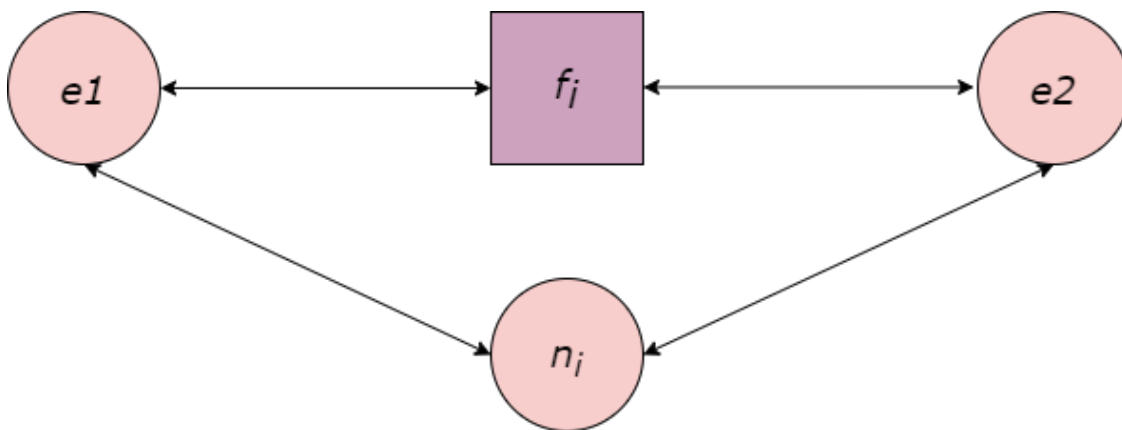


Figure 4.2. Example 2

- The isolation policy from e_2 to e_1 is NOT satisfied in any example.
 - In Example 1, there is the possibility that the firewall f_i allows the traffic.
 - In Example 2, there is a path through n_j where the traffic can always flow.
- The simple reachability policy from e_2 to e_1 is satisfied in BOTH examples.
 - In Example 1, there is the possibility that the firewall f_i allows the traffic.
 - In Example 2, there is a path through n_j where the traffic can always flow.
- The strong reachability policy from e_2 to e_1 is satisfied ONLY in Example 2.
 - In Example 1, there is the possibility that the firewall f_i blocks the traffic.

- In Example 2, there is a path through n_j where the traffic can always flow.
- The conditional reachability policy from e_2 to e_1 is satisfied ONLY in Example 1.
 - In Example 1, there is the stateful firewall f_i that allows the traffic only conditionally.
 - In Example 2, there is a path through n_j where the traffic can always flow.

The described policies are included in the XML input file for the Verification process. Specifically, they are utilized within the *Property* element to define the Network Security Requirements that the network configurations must satisfy.

The XML *Property* attribute refers to the input graph by using its ID and specifies the source and destination where the policy is applied through IP addresses. In Listing 4.3, an example is provided that demonstrates a list of properties defined for various network elements within the same network schema.

```

<PropertyDefinition>
  <Property graph="0" name="StrongReachabilityProperty" src="10.0.0.1"
    dst="20.0.0.1"/>
  <Property graph="0" name="ConditionalReachabilityProperty" src="20.0.0.1"
    dst="10.0.0.1"/>
  <Property graph="0" name="ReachabilityProperty" src="20.0.0.1"
    dst="10.0.0.1"/>
  <Property graph="0" name="StrongReachabilityProperty" src="10.0.0.2"
    dst="20.0.0.1"/>
  <Property graph="0" name="IsolationProperty" src="20.0.0.1"
    dst="10.0.0.2"/>
</PropertyDefinition>

```

Listing 4.3. Example of an XML list of Property

Right away, a complete example of a Verification Test is provided (Listing 4.4). In this test case, three WebClients interact with two WebServers. The clients are hidden behind a NAT that communicates with the public network. Two stateful firewalls are present: the first one between the clients and the NAT, with IP address 20.0.0.3, and the second one between the NAT and the servers, with IP address 20.0.0.2.

In the first firewall, rules are defined for each client, specifying which server they are allowed to connect to. Specifically, clients 10.0.0.1 and 10.0.0.2 can connect to server 30.0.5.2, while client 10.0.0.3 can connect to server 30.0.5.3. The second firewall contains the rules concerning the NAT, with IP address 20.0.0.1, and the servers on the network.

At the end of the XML file, the properties that we want to ensure in this network are defined. To determine whether these properties are satisfied, we need to compare them with the firewall rules and verify their compatibility with the NSRs.

After running the Java code with the XML file as input, the program produces the following output:

```
17:53:48.739 [main] INFO result - SAT
```

This indicates that the problem is satisfied (i.e., the properties conform to the firewall rules). The rest of the output, not shown here, provides additional information regarding the computation process. The properties are satisfied because

- The connection between the WebClient 10.0.0.1 and the WebServer 30.0.5.2 respects the *StrongReachabilityProperty*, as the first firewall contains a rule of ALLOW_COND from 10.0.0.1 to 30.0.5.2 and the second contains a rule of ALLOW_COND from 20.0.0.1 to 30.0.5.2.
- The connection between the WebServer 30.0.5.2 and the WebClient 10.0.0.1 respects the *ConditionalReachabilityProperty*, as the first firewall contains a rule of ALLOW_COND from 10.0.0.1 to 30.0.5.2 and the second contains a rule of ALLOW_COND from 20.0.0.1 to 30.0.5.2.
- The connection between the WebClient 10.0.0.2 and the WebServer 30.0.5.2 respects the *StrongReachabilityProperty*, as the first firewall contains a rule of ALLOW from 10.0.0.2 to 30.0.5.2 and the second contains a rule of ALLOW_COND from 20.0.0.1 to 30.0.5.2.
- The connection between the WebServer 30.0.5.2 and the WebClient 10.0.0.2 respects the *IsolationProperty*, as the first firewall contains a rule of ALLOW from 10.0.0.2 to 30.0.5.2 and the second contains a rule of ALLOW_COND from 20.0.0.1 to 30.0.5.2.
- The connection between the WebClient 10.0.0.3 and the WebServer 30.0.5.3 respects the *ReachabilityProperty*, as the first firewall contains a rule of ALLOW_COND from 10.0.0.3 to 30.0.5.3 and the second contains a rule of ALLOW_COND from 20.0.0.1 to 30.0.5.3.
- The connection between the WebServer 30.0.5.3 and the WebClient 10.0.0.3 respects the *ConditionalReachabilityProperty*, as the first firewall contains a rule of ALLOW_COND from 10.0.0.3 to 30.0.5.3 and the second contains a rule of ALLOW_COND from 20.0.0.1 to 30.0.5.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <node functional_type="WEBCLIENT" name="10.0.0.1">
        <neighbour name="20.0.0.3" />
        <configuration description="A simple description"
          name="confA">
          <webclient nameWebServer="30.0.5.2" />
        </configuration>
      </node>
```

```

<node functional_type="WEBCLIENT" name="10.0.0.2">
  <neighbour name="20.0.0.3" />
  <configuration description="A simple description"
    name="confA">
    <webclient nameWebServer="30.0.5.2" />
  </configuration>
</node>
<node functional_type="WEBCLIENT" name="10.0.0.3">
  <neighbour name="20.0.0.3" />
  <configuration description="A simple description"
    name="confA">
    <webclient nameWebServer="30.0.5.3" />
  </configuration>
</node>

<node functional_type="STATEFUL_FIREWALL" name="20.0.0.3">
  <neighbour name="10.0.0.1" />
  <neighbour name="10.0.0.2" />
  <neighbour name="10.0.0.3" />
  <neighbour name="20.0.0.1" />
  <configuration description="A simple description" name="conf1">
    <stateful_firewall defaultAction="DENY">
      <elements>
        <action>ALLOW_COND</action>
        <source>10.0.0.1</source>
        <destination>30.0.5.2</destination>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </elements>
      <elements>
        <action>ALLOW</action>
        <source>10.0.0.2</source>
        <destination>30.0.5.2</destination>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </elements>
      <elements>
        <action>ALLOW_COND</action>
        <source>10.0.0.3</source>
        <destination>30.0.5.3</destination>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </elements>
    </stateful_firewall>
  </configuration>
</node>
<node functional_type="NAT" name="20.0.0.1">
  <neighbour name="20.0.0.3" />
  <neighbour name="20.0.0.2" />
  <configuration description="A simple description" name="conf2">
    <nat>
      <source>10.0.0.1</source>
      <source>10.0.0.2</source>
      <source>10.0.0.3</source>
    </nat>
  </configuration>
</node>

```

