Master of Science in Computer Engineering

Master Degree Thesis

Analysis on Access Control Policy and Security Groups in Firewall Configuration

Supervisors
prof. Riccardo Sisto
prof. Fulvio Valenza
dott. Daniele Bringhenti

Candidate
Simone ROMANTINI

Academic Year 2023-2024

This work is subject to the Creative Commons Licence

Summary

Thanks to the development of advanced networking solutions, such as Network Functions Virtualization (NFV) and Software-Defined Networking (SDN), there has been crucial improvement regarding the flexibility and efficiency in the building process of Service Function Chains.
With NFV the implementation of multiple, specific network functions, such as NAT or proxy servers, occurs on standard servers. Thus there is no more need for dedicated hardware devices for each function.
This use of standard servers allows for improved efficiency in resource management, both for the system and the device itself since to add a new function it is sufficient to implement it directly on the server, making the purchase of a new physical device no longer necessary; the server will then be able to use its own resources by sharing them with other functions installed on it.
With SDN there's the possibility to create specific routes for the packets traversing the network; this leads to a greater flexibility in the construction and management of Service Function Chains.
It's possible to modify and customize the path taken by each packet by programming a controller to manage the passage of packets through various network devices in real time.

However, problems arise when network devices used to create Service Function Chains are manually configured, as incorrect configurations may lead to significant security breaches or the passage of unwanted traffic.
Furthermore, additional latency during updates or maintenance of the security system is another issue caused by manual configuration of network devices .
By implementing Network Automation, it is possible to solve these problems thanks to the automatic configuration of network security devices, which reduces human errors and minimizes the latency associated with configuration changes.

An example of a Network Automation framework is **VEREFOO (VErified REFinement and Optimized Orchestration)**. VEREFOO receives as input a Service Graph and a set of Network Security Requirements and is then capable of generating an optimized Service Graph as output.
This graph represents a physical network populated with network security functions that are automatically allocated and configured to meet the constraints of the initial set of Network Security Requirements provided as input.

The work of this thesis was mainly focused on extending the firewall options and capabilities considered by VEREFOO in the generation of a solution, in addition to the Packet Filter security function already implemented.
In particular, the focus has been the study and implementation of Access Control

Firewall, a device that filters incoming and outgoing packets not only on the values of the IP quintuple but also by taking in consideration the interfaces which the packet passes through.
Another solution studied and implemented was the Security Groups, virtual firewalls that are associated to chosen interfaces of hosts in a network and allow the passage of selected packets while denying the rest.

Furthermore, to validate the effectiveness of the framework and the solutions implemented, Use Cases were generated and a Nftables Serializer was developed. Some of these Use Cases were used to test the correctness of the serializer and serve as basis for future tests.

Acknowledgements

First of all, I would like to thank professors Valenza and Sisto for supervising my work on this thesis and for giving me the opportunity to contribute to the improvement of a valuable and important tool like VEREFOO. They periodically checked on all the thesis' goals I achieved and were always available in providing me with constant feedback. The work carried out in the last months has helped me expand my knowledge of new aspects of the cybersecurity field and has offered different perspectives on things I already knew.
Furthermore, I want to thank Daniele for all the time he spent giving me guidance and for always being willing to offer advice whenever I had doubts or difficulties.

Finally, I also want to express my gratitude to my entire family and friends, who stood by me throughout this journey, despite various challenges, for their advice and for listening when I needed it.

With the completion of this thesis, a chapter of my life comes to an end, with its ups and downs, giving way to another one that, I am sure, will lead me to further expand my knowledge and bring many more achievements.

Contents

List of Figures 8

List of Tables 9

Listings 10

1 Introduction 12

1.1 Thesis objective 12

1.2 Thesis description 13

2 Network Automation 15

2.1 Service Function Chain 15

2.2 Software Defined Network and Network Function Virtualization . . 17

2.2.1 Definition of Software-Defined Networks and its characteristics 17

2.2.2 Application of the SDN technology to a SFC 18

2.2.3 Definition of Network Function Virtualization and its characteristics 20

2.2.4 Application of the NFV technology to a SFC 21

2.3 Automatizing Network and Security Management 21

2.3.1 Definition and Advantages 21

2.3.2 VEREFOO 22

3 Thesis Objective 24

4 Firewall technologies 26

4.1 Packet Filters 26

4.2 Access Control Firewalls and Security Groups introduction 26

4.3 Access Control Firewalls 27

4.3.1 Nftables 27

4.3.2 CommScope/CISCO ACL . . . 30
4.3.3 AWS Network ACL . . . 30
4.4 Security Groups . . . 31
4.4.1 AWS . . . 31
4.4.2 Oracle . . . 31
4.4.3 IBM . . . 32
4.5 Differences between Packet Filters, Access Control Firewalls and Security Groups . . . 32

5 Approach 34
5.1 Research . . . 34
5.2 Modelling . . . 34
5.2.1 Access Control Firewall . . . 35
5.2.2 Security Group . . . 37
5.2.3 Comparisons between Access Control Firewalls' inputs/outputs and Security Groups' inputs/outputs . . . 39
5.2.4 Differences with Packet Filters . . . 40
5.3 Implementation, Use Cases and Nftables Serializer . . . 40
5.3.1 Implementation . . . 41
5.3.2 Use Cases . . . 45
5.3.3 Nftables Serializer . . . 49

6 Conclusions 58

Bibliography 60

List of Figures

2.1 Example of Service Function Chain. 16

2.2 Example of Software-Defined Network Infrastructure's representation. 18

2.3 Representation of a Service Function Chain built following the SDN architecture. 19

2.4 Example of Service Function Chain modelled with a NFV SDN architecture. 21

2.5 VEREFOO Model. 23

4.1 Packet processing stages and relative Hooks. 27

5.1 Example of Allocation Graph for AC Firewalls. 35

5.2 Example of Service Graph with the allocated AC Firewalls. 36

5.3 Example of Allocation Graph for Security Groups. 37

5.4 Example of Service Graph with the allocated Security Groups. . . . 38

5.5 Representation of Use Case 01 and 02 network. 45

5.6 Representation of Use Case 01 network. 47

5.7 Representation of Use Case 02 network. 48

5.8 Nftables firewall configuration 1. 52

5.9 Nftables firewall configuration 2. 54

5.10 Nftables firewall configuration 3. 56

List of Tables

5.1 Example of a set of Network Security Requirements. 36
5.2 Example of ACL for incoming traffic, associated to interface f1.2 of the firewall f1. 37
5.3 Example of Security Group's Rules, both for incoming and outgoing traffic. 39

Listings

5.1 AC Firewall XML element. 41
5.2 AC Firewall Interface XML element. 41
5.3 AC Firewall Inbound/Outbound List XML element. 42
5.4 AC Firewall Rule XML element. 42
5.5 Security Group XML element. 43
5.6 Security Group inbound/outbound list XML element. 44
5.7 Security Group Rule XML element. 44
5.8 Example of input XML AC Firewall Configuration 1. 50
5.9 Example of output Nftables configuration file 1. 50
5.10 Example of input XML AC Firewall Configuration 2. 52
5.11 Example of output Nftables configuration file 2. 53
5.12 Example of input XML AC Firewall Configuration 3. 54
5.13 Example of output Nftables configuration file 3. 55
5.14 Example of input XML AC Firewall Configuration 4. 57

Chapter 1

Introduction

1.1 Thesis objective

In the past few years we have seen the emergence of two new technologies used in the network field:
Network Functions Virtualization (NFV) and Software-Defined Networks (SDN).
Network Functions Virtualization is a network architecture concept that refers to the ability to run multiple network function on standard hardware, possibly with virtualization support, in order to optimize resource utilization and allowing the flexible deployment of network functions that are part of a Service Function Chain.
Software-Defined Networks,instead, allows the creation of paths traveled by the packets within the physical network by means of software processes.
Thanks to these two solutions, it is now possible to implement specific network functions that make up the graph, such as NATs or firewalls, on a single machine capable of running multiple network functions simultaneously, instead of relying on ad-hoc hardware devices.
This allows a more efficient use of physical resources since they are shared by more than one service function running on the same server.
Furthermore the process of adding more functions is simplified since buying expensive, dedicated hardware is no longer necessary and only requires the implementation at software level on the already available general purpose server.

In the creation of Service Function Chains, a problem may arise due to the manual configuration of network functions, especially network security functions, since incorrect configurations can easily lead to issues such as breached security or acceptance of unwanted traffic.
Additionally, performing these operations manually is slow and tedious, which can lead to delays in updating security defenses when security requirements are changed or added.
Network Automation thus becomes a valuable alternative by making Network and Security Management an automated process, thanks to the automatic handling of configuration changes, which reduces latency and eliminates the risk of human misconfigurations.
The final objective of the work described in this thesis is to define additional security functions accessible to VEREFOO (VErified REfinement and Optimized

Orchestration), a framework capable of allocating and configuring Network Security Functions to fulfill specific network security requirements. Specifically, in this thesis the following functions have been modelled and developed:

- Access Control: the management of packet access to and from individual hosts and/or entire subnets is carried out through Access Control Firewalls that allow or discard the various packets traversing their interfaces.
- Security Groups: a type of firewall that manages the traffic leaving from or coming to all the hosts' interfaces to which it is assigned.

The framework then automatically allocates these NSFs onto a Service Graph, which represents the logical topology of an end-to-end service, made up of multiple network functions (i.e. load balancers) that do not enforce any security defenses. The solution offered must be optimal while simultaneously satisfying the input Network Security Requirements.

1.2 Thesis description

The remaining sections of this thesis are organized as follows:

- Chapter 2 introduces the concept of Service Function Chain (SFC), a chain of multiple functions, capable of providing services and features to a communication.
  It continues by introducing two recent concept used in computer networks and how they enhance the SFC representation: Software Define Networking (SDN) and Network Function Virtualization (NFV).
  In this chapter final part is then discussed how these concepts work in combination with security network and the benefits they bring by automating the process of setting up security in a given network.
  The chapter closes with an introduction of a framewrok that makes use of these innovation and on which this thesis work was carried on.
- Chapter 3 describes the focus of this thesis work and how it is accomplished by dividing the main objective in smaller parts and a brief description for each one.
- Chapter 4 presents the technologies that are first studied, then modelled and finally implemented in the framework.
  First there is a brief introduction to the already implemented Packet Filters and its characteristics and then it moves on analyzing the different solution considerated in order to create new firewall functions.
  Each solution, both for Access Control Firewalls and Security Groups, is described in detail with their main features and differences.
- Chapter 5 describes the approach used to create these two new firewall functions.

The process is divided in three smaller steps: a research part and the solutions currently used in providing some form of access control, the modelling process used to create their abstract model, with description of inputs and outputs and differences with the Packet Filter model, lastly, the description of how this model was implemented with XML structures, the creation of multiple Use Cases that can be used for testing and the development of an Nftables serializer that takes as input the XML configurations and produces a series of Nftables commands as output to set up a suitable firewall.

- Chapter 6 is the final chapter dedicated to the conclusion of this thesis work, which objective were achieved and their summary and lastly there are some suggestions to improve the framework with additional work based on this thesis.

Chapter 2

Network Automation

In this chapter, the concept is introduced of Service Function Chain (SFC), which refers to a series of network functions that compose a chain designed to provide end-to-end services in a communication. Furthermore the main limitations of this basic representation will be highlighted, such as the lack of agility and flexibility due to the principles of the network architecture on which it is based.
To solve these limitations, two new solutions used in computer networking are thus considered: Software-Defined Network (SDN) and Network Function Virtualization (NFV).
Thanks to SDN it's possible to implement a dynamic determination of the path that a packet must follow through a software-driven forwarding process, improving the flexibility and manageability of the SFC.
The second solution, NFV, takes advantage of the concept of running multiple network function images on standard hardware to achieve greater flexibility for the SFC.

The final part is dedicated to the concept of automating the network and security management using the previously mentioned innovations and an improved SFC is discussed, along an example of framework that utilizes use of this implementation. This approach brings several benefits, especially in a contest where a rapid and automatic response is ideal, such as in network security.

2.1 Service Function Chain

In order to provide end-to-end services, a proper set of functions is required to ensure that the traffic passes through them, following a set order to meet user requirements.
The Service Function (SF) and Service Function Chain (SFC) have been formally defined in RFC 7665 [1]:

- **Service Function**: each is responsible for the managing and processing of packets received in the network. It can operate at different layers of the protocol stack and can be implemented as a virtual component or embedded within a physical network element.

