POLITECNICO
DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Automated Policy Enforcement in Software Defined Networking and Network Function Virtualization Environment

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

Candidate
Antonio Amoroso

Academic Year 2020-2021

Summary

The increasing spreading of the amount of exchanging data and the dynamic deployment of applications and services lead to an evolution of the traditional network technology. One possible solution is based on virtualization, in particular, exploiting the Network Function Virtualization (NFV) parading. It is an architectural approach which aim is to decouple the network functions and the hardware appliances, making possible the deployment of network service on general-purpose servers, achieving flexibility during the design of a particular service.

A problem that arises is the service design that usually is performed manually, and this can lead to errors, especially if the service under analysis is related to security functions, such as firewalls. In order to avoid these errors, an automated approach should be used. In this context, it is possible to use a policy-based model that can be refined and translated.

Because of this consideration, this thesis focussed on security inside NFV, in particular, analyzing packet filter behavior and it contributed to the translation from a medium language policy to a low-level configuration taking care of different firewall languages used in different scenarios. Moreover, it contributed to the development of VEREFOO (VErified REFinement and Optimized Orchestration), a framework that aims to provide a Security Automation approach as a solution to the problem highlighted before. Previously inside this framework is already perform the refinement of the policy from high-level language to the medium level one generating also policy configured as IP quintuple (source address, destination address, transport layer protocol, source port, destination port). This work implements a new model used as a base for the translation and it focused on enforcing the configuration generated into Iptables, IpFirewall, BPF-iptables, Open vSwitch, and Fortinet.

The reason under these choices is that Iptables is the standard packet filter inside the Linux kernel and it is also one of the most widely spread, IpFirewall is a packet filter based on FreeBSD operating system and it is the core for other well-known and widely used firewall solutions. The BPF-iptables firewall is used in the extended Berkeley Packet Filter (eBPF) context and developed by this university in order to achieve better performance than the previously defined Iptables and Open vSwitch that is used in Software Defined Networking (SDN) working with OpenFlow protocol. Fortinet is a physical firewall making possible the application of this tool also in a mixed environment and create an opening for future scenarios.

The implementation is finally tested in different network scenarios, finding that most of the translations developed are acting in the same way that is described by

the medium-level abstraction model. In particular, the best results are achieved with IpFirewall, Iptables, and BPF-iptables. The module provides a RESTful API that ensures connectivity to other modules inside the framework. For future works, it can be extended to other types of firewalls and implements different submodules for new packet filters. Moreover, can be implemented a machine learning algorithm that can effectively choose the right packet filter to deploy according to the hardware resources of the machine and the environment in which it should be deployed.

Acknowledgements

I would like to acknowledge everyone who played a role in my academic accomplishments. Firstly, my parents, who supported me since the beginning of my studies. Secondly i would like to thank my academic supervisors Riccardo Sisto, Fulvio Valenza and the PhD student Daniele Bringhenti, which of whom have provided patient advice and guidance throughout the research process. Thank you all for your unwavering support.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Thesis objective

Network technologies are having an increasing impact on our lives not only because of the spreading of smartphones but also other smart devices used for different purposes. This lead to a particular interest in networking and new paradigms arise in order to overcome the traditional way of networking due to the changes of the users which is related to. One of them is *Network Function Virtualization* that united with *Software Defined Networking* want to bring a new approach making an architecture capable to scale and independent from specific vendor appliances. So security inside the network became a popular topic to deal with and since it is configured by humans, it can lead to critical issues of the network, that is the reason why in these years several efforts are put into the automation of security devices in different environments.

The objective of this thesis is to implement a module capable to perform a consistent multi-language translation among several packet filters that are available in the market. The main idea is to develop a data model that is consistent with the already existing framework, for example, the framework VEREFOO (*VErified REFinement and Optimized Orchestration*) which is developed in order to achieve a policy refinement in *Network Function Virtualization* environment. Moreover, another goal that wants to achieve this work is to be able to implement a completely automated solution that can solve many issues related to packet filter configuration, especially for non-expert users.

1.2 Thesis description

The structure of the thesis, excluding this first chapter which has the aim to do an overview of the objective that this research want to reach are:

- Chapter 2 that describes the *Software Defined Networks* and *Network function Virtualization* that are the main area in which this research starts its investigation and the new paradigms for networking.

- Chapter 3 describes the main network security system used during the whole development of this work: the packet filter. Moreover, it presents also the main aspects of a firewall and the policy-based configuration architecture that is the main point for the development of the module. In the end, are analyzed different techniques in literature useful for preliminary researches.
- Chapter 4 describes the tools used during the thesis work, especially underling the main aspects that lead to the choice of particular software or hardware.
- Chapter 5 that describes the data model used as input for the policy translation and are exposed in terms of that particular language as *XML* file, the use cases that will be used to check the correctness of the configuration of the packet filters.
- Chapter 6 describes the main ideas under the development of the code by means of the java class and how they overcome some problems related to the different software chosen to deploy.
- Chapter 7 describes the result of the module showing the configuration obtained and critical issues had over the testing phase.
- Chapter 8 describes the main goals achieved and the further improvement proposed for future researches.
- Appendix A that presents the REST APIs to interface with this module for every framework capable of produce an input compatible to ingress point of translation.
- Appendix B that presents the settings where is it possible to perform the test that is the same environment where the tests of this research were performed.

Chapter 2

Background

2.1 Software Defined Networking

Software Defined Networking is a new paradigm that tries to overcome the traditional IP networks' main problems that are static and rigid. These problems arise because layering the network helps to transport packets but the topology that it is created is complex and difficult to manage and it is in contrast with the incredible evolution of the technology and the needs of the companies in terms of larger networks and data center management.

Moreover, traditional network architecture is made up of three planes that are embedded in a single physical device that belongs to specific vendors, not providing enough space for innovation. The three planes are the data plane that is in charge of forwarding and delivering packets, the control plane that is a management unit capable of applying rules and action on incoming packets, management plane that is where are defined network policies.

To overcome this problem network engineers try to develop *Software Defined Networking* in an open-source environment. Even if it is an open-source project, several organizations tried to standardize it. For example *Open Networking Foundation* (ONF) that is a user-driven organization that is divided in three community that are working on defining standards and specification in order to make possible application of *Software Defined Networking* into the real world. *European Telecommunication Standard Institute* (ETSI) develops standards for *Software Defined Networking* and *Network Function Virtualization* according to needs of industry. *Institute of Electrical and Electronics Engineers* (IEEE) is focused on Standardization and interoperability of *Software Defined Networking*.

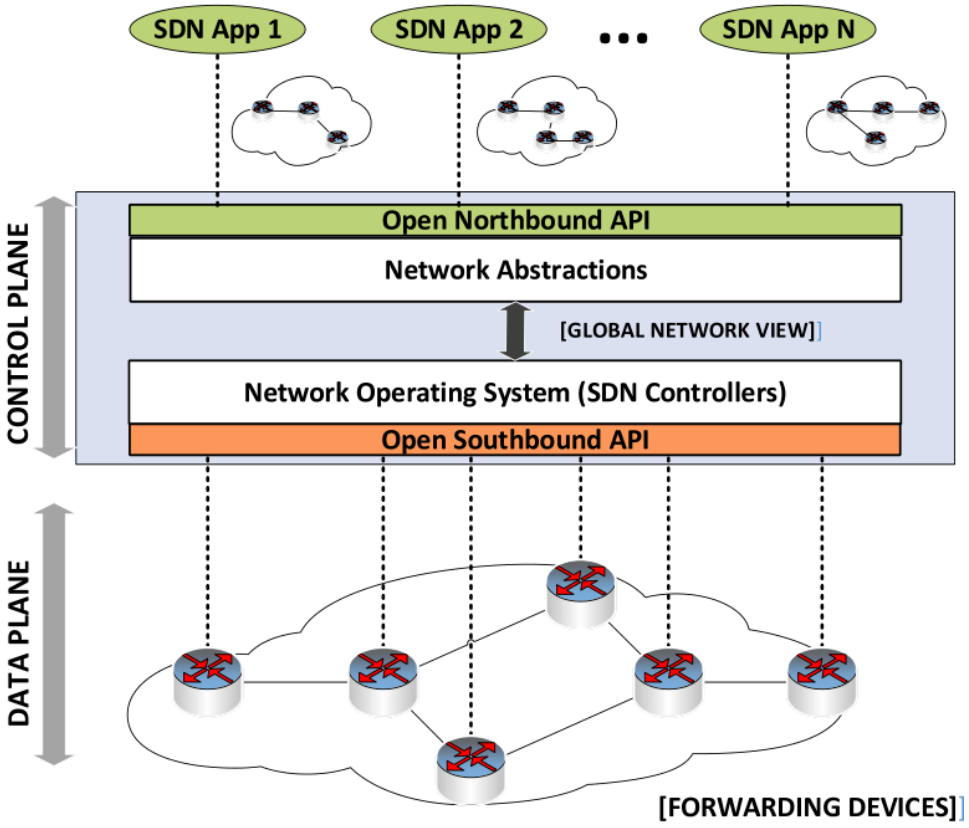

Figure 2.1. Basic architecture of SDN. This image is taken from [1]

2.1.1 Key concept

The first time that comes out *Software Defined Networking* was in 2011 by Martin Casado et al.[7] referring to a network architecture where networks are programmable and able to be flexible, agile, and virtualize. Lately were defined the main concepts that are the pillars of modern networking and are:

- *Separation between the data plane and control plane*: In *Software Defined Networking* based networks having forwarding elements, the data plane, and control logic, the control plane, separated makes it possible to build a simpler architecture.

- *Logical centralization of control*: The control plane is centralized in the controller or *Network Operating System* that has a global view of the network that can be used by the virtualization layer to create abstract views of the physical network to a different control program that belongs to different developers.

- *Flow based control*: The forwarding rules are based on flow entries that are stored in tables inside a switch. If a packet matches one flow of the table, an

action is performed. Otherwise, it is sent to the controller, so the flow can be summarized as a sequence of the packet between source and destination.

- *Programmability*: In *Software Defined Networking* are developed *API* for the controller that is used both for network application and services like load balancer and firewall. Thanks to this also networking is transforming into a software discipline that needs its level of abstraction.

Abstraction is also the main feature that permits *Software Defined Networking* technology to divide network control problems into small pieces easier to manage. Can be identified three-level that are needed in order to achieve a global view of the network. One of them is the distribution abstraction that is in charge to protect the network mechanisms from the problem related to distribution architecture. Only the control logic is centralized, not the physical devices. Another is the configuration abstraction in which it is specified all the behavior that the network has to assume, but it is not in charge of implementing directly that behavior on the physical infrastructure. Moreover, it provides a simplified model of the network making it possible focusing on specific goals that have to achieve. Virtualization plays a fundamental role in this kind of abstraction. The last level of abstraction is the forwarding one that offers an *API* for programming network hardware hiding all the details of the hardware making possible a solution free from specific vendors.

2.1.2 Architecture

Software Defined Networking can be split into three layers like traditional networks: data plane, control plane, and application layers. In this subsection, it will be described an idea of each layer with some of the possible solutions that can be implemented. A general idea can be done using figure 2.2 that will find plenty of explanation below.