```

        </nat>
    </configuration>
</node>
<node functional_type="STATEFUL_FIREWALL" name="20.0.0.2">
    <neighbour name="20.0.0.1" />
    <neighbour name="30.0.5.2" />
    <neighbour name="30.0.5.3" />
    <configuration description="A simple description" name="conf1">
        <stateful_firewall defaultAction="DENY">
            <elements>
                <action>ALLOW_COND</action>
                <source>20.0.0.1</source>
                <destination>30.0.5.2</destination>
                <protocol>ANY</protocol>
                <src_port>0-65535</src_port>
                <dst_port>0-65535</dst_port>
            </elements>
            <elements>
                <action>ALLOW_COND</action>
                <source>20.0.0.1</source>
                <destination>30.0.5.3</destination>
                <protocol>ANY</protocol>
                <src_port>0-65535</src_port>
                <dst_port>0-65535</dst_port>
            </elements>
        </stateful_firewall>
    </configuration>
</node>
<node functional_type="WEBSERVER" name="30.0.5.2">
    <neighbour name="20.0.0.2" />
    <configuration description="A simple description"
        name="confB">
        <webserver>
            <name>30.0.5.2</name>
        </webserver>
    </configuration>
</node>
<node functional_type="WEBSERVER" name="30.0.5.3">
    <neighbour name="20.0.0.2" />
    <configuration description="A simple description"
        name="confB">
        <webserver>
            <name>30.0.5.3</name>
        </webserver>
    </configuration>
</node>
</graph>
</graphs>
<PropertyDefinition>
    <Property graph="0" name="StrongReachabilityProperty" src="10.0.0.1"
        dst="30.0.5.2" />
    <Property graph="0" name="ConditionalReachabilityProperty"
        src="30.0.5.2" dst="10.0.0.1" />

    <Property graph="0" name="StrongReachabilityProperty" src="10.0.0.2"
        dst="30.0.5.2" />

```

```

<Property graph="0" name="IsolationProperty" src="30.0.5.2"
  dst="10.0.0.2" />

<Property graph="0" name="ReachabilityProperty" src="10.0.0.3"
  dst="30.0.5.3" />
<Property graph="0" name="ConditionalReachabilityProperty"
  src="30.0.5.3" dst="10.0.0.3" />
</PropertyDefinition>
<ParsingString></ParsingString>
</NFV>

```

Listing 4.4. Complete Example of an XML test case

If, in another case, we were to add the property shown in Listing 4.5 to the same test case, the output of the problem would be as follows:

```
19:27:04.013 [main] INFO result - UNSAT
```

because the *IsolationProperty* from 30.0.5.3 to 10.0.0.3 cannot be satisfied due to the ALLOW_COND rule from 10.0.0.3 to 30.0.5.3 in the first firewall and the ALLOW_COND rule from 20.0.0.1 to 30.0.5.3 in the second firewall.