One or more service functions can be integrated into a single network element. Additionally, multiple instances of the same service function can exist within the same administrative domain.

- **Service Function Chain**: defines a predefined sequence of Service Functions along with ordering constraints applied to packets, frames, and flows selected. The position assigned to the chain's network functions influences the end-to-end service and its correct operation, as packet flows must pass through them in a defined order.

Here is presented an example of an end-to-end service between a Web Server and a Web Client implemented via Service Function Chain.

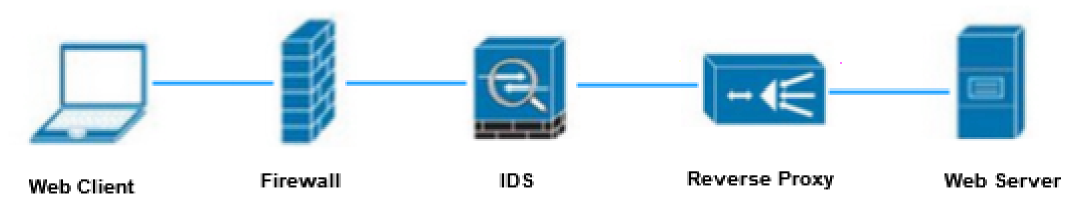

Figure 2.1. Example of Service Function Chain.

On the path between the Client and the Server there are three different functions, from left to right:

1. **Firewall**: manages the traffic according to its configuration.
2. **Intrusion Detection System**: monitors the network and detects possible attacks.
3. **Reverse Proxy**: shields the characteristics of the servers in its local network.

In order to let the packets travel through the service functions in a specific and predetermined order and guarantee the correct working of the end-to-end service, their positions must be fixed.

In the past the network functions were implemented directly on physical devices designed and built with a specific purpose but this led to some limitations affecting the SFC:

- The physical resources are not efficiently used and this results in an imbalance regarding the use of resources. Since each service function was implemented on a separate physical device, it was not possible to share these hardware resources.
- Changes to a SFC, such as adding or removing a SF, resulted in service disruption since the physical devices had to be installed/uninstalled.
- In order to add new functions, purchases of additional dedicated hardware were required, this meant additional costs and time-consuming installation.

- An SFC does not have the flexibility to provide user-specific services.

In order to overcome these issues and improve SFC's flexibility and performance, new paradigms have been developed and used in the recent years.

2.2 Software Defined Network and Network Function Virtualization

2.2.1 Definition of Software-Defined Networks and its characteristics

Software defined Networks are types of network composed of network devices, either physical or virtual, whose packets forwarding and pathing on the network is managed by software that reacts in real time, basing his decisions on the occurrence of certain events by potentially making changes in the packets forwarding.
The three concepts the SDN is based on are:

1. Decoupling the data plane from the control plane.
   The control plane includes all functions and processes that decide on which path to send the various packets/frames.
   The Data plane refers, instead, to the set of functions and processes that have the task of forwarding packets from one interface to another, following the directions given by the control plane logic.

2. The control plane is centralized, concentrating all the intelligence of this technology in one location.
   This centralization can be either logical or physical; however, logical centralization is preferred in order to avoid both single points of failure and scalability issues.

3. Definition of southbound and northbound interfaces.
   The southbound interface is what allows the SDN controller to interact with and handle the network devices.
   The northbound interface, instead, allows the communications between the SDN controller and the user-level application or other higher level controllers.

The image 2.2 shows how the SDN controller interacts with the network devices through the southbound interface and communicates, through the northbound interface, with the user-level applications or a higher-level controller.

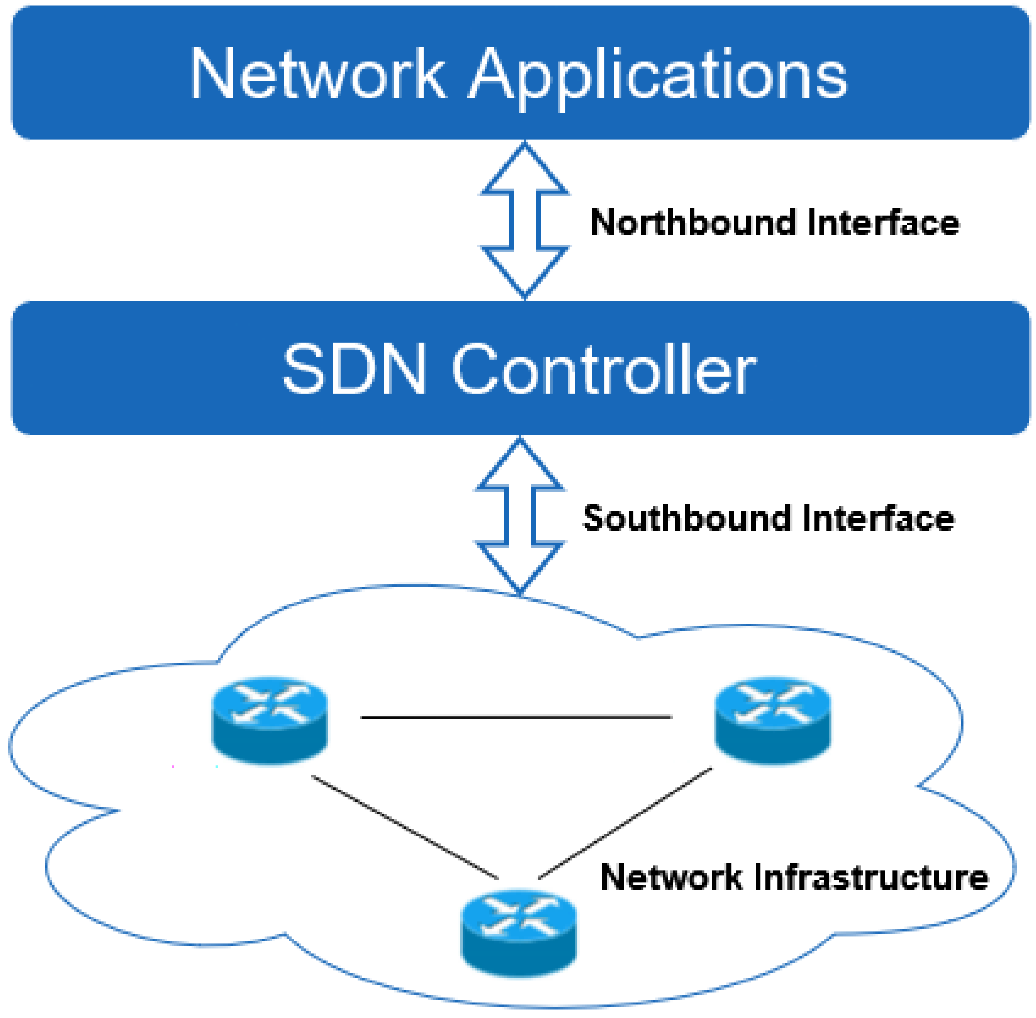

Figure 2.2. Example of Software-Defined Network Infrastructure's representation.

2.2.2 Application of the SDN technology to a SFC

Software-Defined Network (SDN) paradigm allows for overcoming some limitations of the service function chains previously addressed. The service functions that are present in an SFC can be viewed as hardware devices that are interconnected via an SDN switch which is managed by the SDN controller. This setup enables the controller to direct the packet flows and control the order in which the devices are traversed.

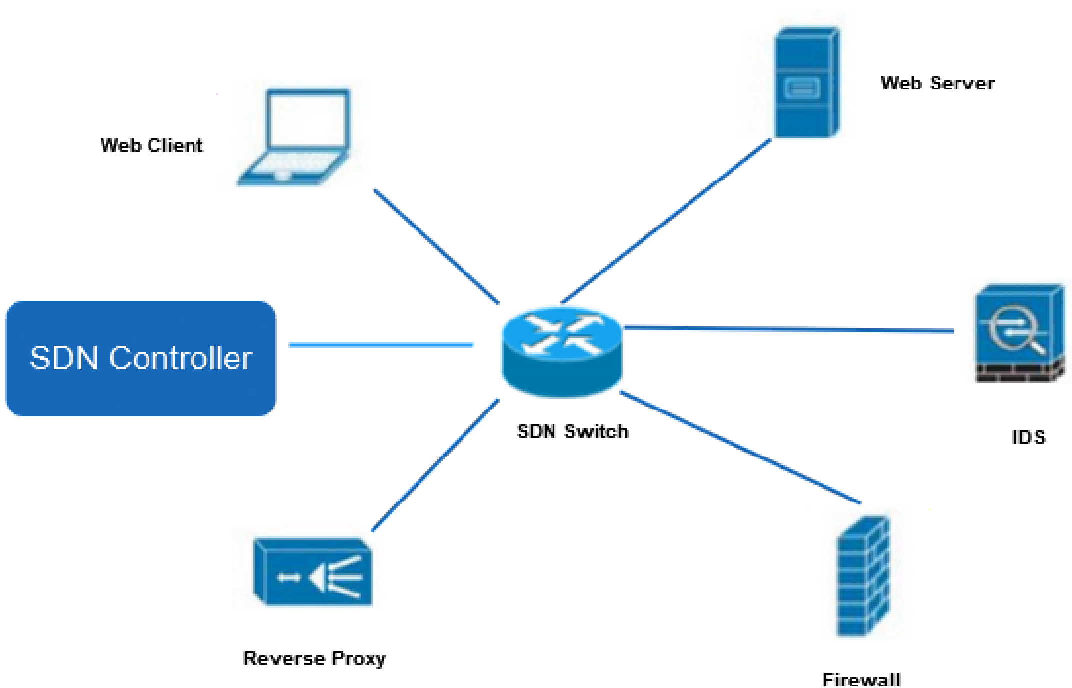

Figure 2.3. Representation of a Service Function Chain built following the SDN architecture. .

The image 2.3 shows how the SFC previously considered can be implemented using an SDN architecture.
This brings many advantages suchs as:

- Agility in provisioning new services: Thanks to software-based routing, new services are available quickly after installing the physical device.
- Maintenance and reliability: Cabling is required only once.
- Differentiated service chains for users: With software-based routing is possible to provide different users with different service functions.
  This opens up the possibility of creating user-dependent service chains.

Still, since these devices are hardware-based, the problem of sharing resources between them remains, along with the inefficiency of installing new functions.

2.2.3 Definition of Network Function Virtualization and its characteristics

Network Function Virtualization (NFV) is the ability to run any network function on standard hardware devices, thanks to computer virtualization, in order to reach a greater level of efficiency regarding the use of resources, as seen in [2].

There are four main characteristics that define NFV:

- Fast standard hardware: With NFV it's possible to make use of cheaper off-the-shelf hardware capable of efficiently running network functions.
- Software-based network function: it's possible to quickly add and remove network functions, since their implementation is performed via software images hosted on one or more servers, rather than implement them as physical devices.
- Computing virtualization: Allows to host and manage multiple network functions together on a single hardware server.
- Standard API: NFV makes use of APIs in order to allow interoperability and facilitate communication between different network functions.

Another key feature of NFV is the ability to automatically scale up or out Virtualized Network Functions (VNFs) when additional resources are needed.
The two different strategies are:

- **Scale Up**: When more resources are allocated to a VNF as needed, such as increasing CPU, memory, or disk space, several problems may arise. This strategy may not be feasible if the limit of resources of the physical server hosting the VNF is reached.
  Additionally, sometimes assigning more resources may not improve the situation. For example, adding more CPUs when the software itself is not designed to exploit the additional cores.
- **Scale Out**: With this strategy the VNF is duplicated multiple times, leading to multiple generation of the same function's instances. Thanks to this approach it's possible to make use of parallelization and enhanced flexibility, allowing traffic to be efficiently partitioned among the different instances thanks to the supervision of a load balancer.

2.2.4 Application of the NFV technology to a SFC

While a Service Function Chain generated using Software-Defined Networking offers various advantages, there are still some limitations due to the reliance on physical devices to implement each Network Function.
Thanks to Virtual Network Functions it's possible to overcome these problems since hardware resources can be now shared between VNFs, additional VNFs can be added at software level instead of requiring the physical addition of more devices. Finally, it's only necessary to duplicate the VNFs in order to provide user-tailored functions.