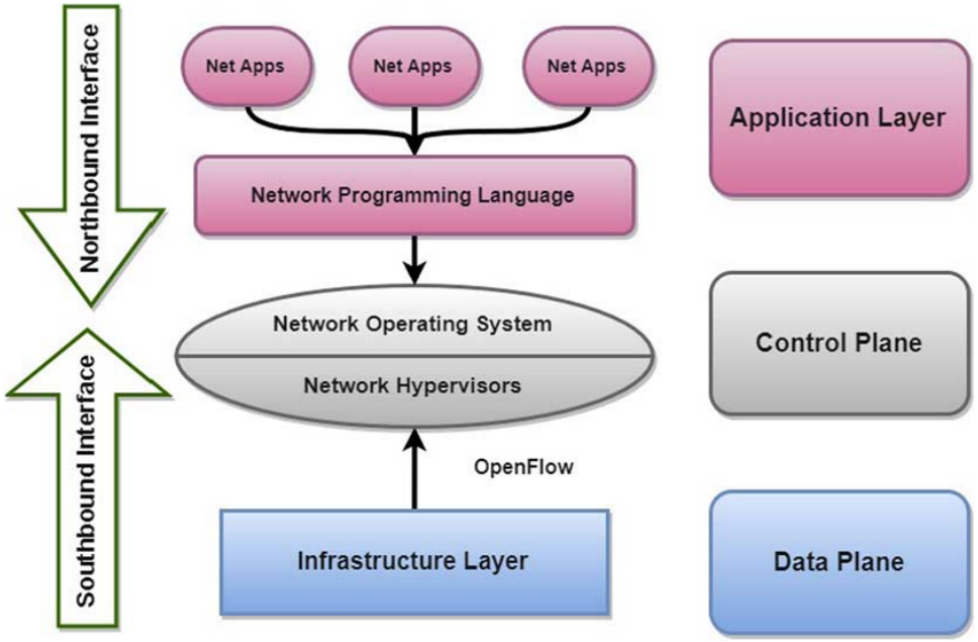

Figure 2.2. Schema of the components of SDN architecture. This image is taken from [2]

The Data plane is the physical network architecture that is made up of all the hardware devices like switches, routers, cables, middleboxes, and so on. In the Software Defined Networking framework, all the middlebox equipment like firewall or intrusion detection system are moved in control plane with all the network intelligence that is orchestrated by network application built on top of a centralized controller that has open and standard interfaces. This leads the data plane into two main entities that are the *Infrastructure layer* and *Southbound interface*. The first one is simply made up of devices that forward packets using OpenFlow protocol that works on flow tables characterized by a rule for matching different field entries, a packet counter, and an action to perform on incoming packets. By contrast, the *Southbound interface* acts as a communication link between the central controller and forwarding elements, generally using an *API*. Thanks to them it is possible to interact with the forwarding elements dynamically controlling the network in real-time. The *Southbound interface* is born from the limitations given by commercial hardware switches because in the traditional network it is a complex task to deploy new policies to existing network infrastructure, but using the *APIs* it is possible to provide an abstraction to the infrastructure facilitating this task.

The Control plane is the core of all decisions taken inside the network, allowing the logically centralized controller to access the network console and has a global view of the network devices. This plane is divided in three layers called *Network hypervisors*, *Network Operating System* and *Northbound interface*. Network hypervisor can be detected between the forwarding elements and the central controller. It is strictly linked with virtualization because network hypervisors manage all the virtual machines, providing multi-tenancy support for cloud service providers. This kind of support can be purchased using an open-source solution or proprietary one

based on the result that wants to be achieved. Network operating system is a central core inside *Software Defined Networking* architecture because it is a keystone between the forwarding devices and the southbound and northbound interfaces. It is responsible for all the decisions taken regarding control activities. Like network hypervisors, it can be chosen between open source or proprietary solutions, but there is also to decide if the controller should be centralized or distributed. Another task is fault management, which consists to distribute again all the tasks of the fault node among its neighbours. *Northbound interface* is another abstraction that is very important for *Software Defined Networking* outcome because it is the link between the network operating system and higher application layer. The communication is reached using *REST API*, like Southbound interfaces and one feature is that it is independent of the provider choose for the implementation of any feature, so it has a map between a service and a specific provider. Moreover, it has to also provide service to users but not specifying to them the implementation used to achieve it.

The application layer is where all the activities related to provisioning and monitoring of the networks are created. This is an additional layer used to abstract more the design that is required for virtualization-based solutions. It is made up of two network layers called programming languages and network applications. The first one is an abstraction level that helps the programmer to build the functionalities of his application. One of the challenges that want to overcome is that every function that has to be developed should not interfere with other services and the management of the controller in order to optimize all the applications that are running over it. Moreover, it has to guarantee the re-use of the code giving the possibility to create software differently from the past making possible the use of the modular application over a monolithic one. Another feature is network virtualization and reduces the latency in forwarding packets from one controller to another. Network applications are the software program used for implementing all the network services.

2.2 Network Function Virtualization

Network Function Virtualization is a way to overcome the problem related to deploying physical proprietary hardware for each function that is part of a service. This problem leads to design network topology chaining in a particular order the physical resources making this process without flexibility.

2.2.1 Key concept

The first time *Network Function Virtualization* comes out, it was used to refer to a flexible network that can deploy services to users in a faster and cheaper way. In order to achieve this result, some key concept was defined in order to make a difference from the previous way of deploy of service and they are:

- *Decoupling software from hardware*: In *Network Function virtualization* environment every network element is not bound by integrated network and

software, but they are developed individually, making possible the evolution in time of the hardware or the software without affecting the execution of the service.

- *Flexible network function deployment*: Due to decoupling of software from hardware, it is possible to assign among the time the resources shared in a Network Function Virtualization environment making possible to run the different task on the same hardware, improving the overall deploy over the same physical hardware and it is faster than the traditional way of deploy of service because editing the network link between resource is flexible.

- *Dynamic scaling*: Taking into consideration that the software and the hardware are independent of one another, furthermore it is fast deploying a new service or reassigns resources of an existing one, *Network Function Virtualization* provides a huge help to scale the performance, making it possible to increase or decrease the resource dynamically with finer granularity than before.

2.2.2 ETSI example

The concept of *Network Function Virtualization* was born in 2012, where a lot of telecommunication service providers write a paper selecting *European Telecommunications Standard Institute* (ETSI) as a place to make specification that will be used in different industry field for the *Network Function Virtualization*. In order to achieve this result, several used cases were defined by this organization. One of this is *Customer Premises Equipment* (CPE) scenario.

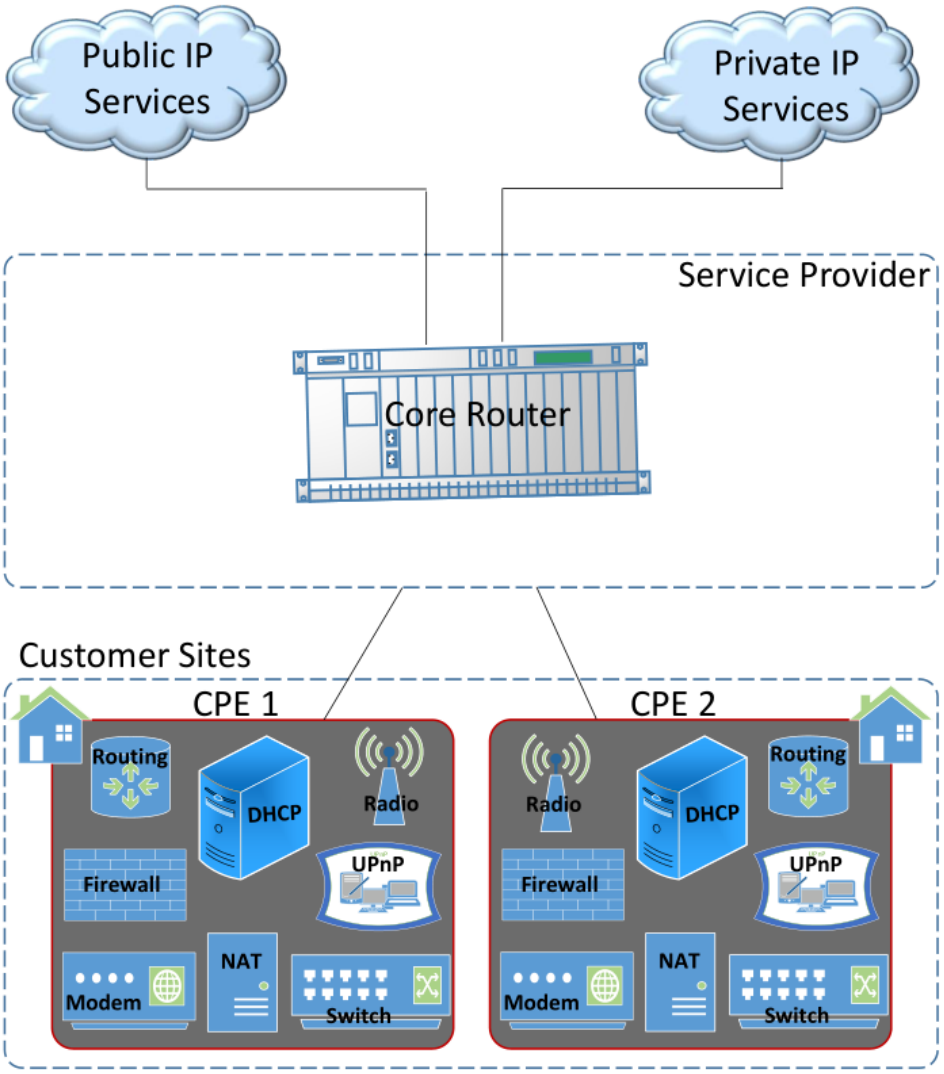

Figure 2.3. Traditional CPE implementation. This image is taken from [3]

Traditionally this scenario is made up of the Service Provider that contains the Core router, that operates in the internet backbone and Customer sites that implements multiple Customers Premises Equipment that implements several functions like *Network Address Translation* (NAT), firewall, *Dynamic Host Configuration Protocol* (DHCP) and so on. In the example shown in figure 2.3, it is possible to notice that service, for example, CPE 1, is made up of eight functions that can be part of a service chain. It is the chain of the functions that are needed to be deployed in order to achieve service, taking care of the order of the link between them. An example can be that the firewall function should be done before NAT and in this scenario, each function is a physical device that should be connected correctly. If some changes are needed inside a service, for example adding a new function or deleting an existing one, the Service Provider needs to go to each customer and perform the changes.

The same scenario is quite different using *Network Function Virtualization* as

shown on figure 2.4. It is possible to see that the Service Provider implements a virtualized core router, without affecting the performance, and all the functions needed to deploy a service inside a generic equipment device, making these functions virtualized. Moreover, the Customer sites are made-up of customers having the same devices used just as switch and router. Making the functions needed for a service chain shared in the Service Provider infrastructure, creating a Data Center, helps to scale the service to multiple users and decrees the cost when a function should be changed, removed, or added.