```
<Property graph="0" name="IsolationProperty" src="30.0.5.3" dst="10.0.0.3" />
```

Listing 4.5. Additional property

In the same way, a series of test cases was executed, and their outputs produced the expected results, demonstrating that the Verification Problem is correctly implemented and solved in VEREFOO.

4.2 Refinement problem

This part thesis addresses the Refinement Problem for stateful firewalls, focusing on VEREFOO's solution for the automatic allocation and configuration of firewalls within a given Service Graph (SG). This methodology ensures that the firewall setup adheres to user-defined Network Security Requirements (NSRs) while minimizing the number of firewalls and the rules associated with them.

A key advantage of this approach is its applicability to real-world scenarios. It supports situations where new security policies need to be applied to an already existing SG or where a manually defined SG requires automatic enhancement with security functions. The process not only computes optimal firewall placements but also generates the required rules, ensuring strict compliance with NSRs. Moreover, the methodology is based on a formal model, ensuring correctness and efficiency, providing a complete and robust solution.

The following sections outline the transition from a stateless to a stateful approach, highlighting key analogies and differences between the two. After this comparison, a logical model is proposed to define how stateful firewalls should be implemented and developed, offering a structured framework for their integration.

4.2.1 From stateless to stateful firewalls

In this problem, it is crucial to differentiate the types of packets that a firewall processes in order to understand its behavior, particularly in highlighting the additional functionality that a stateful firewall provides over a stateless one.

Figure 4.3 illustrates the set of input packets for a stateless firewall, denoted as n_i . Specifically, the input packet class is divided into two categories:

- I_i^a , representing the class of packets that the firewall allows (i.e., it does not drop them);
- I_i^d , representing the class of packets that the firewall drops.

These classes can be derived from any firewall configuration, regardless of whether the firewall operates under a whitelisting or blacklisting mode. The two classes are represented as predicates that are disjunctions of elementary conditions defined over the IP 5-tuple:

- $I_i^a = q_{1,i,a} \vee q_{2,i,a} \vee \cdots \vee q_{n_1,i,a}$;
- $I_i^d = q_{1,i,d} \vee q_{2,i,d} \vee \cdots \vee q_{n_1,i,d}$.

where each elementary predicate q is defined as $(IP_{Src}, IP_{Dst}, p_{Src}, p_{Dst}, t_{Proto})$, specifying the source and destination IP addresses, source and destination ports, and the protocol type.

When applying the concept of atomic predicates, we can express each predicate q , which represents a class of packets, as a disjunction of atomic predicates:

$$q_{x,i,a} = p_{1,x,i,a} \vee p_{2,x,i,a} \vee \cdots \vee p_{n_x,x,i,a}$$

In this expression, each $p_{k,x,i,a}$ is an atomic predicate, representing a minimal, disjoint subset of packets. These atomic predicates cannot be further divided and do not overlap with each other. This way of breaking down packet classes allows for a more granular representation of the traffic a firewall must handle. For instance, if $q_{x,i,a}$ describes a broader class of packets (like all packets from a certain source IP), each atomic predicate $p_{k,x,i,a}$ would represent a smaller, non-overlapping sub-class (e.g., packets from the source IP with a specific protocol and port number).

To refine the firewall's behavior, we introduce notations to describe the relationships between these packet classes and their atomic predicates:

- $q \in I_i^z$ means that the predicate q is a sub-class of the packet class I_i^z . For example, I_i^z might represent all packets the firewall is configured to handle, and q is one of the specific sub-classes (like allowed or denied packets).
- $p \in q$ means that p (an atomic predicate) is a sub-class of q , meaning that p is one of the atomic predicates in the disjunction representing q .

In the MaxSMT problem based on atomic predicates, two key hard constraints must be satisfied:

1. Accepting Packets:

$$\forall q \in I_i^a. \forall p \in q. \neg \text{deny}(p)$$

This constraint ensures that for all packet classes q in the set I_i^a (the set of packets that the firewall allows), and for all atomic predicates p that are part of those packet classes q , the firewall must not deny those packets. This essentially guarantees that all packets in the allowed class are permitted by the firewall.

2. Dropping Packets:

$$\forall q \in I_i^d. \forall p \in q. \text{deny}(p)$$

This constraint ensures that for all packet classes q in the set I_i^d (the set of packets that the firewall denies), and for all atomic predicates p in those packet classes, the firewall must deny them. This guarantees that all packets in the denied class are blocked by the firewall.

These constraints formalize how a firewall processes packets, either allowing or denying them based on the atomic predicates. The firewall must respect these constraints, ensuring that packets in I_i^a are always allowed and packets in I_i^d are always denied. This methodology provides a clear and rigorous approach to enforcing security policies in firewalls, using atomic predicates to manage packet flow in a formal and optimized manner.



Figure 4.3. Stateless schema

The stateless firewall model is simple and functional but can be extended to enhance security in a network. The following section presents the theoretical formulation of a stateful firewall in the VEREFOO model.

In this extended model, the input packet class is divided into four distinct categories:

- I_i^a represents the class of packets that the firewall never drops, and for which no state is saved.
- I_i^d corresponds to the class of packets that the firewall always drops, regardless of state.
- I_i^{as} is the class of packets that the firewall never drops but for which a state is saved.
- I_i^{ac} represents packets that the firewall does not drop, provided a matching state exists.

where a stands for accept, d stands for drop, as stands for accept and save and ac stands for accept conditionally. Figure 4.4 illustrates the set of input packets for a stateful firewall, denoted as n_i .

These classes are defined based on the actions taken by the firewall with respect to saving or requiring state information, which is particularly important for stateful firewalls where the handling of packets depends on the existence of a previous connection.

Also the $\text{cond_permit}_i(p)$ predicate defined in Section 4.1.1 assumes different values based on the input packet set:

- For stateless functions, $\forall p. \text{cond_permit}_i(p) = \text{false}$
- For stateful functions:
 1. $\forall q \in (I_i^a \cup I_i^d \cup I_i^{as}). \forall p \in q. \text{cond_permit}_i(p) = \text{false}$
 2. $\forall q \in I_i^{ac}. \forall p \in q. \text{cond_permit}_i(p) = \text{true}$

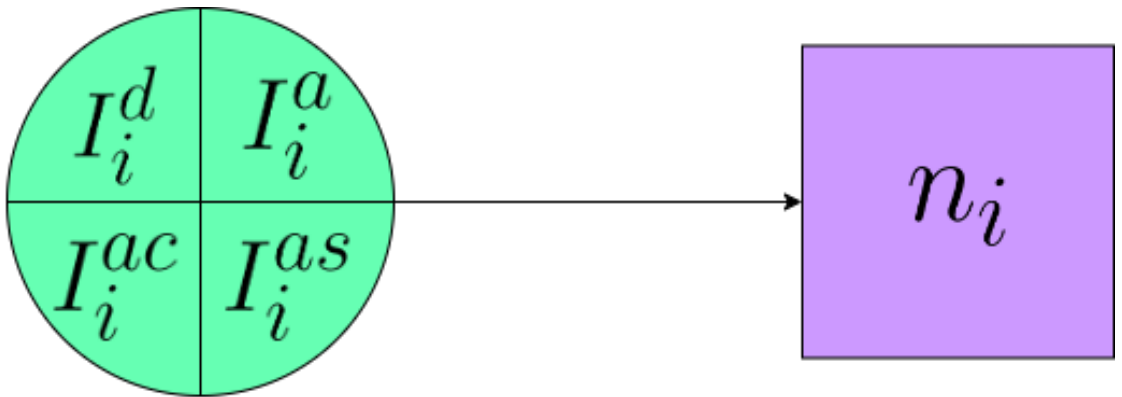


Figure 4.4. Stateful schema

The packet classes I_i^{as} and I_i^{ac} are closely related. Specifically, the set I_i^{as} , which covers packets accepted with saved state, can be expressed as a disjunction of atomic predicates:

$$I_i^{as} = q_{1,i,as} \vee q_{2,i,as} \vee \cdots \vee q_{n_{as},i,as}$$

where the predicates $q_{x,i,as}$ are derived from rules that indicate new connections.

On the other hand, I_i^{ac} corresponds to rules that handles established or related connections. The set I_i^{ac} can also be expressed as a disjunction:

$$I_i^{ac} = q_{1,i,ac} \vee q_{2,i,ac} \vee \dots \vee q_{n_{ac},i,ac}$$

where

$$q_{x,i,ac} = \text{inv } q_{x,i,as} = (q_{1,i,as} \cdot \text{IPDst}, q_{1,i,as} \cdot \text{IPSrc}, q_{1,i,as} \cdot \text{pDst}, q_{1,i,as} \cdot \text{pSrc}, q_{1,i,as} \cdot \text{tProto})$$

Each predicate $q_{x,i,ac}$ is the inverse of a corresponding predicate in I_i^{as} . This inverse is constructed by matching parameters such as destination and source IP addresses, ports, and protocol types from $q_{x,i,as}$, reflecting how established connections are tracked.

A unique feature of the VEREFOO model is the absence of a concept of time. In traditional stateful models, state often involves time, but in this case, the state is defined differently. Particularly, a stateful firewall n_i accepts a class of packets $q_{x,i,ac} \in I_i^{ac}$ if certain conditions are met:

1. A traffic flow that leads packets in the class $\text{inv}(q_{x,i,ac})$ to the firewall n_i is generated.
2. The packet class $\text{inv}(q_{x,i,ac})$ is not blocked by n_i or by any previous node in the path from its source.

Incorporating this stateful behavior into a MaxSMT problem allows for the formulation of hard constraints that enforce these behaviors. Three primary classes of constraints can be defined:

- $\forall q \in I_i^a. \forall p \in q. \neg \text{denyi}(p)$: This ensures that packets in the I_i^a set (those always accepted) are never denied by the firewall.
- $\forall q \in I_i^d. \forall p \in q. \text{denyi}(p)$: This ensures that packets in the I_i^d set (those always dropped) are always denied.
- $\forall q \in I_i^{as}. \forall p \in q. \neg \text{denyi}(p)$: This guarantees that packets that establish a new connection (and save state) are accepted.

A fourth class of constraints governs conditionally accepted packets. For packets in I_i^{ac} , the firewall does not deny them if certain conditions are met. This type of situation is shown in Figure 4.5. This is formalized as:

$$\forall q \in I_i^{ac}. \forall p \in q. \neg \text{denyi}(p) = \otimes$$

where:

$$\otimes = \forall p' \in \text{inv}(q). \exists f \in F^R. (1) \wedge (2) \wedge (3)$$

The three conditions that must hold are:

$$(1) \quad \tau(n_i, f) = p'$$

This condition means that the firewall n_i has observed the "inverse" of the current packet. Here, τ is a function that returns an atomic predicate and p' is the atomic predicate for the inverse packet. The "inverse" packet refers to a packet flowing in the opposite direction of the current one, such as a reply to a request. This ensures that the firewall is aware of the packet's corresponding traffic.

$$(2) \quad \tau(n_i, f) \wedge I_i^{as}$$

This condition ensures that the inverse packet has successfully established a valid connection. By combining $\tau(n_i, f)$ with I_i^{as} , this guarantees that the firewall not only saw the inverse packet but also recorded the necessary state information, indicating that a legitimate connection exists.

$$(3) \quad \forall n_j \in \pi(f) \mid n_j < n_i, j \neq i. \neg \text{deny}_j(\tau(n_j, f))$$

This condition checks that no earlier firewall along the network path has blocked the flow. Here, $\pi(f)$ represents the path taken by the flow f , and the term $n_j < n_i$ indicates that n_j is an earlier node on the path than n_i . The condition $\neg \text{deny}_j(\tau(n_j, f))$ ensures that none of the previous firewalls n_j have denied the atomic predicate $\tau(n_j, f)$ of the flow. Essentially, this condition ensures that the flow remains unblocked as it travels through earlier firewalls before reaching n_i .

The behaviour of the functions τ and π is explained in Section 4.1.1 while the function $F_R \subseteq F$ denote the set of flows that satisfy R .

This formulation integrates both stateless and stateful behaviors into a unified model, enabling the firewall to dynamically handle packets based on their state while ensuring robust security policies are enforced across a network.

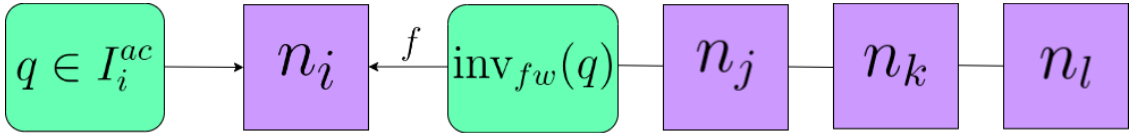


Figure 4.5. Schema for accept conditionally hard constraint

4.2.2 New constraint formulations

To configure a stateful firewall automatically, the version used in stateless firewalls must be modified by adding new rules and constraints, both hard and soft.

In the solver used to tackle the firewall allocation problem, two types of predicates can be applied:

- The first is the *deny* predicate, where conditions can be specified to block certain traffic.
- The second is the *cond_permit* predicate, which regulates traffic conditioned by a prior connection on the reverse traffic.

In the stateless version, there is only one hard constraint that uses the *deny* predicate, formulated as:

$$deny(n, t) = used(n) \wedge [(whitelist(n) \wedge \neg rule(t)) \vee (\neg whitelist(n) \wedge rule(t))]$$

Here, the formula takes as input the node n , where the current firewall is located, and the traffic t , which needs to be either blocked or allowed depending on the policy. The predicates used in the formula are defined as follows:

- *used*(n): indicates whether the firewall is active on node n . If the firewall is not deployed at node n , the rest of the formula is irrelevant, and the firewall doesn't handle the traffic at that node.
- *whitelist*(n): specifies if the firewall at node n is in whitelist mode. In whitelist mode, only explicitly allowed traffic is permitted; everything else is blocked by default.
- *rule*(t): determines if there is a rule allowing traffic t .

The formula means the firewall will block traffic t if it is in whitelist mode and there is no rule allowing t (i.e., anything not explicitly allowed is blocked) or it is in blacklist mode and there is a rule explicitly blocking the traffic t .

For stateful firewalls, which operate exclusively in whitelist mode, two types of rules are possible:

- ALLOW: permits traffic from source A to destination B, as in stateless firewalls.
- ALLOW_COND: always allows traffic from A to B but conditionally allows traffic from B back to A.

Given these additional features, the pre-existing constraint on *deny* must be updated, as it only considered the ALLOW rule, and a new constraint for the *cond_permit* predicate must be introduced to handle conditional connections. The new hard constraints are defined as:

$$deny(n, t) = used(n) \wedge whitelist(n) \wedge \neg rule_{allow}(t) \wedge \neg rule_{allow_cond}(t) \\ \wedge \neg rule_{allow_cond}(t^{-1})$$

$$cond_permit(n, t) = used(n) \wedge whitelist(n) \wedge rule_{allow_cond}(t^{-1}) \wedge \neg rule_{allow}(t) \\ \wedge \neg rule_{allow_cond}(t)$$

where the predicates *used*(n) and *whitelist*(n) are the same as in the previous formulation, while the others are:

- *rule_{allow}*(t): checks if there is a rule allowing traffic t .

- $rule_{allow_cond}(t)$: checks if a rule exists that both allows traffic t and conditionally permits reverse traffic t^{-1} .
- $rule_{allow_cond}(t^{-1})$: checks if a rule exists that both allows traffic t^{-1} and conditionally permits reverse traffic t .

The definition of the $deny(n, t)$ predicate is similar to that of the stateless case, with a few key differences. In the stateful firewall, the possibility of operating in blacklist mode is removed, as stateful firewalls operate exclusively in whitelist mode. Additionally, new rules have been introduced. Specifically, to deny a packet, it is not enough to check for the absence of an explicit rule allowing the packet; any conditional rules that might allow the packet must also be absent. Otherwise, the packet cannot be denied outright.

The $cond_permit(n, t)$ predicate is unique to stateful firewalls and becomes true only when certain specific conditions are met. The first two conditions, $used(n)$ and $whitelist(n)$, are the same as in the $deny(n, t)$ predicate, indicating that the firewall is deployed at node n and operates in whitelist mode, as previously discussed.

The key element that characterizes the $cond_permit(n, t)$ constraint is the predicate $rule_{allow_cond}(t^{-1})$. This predicate must be true because it signifies that traffic t , as the inverse of t^{-1} , is conditionally allowed. Without this predicate, the $cond_permit(n, t)$ constraint would lose its purpose, as it relies on the conditional nature of the traffic being permitted.

For $cond_permit(n, t)$ to be valid, two additional conditions must also hold: there must not be an unconditional rule allowing traffic t (i.e., $\neg rule_{allow}(t)$), and there must not be a conditional rule allowing traffic t (i.e., $\neg rule_{allow_cond}(t)$). These conditions are necessary because if either of those rules existed, traffic t would be allowed unconditionally rather than conditionally, thus invalidating the logic of the $cond_permit(n, t)$ constraint, which only applies to conditional traffic permissions.

In addition, the soft constraints must be adjusted due to the introduction of these additional rules. Whenever possible, the solver should prioritize minimizing the number of rules written in the firewalls. An ALLOW_COND rule is preferable to an ALLOW rule because it covers both the direct traffic from A to B and the conditional reverse traffic from B to A, thus reducing the total number of rules in the firewall.

To reflect this preference, weights should be assigned to rules: a lower weight indicates higher priority, while a higher weight means a less important rule. Consequently, an ALLOW_COND rule should have a lower weight compared to an ALLOW rule.

Chapter 5

Translators for stateful firewall technology

This chapter presents the final part of the contribution to the thesis. Specifically, the objective is to develop a translator that converts the configuration from a medium-level XML format for a stateful firewall to a low-level format compatible with a real stateful firewall technology.

The XML configuration is the same as the one presented in Chapter 4. In particular, the rules already present within the firewall are translated according to the semantics and syntax of the chosen technology. Three technologies were implemented and developed, as described in Chapter 1: iptables, IpFirewall, and Open vSwitch. The translator generates an output file that contains executable rules tailored for the respective platform, allowing users to replicate the configuration specified in the XML file. Essentially, it produces a ready-to-use script that can be directly executed to implement the desired network setup.

5.1 Implementation

To implement these functionalities, a set of Java classes has been developed. The connections between these classes are illustrated in the diagram in Figure 5.1. The main class, *FirewallSerializer*, receives as input the XML file containing the stateful firewall configuration and determines which type of firewall technology to use and create. This class acts as a dispatcher, coordinating all the submodules that may be instantiated. To achieve this, the *FirewallDeploy* enum is used, enabling the selection between the three available configurations, as shown in Listing 5.1. The listing also includes a fourth option, *ALL*, which can be selected when all configurations need to be analyzed, typically for testing purposes.

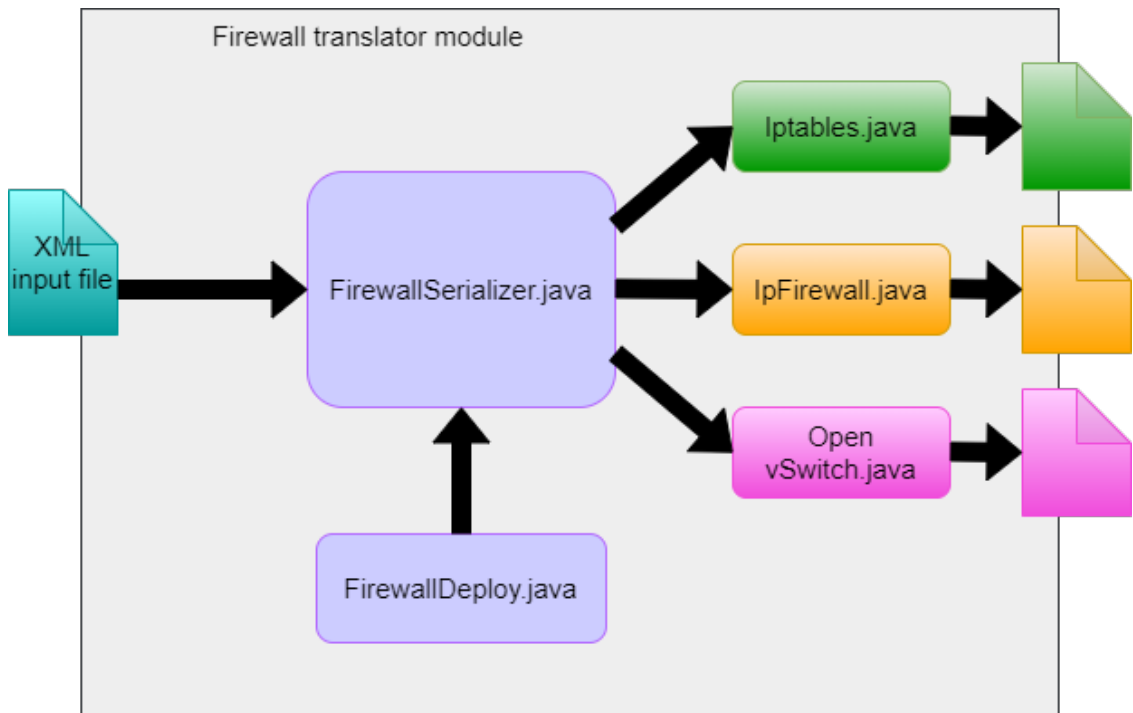


Figure 5.1. Firewall translator classes schema