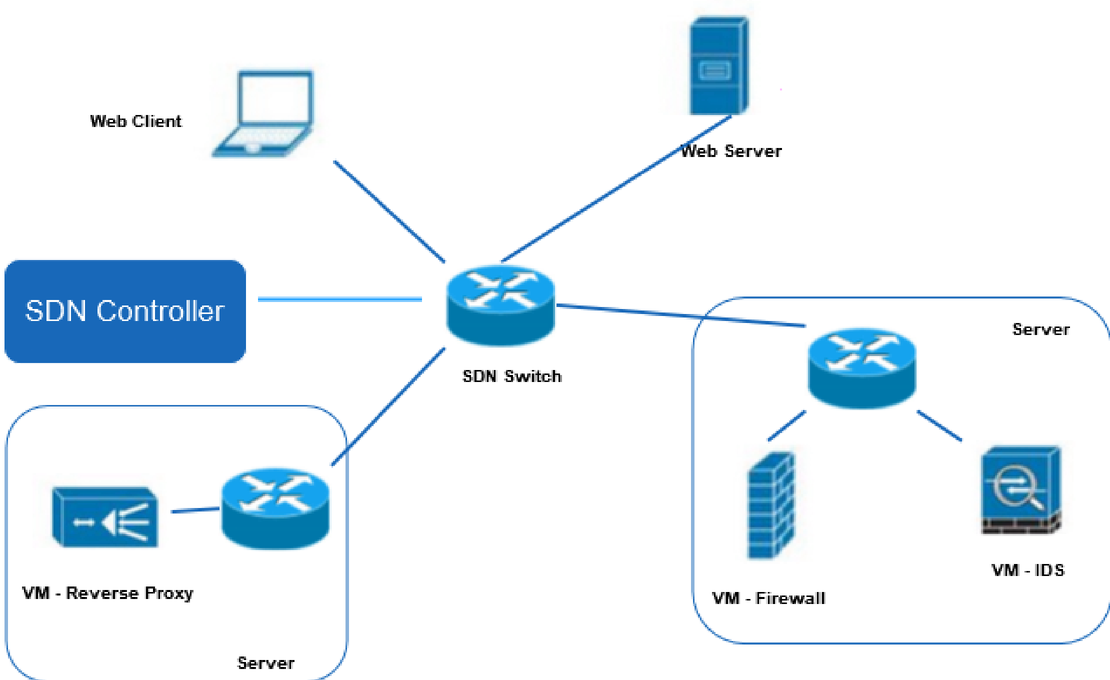

Figure 2.4. Example of Service Function Chain modelled with a NFV SDN architecture. .

2.3 Automatizing Network and Security Management

2.3.1 Definition and Advantages

The new paradigms Network Function Virtualization and Software-Defined Network, previously described, play an important role in the principle of Network Automation.
Network Automation can be described as the practice of automating network operations, manage security and continously monitor the situation thanks to the help of software components.

Its usefulness becomes evident in the presence of new requirements or network's events; the system, in order to answer to this changes, reacts and accordingly modifies the configuration of the network itself. Both the current state and the behavior of the physical level below are taken in consideration when applying these changes.

In the past, hardware-based components were the fundamental components that carried out IT security, requiring manual provisioning and management, and thus limiting the ability to deal with cyber attacks.
In the security field, a rapid response to these incidents is essential to mitigate an increasing range of attacks. It is often difficult or impossible to react fast enough or be adequately prepared in advance without errors when manually configuring devices.
Instead, thanks to an automated network and security management, it's possible to implement automatic configuration changes characterised by lower latencies.

The goal is to not rely on configurations carried out by someone -because that causes interruptions when an event occurs- in order to increase network resiliency; this allows the software to solve problems on its own, retrieve information from the network, and use it to let the network converge on normal behaviour.
It is also important for the system have a comprehensive overview of the entire infrastructure, in order to maintain a properly functioning system capable of automatically reconfiguring the network.

2.3.2 VEREFOO

An example of framework that lets overcome the problems and limitations of a Service Function Chain is VEREFOO (VErified REfinement and Optimized Orchestration).

Its main goal is to receive a set of network security requirements defined by the security administrator and make sure they are satisfied when it allocates and configures Network Security Functions onto a Service Graph, in an automatic and optimal way.
Additionally, it's the framework responsibility to also place the network security functions needed to meet the security constraints on the underlying physical network's servers.

In the figure 2.5 are shown the main parts that compose VEREFOO in order to describe its structure and workflow.

The **Policy ANalysis (PAN)** module checks for conflicts among the requirements of the Network Security Requirements and produces a set composed of only the strictly necessary constraints that must be satisfied in the network.

The **High-to-Medium (H2M)** module job is to translate the high-level Network Security Requirements into a new set of lower level. This new medium-level set is required to both create the Network Security Functions's policies automatically allocated on the graph and configure the lower-level VNFs.

The **NF Selection** module selects the necessary Network Security Functions required to satisfy Security Requirements received as input.

The central part of the VEREFOO framework is the **Allocation, Distribution and Placement (ADP)** module. The intputs it receives are: a set of medium-level network security requirements, the list of the chosen Network Security Functions and either a Service Graph or an Allocation Graph.
A new Service Graph with the new network security functions automatically allocated is then produced as output.
Furthermore, each network security function exploiting a medium level policy language is configured in a way that ensures vendor independence.
All this is done while also ensuring the fulfillment of properties like isolation, reachability, and protection [3, 4, 5, 6, 7] and performing conflict analysis [8, 9, 10]. Finally, the low-level configuration, vendor dependent, is generated starting from the medium level configuration.

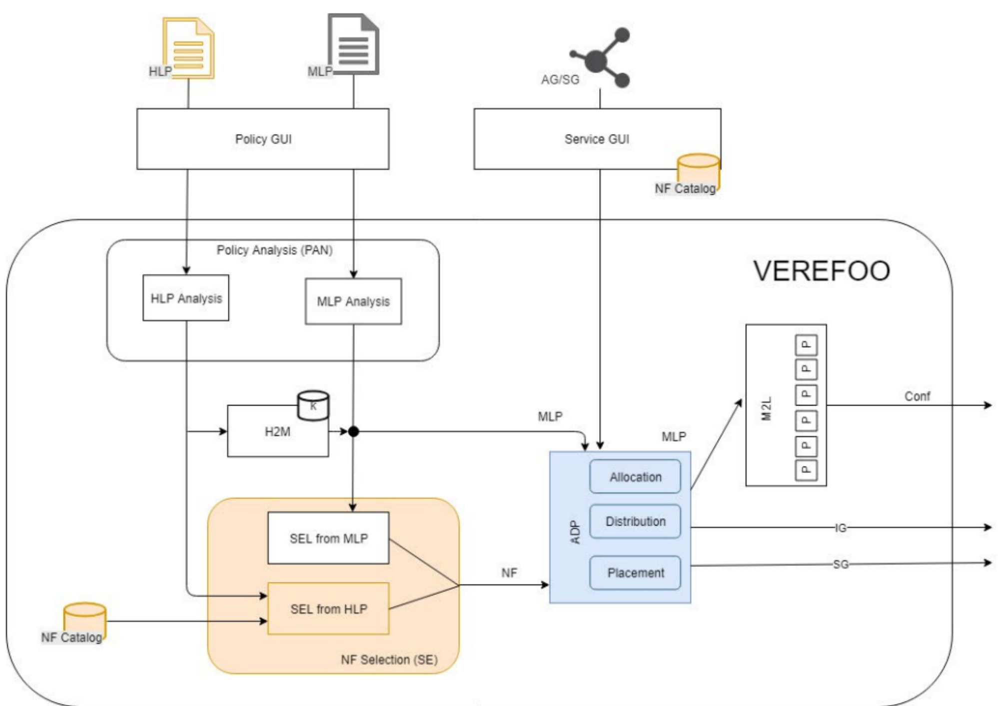

Figure 2.5. VEREFOO Model.

Chapter 3

Thesis Objective

As established in the previous chapters, setting up the security of a network is a complex task that is also critically important.
The difficulty lies in the manual configuration of the network's security features, such as firewalls and VPNs, by network administrators. These tasks require expertise in the security field and a high level of competency, and even in the presence of these, critical errors, sub-optimizations or possible conflicts -generated either between the policies of the same device or between the policies of different devices- often afflict the configuration of these systems.
That's why innovations such as Software Define Network and Network Function Virtualization can help overcome these problems, by automating the process and ensuring flexibility, agility and network resilience.

The main objective of this thesis work was the addition and improvement of VEREFOO possible functions options that it's able to both optimally allocate and configure, while handling any conflicts, in order to satisfy a set of security requirements.
Specifically, the focus was on the research, study, modelling and implementation of Access Control Firewalls and Security Groups; both are a type of firewall capable of filtering packets based on their 5-tuple, with some differences between each other.
This work can be divided into smaller steps, each of which has been essential to achieving the overall goal:

1. An extensive research of the different implementations of Access Control Firewalls (AC Firewalls) and Security Groups (SG) was carried out in order to first understand how they work and and how they differ from each other.

2. The second step consisted in exploring the differences between the AC firewalls, SG and the Packet Filter devices already implemented in VEREFOO, in order to understand how their additional functionalities could be also implemented.

3. The next step was the modelling phase where, based on the informations previously studied, a model of both AC Firewalls and SGs was proposed.
   The modelling consisted in defining the inputs (Allocation Graphs, NSR) and the output (Service Graphs and configurations) of the system.

4. The final step consisted in the implementation of both types of devices as XML elements and of a translator that, given as input the XML configuration of a firewall, produces as output the command lines necessary to implement the firewall with Nftables.

Chapter 4

Firewall technologies

4.1 Packet Filters

In computer networks data is transmitted by means of packets. Each packet has two parts:
the packet header, that contains information about the source IP address, destination IP address, protocol and ports and the packet payload, containing the data to be transmitted.
A packet filter is a firewall technology able to make a decision regarding the forwarding of each incoming packet based on the header information: IP addresses, L4 Protocol and ports of both source and destination, information collocated in the layer 3 (network) and 4 (transport) of the ISO/OSI stack.
The Packet Filter is a firewall function that is available to VEREFOO but it's a firewall technology with some limits; it's not possible, in its current iteration, to implement a form of access control.
That's why this thesis work focuses on the definition and starting implementation of solutions capable of more features like Access Control Firewall and Security Groups.

4.2 Access Control Firewalls and Security Groups introduction

An Access Control Firewall works by allowing or dropping packets going through it and taking the decision based on the header informations, but introduces an additional filtering option given by the interfaces through which the packets enter or leave the device.

A Security Group acts as a stateful virtual firewall that controls the traffic allowed to reach and leave the resources' interfaces that it is associated to. By default, it blocks all the packets that try to reach or leave the instances that it's associated to but allows others explicitly stated to pass through (*whitelisting*).

4.3 Access Control Firewalls

4.3.1 Nftables

Nftables [11] is a Linux kernel packet classification framework, successor of Iptables. Nftables presents some differences compared to Iptables, like a new syntax, support for dual stack, no pre-defined tables and base chains and enhanced generic set and map infrastructure but, other than these and other small differences, they are functionally identical.

Nftables works with rules that specify actions. These rules are grouped in lists called "Chains", each associated to a specific stage of packet processing called "Hooks", and the Chains themselves are in turn stored in "Tables".

The figure 4.1 shows the different Table families and the available Hooks.

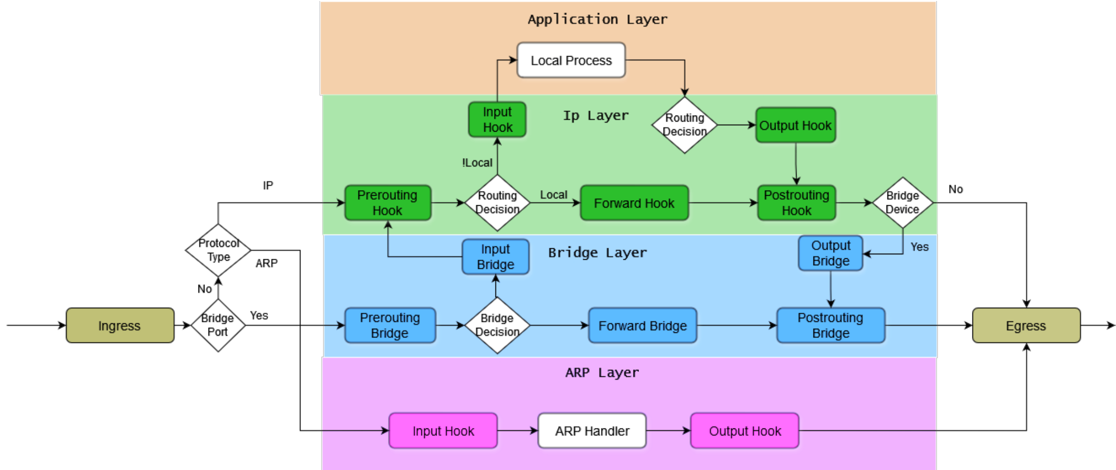

Figure 4.1. Packet processing stages and relative Hooks.