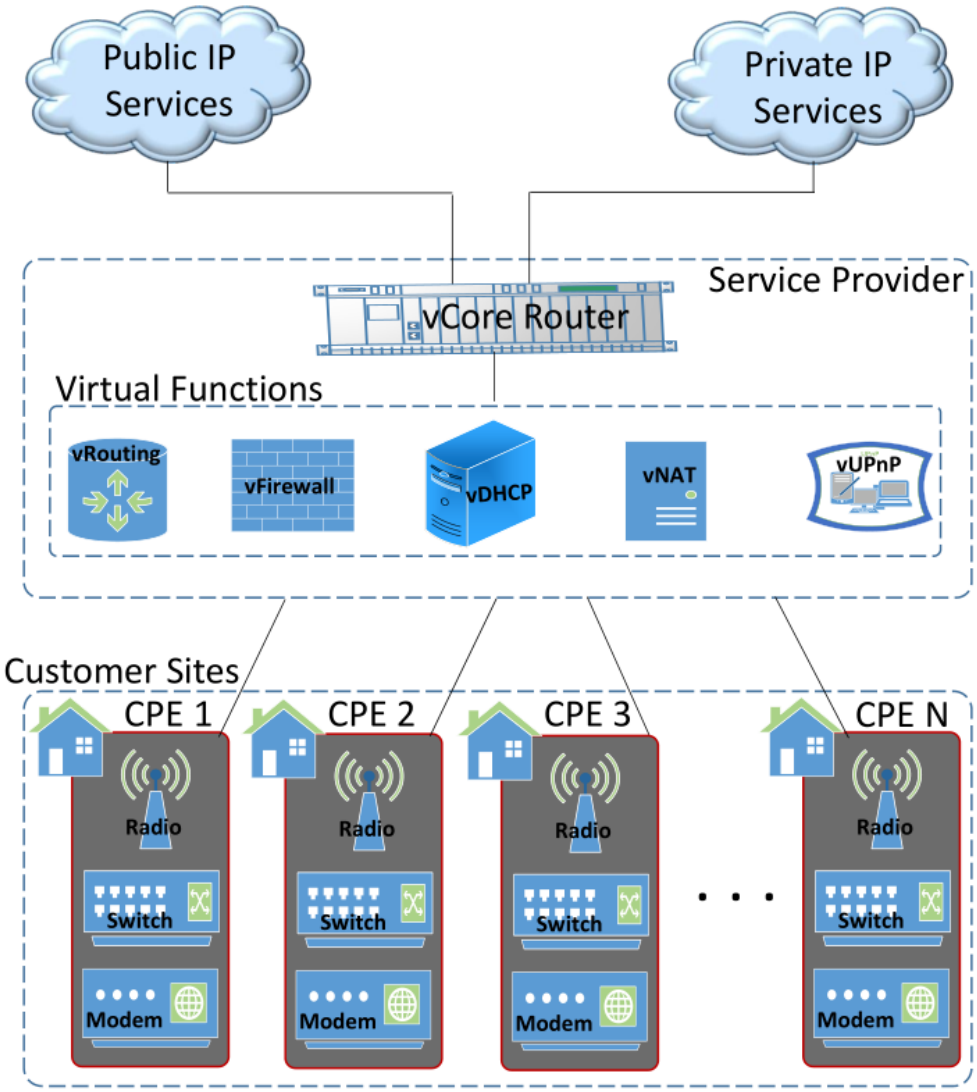

Figure 2.4. CPE implementation with NFV. This image is taken from [3]

2.2.3 Architecture

Network Function Virtualization is divided into three main components: *Network Function Virtualization Infrastructure* (NFVI), *Virtual Network Functions*

(VNFs) and Services, *Network Function Virtualization Management and Orchestration* (NFV MANO). An explicative figure of how they interact is shown below.

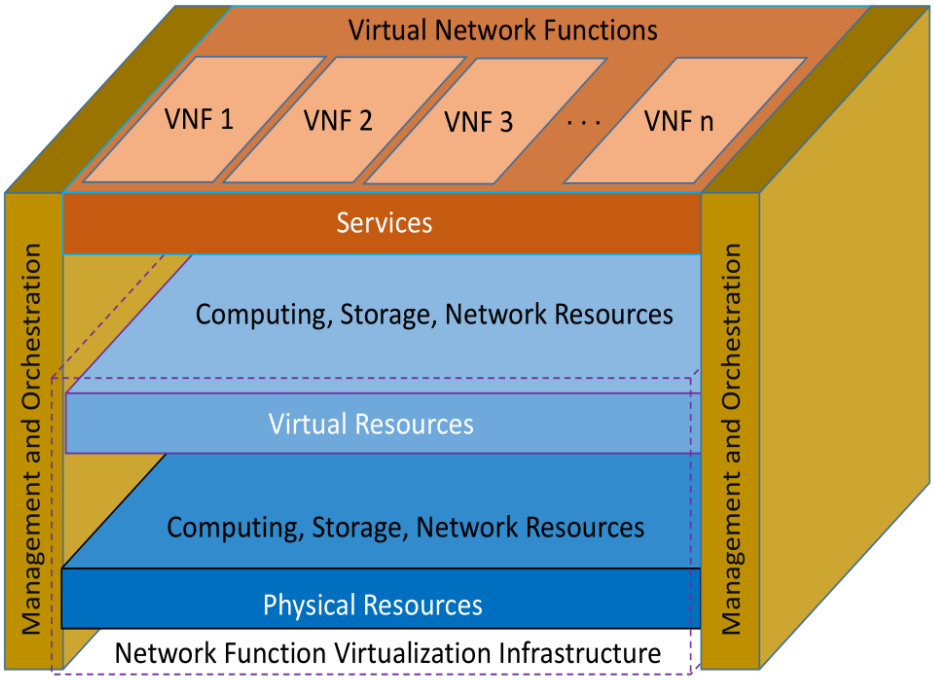

Figure 2.5. Network Function Virtualization Architecture. This image is taken from [3]

Network Function Virtualization Infrastructure is made up of software and hardware resources that generate the environment where the functions and services are deployed. The physical resources are computing hardware, storage, and network, while the virtual ones are an abstraction of the computing, storage, and network resources. Inside NFVI the abstraction is reached using a virtualization layer that is helped by a hypervisor that takes care of the different virtual machines. Thanks to the abstraction layer are possible to decouple virtual resources from physical ones. As for computing and storage virtual resources, they are made up of different virtual machines, while virtual networks are characterized by virtual links and nodes. In particular, a virtual node is a piece of software capable of acting as a host or router, while a virtual link is a logical interconnection between two virtual nodes, that are seen as physical one by the nodes with the advantage that it is possible to dynamically change its properties.

Virtual Network Functions are functional blocks inside the network infrastructure with a well-defined role to perform and interfaces that are used to communicate with other blocks. An example of a network function can be a firewall that in a virtual environment can be fused with other functions in the same virtual machine. In *Virtual Function Virtualization* the function deployed should have the same performance in a traditional network scenario, but the services are achieved only by changing the order of execution of the functions.

Network function Virtualization Management and Orchestration it is used to take care of all the *Virtual Network Functions*, configuring them and deploy on physical resources. This process it is includes also the lifecycle management of the *Virtual Functions* and the databases used for storing information and data models that are used during the effective deployment.Moreover it has other interfaces used for the communications of different components inside *Network Function Virtualization Management and Orchestration* that are *Operations Support System* (OSS) and *Business Support System* (BSS).

2.3 Comparison between SDN and NFV

Both *Network Function Virtualization* and *Software Defined Networking* want to achieve a passage to a new kind of networking standard that has its backbone on open software and standard network hardware. In particular *Network Function Virtualization* aims to run *Network Functions* on industry-standard hardware, while the control plane of *Software Defined Networking* aims to be implemented as open software on industry hardware. These two techniques are complementary and implemented together in a unique solution that will bring great advantages like running a *Software Defined Networking* controller in a virtual machine and exploited as a function in the service chain. Another possible use of these technologies combined could be the flexible and automated deployment offered by *Software Defined Networking* as a way to seep-up the *Network Function Virtualization* functions like security or policy control.

One of the main differences is that they want to solve a different problem related to software-driven networking. An example is the virtualization concept that in *Software Defined Networking* is the allocation of abstract resource for a particular application, while in *Network Function Virtualization* is the way to abstract the network function from dedicated hardware. Another difference is that *Software Defined Networking* are promoted by Data Center and cloud computing areas because it is needed a different network infrastructure where data and control plane are different, while *Network Function Virtualization* is promoted by telecommunications carrier because it can work on an existing network.

Chapter 3

Policy-Based Configuration

3.1 Firewall

A firewall is a combination of hardware and software that isolates an organization's internal network from the rest of the internet. In particular, it allows some kind of packets and blocks others that are determined by a logic defined by a system administrator. Having this concept in mind it is possible to define three goals achieved by a firewall:

- All the traffic from outside to inside and vice versa has to pass through a firewall putting the device at the edge between the organization network and the outside networks. This can be achieved using a single access point and setting the firewall there or using distributed firewalls if the organization is bigger.
- Only the network traffic defined by a system administrator using policy rules is allowed to pass because all the traffic will pass through the firewall, so the device is capable of select which packet forward and which packet block.
- The firewall is immune to penetration, so it should be correctly configured in order to ensure security when correctly configured and deployed.

All the firewalls can be classified according to their functionality in three big categories that are analyzed in the sections below. They are packet filters, stateful filters, and application gateways.

3.1.1 Packet Filters

The packet filter is usually deployed next to the gateway router that connects the internal network with the external one. It takes all the datagram separately and determines if it is allowed to pass or not. This decision is based on the rules deployed by the system administrator and focuses on these fields:

- IP source or destination address.

- Protocol type like *TCP, UDP, ICMP.*
- Transport layer protocol source and destination port number.
- TCP flag bits used to establish a connection like *SYN, ACK.*
- *ICMP* message type.
- Position of the packets, if it is entering or leaving the network.
- Router interface, where each interface can be configured with a particular rule.

A filtering policy can be based on one or multiple of these fields, according to the security level performed on the device. Firewall rules are stored in control lists, one for each interface. The policy is usually executed from top to bottom, making possible complex configuration based on the priority of the rules.

3.1.2 Stateful Filters

Another category of firewalls is the stateful filters that implement a track based on TCP connections in order to achieve better performance in some scenarios where packet filter solutions are limited. An example can block all the incoming transport layer packets coming from the external network based on the TCP *ACK* field, but applying this rule, a traditional packet filter will also make it impossible for the connection from the internal network user to surf on the Web.

A stateful filter can overcome the previous outcome tracking a TCP connection that is characterized by a three-way handshake with the *SYN, SYNACK* and *ACK* packets, then can set a field if the session is over when a *FIN* packet is received. Moreover, it is possible to set a timeout that sets the connection over if there is not a packet exchange during that period. The filter can set rules according to the field defined by the previous firewall category and adding a new field related to the connection if it is already established or not.

3.1.3 Application Gateways

The last category of firewall is the application getaways that allow the network traffic to a specific set of users through a particular application. This goes further beyond the limits of the packet filter and the stateful filter because it is impossible to retrieve this kind of information from network and transport layer headers. An application gateway is an application-specific server where all the data should pass through it. It is implemented one gateway for each application that is needed to filter, but they can run on the same host, but on different processes.

An example of a gateway application is an organization's mail server and Web cache where the user has to identify himself on the application and then the application will perform the connection outside the internal network. The limitation of this kind of filtering technique is where multiple users have to use the same gateway machine and the configuration of the client software that should know how to reach the gateway application and tell which outside server to connect to.

3.2 Policy-Based Configuration

Firewalls are useful for nowadays organizations because they are strictly linked with network traffic. In order to protect themself, it needs a proper configuration and security policy. In RFC 3891 [8] a policy is defined as a method of action that has to guide decisions that are made in the present or future and also it is a rule that manages the access to a specific resource inside the network. The decision is taken by the policy when a condition is satisfied without conflict. In particular:

- a *policy condition* is a representation of all the states corresponding for different fields that have to be set on TRUE in order to execute the action. The rule applied for obtaining the policy condition can be both a set of OR statements or AND making the statement in a *disjunctive normal form* or *conjunctive normal form*.

- a *policy action* is one operation that is performed when all the previously defined conditions are met. The action can be made up of several operations that have to be performed by the device and these operations can be also ordered.