```

public enum FirewallDeploy {

    IPFIREWALL,
    IPTABLES,
    OPENVSWITCH,
    ALL
}

```

Listing 5.1. FirewallDeploy enum

In the following sections, the platforms used in the translation process are examined in greater detail compared to their presentation in Chapter 1, with a particular focus on how the rules in the configuration files are structured.

5.1.1 Iptables

Iptables is the default firewall technology used on Linux systems. Before creating a script that can be executed directly on the platform, some preliminary steps must be completed.

In particular, Listing 5.2 indicate the commands that have to be executed before starting to populate the firewall [17]. The script begins with the shebang `#!/bin/sh`, which specifies that the commands will be executed in the `sh` shell. It then assigns the variable `cmd` to `sudo iptables`, which ensures that all firewall commands will be executed with superuser privileges.

The script clears any existing firewall rules using `${cmd} -F`, which effectively resets the firewall configuration. Next, it sets to DROP the default policies for three main firewall chains: INPUT, FORWARD, and OUTPUT.

```
#!/bin/sh
cmd="sudo iptables"
${cmd} -F
${cmd} -P INPUT DROP
${cmd} -P FORWARD DROP
${cmd} -P OUTPUT DROP
```

Listing 5.2. iptables preliminary commands

The following commands apply to the FORWARD chain, which is responsible for handling packets forwarded between network interfaces. Specifically, the command `-A FORWARD` appends a rule to the end of the FORWARD chain. A crucial component for this implementation is the `-m` option, which specifies the conntrack module needed for tracking states in stateful firewalls. This option will be used exclusively for stateful rules that save the state of packets or conditionally accept them.

The first rule is always inserted after the preliminary operations and ensures that return traffic from RELATED or ESTABLISHED connections is allowed to pass through the system (Listing 5.3). These states characterize a saved connection, meaning packets accepted conditionally will always be permitted.