Unlike Iptables, Nftables has no pre-defined tables (*filter, raw, mangle...*) but they must be defined with an address family (default *ip*) and name.
The family indicates the type of packet to process:

- **ip**: Used for IPv4 related Chains.
- **ip6**: Used for IPv6 related Chains.
- **inet**: Mixed IPv4/IPv6 Chains.
- **arp**: Used for ARP related Chains.
- **bridge**: Used for bridging related Chains.
- **netdev**: Used for Chains that filter very early in the stack.

For the purpose of this thesis only *ip, ip6, inet* families will be taken in consideration.

As with the tables, Nftables does not have any predefined Chains. The Chains are classified in "base" and "regular" types.
Base Chains are associated to one of the hooks while regular Chains are not.
Base Chains must be defined with a Hook, type, name, priority and policy.
In contrast, **regular Chains** are not attached to a Hook and they don't see any traffic by default but they can be used as jump targets to arrange a rule-set in a tree of Chains.

The *type* of a Chain indicates which operations will be executed on the packet in consideration and they are:

- **filter**: standard Chain, generally used for filtering. Possible Hooks: all.
- **nat**: Used to perform Native Address Translation. Only the first packet of a connection is considered by this chain. Hooks: prerouting, input, output, postrouting.
- **route**: Packets that traverse this chain type, if about to be accepted, trigger a route lookup if the IP header has changed. Hooks: output.

For the purpose of this thesis only *filter* type will be taken in consideration.

The *priority* is used to for Chain ordering: a Chain can be assigned a number, even negative, that determines the processing order, with lower numbers processed first.
The *policy* determines the default behaviour of the packet if it reaches the end of the Chain, can be either *drop* or *accept*.

Rules specify what action is taken for a given packet and are attached to Chains. Each rule has a unique handle number by which it can be distinguished and can have an expression to match each packet and one or more actions to perform when the packet matches.
Unlike Iptables, it is possible to specify multiple actions per rule, and counters are off by default and must be specified explicitly in each rule for which packet- and byte-counters are desired.
When a packet matches a rule with the *drop* action it gets immediately dropped and it is not allowed through, on the contrary an *allow* rule doesn't stop the examination and the packet continues onto the other rules and Chains.

The following matches are available:

- ip: IP protocol.
- ip6: IPv6 protocol.
- tcp: TCP protocol.
- udp: UDP protocol.
- udplite: UDP-lite protocol.

- sctp: SCTP protocol.
- dccp: DCCP protocol.
- ah: Authentication headers.
- esp: Encrypted security payload headers.
- ipcomp: IPcomp headers.
- icmp: icmp protocol.
- icmpv6: icmpv6 protocol.
- ct: Connection tracking.
- meta: meta properties such as interfaces.

4.3.2 CommScope/CISCO ACL

An Access Control List is a solution used by CommScope [12] and CISCO [13] to filter traffic's packets passing through their switches and routers. They are essentially lists of rules, called Access Control List (ACL), assigned to an interface to check either incoming or outgoing packets.
Two types of ACL exists:

- **Standard**: only the IP source field is checked to filter packets.
- **Extended**: multiple fields are checked:
  - Ip source
  - Ip destination
  - L4 protocol
  - Source ports
  - Destination ports

Each rule in an extended ACL is composed of multiple parameters and an action (*allow/deny*) that is taken when a packet's header parameters match the rule criterias.
When a packet is inspected it will be compared to each rule, in order, until either it matches one of the rules or it reaches the default *deny* implicit policy and is then dealt accordingly.

An ACL is created and applied following the next steps:

1. Create an ACL with **ip access-list** or **ipv6 access-list**.
2. Create the rules inside an ACL with **permit** or **deny**.
3. Apply the ACL to one or more interface with **ip access-group** or **ipv6 access-group**.

4.3.3 AWS Network ACL

NACL [14] is a service offered by AWS to use in their VPC, manageable by UI or command line that allows to handle packets coming out and in a subnet.
While a subnet can be associated to a single NACL, the latter can be associated to multiple subnets.
Each NACL is composed of two lists: one that checks packets coming into the subnet and one that checks packets exiting the subnet.

4.4 Security Groups

4.4.1 AWS

An AWS Security Group (SG) [15] works as a virtual firewall for the instances in the Virtual Private Cloud (VPC) to manage incoming and outgoing traffic thanks to its security rules.
When an instance is initialized, its interfaces can be associated to one or more SG, if not specified the default SG is used.
Its behaviour is to stop all the traffic and its rules are always permissive (*whitelisting*); it's not possbile to create rules to deny passage to some packets.
If more than one SG is associated with an interface, all their rules are pooled together and then evaluated.
When a packet has to be managed, it doesn't stop at the first match, but all the rules are checked and the most permissive one is applied.

Each rule has the following fields:

- **Source/Destination IP**: either one, depends if the rules are for inbound-/outbound traffic. It's possible to specify another Security Group ID, this way the rule is applied to instances associated to the SG.
- **Protocol**: e.g. TCP/UDP/ICMP.
- **ICMP type and code**
- **Source/Destination ports**: either one, depends if the rules are for inbound-/outbound traffic.
- **Description**: Optional.

The SG are **stateful** firewalls, this means that they keep track of the connections involving the hosts and their interfaces associated to the SG by means of a connection state table.
When a packet is inspected by the SG, it checks if the former belongs to an active connection: if it doesn't it's discarded, otherwise it is allowed to pass through regardless of any rule that would otherwise block it.
This is an inherent feature of AWS SGs while the AC firewalls previously seen require additional support to implement it.

4.4.2 Oracle

Oracle's Network Security Groups [16] work similarly to AWS's but their VPCs aren't provided with a default NSG.
Another difference is found in the security rules: while AWS SGs' rules are always stateful, each Oracles's NSG rule can either be declared stateful or stateless.
Oracle's NSGs also have, in their security rules, destination and source ports for both inbound and outbound rule lists while AWS's security lists have either one

based on if we're considering inbound or outbound rules.
Furthermore Oracle makes available another similar service called Security Lists [17].
Security lists differ from NSGs because a default Security List is provided for the VPC and because they work at subnet level: they allow/deny packets using security rules just like SGs, but they manage the traffic directed to all the instances contained in the subnet to which the Security List is associated.

4.4.3 IBM

Like the previous two implementations of security groups, an IBM Cloud Security Group [18] is composed of two sets of IP filter rules, based of the 5-tuple, that define how to handle incoming (ingress) and outgoing (egress) traffic to the interfaces of a virtual server instance.
The rules of a security group are known as security group rules and, like AWS SGs', are stateful.

It's possible to assign one or more security groups to a network interface and the security group rules of each SG apply to the associated virtual server instances. IBM's SGs also follow a *whitelist* policy; this means that, in order to allow inbound-/outbound traffic, some rules that allow the passage of this traffic is required.
The order of rules within a SG does not matter since they are all evaluated against the packet being checked and the priority always falls to the least restrictive rule.

4.5 Differences between Packet Filters, Access Control Firewalls and Security Groups

After the introduction and description of these technologies, it's possible to compare them to the Packet Filter function already implemented in VEREFOO.

Regarding Access Control Firewalls, they don't differ much from Packet Filters, except for a fundamental distinction: AC Firewalls make use of an additional parameter when checking the packets passing through them -the interfaces through which the packet is entering or exiting.
Each interface can have two Access Control Lists, one for incoming packets and one for outoing ones.
This is important because it allows to regulate access to individual hosts or entire subnets, a feature that a simple Packet Filter cannot provide.

Security Groups are different from Packet Filters for multiple characteristics:

- They are virtual firewalls that can be associated to multiple hosts' interfaces, instead of a firewall put between two nodes.
- They block all traffic and their rules are permissive (*whitelisting* policy), while a Packet Filter default policy can be either *allow* or *drop*, like its rules' actions.

- A Security Group has two ACLs, one for inbound traffic and one for the outbound.
- Each time a packet is examined against the rules of a Security Group, all its rules are evaluated and only the most permissive one is considered, in case of multiple matches.
- A Security Group's rule can have other Security Groups IDs as Source or Destination, instead of IP addresses.
- Security Groups are inherently stateful, Packet Filters are not.

Chapter 5

Approach

5.1 Research

The first step of this thesis' work was the research and study of different examples of implementations, regarding AC Firewalls and SGs, and led to the solutions previously described.
This has been done in order to understand how they work and to identify similarities and differences both between the two kind of firewalls and between the various implementations.

Another step of this research was to also determine what differences there were between the Packet Filters already implemented in VEREFOO and this two new firewalls.

This led to consider as possible candidates to study, regarding Access Control Firewalls: *Nftabels* for Linux, the *Access Control Lists* (ACL) used by CISCO and CommScope on their routers and switches devices and the *Network Access Control Lists* (NACL) used by AWS in Virtual Private Networks.
The Security Groups solutions considered were the SGs offered as services on their VPN by AWS, Oracle and IBM.

After their characteristics and differences were identified, a general model for each of these two types of firewalls was then developed.

5.2 Modelling

After the research part was completed, such that the appropriate solutions were studied and the similarities and differences were defined, it was necessary to develop a theoretical model to represent the AC Firewalls and the SGs.
This modelling was done in a way that it didn't represent any specific solution considered but a generic model that made use of their particular and general features.

Each model was developed by defining, first, the input parameters required by the system and then the corresponding outputs produced.

5.2.1 Access Control Firewall

The Access Control Firewall was modelled as a **Network Security Function** (NSF) that can be allocated in a special node called **Allocation Place** found in the Allocation Graph.
For each of these devices, their interfaces connecting them to other nodes are also represented.
On these interfaces it's possible to generate Access Control Lists (ACL) that check and filter traffic packets, incoming and/or outgoing, based on their 5-tuple.

Inputs

The inputs required for the system to decide where to put the AC Firewalls and automatically configure them are:

- An **Allocation Graph**: it's a structure obtained starting from a **Service Graph**.
  The Allocation Graph is an extension of the Service Function Chain concept that models the logical topology of an end-to-end service, excluding functions dedicated to system security. In this new graph, Allocation Places are also incorporated, one for each pair of nodes, to allow the allocation of Network Security Functions.

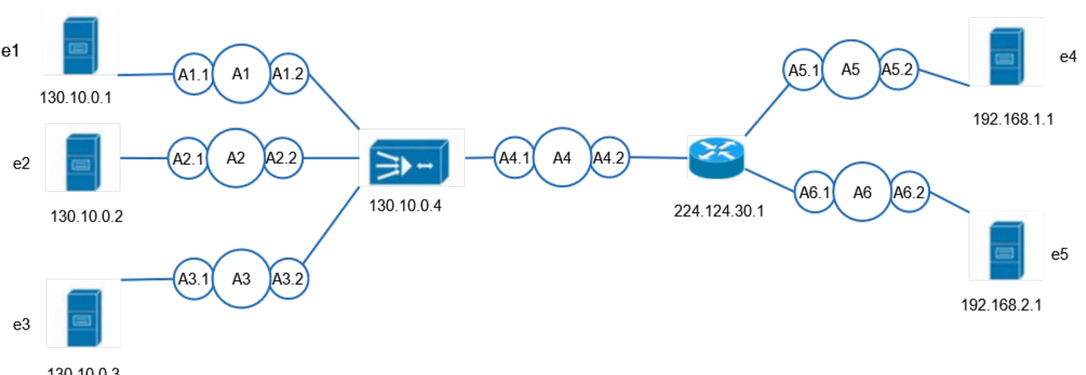

Figure 5.1. Example of Allocation Graph for AC Firewalls.

- A set of **Network Security Requirements**, security constraints that must be satisfied by the automatic allocation of firewalls on the Allocation Graph also received as input.

  Each NSR is defined by:

  - an *action* that specify if the traffic of a communication between two nodes or subnets, matching the IP quintuple parameters, is to be allowed or blocked.