- a *policy conflict* occurs when two or more rules performed different actions that contradict each other leading the device to a state where it is not able to choose which action to perform. The policy system should avoid this behavior that is a problem linked to the misconfiguration by the system administrator. If an error occurs due to the hardware because it can not implement the action it is no more a *policy conflict* but it becomes a *policy error*.

A policy system architecture defined in RFC 3060 [9] exploits these policies definitions discussed before and it is characterized by these main components:

- *Policy Management Tool*: it is the place where the user configures the rules that are going to be translated into a real device and will be activated through acting when all the conditions are satisfied.

- *Policy Repository*: it is the datastore that contains all the data regarding policies with their conditions and actions. The policy stored inside can be retrieved multiple times defining in this way a data model for them.

- *Policy Decision Point*: it is a logical entity that processes the conditions needed in order to activate a policy and use the result obtained in order to execute other network elements.

- *Policy Enforcement Point*: it is a logical entity that executes the policy decision at the bottom of the hardware configuration.

An example of this kind of Policy Management System is illustrated in the figure below.

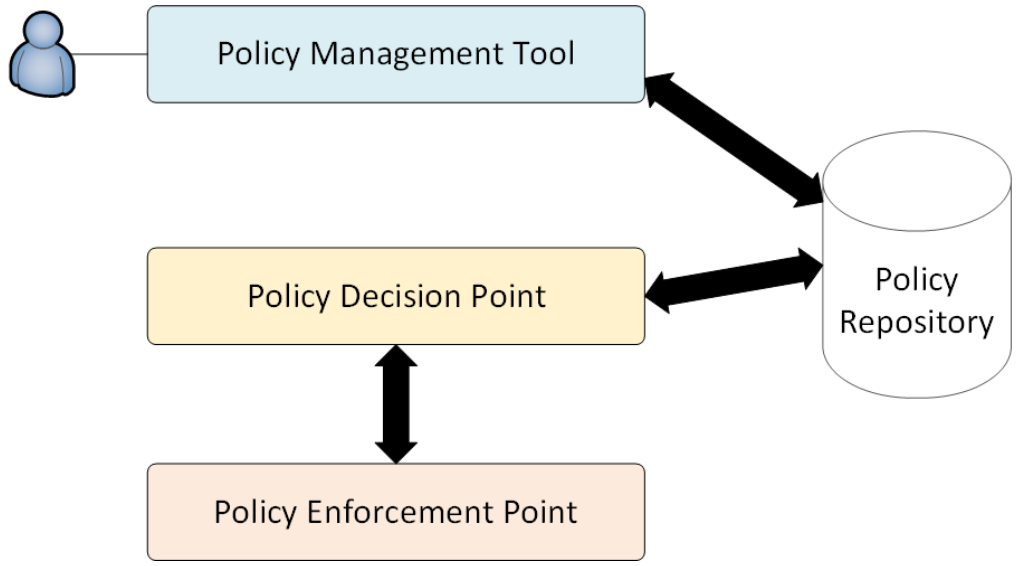

Figure 3.1. Example of Policy Management System

3.2.1 Policy Abstraction

The model previously proposed can lead to several errors if configured by non-expert users. A way to overcome this issue is to have two levels of abstraction while configuring a policy [4]. The first level is defined as *High-Security Policy Language* (HSPL) that is composed of a statement made-up by a subject, action and object declared as a high-level sentence. The subject is the user or the set of users that will be affected by the policy, the action is the operation that will be permitted to those users like access or deny particular resources, the object is the resource affected by the action that can be a network traffic type or particular services. This kind of policy is defined for non-expert users, in order to achieve complete protection and it is based on:

- simplicity, because the user-defined for the use of this kind of policy needs to be helped using auto-completion techniques or statement already defined in a database.
- flexibility, because in order to ensure correct protection the user should be able to set different types of constraints based on time or network traffic.
- extensibility, because this kind of abstraction can receive updates related to new security improvements without changing the already defined structure.

The second level is defined for expert users and it is called *Medium Security Policy Language* (MSPL) defined by a set of statements that are going to be related to the structure of a real firewall configuration like defining the IP addresses of both source and destination, the priority of the policy execution, a filtering action to be performed, transport layer information and so on. The idea behind this kind of level is:

- abstraction, because the model defined should not be related by a particular firewall vendor, but can be applied to most of them.
- diversity, because it should define all the possible security functions that can be related to different concepts and policies.
- flexibility and extensibility, because it should be possible t add new features introducing new security controls.
- continuity, because it should be linked with the previously defined level and it is used as a way to track a particular HSPL in its actual firewall policy configuration.

An example of the the use of these policies is underlined in the figure below:

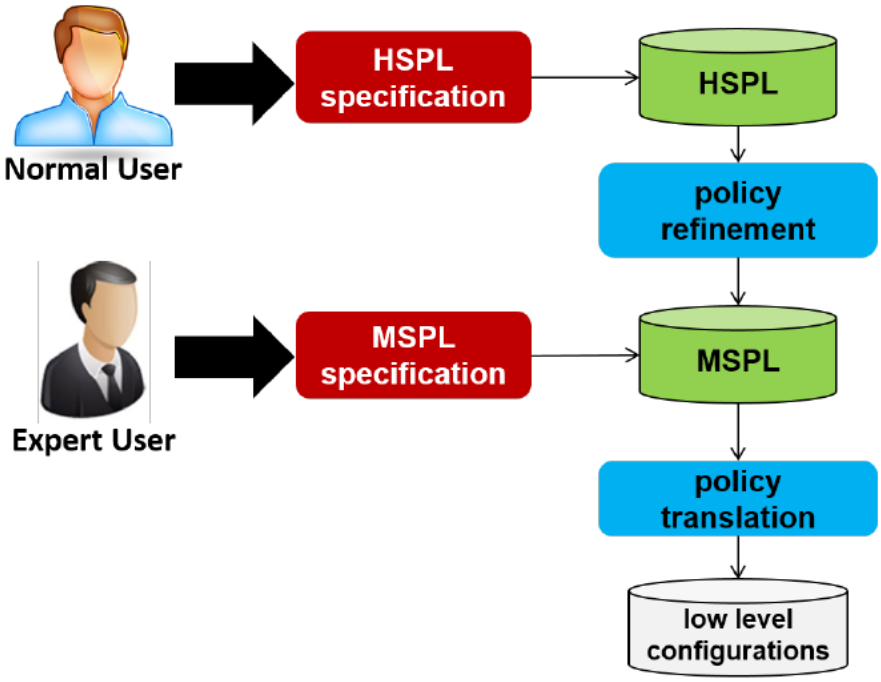

Figure 3.2. Example use of security policy languages. This image is taken from [4]

It is possible to notice that it is identified another policy language defined as a low-level configuration that is the language defined by every firewall vendor and changes among them. Moreover are defined also two processes called policy refinement and policy translation are described in the following subsections.

3.2.2 Policy Refinement

Policy refinement is the first level of translation that occurs when is needed to enforce an HSPL policy. The policy should be synthesized in a policy in MSPL, without changing the logic behind the previous model. It is performed creating a

map between values of subject and object field into its lower-level configuration, then it selects the filtering action to perform helped by a set of interface rules.

3.2.3 Policy Translation

The *Policy translation* is the last service that is performed by the framework and it is the actual enforcement of the MSPL inside a firewall. As seen in the figure 3.2, this service can be the second translation of the refinement if the policy was created in HSPL by a normal user or it is the first translation of the policy from a new policy by an expert user based on MSPL. This translation can be performed multiple times from a given MPLS if is needed to enforce the same policy in a different firewall. This is need because the HSPL and MSPL are created based on the principle to have the same syntax if the meaning of a policy is the same, while the low-level configuration does not have a common standard, so are need different services they are executing the same task.

3.3 Related Work

In literature, different works try to achieve the policy configuration of a system that inspired this thesis work. An example is *Transcompiling firewall* [10] that tries to take a complete configuration of a specific vendor firewall, then it extracts an abstract level of the policies implemented building each policy in an intermediate language called *IFCL* and then it converts them into a different vendor firewall policy. Other works that were useful for this thesis work are the papers regarding firewall automation and orchestration in virtualized environment [5] [11] [12] [13] that build a framework called *VErified REFinement and Optimized Orchestrator* (VEREFOO) that is possible to see in figure 3.3. This framework aims to provide an automatic deployment of *Network Functions* inside a given set of nodes. It performs also an optimization of the position of the different nodes exploiting z3 as a solver for the *MaxSMT* problem for correct enforcement of the nodes. Moreover given a set of isolation and reachability requirements for a Virtual Network system, it can create the correct number of firewalls needed and obtain a configuration made up of abstract policies created using the MSPL syntax.

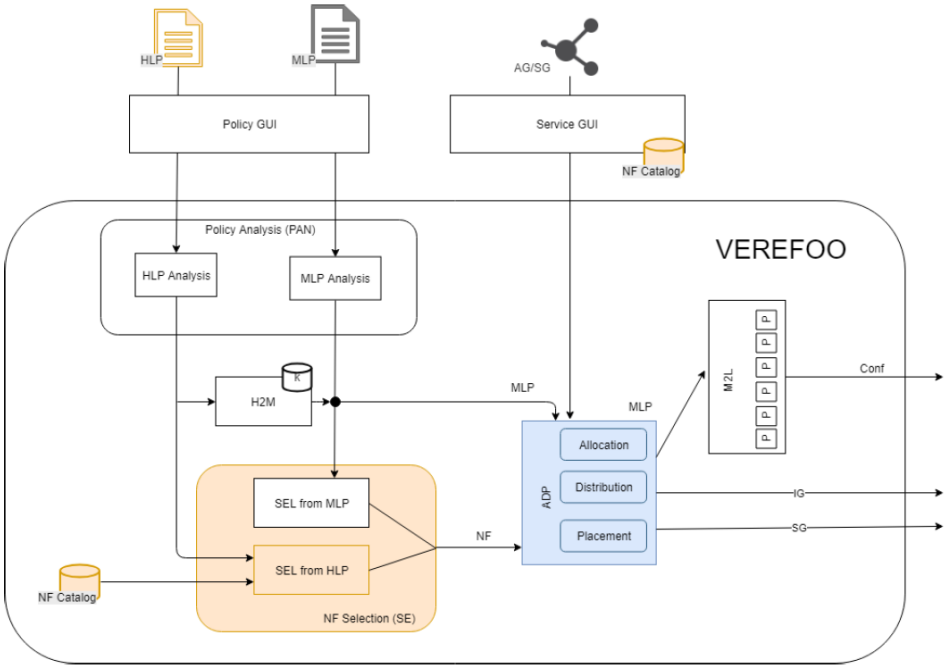

Figure 3.3. VEREFOO architecture. This image is taken from [5]

Chapter 4

Tools

4.1 Iptables

Iptables is a standard packet filter that operates on top of Netfilter that is the standard packet processing inside of Linux kernel version above 2.6.x. The filters are possible thanks to a set of tables that are useful to organize a set of chains of different rules that will deal with different typologies of packets that are approaching the interface of the machine where it is set. The system administrator can define tables or using already existing ones, each of them builds for a specific purpose that is listed below:

- Filter Table: it is the default table set and it is used to decide allowing or denying the incoming packet to its destination.
- NAT Table: it is used for network address translation purposes. The main use of this table is to modify the packet's source or destination in order to change how they are forwarded.
- Mangle Table: it is used to alter IP headers of the incoming packet. It can be used for example to change the number of valid network hop, but cannot change the packet itself. Another use of this table is to mark the packet by internal kernel making possible the interaction with other networking tools.