```
${cmd} -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j
ACCEPT
```

Listing 5.3. iptables default rule for accepting return traffic

Packets that need to be tracked will use the option `-m conntrack --ctstate NEW`, which allows for the saving of a new connection state. The other commands supported by the translator for creating additional rules include:

- `-p protocol`: Specifies the protocol, which can be tcp or udp. If the protocol is not specified (written as ANY in the XML file), two rules will be generated for each possible value.
- `-s source IP`: Specifies the source IP address. When generated from the XML file, the translator will automatically add the netmask.
- `-d destination IP`: Specifies the destination IP address. Similar to the source IP, the translator will automatically add the netmask.
- `-sport sourcePort`: Specifies the source port, which can be a single number or a range (e.g., 1000:2000). If the XML file contains the * symbol, indicating all ports, this option will not appear in the rule.

- `-dport destinationPort`: Specifies the destination port, formatted similarly to the source port. The presence of the `*` symbol in the XML file means this option will be omitted.
- `-j action`: Specifies the action to take. Since stateful firewalls operate in whitelisting mode, the only possible action is ACCEPT, allowing packets that conform to the rules.

The practical application of the presented concepts can be seen in the example provided in Listing 5.4. In the first line, a rule is appended to allow TCP packets from the source 10.0.0.3/32 to the destination 30.0.5.3/32, without involving any stateful operations. The following lines show similar rules, but duplicated for both the TCP and UDP protocols. In these lines, along with the IP addresses, a source port range of 6:10 is specified, and the option to track the state of new connections is enabled using the `conntrack` module.

```
{cmd} -A FORWARD -p tcp -s 10.0.0.3/32 -d 30.0.5.3/32 -j ACCEPT
{cmd} -A FORWARD -p tcp -s 10.0.0.4/32 -d 30.0.5.2/32 --sport
6:10 -m conntrack --ctstate NEW -j ACCEPT
{cmd} -A FORWARD -p udp -s 10.0.0.4/32 -d 30.0.5.2/32 --sport
6:10 -m conntrack --ctstate NEW -j ACCEPT
```

Listing 5.4. iptables example

5.1.2 IpFirewall

IpFirewall, commonly referred to as `ipfw`, is the packet filtering technology utilized in FreeBSD operating systems. It is distinguished by two key concepts: sets and priorities [18].

Sets enable the grouping of rules for more efficient management. Each set is identified by a number ranging from 0 to 31, allowing rules within the same set to be manipulated collectively. If no set is specified, rules are added to set 0 by default. Each rule in `ipfw` is assigned a priority (or rule number), which dictates the order in which the rules are evaluated. Lower numbers indicate higher priority, meaning those rules are processed first. The range for rule numbers is from 0 to 65535, with 65535 representing the lowest priority. It is crucial to note that a rule with a lower priority but within a certain set will be executed before a higher-priority rule located in a set with a higher number.

Also with `ipfw` scripts preliminary commands must be executed. The script begins with the shebang line `#!/bin/sh` and assigns the variable `cmd` to the command `/sbin/ipfw -q`, which is used to execute `ipfw` on FreeBSD. Subsequently, all current rules are removed using the command `$cmd -f flush` (see Listing 5.5).

```
#!/bin/sh
cmd="/sbin/ipfw -q"
```



```

${cmd} -f flush
${cmd} delete set 31

```

Listing 5.5. ipfw preliminary commands

Set 31, which typically contains the default rules, is not affected by the flush command. Therefore, a specific rule is necessary to delete any previously inserted rules. This set is used at the end of the script, where the last default rule is added. This rule has the lowest priority (65534 and not 65535 because it's reserved) and is placed in the final set to ensure it is evaluated last. Specifically, it denies traffic that does not match any of the preceding rules, adhering to the whitelisting mode, as illustrated in Listing 5.6.

Once the old rules have been deleted, the first rule added is the `check-state` default rule that permits existing stateful connections, allowing packets that are already being tracked. This rule is assigned a priority of 0 to ensure it is evaluated first.

```

${cmd} add 0 check-state
${cmd} add 65534 set 31 deny ip from any to any

```

Listing 5.6. ipfw default rules

Rules in ipfw are inserted using the command `add`, and the following parameters can be specified when creating a rule:

- *action*: This parameter defines the rule's action, such as allow or deny. For stateful firewalls, the only available action is allow.
- *priority*: This specifies the rule's priority, which determines its order of evaluation relative to other rules following the `add` command.
- `set setNumber`: This indicates the set number to which the rule will be added, facilitating easier management of groups of rules.
- *protocol*: This defines the protocol to be used, which can be `tcp`, `udp`, or `ip` (for all IP traffic).
- `from sourceIP`: This parameter specifies the source IP address. IP addresses with this translator are defined with a netmask. The keyword `any` can be used to indicate all possible addresses.
- `to destinationIP`: This parameter specifies the destination IP address. IP addresses with this translator are defined with a netmask. The keyword `any` can be used to indicate all possible addresses.
- *sourcePort*: This follows the source IP address and specifies the source port. It can be a single port number or a range (e.g., '1024-2048').
- *destinationPort*: This follows the destination IP address and specifies the destination port. It can be a single port number or a range (e.g., '1024-2048').

- **keep-state:** This option can be included in a rule when traffic needs to be conditionally accepted. It keeps track of whether a connection has been established, allowing the firewall to permit or deny subsequent packets based on the connection's state.

In general, rules are added with priority 1 and assigned to set 1 to ensure they are executed after the rule for existing stateful connections and before the default rule. In the following example, two rules are presented.

The first rule allows TCP packets from the address 10.0.0.3 to the address 30.0.5.3, permitting traffic from all source ports to the destination port range 100-200.

The second rule allows both TCP and UDP packets from the address 10.0.0.4 to the address 30.0.5.2, permitting traffic from source ports 6-10 to all destination ports. This second rule is stateful because it tracks the state of the connection, allowing return traffic as part of the established connection.

```
{cmd} add 1 set 1 allow tcp from 10.0.0.3/32 0-65535 to
      30.0.5.3/32 100-200
{cmd} add 1 set 1 allow ip from 10.0.0.4/32 6-10 to 30.0.5.2/32
      0-65535 keep-state
```

Listing 5.7. ipfw example

5.1.3 Open vSwitch

Open vSwitch (OVS) is an open-source virtual switch designed to provide switching capabilities for virtualized and cloud environments, primarily within hypervisors. Additionally, Open vSwitch manages the priorities that determine the order in which rules are evaluated [19].

If multiple rules match a packet, the rule with the highest priority will be applied. Priorities range from 0 (highest) to 65535 (lowest), with the default priority typically set to 0 when not specified. A rule with a priority of 0 is evaluated first, while a rule with a priority of 65535 is evaluated last. In most configurations, low-priority rules act as a default action, allowing or dropping any unmatched traffic. This priority evaluation method is similar to that of ipfw, where 0 is the highest priority and 65535 is the lowest.

The command used in the shell to execute Open vSwitch (OVS) is `sudo ovs-ofctl`. Before adding rules, the bridges corresponding to the firewalls must already be created. For instance, in the example provided in the listing, the `add-flow` command is used to add a new rule to a bridge. The name that follows this command refers to the bridge name (in this case, A).

To accept return traffic for stateful commands, a rule is inserted first with the highest possible priority, which is 0, ensuring that these types of packets are always permitted. At the end of the script, there is a default rule to deny other packets

(action: `drop`). This rule has a priority of 65534, as it should only be executed when no other rules match the packets. The priority cannot be set to 65535, as that is reserved for default rules on the bridge, which we intend to override. These rules are illustrated in the following Listing:

```
sudo ovs-ofctl add-flow A
    priority=0,ct_state=+est+rel,dl_type=0x800,action=NORMAL
sudo ovs-ofctl add-flow A
    priority=65534,dl_type=0x800,action=drop
```

Listing 5.8. ovs default rules

To create the OVS rules generated by the translator, the following options can be specified:

- `priority=priorityNumber`: Defines the order in which rules are evaluated. Rules with higher priority are applied before those with lower priority, as previously described.
- `ct_state=stateType`: Matches the state of connections tracked by OVS's connection tracker. The `+new` option matches packets that are part of a new connection, while `+est` (established) and `+rel` (related) match packets that are part of existing or established connections.
- `dl_type=packetType`: Specifies the type of payload. In this context, `0x800` represents IP packets and it's the only value used.
- `nw_src=sourceIp`: Defines the source IP address along with its netmask. If omitted, the rule applies to all source addresses.
- `nw_dst=destinationIp`: Defines the destination IP address along with its netmask. If omitted, the rule applies to all destination addresses.
- `nw_proto=protocolNumber`: Specifies the protocol number. For instance, `6` represents TCP packets and `17` represents UDP packets. If both protocols are needed, separate rules must be created for each.
- `tp_src=portNumber`: Refers to the source port at the transport layer, represented in hexadecimal format. For port ranges, bitmasks are used to efficiently divide the source port space into progressively smaller ranges, reducing the number of required rules. This optimization enhances performance when handling large volumes of traffic.
- `tp_dst=portNumber`: Refers to the destination port at the transport layer, represented in hexadecimal format. Like the source port, destination ports are also handled using bitmasks to divide port ranges, improving performance under heavy traffic conditions by minimizing the number of rules.
- `action=actionType`: Defines the action to take when a packet matches the rule. The actions used by the translator are `NORMAL` that forward the packet normally and `drop` which discard the packet.

The rules in the script are added with a priority of 1 to ensure they are executed after the rule allowing packets from existing connections, but before the default deny rule. An example of an OVS script is shown in Listing 5.9.

The first rule is a stateless rule that permits TCP packets from the source address 10.0.0.3 to the destination address 30.0.5.3. The subsequent rules address the same traffic but are categorized based on the protocols and ports. Specifically, there are six rules: the first three apply to TCP packets, while the remaining three apply to UDP packets, both originating from the address 10.0.0.4 and destined for 30.0.5.2. Three of these rules are necessary to segment the port range, which spans from 6 to 10.

```
sudo ovs-ofctl add-flow B
  priority=1,dl_type=0x800,nw_src=10.0.0.3/32,
  nw_dst=30.0.5.3/32,nw_proto=6,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.4/32,
  nw_dst=30.0.5.2/32,nw_proto=6,tp_src=0x8/0xfffe,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.4/32,
  nw_dst=30.0.5.2/32,nw_proto=6,tp_src=0xa,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.4/32,
  nw_dst=30.0.5.2/32,nw_proto=6,tp_src=0x6/0xfffe,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.4/32,
  nw_dst=30.0.5.2/32,nw_proto=17,tp_src=0x8/0xfffe,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.4/32,
  nw_dst=30.0.5.2/32,nw_proto=17,tp_src=0xa,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.4/32,
  nw_dst=30.0.5.2/32,nw_proto=17,tp_src=0x6/0xfffe,action=NORMAL
```

Listing 5.9. ovs example

5.2 Examples

After explaining how the rules are structured in the scripts generated by the translator, we will now present some examples to demonstrate the translation results. The following XML file is provided as input to the translator. It defines two stateful firewalls: the first contains two ALLOW_COND rules and one ALLOW rule for three clients, while the second includes two ALLOW_COND rules from the same address (a NAT), along with three additional rules for web clients.

```

<node functional_type="STATEFUL_FIREWALL" name="20.0.0.3">
  <neighbour name="10.0.0.1" />
  <neighbour name="10.0.0.2" />
  <neighbour name="10.0.0.3" />
  <neighbour name="20.0.0.1" />
  <configuration description="A" name="conf1">
    <stateful_firewall defaultAction="DENY">
      <elements>
        <action>ALLOW_COND</action>
        <source>10.0.0.1</source>
        <destination>30.0.5.2</destination>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </elements>
      <elements>
        <action>ALLOW</action>
        <source>10.0.0.2</source>
        <destination>30.0.5.2</destination>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </elements>
      <elements>
        <action>ALLOW_COND</action>
        <source>10.0.0.3</source>
        <destination>30.0.5.3</destination>
        <protocol>UDP</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </elements>
    </stateful_firewall>
  </configuration>
</node>
<node functional_type="STATEFUL_FIREWALL" name="20.0.0.2">
  <neighbour name="20.0.0.1" />
  <neighbour name="30.0.5.2" />
  <neighbour name="30.0.5.3" />
  <neighbour name="30.0.5.4" />
  <neighbour name="20.0.0.4" />
  <configuration description="B" name="conf1">
    <stateful_firewall defaultAction="DENY">
      <elements>
        <action>ALLOW_COND</action>
        <source>20.0.0.1</source>
        <destination>30.0.5.2</destination>
        <protocol>ANY</protocol>
        <src_port>0-65535</src_port>
        <dst_port>0-65535</dst_port>
      </elements>
      <elements>
        <action>ALLOW_COND</action>
        <source>20.0.0.1</source>
        <destination>30.0.5.3</destination>
        <protocol>UDP</protocol>
        <src_port>0-65535</src_port>
        <dst_port>0-65535</dst_port>
      </elements>
    </stateful_firewall>
  </configuration>
</node>

```

```

    </elements>
    <elements>
      <action>ALLOW_COND</action>
      <source>10.0.0.4</source>
      <destination>30.0.5.4</destination>
      <protocol>ANY</protocol>
      <src_port>*</src_port>
      <dst_port>*</dst_port>
    </elements>
    <elements>
      <action>ALLOW</action>
      <source>10.0.0.5</source>
      <destination>30.0.5.4</destination>
      <protocol>ANY</protocol>
      <src_port>*</src_port>
      <dst_port>*</dst_port>
    </elements>
    <elements>
      <action>ALLOW</action>
      <source>10.0.0.6</source>
      <destination>30.0.5.4</destination>
      <protocol>ANY</protocol>
      <src_port>*</src_port>
      <dst_port>*</dst_port>
    </elements>
  </stateful_firewall>
</configuration>
</node>

```

Listing 5.10. XML input file example

Since the XML file defines two firewalls, two separate scripts are generated, one for each firewall, and the firewall name is included in the `description` field (A and B). The first translator tested is iptables, which produced the following rules script. As shown, there are duplicate rules for TCP and UDP protocols, as they cannot be combined. It also includes commands to remove previous rules, as well as default commands to either accept or reject packets.

```

#!/bin/sh
cmd="sudo iptables"
${cmd} -F
${cmd} -P INPUT DROP
${cmd} -P FORWARD DROP
${cmd} -P OUTPUT DROP
${cmd} -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.0.1/32 -d 30.0.5.2/32 --sport 80 -m
  conntrack --ctstate NEW -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.0.2/32 -d 30.0.5.2/32 --dport 110:120 -j
  ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.0.2/32 -d 30.0.5.2/32 --dport 110:120 -j
  ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.0.3/32 -d 30.0.5.3/32 -m conntrack --ctstate
  NEW -j ACCEPT

```

Listing 5.11. Firewall A iptables translation

```
#!/bin/sh
cmd="sudo iptables"
${cmd} -F
${cmd} -P INPUT DROP
${cmd} -P FORWARD DROP
${cmd} -P OUTPUT DROP
${cmd} -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
${cmd} -A FORWARD -p tcp -s 20.0.0.1/32 -d 30.0.5.2/32 -m conntrack --ctstate
    NEW -j ACCEPT
${cmd} -A FORWARD -p udp -s 20.0.0.1/32 -d 30.0.5.2/32 -m conntrack --ctstate
    NEW -j ACCEPT
${cmd} -A FORWARD -p udp -s 20.0.0.1/32 -d 30.0.5.3/32 -m conntrack --ctstate
    NEW -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.0.4/32 -d 30.0.5.4/32 --sport 25 --dport
    90:92 -m conntrack --ctstate NEW -j ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.0.4/32 -d 30.0.5.4/32 --sport 25 --dport
    90:92 -m conntrack --ctstate NEW -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.0.5/32 -d 30.0.5.4/32 --dport 73 -j ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.0.5/32 -d 30.0.5.4/32 --dport 73 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.0.6/32 -d 30.0.5.4/32 -j ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.0.6/32 -d 30.0.5.4/32 -j ACCEPT
```

Listing 5.12. Firewall B iptables translation

Next, IpFirewall was tested, which generated the following scripts. Unlike iptables, ipfw is less verbose, as it groups different protocols together, preventing rule duplication. This results in a more concise and readable output. Additionally, ipfw inserts commands to delete previously added rules and manage default traffic handling.