  - the *Ip quintuple* composed of IP source, IP destination, Ports source, Ports destination and the Protocol (TCP, UDP,...).

NETWORK SECURITY REQUIREMENTS					
Action	IPSrc	IPDst	pSrc	pDst	tProto
Allow	192.168.1.1	192.168.2.1	*	*	*
Allow	192.168.2.1	192.168.1.1	*	*	*
Deny	192.168.1.1	130.10.0.*	*	!=80	TCP
Allow	130.10.0.*	192.168.2.1	*	*	*

Table 5.1. Example of a set of Network Security Requirements.

Outputs

The outputs generated by the system after the allocation and configuration of the NSF are:

- a **Service Graph**, where the NSF representing the AC Firewalls and relative interfaces have been allocated in the appropriate APs.

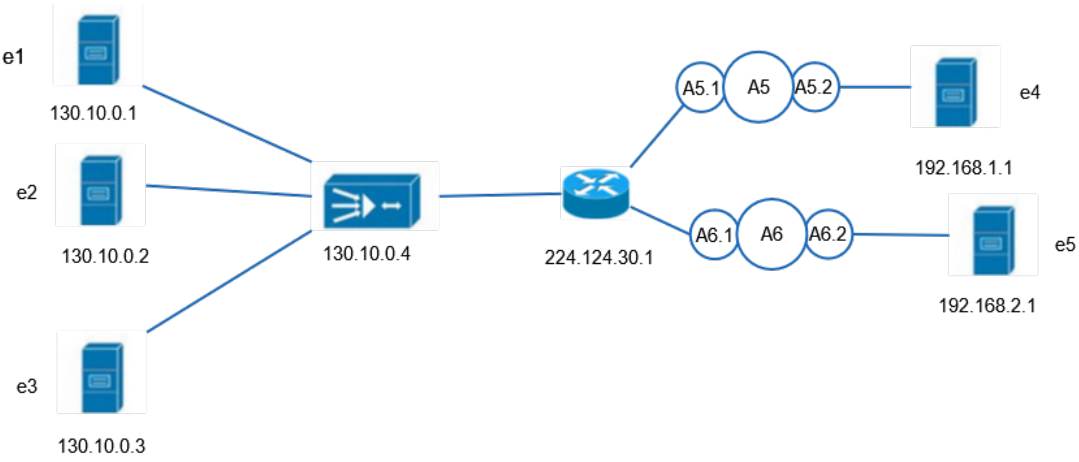

Figure 5.2. Example of Service Graph with the allocated AC Firewalls.

- An **Access Control List** (ACL) for incoming traffic and/or an ACL for outgoing traffic for each interface that might requires it.
  An ACL is a set of rule, each one characterized by an action (*allow* or *deny*), that expresses if the packet matching that rule must be dropped or allowed to pass throught followed by the IP quintuple.
  When a packet matches a rule it is handled accordingly and the inspection stops but each ACL has a default rule that may block or allow the rest of the traffic that doesn't match any of the previous ones.

ACL f1 interface f1.2 IN						
N.	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	192.168.1.1	192.168.2.1	*	*	*
2	Deny	192.168.1.1	130.10.0.*	*	!=80	TCP
D	Deny	*	*	*	*	*

Table 5.2. Example of ACL for incoming traffic, associated to interface f1.2 of the firewall f1.

5.2.2 Security Group

The Security Group was also modelled as a Network Security Function (NSF) that can be allocated in the special nodes Allocation Places found in the Allocation Graph.
Since the Security Groups of a VPC, in general, work as firewalls to the interfaces of the virtual instances or load balancers it is associated to, the AG for SG placements has APs only next to their interfaces.
Each SG has it's own set of Security Group Rules that check and filter traffic packets, incoming and/or outgoing, based on their parameters.

Inputs

The inputs required for the system to decide where to put the SGs and automatically configure them are:

- An **Allocation Graph** with its APs located next to virtual instances and load balancers' interfaces.

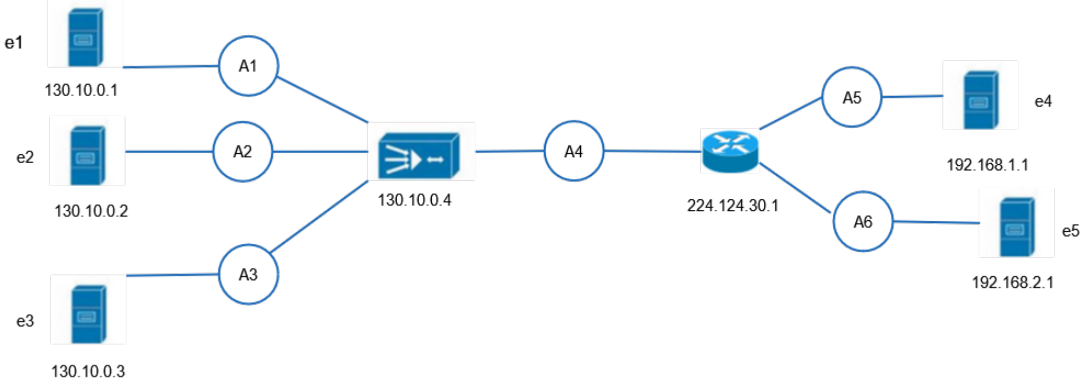

Figure 5.3. Example of Allocation Graph for Security Groups.

- A set of **Network Security Requirements**, security constraints that must be satisfied by the automatic allocation of firewalls on the Allocation Graph also received as input.
  Each NSR is defined by:

- an *action* that specify if the traffic of a communication between two nodes or subnets, matching the IP quintuple parameters, is to be allowed or blocked.

- the *Ip quintuple* composed of IP source, IP destination, Ports source, Ports destination and the Protocol (TCP, UDP,...).

Outputs

The outputs generated by the system after the allocation and configuration of the NSF are:

- a **Service Graph**, where the NSF representing the SG have been allocated in the appropriate APs.

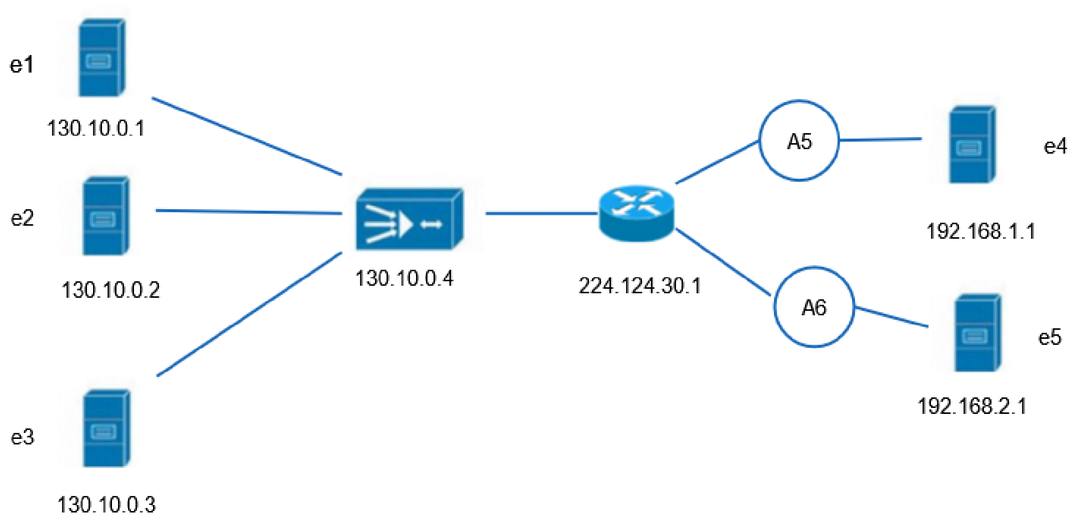

Figure 5.4. Example of Service Graph with the allocated Security Groups.

- A **Set of Security Rules** for incoming traffic and/or a set for outgoing traffic for each SG that might requires it.
  Every Security Rule is implicitly an *allow* rule, since SGs follow a *whitelisting* policy: they block all the traffic except the one explicitly allowed.
  Each rule is characterised by an *IP address* (*source address* if the rule checks inbound traffic, *destination address* if the rule checks outbound traffic), *Ports source*, *Ports destination* and the *Protocol*.
  When a packet is inspected it is matched against all the rules and it doesn't stop at the first matching rule. After all the rules are evaluated only then the most permissive, matching rule is considered. Each SG list of rules, both for incoming and outgoing traffic has a default rule that block all the traffic that doesn't match any of the previous rules.

SG 1 INBOUND			
IPSrc	pSrc	pDest	tProto
192.168.2.1	*	*	*
SG 1 OUTBOUND			
IPDst	pRange	pDest	tProto
192.168.2.1 130.10.0.*	* 80	* *	* *

Table 5.3. Example of Security Group's Rules, both for incoming and outgoing traffic.

5.2.3 Comparisons between Access Control Firewalls' inputs/outputs and Security Groups' inputs/outputs

In the last section was described the AC Firewalls' and Security Groups' modelling proposed in order to better represent them as Network Security Functions.

The Network Security Requirements is an input, used by the system to automatically place and configure these NSFs, that they share.
It's a set of constraints, characterised by a policy and an IP 5-tuple, that has to be satisfied and it is expressed the same way for both the options.

The other input is the Allocation Graph, similar for both the firewall options but with the difference that is the placement of the Allocation Places:
while for the SG they are located next to the virtual instances' and load balancers' interfaces, for the AC Firewalls they are allocated to each pair of nodes, unless indicated otherwise by the security administrator.

Regarding the outputs, one of the two is the Service Graph, an AG with all the necessary NSFs configured and allocated in the appropriate APs.
This output is conceptually the same for both the NSFs but we can find the same Security Group in different Places since multiple interfaces can be assigned to the same Security Group.

The other output is the configuration of each of the firewall configured in the Service Graph.
The configuration of an AC consists in a set of rules (ACL) characterised by the IP 5-tuple and an action to be taken in the event of a matching packet. Each interface of the firewall can have two set of rules that checks only for the type of traffic specified, one for the incoming and one for the outgoing traffic.
The order in which these rules are listed it's important since the handling of the packet that is then discarded or allowed through stops at the first matching rule.
Another feature of these sets of rules is the presence of an implicit default rule that may, depending on the configuration, allow through or discard all the packets that don't match any of the previous rules.

The Security Groups' configuration is different from the other one:
for each Security Group exists two set of security rules, one for the inbound traffic that is coming to the interfaces associated to the firewall and one for the outbound traffic that is leaving the interfaces.

Each rule allows the packets that match the listed parameters: IP source (destination for outbound traffic), Source and Destination Ports and Protocol. Rules that block traffic are not necessary since this kind of firewall follow a *whitelisting* policy and blocks by default all the traffic that it doesn't let through.
A feature that is possible to use with the Security Groups and not with the AC Firewall is the use of Security Groups IDs in place of IP Source and Destination; this possibly for a better management of different groups of IPs.
Another difference is the fact that, contrary to the ACL used by the AC Firewalls, all the rules are evaluated before applying the most permissive one.

5.2.4 Differences with Packet Filters

The Packet Filter Network Security Function already implemented in VEREFOO has limited capabilities when it comes to filter traffic packets. They work by analysing packets that traverse them and checking them against a single ruleset.

While their ruleset is fundamentally the same as the one used by an Access Control Firewall, with a default action, the rule's action and the IP-tuple, they don't make the distinction on which interface the packet passing through and its direction.
This makes it impossible to conduct a thorough access control to individual devices or subnets that is not just a check on every packet that passes through the Packet Filter, based exclusively on the IP tuple.

The Security Groups differ from the Packet Filters since they are basically a couple of ruleset (one for inbound and one outbound traffic) that can be associated to multiple web interfaces, instead of a device that checks and filter every packet passing through it.
Another difference lies in the rules themselves since they don't require an order and don't need expressly an action to follow (they always let through the packet matching atleast one rule) and they also require only the IP Source (inbound ruleset) or IP Destination (outbound ruleset).

5.3 Implementation, Use Cases and Nftables Serializer

The following step of this thesis' work can be split up in three smaller parts:

1. The first thing approached was how to represent the functions as abstract entities and this was done by creating suitable elements in a .xsd file.

2. The next step was the creation of testfiles .xml that represented small, configured networks, with their validity checked against the previous developed .xsd elements, and, following the conclusion of step three, the correct functioning of the latter.