Each table is made up of a different chain where the rule is placed. The chains can have different lengths of rules or can be empty, and if a packet reaches the end of the chain without action, it will return to the chain that is called it. There are five built-in chains, but the system administrator can create more. The chains already available are:

- *PREROUTING*: in this chain, the packet will be check before a routing decision is made.
- *INPUT*: packets that are entering this chain are going to be locally derived.
- *FORWARD*: different from the previous chain, in this one the packet will go in this chain if have not to routed to the local destination.

- *OUTPUT*: all the packets that are generated by the machine in which are enabled Iptables will be redirected to this chain.
- *POSTROUTING*: packets are passing through this chain after a routing decision it is made, just before been sent outside the hardware.

Each rule can perform different actions like accept the packet, deny it or jump to another chain, making possible multiple combinations between them.

4.2 Ipfirewall

Ipfirewall is a stateful firewall and packet filter written as an external module for FreeBSD operating systems. Later it became integrated into the kernel and it can deal with ipv4 and ipv6 network traffic. It is an open-source firewall and it is used as a base for more complex and well-known firewalls like *M0n0wall*[14] that is the main idea for the development of *OPNsense*[15]. Since the syntax that defines the rules for a policy is complex, it is easy for a network administration made configuration errors or most users can find it difficult to build them. This is one of the reasons why it is used for this thesis work because thanks to the automatic configuration created by the software, all the problems related to humans configuration are deleted.

The internal structure of the firewall can be identified by several components[16] that are:

- packet accounting facility.
- bridge facility.
- forward facility.
- logging facility.
- NAT facility.
- dummynet traffic shaper facilities.
- ipstealth facility.
- advanced special-purpose facilities.

4.3 BPF-Iptables

BPF-Iptables is a firewall developed inside the *Polycube* project inside Politecnico di Torino, that is a framework that allows the development of fast and dynamically loadable network function exploiting modern network technologies. The main idea under this development was to create a firewall with the semantics similar to Iptables but exploiting the performance given by *extended Berkley Packet Filter* (eBPF) that will be explained in the subsection below.

The components of the data plane of this firewall architecture are:

- *Header Parser*: it is a module in the ingress and egress pipeline that extracts the packet headers that are used by eBPF programs for filtering rules. One peculiarity of this module is that if a new rule requires an additional field, the code is dynamically generated, compiled again, and injected into the kernel.

- *Chain Selector*: it is the second module that appeared in the bpf-iptables pipeline and it is responsible for select the correct chain where the packet should be processed. In this firewall it is not possible to create a new chain, so the choice is limited to three chains that are INPUT, OUTPUT, and FORWARD having the same definition of the corresponding Iptables chains. In order to achieve a correct behavior, two instances of this module are created inside the firewall. One is used for the ingress pipeline and the other is used for the egress one.

- *Matching algorithm*: this firewall implements a Linear Bit-Vector Search (LBVS) that is more flexible than the linear search adopted by iptables. The algorithm used requires a bi-dimensional table for every field that will be used for the match that is filled by values of that field presented in the ruleset.

- *Classification Pipeline*: it is the module where the packet is filtered. It is characterized by a cascade of eBPF programs calling each other. The action performed by each program composing this module are:

  - ☐ *extracts* the packet field data from a map filled by Header Parser module.
  - ☐ *performs* a lookup on its own BPF map to find the bivector associated with the field on the current packet.
  - ☐ *performs* a tail-call to the next program of the module chain after saving the bivector.

- *Connection Tracking*: it is a module implemented for stateful filters, that adds multiple eBPF programs both in the ingress and egress pipeline. Moreover, there is an additional matching component added in the classification pipeline which is used as a filter based on the state of the connection.

The control plane architecture is characterized by actions that are taken when one of the following events occurs and they are:

- *Bpf-iptables start-up*: when the program started, it sets an eBPF program that takes the incoming and outgoing packets to the network interfaces and sends them to the ingress and egress hook starting the first pipeline. After all, this is correctly implemented, it is configured the Chain Selector module by subscribing to any Netlink event related to the status or address changes on the host's interfaces.

- *Netlink notification*: when a new Netlink notification is received, bpf-iptables checks the nature of that notification and updates the new address in the Chain Selector, otherwise sets the eBPF program used as redirection from the network interface to the ingress or egress pipeline on the new interfaces.

- *Ruleset changes*: when the rule configuration is changed it is called a process that calculates the new value-bivector pairs for each field. The new values generated are finally inserted inside a new eBPF map and the program creates a new parallel chain.

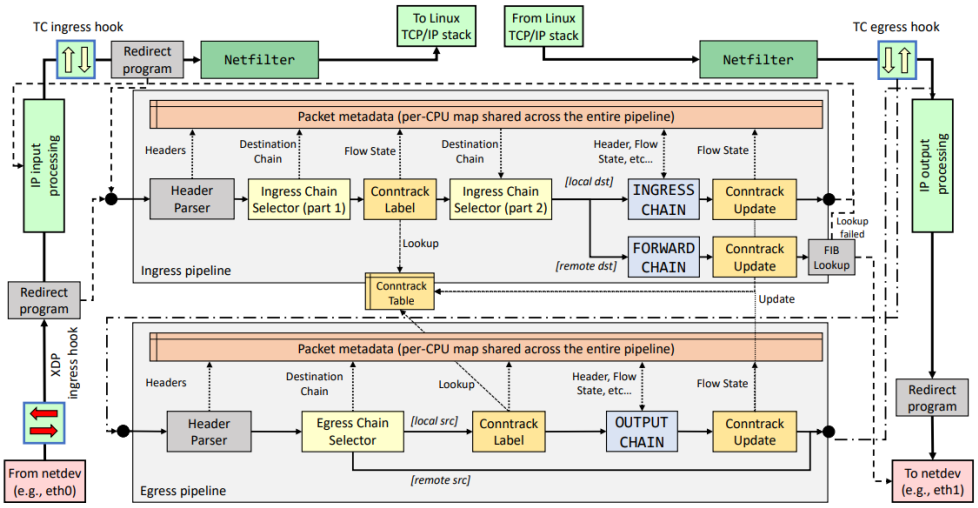

Figure 4.1. Architecture of bpf-iptables. This image is taken from [6]

4.3.1 eBPF

Extended Berkeley Packet Filter (eBPF) is an in-kernel virtual machine for packet filtering introduced in Linux kernel version 3.15. It was proposed the first time by Alexei Starovoitov in 2013 as a way to overcome the previous version on Berkeley Packet Filter, now known as cBPF. This new technology brought several improvements especially inside its architecture that can be summarized as follows:

- making possible the interaction with the generic event inside the kernel that can lead the evolution of eBPF not only linked with packet filtering but also with other system calls.

- availability of networking hook points that are socket filter, responsible of intercepts packets after the IP routing; traffic control hook capable of takes ingress and egress packet; eXpress Data Path, also known as XDP, this hook is located after the network card drivers and it makes possible two levels of approaches called Driver mode and Generic mode used for performance improvement.

- introducing maps as data structures is possible to make a stateful firewall or in general taking care about stateful processing.

- implementing tail-calls that are mechanism useful for connecting programs and use a function call to return to a different place that it is not the caller.

- overcoming limitation of BPF code that is not capable to invoke kernel runtime through system calls. Since the interaction with the kernel is limited to a safer one, these functionalities can be called helpers functions, so eBPF can use them to modify packets or make more sophisticated execution like he previously analyzed tail calls.
- increasing the flexibility and performance exploiting the power of a 64-bit architecture and a Just-In-Time compilation.

4.4 Open vSwitch

Open vSwitch is an open-source implementation of a distributed virtual multilayer switch. It is mostly used for hardware virtualization and can be exploited by *Software Defined Networking* as a way for distributing packets using Open Flow protocol that will be better analyzed inside the next subsection. Even if it is not a firewall, there are different applications in *Software Defined Networking* environment[17], so it was selected as a packet filter in a distributed scenario.

There are several components inside the Open vSwitch environment that are[18]:

- *ovs-vswitchd*: it is the daemon that implements the switch itself.
- *ovsdb-server*: it is a database used for obtaining the configuration of the switch thanks to database queries.
- *ovs-dpctl*: it is used to configure the kernel module of the switch.
- *XenServer RPMs*: it is used to allow an Open vSwitch to be installed on a host where is running a Citrix XenServer and increase the possible functionalities of the switch.
- *ovs-vsctl*: it is call useful for updating the database configuration needed for future deployment.
- *ovs-appctl*: it is used to send commands to a running Open vSwitch.

4.4.1 Open Flow

Open Flow is the first protocol designed for implementing communications between forwarding elements and controller inside a *Software Defined Networking* architecture. It was standardized by *Open Networking Foundation* (ONF) putting several efforts taking care of the principle defined when it was presented the first time and they are:

- separation of control plane and data plane.
- centralization of the control.
- flow-based control.

Currently, Open Flow is used for accessing and manipulating the forwarding plane of network devices that can be physical and virtual.

Elements capable of doing network functions based on Open Flow are called Open Flow switches that are characterized by flow tables in the place of commonly used routing tables. A flow table is made up of multiple field entries that are used for different network functions, for example, the source of destination address. Each flow can be distinguished in three main segments[19]:

- *Header*: it is unique for each flow and defines it as containing a tuple composed at most by ten fields.
- *Action*: it specifies how to handle the packets incoming inside the specific flow. Possible actions are to forward a packet to a port or the controller or drop it.
- *Statistics*: it has information related to the number of packets inside a flow, the time from the last matched packet, or the number of bytes that used that flow.

The communication between switch and controller used a special communication channel also known as *Secure Sockets Layer*. This channel is not only used for Open Flow messages, but also for packets with unknown flow entries, that are directly sent to the controller that will take an action to perform on it.

4.5 FortiGate 50E

FortiGate 50E series is a physical firewall that provides security against different types of cyber attack and it is easy to configure and deploy thanks to *FortiOS*, a proprietary operating system that provides a Graphical User Interface for entry-level users and Command Line interface designed for advanced users. The operating system is combined with *FortiASIC* processors provides also high performance, especially for medium-sized businesses. It is developed and build by *Fortinet* that is an American company that aims to provide security services like firewalls and VPN since 2000.

It is used as a physical firewall in order to achieve a complete development of the module of this thesis and for testing the effects of the new technologies on a traditional network scenario and try to extend the paradigm exploited to hybrid network solutions. The model specification that it is used can be found in its data-sheet[20].