```
#!/bin/sh
cmd="/sbin/ipfw -q"
${cmd} -f flush
${cmd} delete set 31
${cmd} add 0 check-state
${cmd} add 1 set 1 allow tcp from 10.0.0.1/32 80-80 to 30.0.5.2/32 0-65535
    keep-state
${cmd} add 1 set 1 allow ip from 10.0.0.2/32 0-65535 to 30.0.5.2/32 110-120
${cmd} add 1 set 1 allow udp from 10.0.0.3/32 0-65535 to 30.0.5.3/32 0-65535
    keep-state
${cmd} add 65534 set 31 deny ip from any to any
```

Listing 5.13. Firewall A ipfw translation

```
#!/bin/sh
cmd="/sbin/ipfw -q"
${cmd} -f flush
${cmd} delete set 31
${cmd} add 0 check-state
${cmd} add 1 set 1 allow ip from 20.0.0.1/32 0-65535 to 30.0.5.2/32 0-65535
    keep-state
${cmd} add 1 set 1 allow udp from 20.0.0.1/32 0-65535 to 30.0.5.3/32 0-65535
    keep-state
${cmd} add 1 set 1 allow ip from 10.0.0.4/32 25-25 to 30.0.5.4/32 90-92
    keep-state
```

```
{cmd} add 1 set 1 allow ip from 10.0.0.5/32 0-65535 to 30.0.5.4/32 73-73
{cmd} add 1 set 1 allow ip from 10.0.0.6/32 0-65535 to 30.0.5.4/32 0-65535
{cmd} add 65534 set 31 deny ip from any to any
```

Listing 5.14. Firewall B ipfw translation

Lastly, Open vSwitch was tested in the translation, producing the following results. As anticipated, the generated scripts are significantly more verbose than those from the other tools, primarily due to the way Open vSwitch manages source and destination ports.

```
sudo ovs-ofctl add-flow A
  priority=0,ct_state=+est+rel,dl_type=0x800,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.1/32,
  nw_dst=30.0.5.2/32,nw_proto=6,tp_src=0x50,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,dl_type=0x800,nw_src=10.0.0.2/32,nw_dst=30.0.5.2/32,
  nw_proto=6,tp_dst=0x70/0xffff8,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,dl_type=0x800,nw_src=10.0.0.2/32,nw_dst=30.0.5.2/32,
  nw_proto=6,tp_dst=0x78,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,dl_type=0x800,nw_src=10.0.0.2/32,nw_dst=30.0.5.2/32,
  nw_proto=6,tp_dst=0x6e/0xffffe,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,dl_type=0x800,nw_src=10.0.0.2/32,nw_dst=30.0.5.2/32,
  nw_proto=17,tp_dst=0x70/0xffff8,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,dl_type=0x800,nw_src=10.0.0.2/32,nw_dst=30.0.5.2/32,
  nw_proto=17,tp_dst=0x78,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,dl_type=0x800,nw_src=10.0.0.2/32,nw_dst=30.0.5.2/32,
  nw_proto=17,tp_dst=0x6e/0xffffe,action=NORMAL
sudo ovs-ofctl add-flow A
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.3/32,
  nw_dst=30.0.5.3/32,nw_proto=17,action=NORMAL
sudo ovs-ofctl add-flow A priority=65534,dl_type=0x800,action=drop
```

Listing 5.15. Firewall A ovs translation

```
sudo ovs-ofctl add-flow B
  priority=0,ct_state=+est+rel,dl_type=0x800,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=20.0.0.1/32,
  nw_dst=30.0.5.2/32,nw_proto=6,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=20.0.0.1/32,
  nw_dst=30.0.5.2/32,nw_proto=17,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=20.0.0.1/32,
  nw_dst=30.0.5.3/32,nw_proto=17,action=NORMAL
sudo ovs-ofctl add-flow B
  priority=1,ct_state=+new,dl_type=0x800,nw_src=10.0.0.4/32,
  nw_dst=30.0.5.4/32,nw_proto=6,tp_src=0x19,tp_dst=0x5c,action=NORMAL
```



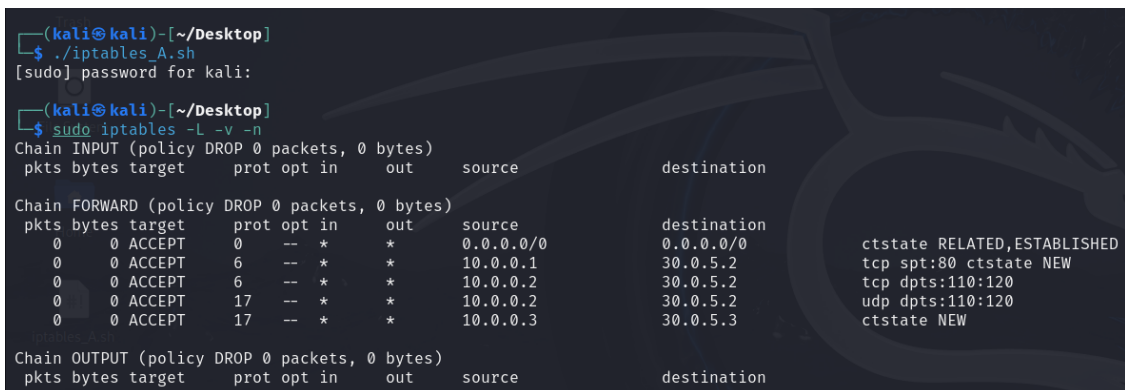
```

sudo ovs-ofctl add-flow B
    priority=1,ct_state=new,dl_type=0x800,nw_src=10.0.0.4/32,
    nw_dst=30.0.5.4/32,nw_proto=6,tp_src=0x19,tp_dst=0x5a/0xfffe,action=NORMAL
sudo ovs-ofctl add-flow B
    priority=1,ct_state=new,dl_type=0x800,nw_src=10.0.0.4/32,
    nw_dst=30.0.5.4/32,nw_proto=17,tp_src=0x19,tp_dst=0x5c,action=NORMAL
sudo ovs-ofctl add-flow B
    priority=1,ct_state=new,dl_type=0x800,nw_src=10.0.0.4/32,
    nw_dst=30.0.5.4/32,nw_proto=17,tp_src=0x19,tp_dst=0x5a/0xfffe,
    action=NORMAL
sudo ovs-ofctl add-flow B
    priority=1,dl_type=0x800,nw_src=10.0.0.5/32,nw_dst=30.0.5.4/32,
    nw_proto=6,tp_dst=0x49,action=NORMAL
sudo ovs-ofctl add-flow B
    priority=1,dl_type=0x800,nw_src=10.0.0.5/32,nw_dst=30.0.5.4/32,
    nw_proto=17,tp_dst=0x49,action=NORMAL
sudo ovs-ofctl add-flow B
    priority=1,dl_type=0x800,nw_src=10.0.0.6/32,nw_dst=30.0.5.4/32,
    nw_proto=6,action=NORMAL
sudo ovs-ofctl add-flow B
    priority=1,dl_type=0x800,nw_src=10.0.0.6/32,nw_dst=30.0.5.4/32,
    nw_proto=17,action=NORMAL
sudo ovs-ofctl add-flow B priority=65534,dl_type=0x800,action=drop

```

Listing 5.16. Firewall B ovs translation

To verify that the scripts are correctly written, the following screenshots display their execution on a Linux machine. Specifically, Figure 5.2 shows the execution of the first script for Iptables, named *iptables_A.sh*. As seen in the figure, the script runs without errors. By using the command `sudo iptables -L -v -n`, it is possible to check the active firewall rules. Upon inspection, we can confirm that the applied rules match the intended configuration.



```

(kali@kali)-[~/Desktop]
└─$ ./iptables_A.sh
[sudo] password for kali:
(kali@kali)-[~/Desktop]
└─$ sudo iptables -L -v -n
Chain INPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target    prot opt in     out     source destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target    prot opt in     out     source destination
 0      0 ACCEPT    0    --  *     *       0.0.0.0/0 0.0.0.0/0          ctstate RELATED,ESTABLISHED
 0      0 ACCEPT    6    --  *     *       10.0.0.1  30.0.5.2          tcp spt:80 ctstate NEW
 0      0 ACCEPT    6    --  *     *       10.0.0.2  30.0.5.2          tcp dpts:110:120
 0      0 ACCEPT    17   --  *     *       10.0.0.2  30.0.5.2          udp dpts:110:120
 0      0 ACCEPT    17   --  *     *       10.0.0.3  30.0.5.3          ctstate NEW

Chain OUTPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target    prot opt in     out     source destination

```

Figure 5.2. Execution of an Iptables script on a Linux machine

In Figure 5.3, the same firewall configuration is applied on the OVS platform using the script *ovs_A.sh*. As with the previous case, no errors occur during execution. To run the script, Open vSwitch must first be activated on the machine by

running the command `sudo systemctl start openvswitch-switch`. After that, the bridge must be created with the name of the firewall (in this case, `A`) using the command `sudo ovs-vsctl add-br A`. Once these steps are completed, the script can be executed, and the resulting configuration can be viewed with the command `sudo ovs-ofctl dump-flows A`. The output will show the active flows, which correspond to the rules defined in the script.