3. The final step consisted in the development of a firewall serializer, with focus on a firewall configured by means of **Nftables**, that converted an Access Control Firewall configuration in multiple Nftables commands.
   Its working was then checked on the tests previously written.

5.3.1 Implementation

The first phase covers the study and creation of suitable .XML elements that represent accurately the function considered, but kept generic enough to still be able to represent their multiple solutions and real-life implementation.

Access Control Firewall Implementation

Regarding Access Control Firewall, its .xml implementation was done by first creating a structure that represented the firewall function itself:

```
<xsd:element name="access_control_firewall">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="ac_interface_elements" minOccurs="0"
maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Listing 5.1. AC Firewall XML element.

In this data structure, called *access_control_firewall*, there is a simple collection of elements called *ac_interface_elements*.
Each one of them represents the interfaces of the device that connect it to other nodes in the network. It goes from 0 to infinite, although both these extreme cases are unlikely to occur.

Each *ac_interface_elements* component is defined in the following way:

```
<xsd:element name="ac_interface_elements">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="ac_inbound_list" minOccurs="0" maxOccurs="1"
/>
            <xsd:element ref="ac_outbound_list" minOccurs="0"
                maxOccurs="1" />
        </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="description" type="xsd:string" use="optional"/>
    </xsd:complexType>
</xsd:element>
```

Listing 5.2. AC Firewall Interface XML element.

This data structure represents a single interface of the AC Firewall that connects it to the other adjacent nodes in the network.
Each Interface element is characterized by two attributes and a list composed of two elements:

1. The first attribute is the name given to the interface (i.e. "eth0"). It's used in order to easily refer to a particular interface in case of necessity.

2. The second attribute is a simple description of the interface (i.e. "Interface that connects to node xyz").

3. One of the elements of the list is called *ac_inbound_list* and is in itself a list; it contains the rules that allow or block matching packets inbound to the interface.

4. The other element is another list called *ac_outbound_list*, similarly to the other one it allows or blocks passage to the packet leaving through the interface.

The two lists have a limit of minimum 0 and at max 1, since multiple lists of the same traffic direction are superfluous.
Both *ac_inbound_list* and *ac_outbound_list* are defined in the same way:

```
<xsd:element name="ac_inbound_list">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="elements" minOccurs="0"
maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="defaultAction" type="ActionTypes"/>
    </xsd:complexType>
</xsd:element>
```

Listing 5.3. AC Firewall Inbound/Outbound List XML element.

Each inbound/outbound list has an attribute *defaultAction* that states what is the default action (allow/drop) that occurs when a packet reaches the end of the list, thus doesn't match any of the previous rules.
In the real-life implementations previously studied the default action was typically an implicit "drop everything" rule but, since it is easy to bypass this restriction by adding an "allow everything" rule as the last explicit one, it was opted to add a choice for the default action.

The other parameter of this data structure is the list of elements representing a rule (min 0, max infinite):

```
<xsd:element name="elements">
    <xsd:complexType>
        <xsd:sequence>
   <xsd:element name="id" type="xsd:long" minOccurs="0" />
            <xsd:element name="action" type="ActionTypes" minOccurs="0"
                default="DENY"/>
            <xsd:element name="source" type="xsd:string"/>
            <xsd:element name="destination" type="xsd:string" />
            <xsd:element name="protocol" type="L4ProtocolTypes"
                minOccurs="0" default="ANY"/>
            <xsd:element name="src_port" type="xsd:string" minOccurs="0"/>
            <xsd:element name="dst_port" type="xsd:string" minOccurs="0"/>
            <xsd:element name="priority" type="xsd:string" minOccurs="0"
                default="*"/>
            <xsd:element name="directional" type="xsd:boolean"
                minOccurs="0" default="true"/>
```

```
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Listing 5.4. AC Firewall Rule XML element.

This element was already developed in previous works but is still suitable to be used by the AC Firewalls, except for the *directional* field.
The other fields are:

- The *id* by which a rule can be referred with; can be useful if a specific rule need to be deleted or a rule has to be added before/after another one.
- The action to take in case of a matching packet.
- Source and Destination IP addresses.
- L4 Protocol (TCP, UDP, ANY).
- Source and Destination port(s).
- Priority, useful to order the rules since the evaluation stops at the first match.

Security Group Implementation

The Security Group function followed a similar implementation but with some differences.

First we have the main structure:

```
<xsd:element name="security_group">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="sg_inbound_list" minOccurs="0" maxOccurs="1"
/>
            <xsd:element ref="sg_outbound_list" minOccurs="0"
                maxOccurs="1" />
        </xsd:sequence>
    <xsd:attribute name="defaultAction" type="ActionTypes" fixed="DENY"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    </xsd:complexType>
</xsd:element>
```

Listing 5.5. Security Group XML element.

It's characterized by a couple of attributes and a list composed itself by two other lists:

1. The first attribute is the *ID* by which the Security Group can be referred to in the rules of its own or other SG rules.
2. The *Default Action* that is always *DENY* since the policy followed by all the SGs is a *whitelisting* one: all the traffic is blocked except the packets expressly permitted.

3. One of the two lists is called *sg_inbound_list* and represents the list containing rules that manages the traffic incoming to the interfaces associated to the Security Group.

4. The other is called *sg_outbound_list* and represents instead the list of rules that manages the traffic that exits the interfaces associated to the Security Group.

Each list is then implemented in the same way:

```
<xsd:element name="sg_inbound_list">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="sg_inbound_elements" minOccurs="0"
maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Listing 5.6. Security Group inbound/outbound list XML element.

It's a simple list of newly defined *sg_inbound_elements* (*sg_outbound_elements*), each of them representing a rule with different parameters:

```
<xsd:element name="sg_inbound_elements">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="id" type="xsd:long" minOccurs="0" />
            <xsd:element name="action" type="ActionTypes" minOccurs="0"
                fixed="ALLOW"/>
            <xsd:element name="source" type="xsd:string" minOccurs="0"/>
            <xsd:element name="protocol" type="L4ProtocolTypes"
                minOccurs="0" default="ANY"/>
            <xsd:element name="src_port" type="xsd:string" minOccurs="0"/>
            <xsd:element name="dst_port" type="xsd:string" minOccurs="0"/>
            <xsd:element name="description" type="xsd:string"
                minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Listing 5.7. Security Group Rule XML element.

These rule elements are different from the AC Firewall's since they work on sligthly different parameters:

- An *id*, in case the rule has to be referred directly, i.e. for a change in its parameters or to delete it.
- The *action*, fixed on *ALLOW* because of the *whitelisting* policy.
- The *IP source address* (*destination* for outbound rules).
- The eventual *L4 Protocol*.
- *Source and destination ports*.
- An optional *description*.

5.3.2 Use Cases

Some Use Cases were developed for future testing, both for Security Groups and Access Control Firewalls.
The Access Control Firewalls Use Cases were further exploited to develop a serializer capable of converting a firewall configuration taken from a .xml Use Case file and provide as output Nftables commands necessary to set up a real working firewall.
Following there are a couple of examples for each of the security functions:

AC Firewall

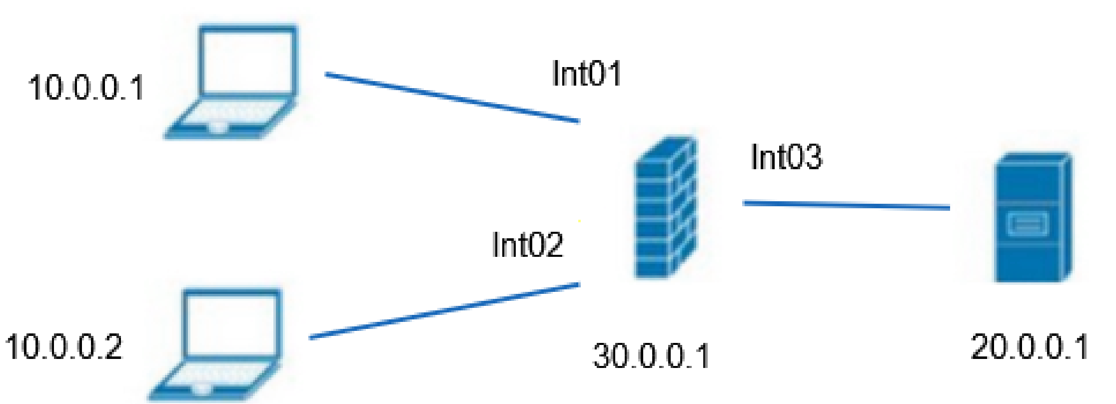

Figure 5.5. Representation of Use Case 01 and 02 network.

In this UC there are the following properties that need to be satisfied:

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

An example of configuration can then be:

```
<access_control_firewall >
           <ac_interface_elements name="Interface01"
               description="Interface␣on␣WEBCLIENT␣10.0.0.1">
             <ac_inbound_list defaultAction="DENY">
               <elements>
                 <action>ALLOW</action>
                 <source>10.0.0.1</source>
                 <destination>20.0.0.1</destination>
                 <protocol>ANY</protocol>
                 <src_port>*</src_port>
                 <dst_port>*</dst_port>
               </elements>
             </ac_inbound_list>
             <ac_outbound_list defaultAction="ALLOW">
```

```
                <elements>
                  <action>DENY</action>
                  <source>20.0.0.1</source>
                  <destination>10.0.0.1</destination>
                  <protocol>ANY</protocol>
                  <src_port>*</src_port>
                  <dst_port>*</dst_port>
                </elements>
              </ac_outbound_list>
            </ac_interface_elements>
          </access_control_firewall>
```

If the network structure remains the same but the properties to satisfy change in:

```
<PropertyDefinition>
      <Property graph="0" name="ReachabilityProperty" src="10.0.0.1"
          dst="20.0.0.1" lv4proto="TCP" src_port="80" dst_port="80"/>
  <Property graph="0" name="ReachabilityProperty" src="10.0.0.1"
      dst="20.0.0.1" lv4proto="UDP" src_port="68" dst_port="68"/>
</PropertyDefinition>
```

The firewall configuration will be different and it changes in:

```
<access_control_firewall >
  <ac_interface_elements name="Interface01" description="Interface␣
      on␣WEBCLIENT␣10.0.0.1">
    <ac_inbound_list defaultAction="DENY">
      <elements>
        <action>ALLOW</action>
        <source>10.0.0.1</source>
        <destination>20.0.0.1</destination>
        <protocol>TCP</protocol>
        <src_port>80</src_port>
        <dst_port>80</dst_port>
      </elements>
      <elements>
        <action>ALLOW</action>
        <source>10.0.0.1</source>
        <destination>20.0.0.1</destination>
        <protocol>UDP</protocol>
        <src_port>68</src_port>
        <dst_port>68</dst_port>
      </elements>
    </ac_inbound_list>
    <ac_outbound_list defaultAction="DENY"/>
  </ac_interface_elements>
</access_control_firewall>
```

Security Group

In this network representation the Security Group 01 is associated to the webserver's interface and is represented as an entity for clarity purpose.
An example of properties to satisfy is:

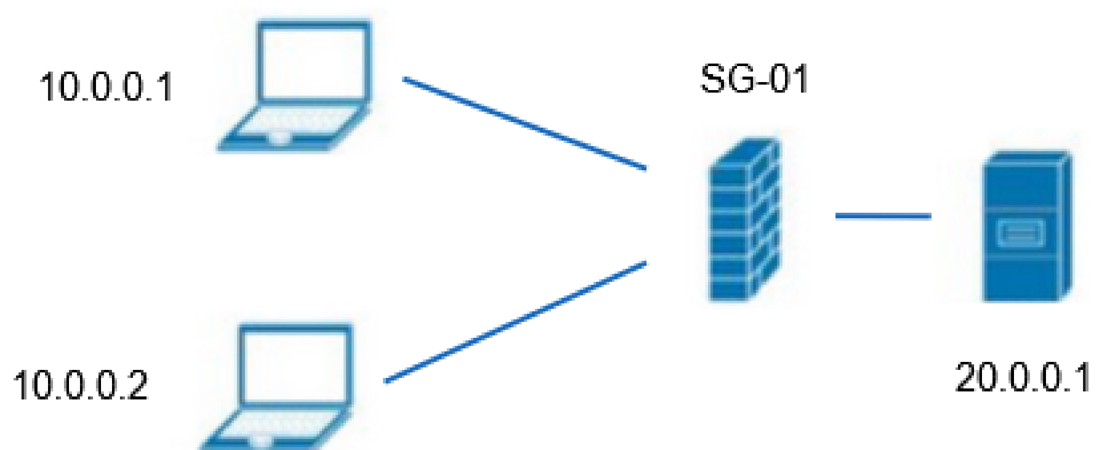

Figure 5.6. Representation of Use Case 01 network.

```
<PropertyDefinition>
            <Property graph="0" name="ReachabilityProperty" src="10.0.0.1"
                dst="20.0.0.1"/>
  <Property graph="0" name="ReachabilityProperty" src="20.0.0.1"
      dst="10.0.0.1"/>
</PropertyDefinition>
```

And an example of configuration for the SG-01 that satisfies the properties is:

```
         <security_group defaultAction="DENY" id="sg-01">
                      <sg_inbound_list>
                        <sg_inbound_elements>
                          <action>ALLOW</action>
                          <source>10.0.0.1</source>
                          <protocol>ANY</protocol>
                          <src_port>*</src_port>
                          <dst_port>*</dst_port>
                        </sg_inbound_elements>
                      </sg_inbound_list>
                      <sg_outbound_list>
                        <sg_outbound_elements>
                          <action>ALLOW</action>
                          <destination>10.0.0.1</destination>
                          <protocol>ANY</protocol>
                          <src_port>*</src_port>
                          <dst_port>*</dst_port>
                          </sg_outbound_elements>
                        </sg_outbound_list>
                      </security_group>
      </configuration>
```

Notice the fixed *DENY* Default Action and the *ALLOW* Action of the rules. These options have their values fixed this way because of the *whitelisting* policy that all the SG follow.