Chapter 5

Approach

5.1 Data model

The data model used as input for the dispatcher can be identified as an XML schema of the resource *NFV*. It is characterized by many elements exploited by other modules inside the framework. In particular, this thesis work focuses on *graphs* element that is the container of all the graph that has to be analyzed.

```
<xsd:element name="NFV">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="graphs" minOccurs="1" maxOccurs="1"/>
                        <xsd:element ref="Constraints" minOccurs="0"
                            maxOccurs="1"/>
                        <xsd:element ref="PropertyDefinition" minOccurs="1"
                            maxOccurs="1"/>
                        <xsd:element ref="Hosts" minOccurs="0"/>
                        <xsd:element ref="Connections" minOccurs="0"/>
                        <xsd:element ref="NetworkForwardingPaths" minOccurs="0"
                            maxOccurs="1"/>
                        <xsd:element name="ParsingString" type="xsd:string"
                            minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:key name="hostKey">
                <xsd:selector xpath="Hosts/Host"/>
                <xsd:field xpath="@name"/>
        </xsd:key>
        <xsd:keyref name="ConnectionSourceHost" refer="hostKey">
                <xsd:selector xpath="Connections/Connection"/>
                <xsd:field xpath="@sourceHost"/>
        </xsd:keyref>
        <xsd:keyref name="ConnectionDestHost" refer="hostKey">
                <xsd:selector xpath="Connections/Connection"/>
                <xsd:field xpath="@destHost"/>
        </xsd:keyref>
        <xsd:unique name="ConnectionUniqueness">
                <xsd:selector xpath="Connections/Connection"/>
                <xsd:field xpath="@sourceHost"/>
```

```
                <xsd:field xpath="@destHost"/>
        </xsd:unique>
        <xsd:keyref name="keyRefNode" refer="keyNode">
                <xsd:selector xpath="NetworkForwardingPaths/Path/node"/>
                <xsd:field xpath="@name"/>
        </xsd:keyref>
        <xsd:unique name="PathUniqueness">
                <xsd:selector xpath="NetworkForwardingPaths/Path"/>
                <xsd:field xpath="@id"/>
        </xsd:unique>
        <xsd:unique name="BandwidthUniqueness">
                <xsd:selector
                    xpath="Constraints/BandwidthConstraints/BandwidthMetrics"/>
                <xsd:field xpath="@src"/>
                <xsd:field xpath="@dst"/>
        </xsd:unique>
        <xsd:key name="keyGraph">
                <xsd:selector xpath="graphs/graph"/>
                <xsd:field xpath="@id"/>
        </xsd:key>
        <xsd:keyref name="PropertyRef" refer="keyGraph">
                <xsd:selector xpath="PropertyDefinition/Property"/>
                <xsd:field xpath="@graph"/>
        </xsd:keyref>
</xsd:element>
```

Listing 5.1. XML schema of NFV

The *graphs* element is made up of several *graph* that are entity defined as a collection of *nodes*, each one of them is characterized by its *function* like firewall, NAT, load balancer, a *name* that is the IP address given to it, a *neighbors* list containing all the addresses where this node should be linked to and a configuration that depends on the type used on the function field. An example of the node schema is presented below.

```
<xsd:element name="node">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="neighbour" maxOccurs="unbounded"
                            minOccurs="0"/>
                        <xsd:element ref="configuration" maxOccurs="1"
                            minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="id" type="xsd:long" use="optional"/>
                <xsd:attribute name="name" type="xsd:string" use="required"/>
                <xsd:attribute name="functional_type" type="functionalTypes"
                    use="optional"/>
        </xsd:complexType>
        <xsd:unique name="uniqueNeighbourId">
                <xsd:selector xpath="neighbour"/>
                <xsd:field xpath="@id"/>
        </xsd:unique>
        <xsd:unique name="uniqueConfigurationId">
                <xsd:selector xpath="neighbour"/>
                <xsd:field xpath="@id"/>
        </xsd:unique>
```

```
    </xsd:element>
```

Listing 5.2. XML schema of node element

In this thesis work is only the analyzed the node element with a specific function, the firewall one. Its configuration is characterized by:

- a *name* field that is the peculiar name of that configuration.
- a *description* field that is a description of the configuration.
- a *default action* field which specifies if the firewall will be in whitelisting mode, enabling all connections, or blacklisting mode, disabling all connections.
- a set of *elements* that are the policy to enforce inside the firewall.

In particular *elements* element is structured in this way:

- ☐ *id*: it is an optional attribute and it is a Long type variable used to store the identification number of the policy. It is used in the REST API implementation.
- ☐ *action*: it is an attribute that defines if the policy will allow or deny the network traffic described by the others parameter.
- ☐ *source*: it is the IP address of the source from which the traffic will be generated. It is a String having four 8-bit numbers followed by a dot. Moreover can be used a special character, *"-1"* in this case, that is used to identify a subnet mask different from /32.
- ☐ *destination*: it is the IP address of the destination that should be reached by this policy. Like the source field, it t is a String having four 8-bit numbers followed by a dot and implements, in the same way, the use of different netmasks using the special character *"-1"*.
- ☐ *protocol*: it is an optional attribute used when the policy is referred to as a transport layer protocol. The possible value of this field is TCP if the policy will affect only tcp protocol, UDP if the policy will care about udp connections, and ANY for both.
- ☐ *src port*: it is usually an optional field that becomes necessary if the transport layer attribute is set. Possible values are * if the policy should be applied to every port, a single integer if only one part should be affected by the policy, or two integers linked with - in order to underline a range of port that the policy should take care.
- ☐ *dst port*: it is an optional field linked to the transport layer protocol and the destination address of the policy defined. In particular, can assume value equals to * if the policy should be applied to every port on the destination IP address. It can be also a single value specifying the port that will allow or deny that traffic or a range of port written as the minor port number plus a - followed by the major number of the port range.

- *priority*: it is an optional attribute that specifies the execution order of the policies. It can be omitted or defined ad * if the policy does not have an order of execution, otherwise, it is an integer number. Lower is the number, higher will be the priority assigned to that particular policy. Policies with the same priority number are treated like there is no priority among them.

- *directional*: it is an optional attribute that can assume two possible values. It can be true if the policy is applied also from destination to source without changing the transport layer protocol. The second value can be false, meaning that the policy will affect only the traffic described by the other attributes. If omitted is considered false.

The XML schema of the elements element is shown below:

```
<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 5.3. XML schema of elements

A firewall example in XML language is shown below. In particular, it is a firewall with IP address 10.0.0.2, connected to other two devices, one with address 10.0.0.1 and the other 20.0.0.1, it has a configuration called *conf1* with related description and it is set in whitelisting mode, enabling all the connections. It has one policy that blocks all the TCP connections that coming from the subnet 10.0.0.0/24 on every port available and directed to 20.0.0.1/32 on port 80.

```
<node name="10.0.0.2" functional_type="FIREWALL">
       <neighbour name="10.0.0.1"/>
       <neighbour name="20.0.0.1"/>
       <configuration name="conf1" description="description␣of␣confi1">
              <firewall defaultAction="ALLOW">
                     <elements>
                            <action>DENY</action>
                            <source>10.0.0.-1</source>
```

```
                        <destination>20.0.0.1</destination>
                        <protocol>TCP</protocol>
                        <src_port>*</src_port>
                        <dst_port>80</dst_port>
                    </elements>
                </firewall>
            </configuration>
    </node>
```

Listing 5.4. Example of firewall in whitelisting mode

5.2 Use Case

Thanks to the data structure defined above it are possible to create different uses cases that can be exploited by all the module created for this thesis work and checked the correctness between the input and the output. One element that characterizes all the use cases is the description attribute of the configuration because it is formatted in a special way due to a particular module described later.

5.2.1 UC1

The first use case defined it is one of the simplest configurations. It is a firewall in whitelisting mode, so able to forward all the traffic received without any other policy. The XML schema is characterized by a *name* with the corresponding IP address, the *functional-type* that is a firewall, the IP address of the devices to which it is linked to and a configuration called *conf1* because it is referred to this specific use case. The *description* presents the name of the bridge that will be used in a particular type of firewall developed during this thesis work. In the end, it is possible to notice the *default action* set on ALLOW that will describe the behavior of the firewall.

```
<node name="192.168.56.2" functional_type="FIREWALL">
        <neighbour name="192.168.56.3"/>
        <neighbour name="192.168.57.4"/>
        <configuration name="conf1" description="b0:␣UC1">
                <firewall defaultAction="ALLOW"/>
        </configuration>
</node>
```

Listing 5.5. Use case 1 node XML schema

5.2.2 UC2

In order to check the correctness of the configuration of the firewall deployed, the second use case is defined as a firewall in blacklisting mode, so blocking all the traffic received, without any policy allowing any kind of connection. This time the XML schema is a firewall with an IP address defined by the *name*, two devices

linked to it, and their corresponding address. There is a configuration called *conf2* related to the number of this use case and in the *description* field there is present also the name of the bridge used by a particular submodule. Since it is complementary to the first use case, its *defaultAction* is set to DENY, blocking all the incoming network traffic.

```
<node name="192.168.56.5" functional_type="FIREWALL">
        <neighbour name="192.168.56.3"/>
        <neighbour name="192.168.57.4"/>
        <configuration name="conf2" description="b0: UC2">
                <firewall defaultAction="DENY"/>
        </configuration>
</node>
```

Listing 5.6. Use case 2 node XML schema

5.2.3 UC3

The third use case can be described as a whitelisting firewall with one policy that denies particular network traffic. This configuration aims to test the correct translation of the policy that has different netmasks. This firewall is characterized by a *name* that contains its network address, the device which it is connecting in sense of graph connection, a configuration name that is *conf3* directly linked to the use case number, and its *description* that includes the name of the bridge to use, need by a particular firewall. Its *defaultAcation* is ALLOW, setting the firewall in whitelisting mode and the policy describes an action that will block the network traffic from all the IP addresses identified by the source subnet (/24 in this case) that want to use any transport layer protocol to a specific IP address, blocking all the ports from both source and destination.

```
<node name="192.168.56.6" functional_type="FIREWALL">
        <neighbour name="192.168.56.3"/>
        <neighbour name="192.168.57.4"/>
        <configuration name="conf3" description="b0: UC3">
                <firewall defaultAction="ALLOW">
                        <elements>
                                <action>DENY</action>
                                <source>192.168.56.-1</source>
                                <destination>192.168.57.4</destination>
                                <protocol>ANY</protocol>
                                <src_port>*</src_port>
                                <dst_port>*</dst_port>
                        </elements>
                </firewall>
        </configuration>
</node>
```

Listing 5.7. Use case 3 node XML schema

5.2.4 UC4

This use case aims to check if the directional feature is translated correctly. In order to achieve this, it is implemented a whitelisting firewall with one policy that exploits the directional attribute, blocking all the tcp traffic from every port of the source to a specific port of the destination address and vice versa. The XML settings of this firewall have its IP address in the name field, followed by the devices connected to it. The configuration is characterized by *conf4* name, before it belongs to the fourth use case, a description with the bridge used in one particular firewall and the *defaultAction* ALLOW. The policy is made up by a DENY action, in order to block the network traffic over tcp connection thanks to protocol field set to TCP, a source IP address with all possible network ports, and a destination address targetting port 8080. Moreover having the attribute directional set on true, there will be another policy that will deny tcp traffic from the destination at port 8080 to the source IP address at every port number.

```
<node name="192.168.56.7" functional_type="FIREWALL">
        <neighbour name="192.168.56.3"/>
        <neighbour name="192.168.57.4"/>
        <configuration name="conf4" description="b0: UC4">
                <firewall defaultAction="ALLOW">
                        <elements>
                                <action>DENY</action>
                                <source>192.168.56.3</source>
                                <destination>192.168.57.4</destination>
                                <protocol>TCP</protocol>
                                <src_port>*</src_port>
                                <dst_port>8080</dst_port>
                                <directional>true</directional>
                        </elements>
                </firewall>
        </configuration>
</node>
```

Listing 5.8. Use case 4 node XML schema

5.2.5 UC5

The fifth use case is in charge to verify the priority settings of a firewall. It implements a whitelisting firewall, that allows all the network traffic, with two policies. One policy blocks an amount of traffic with a certain priority while the other allows a subset of the previous traffic with higher priority. To better understand the XML schema of this firewall is made up of a node element having as a name the IP address of the machine that will deploy the firewall, as *functional-type* FIREWALL, followed by the IP address of the devices linked to it. The configuration has its name related to the number of the use case, *conf5* in this scenario, inside the description there is the name of the bridge used by a certain firewall. The whitelisting mode is enabled setting the defaultAction on ALLOW. As for the policies, the first one with priority equals 10 will block all the tcp connections from any port of the source field IP address to port 45 up to 56 on the destination field. The second

policy with priority equals 5 will allow the traffic coming from the same IP address of the previous policy and same destination, but only enabling port 50. Since the priority of the second policy is higher than the first one, the packet that has to reach the destination on port 50 will be forwarded.