```
(kali@kali) [~/Desktop]
└─$ sudo systemctl start openvswitch-switch

(kali@kali) [~/Desktop]
└─$ sudo ovs-vsctl add-br A

(kali@kali) [~/Desktop]
└─$ ./ovs_A.sh

(kali@kali) [~/Desktop]
└─$ sudo ovs-ofctl dump-flows A

cookie=0x0, duration=29.529s, table=0, n_packets=0, n_bytes=0, priority=65534,ip actions=drop
cookie=0x0, duration=29.878s, table=0, n_packets=0, n_bytes=0, priority=1,ct_state=new,tcp,nw_src=10.0.0.1,nw_dst=30.0.5.2,tp_src=80 actions=NORMAL
cookie=0x0, duration=29.830s, table=0, n_packets=0, n_bytes=0, priority=1,tcp,nw_src=10.0.0.2,nw_dst=30.0.5.2,tp_dst=0x70/0xffff8 actions=NORMAL
cookie=0x0, duration=29.667s, table=0, n_packets=0, n_bytes=0, priority=1,udp,nw_src=10.0.0.2,nw_dst=30.0.5.2,tp_dst=0x70/0xffff8 actions=NORMAL
cookie=0x0, duration=29.777s, table=0, n_packets=0, n_bytes=0, priority=1,tcp,nw_src=10.0.0.2,nw_dst=30.0.5.2,tp_dst=120 actions=NORMAL
cookie=0x0, duration=29.634s, table=0, n_packets=0, n_bytes=0, priority=1,udp,nw_src=10.0.0.2,nw_dst=30.0.5.2,tp_dst=120 actions=NORMAL
cookie=0x0, duration=29.724s, table=0, n_packets=0, n_bytes=0, priority=1,tcp,nw_src=10.0.0.2,nw_dst=30.0.5.2,tp_dst=0x6e/0xffffe actions=NORMAL
cookie=0x0, duration=29.598s, table=0, n_packets=0, n_bytes=0, priority=1,udp,nw_src=10.0.0.2,nw_dst=30.0.5.2,tp_dst=0x6e/0xffffe actions=NORMAL
cookie=0x0, duration=29.565s, table=0, n_packets=0, n_bytes=0, priority=1,ct_state=new,udp,nw_src=10.0.0.3,nw_dst=30.0.5.3 actions=NORMAL
cookie=0x0, duration=36.042s, table=0, n_packets=0, n_bytes=0, priority=0 actions=NORMAL
cookie=0x0, duration=29.929s, table=0, n_packets=0, n_bytes=0, priority=0,ct_state=est+rel,ip actions=NORMAL
```

Figure 5.3. Execution of an OVS script on a Linux machine

Conclusions

This thesis has explored the integration and enhancement of stateful firewalls within the VEREFOO framework, aiming to improve network security and automate firewall configuration. The work was driven by the increasing complexity of modern networks and the rising need for robust cybersecurity solutions capable of adapting to dynamic and sophisticated threats. Through the development and testing of stateful firewalls, this research has demonstrated their significant advantages over stateless firewalls, particularly in their ability to monitor the state of active connections and provide a more comprehensive layer of security.

The first key contribution of this thesis was the verification of VEREFOO's ability to enforce Network Security Requirements (NSRs) when using stateful firewalls. Through a series of verification tests, it was shown that the framework can accurately verify whether network configurations meet the required security standards. This achievement is crucial as it confirms the framework's reliability in detecting compliance with security policies, ensuring that stateful firewall rules are correctly applied and managed across different network scenarios.

The second major contribution was the formulation of new logical statements to solve the refinement problem for stateful firewalls. This step enhanced the framework's capability to automatically allocate firewalls in an optimal manner, ensuring that traffic is filtered efficiently while meeting security demands. The introduction of new constraints to model the behavior of stateful firewalls enabled a more refined and accurate control over how traffic is handled, addressing the unique challenges posed by stateful traffic analysis and packet filtering.

The final contribution involved the development of a translation mechanism from medium-level XML configurations to low-level configurations for real-world firewall technologies such as Iptables, IpFirewall, and Open vSwitch. This translation process was essential for bridging the gap between the abstract firewall rules defined in the framework and their actual deployment on network devices.

The results of this thesis not only validate the VEREFOO framework's functionality in managing stateful firewalls but also highlight the importance of incorporating stateful functions into modern network security systems.

For future work, the proposed logical statements for solving the Refinement problem can be implemented into the VEREFOO code to enable optimal allocation of stateful firewalls. Additionally, further testing should be conducted to assess VEREFOO's scalability after the integration of stateful functions, ensuring that it maintains both performance and accuracy as networks increase in size and complexity.

Bibliography

- [1] A. Akhter, M. Fragkoulis, and A. Katsifodimos, “Stateful functions as a service in action,” *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2073–2086, 2020, available at <https://doi.org/10.14778/3352063.3352092>.
- [2] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, “Cloudburst: Stateful functions-as-a-service,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 11, pp. 2438–2452, 2020. [Online]. Available: <https://doi.org/10.14778/3407790.3407836>
- [3] D. Bansal, G. DeGrace, R. Tewari, M. Zygmunt, J. Grantham, S. Gai, M. Baldi, K. Doddapaneni, A. Selvarajan, A. Arumugam, B. Raman, A. Gupta, S. Jain, D. Jagasia, E. Langlais, P. Srivastava, R. Hazarika, N. Motwani, S. Tiwari, S. Grant, R. Chandra, and S. Kandula, “Disaggregating stateful network functions,” in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA, USA: USENIX Association, April 17–19 2023. [Online]. Available: <https://www.usenix.org/conference/nsdi23/presentation/bansal>
- [4] C. Diekmann, L. Hupel, J. Michaelis, M. Haslbeck, and G. Carle, “Verified iptables firewall analysis and verification,” *Journal of Automated Reasoning*, vol. 61, no. 2, pp. 191–242, 2018. [Online]. Available: <https://doi.org/10.1007/s10817-017-9445-1>
- [5] FreeBSD Document Project, *FreeBSD. Firewalls*, February 2010, accessed 16 May 2010. [Online]. Available: <https://www.freebsd.org/doc/handbook/firewalls.html>
- [6] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, “The design and implementation of open vswitch,” in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’15)*. Oakland, CA, USA: USENIX Association, May 2015, pp. 117–130. [Online]. Available: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>
- [7] Open vSwitch Project, “Open vswitch with conntrack integration,” <https://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/>, 2023, accessed: 10 August 2024.
- [8] L. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

- [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24
- [9] D. Kreutz, F. M. V. Ramos, P. J. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015, [Online]. Available: <https://doi.org/10.1109/JPROC.2014.2371999>.
- [10] R. Chayapathi, S. F. Hassan, and P. Shah, *Network Functions Virtualization (NFV) with a Touch of SDN*. Addison-Wesley Professional, 2016.
- [11] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, “Automated optimal firewall orchestration and configuration in virtualized networks,” in *2020 IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE, 2020.
- [12] —, “Automated firewall configuration in virtual networks,” *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1559–1576, March/April 2023, manuscript received 25 Mar. 2021; revised 2 Feb. 2022; accepted 13 Mar. 2022. Published 17 Mar. 2022, current version 14 Mar. 2023. Supported by EU H2020 Projects ASTRID (Grant No. 786922) and CyberSec4Europe (Grant No. 830929). Corresponding author: Fulvio Valenza.
- [13] D. Bringhenti and F. Valenza, “Greenshield: Optimizing firewall configuration for sustainable networks,” *IEEE Transactions on Network and Service Management*, 2024, in corso di stampa. [Online]. Available: <https://doi.org/10.1109/tnsm.2024.3452150>
- [14] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, “Improving the formal verification of reachability policies in virtualized networks,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, p. 713, March 2021.
- [15] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, “A two-fold traffic flow model for network security management,” *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, August 2024, accepted for publication.
- [16] S. M. Mohammad and S. Lakshmisri, “Security automation in information technology,” *International Journal of Creative Research Thoughts (IJCRT)*, vol. 6, no. 2, p. 901, June 2018. [Online]. Available: <http://www.ijcrt.org>
- [17] The Linux Man-Pages Project, *Linux Manual Pages: iptables(8)*, 2024, accessed: 2024-10-04. [Online]. Available: <https://man7.org/linux/man-pages/man8/iptables.8.html>
- [18] The FreeBSD Project, *FreeBSD Manual Pages: ipfw(8)*, 2024, accessed: 2024-10-04. [Online]. Available: [https://man.freebsd.org/cgi/man.cgi?ipfw\(8\)](https://man.freebsd.org/cgi/man.cgi?ipfw(8))
- [19] Open vSwitch. (n.d.) Ovs conntrack tutorial. Accessed: 2024-10-03. [Online]. Available: <https://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/>