Another network example is:

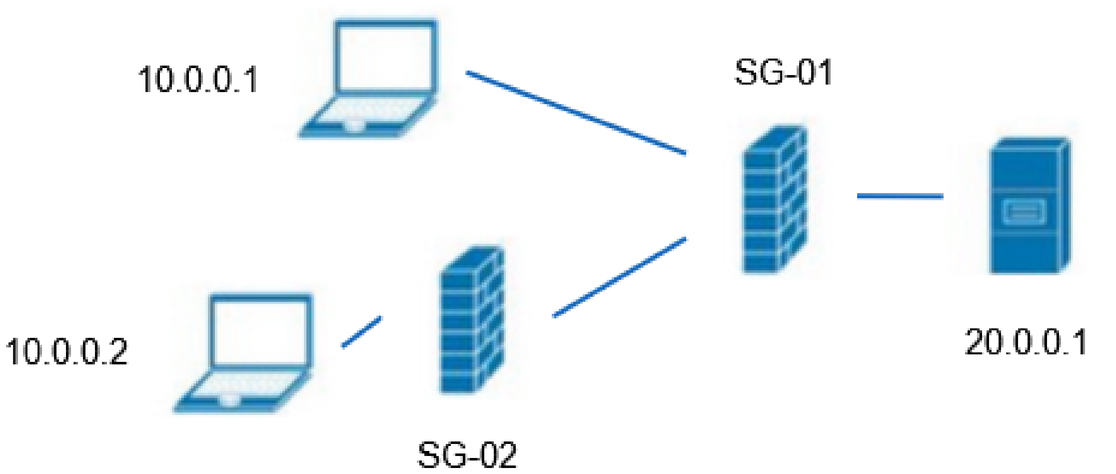

Figure 5.7. Representation of Use Case 02 network.

With the following properties:

```
<PropertyDefinition>
            <Property graph="0" name="ReachabilityProperty" src="10.0.0.1"
                dst="20.0.0.1"/>
  <Property graph="0" name="ReachabilityProperty" src="20.0.0.1"
      dst="10.0.0.1"/>
</PropertyDefinition>
```

And the following configuration regarding SG-01 associated to the webserver's interface:

```
  <security_group defaultAction="DENY" id="sg-01">
    <sg_inbound_list>
      <sg_inbound_elements>
        <action>ALLOW</action>
        <source>sg-02</source>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </sg_inbound_elements>
    </sg_inbound_list>
    <sg_outbound_list>
      <sg_outbound_elements>
        <action>ALLOW</action>
        <destination>10.0.0.1</destination>
        <protocol>ANY</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
      </sg_outbound_elements>
    </sg_outbound_list>
</security_group>
```

And SG-02, associated to the webclient 10.0.0.2 interface:

```
<security_group defaultAction="DENY" id="sg-02">
  <sg_inbound_list>
    <sg_inbound_elements>
      <action>ALLOW</action>
      <source>sg-01</source>
      <protocol>ANY</protocol>
      <src_port>*</src_port>
      <dst_port>*</dst_port>
    </sg_inbound_elements>
  </sg_inbound_list>
  <sg_outbound_list>
    <sg_outbound_elements>
      <action>ALLOW</action>
      <destination>sg-01</destination>
      <protocol>ANY</protocol>
      <src_port>*</src_port>
      <dst_port>*</dst_port>
      </sg_outbound_elements>
    </sg_outbound_list>
</security_group>
```

Notice the *source* designated by the SG-01 id to show this feature. It still possible to use IP addresses but the SG IDs make it easier to manage multiple devices, without the need of a rule for each one.

5.3.3 Nftables Serializer

This section is dedicated to discuss the development of a serializer that is capable of translating the Access Control Firewall configuration into a series of commands for the Nftables framework, which sets up a firewall.
This translator consists in a java class that takes as input the AC firewall's configuration, written as an XML object, and produces as output a series of Nftables commands in a script file.
This file can then be run on a Linux machine to automatically set up an actual firewall that is configured in the same way of the XML configuration.

While Nftables allows the option to set up a firewall on the endpoints by creating Chains associated to *input* and *output* Hooks instead of the *forward* one, in this Nftables serializer, all the firewall configurations taken as input are considered as the configurations of AC firewall functions placed between two other functions.

Subsequently are shown some examples of translation, all based on the network showed in the figure 5.5, each with a different configuration to demonstrate its capabilities.

The first firewall is configured with an interface that presents both an inbound access control list and an outbound access control list:

```
<access_control_firewall >
          <ac_interface_elements name="Interface01"
            description="Interface␣on␣WEBCLIENT␣10.0.0.1">
           <ac_inbound_list defaultAction="DENY">
             <elements>
               <action>ALLOW</action>
               <source>10.0.0.1</source>
               <destination>20.0.0.1</destination>
               <protocol>ANY</protocol>
               <src_port>*</src_port>
               <dst_port>*</dst_port>
             </elements>
           </ac_inbound_list>
           <ac_outbound_list defaultAction="ALLOW">
             <elements>
               <action>DENY</action>
               <source>20.0.0.1</source>
               <destination>10.0.0.1</destination>
               <protocol>ANY</protocol>
               <src_port>*</src_port>
               <dst_port>*</dst_port>
             </elements>
          </ac_outbound_list>
        </ac_interface_elements>
      </access_control_firewall>
```

Listing 5.8. Example of input XML AC Firewall Configuration 1.

The Nftables serializer will then produce, as output, the following script file containing the series of commands necessary to set up a working firewall that respects the policies found in the input configurarion:

```
nft add table ip t_esempio

nft add chain ip t_esempio c_Interface01 '{ type filter hook
    forward priority 0; policy accept ;}'

nft add chain ip t_esempio IN_Interface01

nft add rule ip t_esempio c_Interface01 meta iifname Interface01
    jump IN_Interface01

nft add rule ip t_esempio IN_Interface01 ip saddr 10.0.0.1/32 ip
    daddr 20.0.0.1/32 accept

nft add rule ip t_esempio IN_Interface01 drop

nft add chain ip t_esempio OUT_Interface01

nft add rule ip t_esempio c_Interface01 meta oifname Interface01
    jump OUT_Interface01
```

```
nft add rule ip t_esempio OUT_Interface01 ip saddr 20.0.0.1/32
    ip daddr 10.0.0.1/32 drop

nft add rule ip t_esempio OUT_Interface01 accept
```

Listing 5.9. Example of output Nftables configuration file 1.

From the listing 5.13 it's possible to see how the AC Firewall configuration is adapted to the Nftables framework:

1. First, a Table, in this example called *t_esempio* and filtering the ipv4 traffic, is created.
2. Subsequently, for each firewall's interface, a Chain is created; these Chains will be of *filter* type, associated to the *forward* Hook, with default policy set to *accept*.
   The choice of Hook is given by the fact that this device that is currently considered is placed between two nodes and it analyzes the packets passing throught it.
3. After these Chains are created, at most two other Chains are created, one for the list of rules that checks the inbound traffic (IN_Interface01) on that interface and another for the traffic leaving the same interface (OUT_Interface01). If the list(s) isn't present no Chain for it is created.
   These Chains are Regular Chains, they are not associated to any Hook, don't have any Default Action and they only inspect a packet when it is sent to them from other Chains' rules.
   They are very useful to better organize the Chains structure, render it more efficient and easier to read.
4. After this, two rules that forward the packets coming to or leaving from the interface to their respective Regular Chain are added inside the Interfaces' Chains.
   When a packet comes throught an interface it matches the rule with the parameter "meta iifname InterfaceName" and is sent to the designated Regular Chain to further check the other parameters of the 5-tuple.
   The same happens to packets leaving the interface and the rule with parameter "meta oifname InterfaceName".
5. The last procedure is to add the necessary rules inside the Regular Chains representing the Access Control Lists.
   A particular rule is present at the end, in each Chain, after all the others; since the Regular Chains are missing a policy of their own policy, this rule is necessary to act as Default Action, to let the packet through or drop it.

The next figure shows the Nftables firewall configuration set up on a Linux machine thanks to the script produced by the translator.

```
table ip t_esempio {
        chain c_Interface01 {
                type filter hook forward priority filter; policy accept;
                iifname "Interface01" jump IN_Interface01
                oifname "Interface01" jump OUT_Interface01
        }

        chain IN_Interface01 {
                ip saddr 10.0.0.1 ip daddr 20.0.0.1 accept
                drop
        }

        chain OUT_Interface01 {
                ip saddr 20.0.0.1 ip daddr 10.0.0.1 drop
                accept
        }
}
```

Figure 5.8. Nftables firewall configuration 1.

Following there's a configuration where *Interface01* presents both inbound and outbound lists, with rules that check on packets with specific protocols and ports.

```xml
<access_control_firewall >
  <ac_interface_elements name="Interface01" description="Interface␣
    on␣WEBCLIENT␣10.0.0.1">
    <ac_inbound_list defaultAction="DENY">
      <elements>
        <action>ALLOW</action>
        <source>10.0.0.1</source>
        <destination>20.0.0.1</destination>
        <protocol>TCP</protocol>
        <src_port>80</src_port>
        <dst_port>80</dst_port>
      </elements>
      <elements>
        <action>ALLOW</action>
        <source>10.0.0.1</source>
        <destination>20.0.0.1</destination>
        <protocol>UDP</protocol>
        <src_port>68</src_port>
        <dst_port>68</dst_port>
      </elements>
    </ac_inbound_list>
    <ac_outbound_list defaultAction="DENY"/>
  </ac_interface_elements>
</access_control_firewall>
```

Listing 5.10. Example of input XML AC Firewall Configuration 2.

And its relative output script file:

```
nft add table ip t_esempio

nft add chain ip t_esempio c_Interface01 '{ type filter hook
    forward priority 0; policy accept ;}'

nft add chain ip t_esempio IN_Interface01

nft add rule ip t_esempio c_Interface01 meta iifname Interface01
    jump IN_Interface01

nft add rule ip t_esempio IN_Interface01 ip saddr 10.0.0.1/32 ip
    daddr 20.0.0.1/32 tcp sport 80 tcp dport 80 accept

nft add rule ip t_esempio IN_Interface01 ip saddr 10.0.0.1/32 ip
    daddr 20.0.0.1/32 udp sport 68 udp dport 68 accept

nft add rule ip t_esempio IN_Interface01 drop

nft add chain ip t_esempio OUT_Interface01

nft add rule ip t_esempio c_Interface01 meta oifname Interface01
    jump OUT_Interface01

nft add rule ip t_esempio OUT_Interface01 drop
```

Listing 5.11. Example of output Nftables configuration file 2.

1. First, a Table, in this example called *t_esempio* and filtering the ipv4 traffic, is created.
2. Second, the Chains associated to Interface01 are created and populated with the jump rule, in order to delegate the check on the packets to the Regular Chains.
3. Finally, in each Regular Chain, rules are created that block or let through the packets, this time by also checking their protocol and ports.
   At the end of each Chain a rule is added to work as Default Action.

And following is the Nftables configuration:

```
table ip t_esempio {
        chain c_Interface01 {
                type filter hook forward priority filter; policy accept;
                iifname "Interface01" jump IN_Interface01
                oifname "Interface01" jump OUT_Interface01
        }

        chain IN_Interface01 {
                ip saddr 10.0.0.1 ip daddr 20.0.0.1 tcp sport 80 tcp dport 80 accept
                ip saddr 10.0.0.1 ip daddr 20.0.0.1 udp sport 68 udp dport 68 accept
                drop
        }

        chain OUT_Interface01 {
                drop
        }
}
```

Figure 5.9. Nftables firewall configuration 2.

Next there's an example of configuration with multiple interfaces, one with an inbound and outbound list and another with a outbound list.
The rules present an additional check on specific protocols and ports.

```xml
<access_control_firewall >
                        <ac_interface_elements name="Interface01"
                            description="Interface on WEBCLIENT 10.0.0.1">
                          <ac_inbound_list defaultAction="ALLOW">
                            <elements>
                              <action>DENY</action>
                              <source>10.0.0.1</source>
                              <destination>20.0.0.1</destination>
                              <protocol>TCP</protocol>
                              <src_port>80</src_port>
                              <dst_port>80</dst_port>
                            </elements>
                          </ac_inbound_list>
                          <ac_outbound_list defaultAction="ALLOW">
                            <elements>
                              <action>DENY</action>
                              <source>20.0.0.1</source>
                              <destination>10.0.0.1</destination>
                              <protocol>TCP</protocol>
                              <src_port>80</src_port>
                              <dst_port>80</dst_port>
                            </elements>
                          </ac_outbound_list>
                        </ac_interface_elements>
                        <ac_interface_elements name="Interface03"
                            description="Interface on WEBSERVER">
                          <ac_outbound_list defaultAction="ALLOW">
                            <elements>
                              <action>DENY</action>
                              <source>10.0.0.2</source>
                              <destination>20.0.0.1</destination>
```

```
                        <protocol>ANY</protocol>
                        <src_port>*</src_port>
                        <dst_port>*</dst_port>
                    </elements>
                </ac_outbound_list>
            </ac_interface_elements>
        </access_control_firewall>
```

Listing 5.12. Example of input XML AC Firewall Configuration 3.

And as output there's the following script file:

```
nft add table ip t_esempio

nft add chain ip t_esempio c_Interface01 '{ type filter hook
    forward priority 0; policy accept ;}'

nft add chain ip t_esempio IN_Interface01

nft add rule ip t_esempio c_Interface01 meta iifname Interface01
    jump IN_Interface01

nft add rule ip t_esempio IN_Interface01 ip saddr 10.0.0.1/32 ip
    daddr 20.0.0.1/32 tcp sport 80 tcp dport 80 drop

nft add rule ip t_esempio IN_Interface01 accept

nft add chain ip t_esempio OUT_Interface01

nft add rule ip t_esempio c_Interface01 meta oifname Interface01
    jump OUT_Interface01

nft add rule ip t_esempio OUT_Interface01 ip saddr 20.0.0.1/32
    ip daddr 10.0.0.1/32 tcp sport 80 tcp dport 80 drop

nft add rule ip t_esempio OUT_Interface01 accept

nft add chain ip t_esempio c_Interface03 '{ type filter hook
    forward priority 0; policy accept ;}'

nft add chain ip t_esempio OUT_Interface03

nft add rule ip t_esempio c_Interface03 meta oifname Interface03
    jump OUT_Interface03

nft add rule ip t_esempio OUT_Interface03 ip saddr 10.0.0.2/32
    ip daddr 20.0.0.1/32 drop

nft add rule ip t_esempio OUT_Interface03 accept
```

Listing 5.13. Example of output Nftables configuration file 3.

1. First, a Table, in this example called *t_esempio* and filtering the ipv4 traffic, is created.
2. Then the Chains associated to Interface01 are created and populated with the jump rule, in order to delegate the check on the packets to the Regular Chains.
3. In each Regular Chain, rules are created that block or let through the packets and at the end of each a rule is added to work as Default Action.
4. The same is then done for the *Interface03*.

And here's the relative Nftables configuration:

```
table ip t_esempio {
        chain c_Interface01 {
                type filter hook forward priority filter; policy accept;
                iifname "Interface01" jump IN_Interface01
                oifname "Interface01" jump OUT_Interface01
        }

        chain IN_Interface01 {
                ip saddr 10.0.0.1 ip daddr 20.0.0.1 tcp sport 80 tcp dport 80 drop
                accept
        }

        chain OUT_Interface01 {
                ip saddr 20.0.0.1 ip daddr 10.0.0.1 tcp sport 80 tcp dport 80 drop
                accept
        }

        chain c_Interface03 {
                type filter hook forward priority filter; policy accept;
                oifname "Interface03" jump OUT_Interface03
        }

        chain OUT_Interface03 {
                ip saddr 10.0.0.2 ip daddr 20.0.0.1 drop
                accept
        }
}
```

Figure 5.10. Nftables firewall configuration 3.

In the final listing is shown an example of erroneous configuration: two inbound lists are defined for *Interface03*.

```
<access_control_firewall >
            <ac_interface_elements name="Interface03"
                description="Interface␣on␣WEBSERVER">
              <ac_inbound_list defaultAction="DENY">
                <elements>
                  <action>ALLOW</action>
                  <source>10.0.0.2</source>
                  <destination>20.0.0.1</destination>
                  <protocol>ANY</protocol>
                  <src_port>*</src_port>
                  <dst_port>*</dst_port>
                </elements>
            </ac_inbound_list>
            <ac_inbound_list defaultAction="ALLOW"/>
          </ac_interface_elements>
        </access_control_firewall>
```

Listing 5.14. Example of input XML AC Firewall Configuration 4.

In this case no output script file will be produced since an erroneous configuration may lead to serious problems.

Chapter 6

Conclusions

During this thesis work two new firewalls functions have been studied, modelled and implemented. These firewalls allow the security administrator to provide access control capabilities to a network and its nodes.
While these new functions provide similar capabilities, their differences allow them to be used together in order to reach a better security level via some form of defense in depth.

Despite the already implemented and working firewall function representing a Packet Filter, its features don't allow the set up of access control policies.
Its filtering capabilities are limited to only accept or drop packets passing through it by means of a single list of rules.
In order to allow the ADP module of VEREFOO options to automatically allocate and configure firewalls capabale of access control, implementation of Security Groups and Access Control Firewalls was considered.
Access Control Firewalls were implemented as functions that are located, similarly to the Packet Filters, between two generic nodes. Unlike Packet Filters, Access Control Firewalls make use of multiple Access Control Lists, lists of rules that manage the flow of the packets that match one of them by checking their IP-tuple composed of IP source and destination, the eventual L4 Protocol and source and destination Port(s).
Another important difference with the Packet Filters is the fact that these Access Control Firewalls also take into consideration the interfaces through which the packets arrive and/or leave. For each interface that requires it, a list of rules that accept or drop the matching incoming packets and a list for the ones leaving it are created.
The other firewall function, Security Group, is a firewall that, like the Access Control Firewalls, also exploits two Security Lists for incoming and outgoing traffic packets but takes into consideration only the IP source for the former and the IP destination for the latter. The difference with the other functions is the fact that Security Groups work by filtering the traffic that concerns the interfaces to which they are applied, the whitelisting policy that blocks all traffic except for the packets allowed by its security rules, and their inherently stateful property that allows responses belonging to a connection regardless of any rule that may otherwise block it. Furthermore, while in the other two firewalls the scan of a packet stops at the first matching rule, the Security Lists first evaluate all of them and then apply

the most permissive one. Lastly, another feature of the rules is the fact that it's possible to use, instead of IP source and destination, other Security Group IDs; this allows for a more efficient security ruleset and better management of groups of IP addresses.

In order to implement these new security functions, first thing to do was to conduct an in-depth research on the various solutions available and currently in use.
Once these solutions had been identified, their documentations studied and their differences and similarities analyzed, both in relation to each other and to the Packet Filters, it was necessary to establish an abstract model that represented them as a function in an Aloccation Graph.
Of this model, the inputs were defined, consisting of an Allocation Graph with the Allocation Places needed to place the functions and the set of Network Security Requirements, and the outputs, consisting of the Service Graph with the allocated functions and their configurations, set up in order to satisfy the initial Network Security Requirements.

The next step involved implementing these functions as XML entities. This was achieved by using the Packet Filter data structure as a foundation and then enhancing it with additional features developed from scratch, like interfaces.
Following this implementation, some Use Cases were developed to validate the XML structures and as input for future testing.

The final step involved developing a serializer, specifically a translator capable of taking as input one of the configurations present in the Use Cases. This translator would then generate a list of commands that, when executed on a Linux machine with the Nftables framework, configures a firewall in accordance with the policies defined in the firewall's XML configuration.

A possible future development for the ADP module of VEREFOO could involve the implementation of stateful features, which are inherently characteristic of Security Groups and can be implemented on the other Access Control Firewalls by using additional structures such as a Connection State Table.
Another possibility could be the possibility of a firewall setup on one of the end-nodes (an option possible with Nftables), rather than placing the security functions between two generic nodes.
Additionally, further improvements could include the introduction of more Network Security Functions to enhance the overall capabilities of the framework.

Bibliography

[1] J. M. Halpern and C. Pignataro, “Service function chaining (sfc) architecture,” RFC 7665, pp. 1–32, 2015, [Online]. Available: https://doi.org/10.17487/RFC7665.

[2] D. Bringhenti, R. Sisto, and F. Valenza, “A novel abstraction for security configuration in virtual networks,” *Comput. Networks*, vol. 228, p. 109745, 2023.

[3] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, “Automated optimal firewall orchestration and configuration in virtualized networks,” in *NOMS*. IEEE, 2020, pp. 1–7.

[4] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, “Automated firewall configuration in virtual networks,” *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, pp. 1559–1576, 2023.

[5] S. Bussa, R. Sisto, and F. Valenza, “Security automation using traffic flow modeling,” in *NetSoft*. IEEE, 2022, pp. 486–491.

[6] D. Bringhenti and F. Valenza, “Optimizing distributed firewall reconfiguration transients,” *Comput. Networks*, vol. 215, p. 109183, 2022.

[7] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, “Introducing programmability and automation in the synthesis of virtual firewall rules,” in *NetSoft*. IEEE, 2020, pp. 473–478.

[8] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, “Improving the formal verification of reachability policies in virtualized networks,” *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 1, pp. 713–728, 2021.

[9] F. Valenza and M. Cheminod, “An optimized firewall anomaly resolution,” *J. Internet Serv. Inf. Secur.*, vol. 10, no. 1, pp. 22–37, 2020.

[10] C. Basile, D. Canavese, A. Lioy, and F. Valenza, “Inter-technology conflict analysis for communication protection policies,” in *CRiSIS*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8924, pp. 148–163.

[11] Netfilter Project, “Nftables man page,” Available online: https://www.netfilter.org/projects/nftables/manpage.html, 2024, accessed: 2024-10-04.

[12] CommScope, “Fastiron security guide,” Available online: https://docs.commscope.com/bundle/fastiron-08095-securityguide/page/GUID-7B3F3989-7694-499D-A828-9AF2D6FB8169.html, 2024, accessed: 2024-10-04.

[13] Cisco Systems, “Access lists,” Available online: https://www.cisco.com/c/en/us/support/docs/security/ios-firewall/23602-confaccesslists.html, 2024, accessed: 2024-10-04.

[14] Amazon Web Services, “Vpc network acls,” Available online: https://

docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html, 2024, accessed: 2024-10-04.

[15] ——, "Amazon vpc security groups," Available online: https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html, 2024, accessed: 2024-10-04.

[16] Oracle, "Network security groups," Available online: https://docs.oracle.com/en-us/iaas/Content/Network/Concepts/networksecuritygroups.htm, 2024, accessed: 2024-10-04.

[17] ——, "Security lists," Available online: https://docs.oracle.com/en-us/iaas/Content/Network/Concepts/securitylists.htm, 2024, accessed: 2024-10-04.

[18] IBM Cloud, "Ibm security groups," Available online: https://cloud.ibm.com/docs/security-groups?topic=security-groups-about-ibm-security-groups, 2024, accessed: 2024-10-04.