```
<node name="192.168.56.8" functional_type="FIREWALL">
        <neighbour name="192.168.56.3"/>
        <neighbour name="192.168.57.4"/>
        <configuration name="conf5" description="b0: UC5">
                <firewall defaultAction="ALLOW">
                        <elements>
                                <action>DENY</action>
                                <source>192.168.56.3</source>
                                <destination>192.168.57.4</destination>
                                <protocol>TCP</protocol>
                                <src_port>*</src_port>
                                <dst_port>45-56</dst_port>
                                <priority>10</priority>
                        </elements>
                        <elements>
                                <action>ALLOW</action>
                                <source>192.168.56.3</source>
                                <destination>192.168.57.4</destination>
                                <protocol>TCP</protocol>
                                <src_port>*</src_port>
                                <dst_port>50</dst_port>
                                <priority>5</priority>
                        </elements>
                </firewall>
        </configuration>
</node>
```

Listing 5.9. Use case 5 node XML schema

Chapter 6

Implementation

The structure of this module can be divided into two main groups. The first one is identified as a dispatcher that takes the medium-level resource and distributes them among all the translations available. The second is made up of all the submodules that made possible the policy enforcement on the vendor-specific firewall. The figure below represents an abstract view of the whole process.

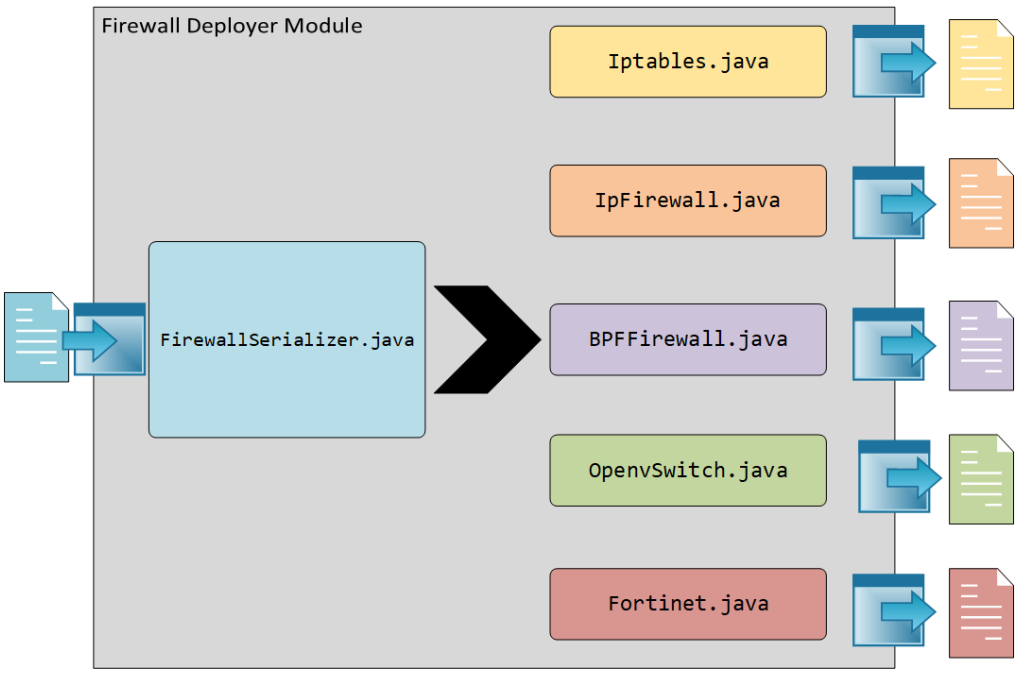

Figure 6.1. Module workflow

6.1 Firewall Serializer

This is the main class that can be found in the package *it.polito.verefoo.firewall*. Its constructor is characterized by two parameters:

- an *NFV* data structure that is described in the previous chapter.

- a *type* of firewall that is needed the translation.

The type of the translation of one particular firewall happens related with the enum java class *FirewallDeploy* that is shown below:

```
public enum FirewallDeploy {

FORTINET,
IPFIREWALL,
IPTABLES,
EBPF,
OPENVSWITCH,
ALL
}
```

Listing 6.1. Definition of FirewallDeploy

In particular, the options that are represented in this class are explained as:

- *FORTINET* that implies that the translation should be done for a physical firewall.
- *IPFIREWALL* that is used for FreeBSD support, generating an ipfw set of rules.
- *IPTABLES* that creates a script that sets an Iptables firewall in Linux environment.
- *EBPF* that generates a script for bpf-iptables that can be deployed if need a packet filter working on eBPF.
- *OPENVSWITCH* that is set if the output of the program is only a configuration to deploy on SDN environment, pointing on a particular OpenvSwitch bridge.
- *ALL* that is mainly used to have all the configuration available and check if there are problems during translation in one or more firewalls.

6.2 Iptables

This class implements the translation from the firewall abstraction model to Iptables packet filter. This firewall is an open-source firewall build inside Linux 2.4.x and 2.6.x kernels and operates on top of Netfilter, which is the standard packet processing of the previously defined kernels. Its mechanism is based on chains of rules that can be combined using the jump instruction. A chain is characterized by a default rule that can assume two possible values: DROP for discard the packets, ALLOW for forwarding them. This class manipulates three chains of the five there are added by default and they are:

- *INPUT* that is the chain taking care of the packets that will be delivered locally.

- *OUTPUT* that is the chain used by the packet that is sent by this machine.

- *FORWARD* is the chain where all the packet that is routed by the firewall.

The script generated by this class has a first part that deletes all the policies already installed on the machine identified as firewall and sets the default action of each chain based on the value on the abstract model. Then are applied some policy based on the elements element of the model following the logic explained below:

- ☐ iptables -A FORWARD: it is the fix part of the command generated by the class for a single policy that inserts this rule after the last one on the FORWARD chain.

- ☐ -p *protocol*: it is generated when a transport layer protocol is defined inside the model.It can assume tcp or udp values, but when ANY in the model has to be translated, it will generate one rule with tcp and another with udp protocol, having all the other features without changes.

- ☐ -s *sourceIP/NetmaskSrc*: it shows the source address with its automatically generated netmask that will trigger the action of this policy.

- ☐ -d *destinationIP/NetmaskDs*: it shows the destination address with its automatically generated netmask that will trigger the action of this policy.

- ☐ - -sport *lowerbound[:upperbound]*: it is generated if it is setted the transport layer protocol and the model it is different from the value *"*"*. This field indicates the port number of the source address to which the policy should be activated. It can also assume a ranged port number using the lower-bound and the upper-bound having between *":"* character as a separator.

- ☐ - -dport *lowerbound[:upperbound]*: it is generated if it is setted the transport layer protocol and the model it is different from the value *"*"*. This field indicates the port number of the destination address to which the policy should be activated. It can also assume a ranged port number using the lower bound and the upper bound having between *":"* character as a separator.

- ☐ -j *action*: it is the action that will be executed by this policy translating the two values on the model into ALLOW and DROP respectively.

If the priority is set, all the rules are stored in memory and ordered based on its priority number having the result that the rule with the higher number will be inserted before inside the chain. It deals with directional field adding another ruler with the right parameter modified having the same priority as the original policy.

6.3 IpFirewall

This class aims to implement a complete configuration of a packet filter on FreeBSD operating system based on Ipfirewall. It operates on 31 numbered sets, having the last one used for default rules. Moreover, each policy is characterized by a rule number that can assume values between 1 and 65534 that indicates the priority of execution of the rules, having more priority if the number is smaller. The sets are implemented just for flexible enabling and disabling of a group of rules belonging to the same set.

The output configuration file produced by this class will first remove all the rules of previous configurations and then delete set number 31 that is not affected by the flush command. After this preliminary operation are added all the policy that based on the abstract configuration can have different fields that are summarized below:

- ipfw -q add: it is the fixed part generated by the class that ensures the correct working of the Ipfirewall without showing any output to the machine console.
- *ruleNumber*: it is the number which the policy are processed when a packet goes inside the firewall.
- set *setNumber*: usually, the algorithm implements only set number 1 for all the policy received and set 31 for the default action. They are to be considered as a container that is used for better management of the policies.
- *action*: it is the action to perform if the packet matches the rule. It can assume two possible values that are *"deny"* for block the packet and *"allow"* to let it pass and deliver to its destination.
- *protocol*: this field is setted based of the protocol used. If the transport layer part of the policy is absent, it will set the protocol to *"ip"*, otherwise, it will set *"tcp"* or *"udp"* based on the choice inside the abstract policy.
- from *sourceIP/NetmaskSrc*: it shows the source address with its automatically generated netmask that will trigger the action of this policy.
- *sourcePortLowerbound[-upperbound]*: it is generated if it is setted the transport layer protocol. This field indicates the port number of the source address to which the policy should be activated. It can also assume a ranged port number using the lower bound and the upper bound having between "-" character as a separator. If the model has any value *"*"*, the algorithm will generate a rule that blocks or allows all the incoming packets with that source ports.
- to *destinationIP/NetmaskDs*: it shows the destination address with its automatically generated netmask that will trigger the action of this policy.
- *destinationPortLowerbound[-upperbound]*: it is generated if it is setted the transport layer protocol. This field indicates the port number of the destination address to which the policy should be activated. It can also assume

a ranged port number using the lower bound and the upper bound having between *"-"* character as a separator. If the model has any value *"*"*, the algorithm will generate a rule that blocks or allows all the incoming packets with that destination ports.

If the directional flag is set, another rule having the same priority but the opposite source and destination addresses and ports, if dealing with a transport layer policy, is added. After all the policies are correctly inserted inside the configuration file, also a special rule on set 31 is added based on the default action of the abstract firewall.

6.4 BPFFirewall

This class is used to implement a firewall capable to exploit eBPF technology, making a configuration for bpf-iptables. The firewall used for the configuration was developed in 2018 and it aims to replicate the Iptables syntax making some update on the architecture behind. However, there are several differences from Iptables regarding the number of chains available and the port range either source and destination.

The current chain available that are the same affected by the configuration created by this class are:

- *INPUT* that is the chain taking care of the packets that will be delivered locally.
- *OUTPUT* that is the chain used by the packets that are sent by this machine.
- *FORWARD* is the chain where all the packets are routed by the firewall.

As for the port range, since it is not implemented, the class uses an algorithm that takes as input the lower bound and upper bound of the range and it creates multiple rules in order to achieve the same result of the abstract model. If it is present the default value "*" used to describe that the policy will affect all the ports, that field is omitted in the policy created.

The configuration generated by the whole algorithm deletes the previous policy that is stored in the three chains and sets the default action of that chains based on the same parameter in the abstract firewall. Then the first rule is inserted in the chain having id set to 0 while the others have this pattern:

- ☐ polycubectl pcn-iptables chain FORWARD append: it is the fixed part generated by the class that ensures the correct working of this firewall.
- ☐ src=*sourceIP/NetmaskSrc*: it shows the source address with its automatically generated netmask that will trigger the action of this policy.
- ☐ dst=*destinationIP/NetmaskDs*: it shows the destination address with its automatically generated netmask that will trigger the action of this policy.

- l4proto=*protocol*: it sets the transport layer protocol according to the model. The possible options are *"tcp"* and *"udp"* but if both have to be enforced, the class will generate two rules, one for each transport layer protocol.

- sport=*sourcePort*: it is the source port related to the transport layer protocol. Since this firewall does not support ranged port for this field, it is implemented an algorithm that generates one rule for each source port and is combined with the destination port in order to avoid duplicates of the same policy.

- dport=*destinationPort*: it is the destination port related to the transport layer protocol.Since this firewall does not support ranged port for this field, it is implemented an algorithm that generates one rule for each destination port and is combined with the source port in order to avoid duplicates of the same policy.

- action=*action*: it is the action performed by the rule that can take two values according to the abstract configuration of the firewall. The values can be ACCEPT for allowing traffic that matches the previously defined field and DROP to block it.

If the priority is set inside the abstract model of the firewall, all the policies are sorted in ascending mode and then inserted based on the value of that field. If there are multiple rules with the same priority the order will be casual. If the direction field is set to true when the rule is processed, it is generated a copy of that rule in terms of action, priority, and protocol, but with an inverted source address and port with the destination one.

6.5 OpenvSwitch

The translation performed by this class is based on Open vSwtch which is a distributed multi-layer switch used in Software Defined Networking environment. Inside this thesis work, it is used as a way to configure a device in order to achieve a packet filter for OpenFlow protocol. Before explaining the main characteristics of the algorithm implemented, it is useful to underline that Open vSwitch is not developed like a packet filter, but some studies have tested this kind of scenario achieving good results[17][21]. In order to generate a good configuration, it is needed an Open vSwitch bridge inside the machine has to be configured and it is needed the name of the bridge at the beginning of the description field followed by ":" character.

This firewall work using flows, where each one is characterized by a policy. The configuration script generated by this class deletes all the existing flows available on the Open vSwitch switch and then created a set of rules based on the incoming policies of the abstract model. An example of the comma-separated fields need for a complete creation of one is policy is:

- ovs-ofctl add-flow *bridgeName*: it identifies the name of the bridge where the flow should be inserted. It is extracted from the description of the abstract firewall configuration model.

- priority=*priorityNumber*: it is the priority of the flow that has to be executed. Since the priority in Open vSwitch is define in decrementing order, an algorithm is performed in order to achieve the same result as the other modules.
- dl_type=*0x800*:it is a fixed part generated by the algorithm that identifies the Ethernet interfaces.
- nw_src=*sourceIP/NetmaskSrc*: it shows the source address with its automatically generated netmask that will trigger the action of this policy.
- nw_dst=*destinationIP/NetmaskDs*: it shows the destination address with its automatically generated netmask that will trigger the action of this policy.
- nw_proto=*protocolNumber*: it is the number related to the transport layer protocol used referred to RFC 768[22] and RFC793[23]. It is used numbers 6 for tcp and 17 for udp and a double rule is generated if the abstract model has to implement both.
- tp_src=*sourcePort*: it represents the port of the source address that will trigger this flow. It is expressed in hexadecimal, supporting also masks in order to perform multiple port scan. There is an algorithm that takes the lower bound and the upper bound of the policy in the abstract model if presented and generates a set of strings, where each one should be policy to enforce in order to cover all the ports needed. Then multiple rules are generated according to the destination port list of the port to enforce.
- tp_dst=*destinationPort*: it represents the port of the destination address that will trigger this flow. It is expressed in hexadecimal, supporting also masks in order to perform multiple port scan. There is an algorithm that takes the lower bound and the upper bound of the policy in the abstract model if presented and generates a set of strings, where each one should be policy to enforce in order to cover all the ports needed. Then multiple rules are generated according to the source port list of the port to enforce.
- action=*action*: it defines the action that is performed by the flow. Only two values are allowed that are *"NORMAL"* to allow traffic *"drop"* to block it, according to the model.

If the abstract policy contains the directional field set on true, the module will generate an additional flow having the same priority and protocol but opposite network ports and addresses. When all the policy is correctly translated, it is created a new flow with the lowest priority, for example, *0* reading Open vSwitch documentation[18], having only the action set having the same value of the default action that is present in the abstract firewall configuration.

6.6 Fortinet

The last class implemented by the module translates the abstract firewall model to a physical firewall. The firewall used is a FortiGate 50E, provided by Kyoto Institute

of Technology laboratories, and developed by *Fortinet* which is an American leader company in security hardware and deployment. It was hard dealing with this device because the main idea under this product was to create a firewall for a system administrator that is not an expert that is helped by a Graphical User Interface.

The output of this class is a configuration script that has to be deployed using a USB key inside the physical device and it is based on the *Command Line Interface* proposed by *Fortinet* and following *FortiOS* configuration principles[24]. The main components that are configured by the class are the following:

- *system settings*: it is used to configure the device in the transparent mode that processes all the incoming and outgoing packets applying the policy defined by the set of rules previously configured.
- *firewall service category*: this group identifies the category in which all the policy will be stored. It is characterized by a name and a comment. It is usually set by the class having default value *"custom_verefoo"*.
- *firewall schedule recurring*: it is possible to define when the policy will be executed. It is defined by a name and a day in which the policy will be active. This class defines a custom schedule setter every day.
- *firewall address*: the source and destination addresses need this group to be configured, one for each network address, in order to be used inside a policy. The main fields that made up this group are:
  - *name*: it is the name of the address created that has the prefix "src" if it is a source address or "dst" if it is a destination address.
  - *uuid*: it is a *universal unique identifier* used in network infrastructure in order to obtain coordination between elements in a distributed environment without a controller.
  - *subnet*: it is the union between the ipv4 address and the netmask that identifies a set of addresses where the policy will be applied.
- *firewall service custom*: this portion of the configuration defines a service that is placed inside the category created before and sets the transport layer protocol that will trigger the policy. It has a field for tcp and one for udp and is it possible to define a range for both destination and source port for which the policy is applied.
- *firewall policy*: this is the piece of the created configuration that will create the policy. It has several fields previously generated:
  - *uuid*: it is a *universal unique identifier* used in network infrastructure in order to obtain coordination between elements in a distributed environment without a controller.
  - *srcintf*: it is the interface from which the traffic will arrive in order to apply the filter policy. It is configured by the algorithm like all the available LAN interfaces.

- *dstintf*: it is the interface from which the traffic will be sent in order to apply the filter policy. It is configured by the algorithm like all the available LAN interfaces.
- *srcaddr*: it is the source address that will match the incoming packet in order to be filtered. It is set using the previously defined group of firewall addresses.
- *dstaddr*: it is the destination address that will match the incoming packet in order to be filtered. It is set using the previously defined group of firewall addresses.
- *action*: it is the filtering action to be performed. It is set according to the model to "accept" if the traffic will be permitted or "deny" if the traffic should be blocked.
- *schedule*: it is when the policy should be active and takes as argument the previously created schedule group.
- *service*: it is related to the transport layer protocol. If it is set will take the service group previously created and put its name in this portion of the configuration.

If the priority field is set the policy is sorted by that field before the translation through the algorithm. If the direction boolean is set to true, it is created just a new firewall policy group and a firewall service custom, but exploiting the other groups with inverted source and destination addresses. After all the policies are processed if the default action of the abstract configuration is set on allow, the class will generate an additional policy with the lowest priority because it is not possible to change the default behavior of the device that is in blacklisting mode.

Chapter 7

Validation

7.1 Use Case 1

In this section, it is analyzed for every possible firewall scenario, the outcome, and the configuration of the firewall generated by its relative class. The first case analyzed is the configuration of the Iptables firewall starting from an abstract model that simply sets the device in whitelisting mode. It is possible to notice from the configuration listed below that the machine at the beginning deletes all the policy configured previously and it sets the INPUT, FORWARD, and OUTPUT chain on ACCEPT value due to the use case 1 where the default action allows every incoming connection.

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

Listing 7.1. Output of UC1 iptables

The same use case applied to Ipfirewall produces a different configuration, based on the java class that implements it. In particular, this script makes all the *sets* empty, then it is added a default rule on the default setting that allows every traffic from any ipv4 address to any others. The configuration applied to the machine under test is shown below.

```
#!/bin/sh
cmd="/sbin/ipfw␣-q"
${cmd} -f flush
${cmd} delete set 31
${cmd} add 65534 set 31 allow ip from any to any
```

Listing 7.2. Output of UC1 ipfirewall

After the test was performed on Linux and FreeBSD operating systems with a positive outcome, it took place a test of the firewall exploiting eBPF on Use Case 1. The configuration produced this time is pretty familiar compared to the Iptable one thanks to the effort of the developer in order to achieve the same semantics of the famous Linux firewall. The script that is presented below shows that INPUT, OUTPUT, and FORWARD chains are manually deleted by previous chains and it sets them allowing the incoming packets, according to the abstract model configuration.

```
#!/bin/sh
cmd="polycubectl␣pcn-iptables␣chain"
${cmd} INPUT rule del
${cmd} FORWARD rule del
${cmd} OUTPUT rule del
${cmd} INPUT set default=ACCEPT
${cmd} FORWARD set default=ACCEPT
${cmd} OUTPUT set default=ACCEPT
```

Listing 7.3. Output of UC1 bpf-iptables

The solution based on *Software Defined Networking* that exploits the Open vSwitch switch use the *description* attribute inside the configuration of the abstract packet filter and set the bridge: firstly eliminating all the existing flows, then adding a flow with the smallest priority available that allows all incoming packets to reach their destinations. The configuration is shown below.

```
#!/bin/sh
sudo ovs-ofctl del-flows b0
sudo ovs-ofctl add-flow b0 priority=0,dl_type=0x800,action=NORMAL
```

Listing 7.4. Output of UC1 open vSwitch

All the configurations created by the java class of the physical firewall are not tested in real network scenarios due to the lockdown established after *SARS-CoV-2*. It was just verified that the script worked on a real device deploying it through a USB key. The whitelisting packet filter in this configuration is characterized by different groups that set the service category, schedule recurring, firewall service custom, the transparent mode in order to perform the filter in any position inside the network. Moreover, it is set a policy that allows network traffic from every direction. The configuration created is shown below.

```
config system settings
       set opmode transparent
end
config firewall service category
       edit "custom_verefoo"
              set comment "new category for verefoo services"
       next
end
```

```
config firewall schedule recurring
       edit "always_custom"
              set day sunday monday tuesday wednesday thursday
                  friday saturday
       next
end
config firewall address
       edit "src_1"
              set uuid 7701c7b1-5b08-4470-bc1b-c5f2a616bb7a
       next
end
config firewall address
       edit "dst_1"
              set uuid 4e39c136-163d-4863-b214-30f245da80ba
       next
end
config firewall service custom
       edit "custom_service_1"
              set category "custom_verefoo"
              set comment "default action service"
              set tcp-portrange 0-65535:0-65535
              set udp-portrange 0-65535:0-65535

       next
end
config firewall policy
       edit 1
              set uuid 2df68c98-6377-48a5-a242-379a15f5814e
              set srcintf "lan"
              set dstintf "lan"
              set srcaddr "src_1"
              set dstaddr "dst_1"
              set action accept
              set schedule "always_custom"
              set service "custom_service_1"
       next
end
```

Listing 7.5. Output of UC1 fortinet

7.2 Use Case 2

The second section describes the configuration of real firewalls starting from the abstract model that is created for an abstract firewall model that blocks every kind of connection, so it is in the so-called blacklisting mode. The Iptables java class translation is shown below and it is characterized by the replacement of all existing policies with the default action of the INPUT, FORWARD, and OUTPUT chains

set in dropping packets mode, how underlined by the abstract model.

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

Listing 7.6. Output of UC2 iptables

The Ipfirewall configuration for this second use case is characterized by a set of commands that made all the *sets* previously loaded with any rules, empty and sets the default behavior of the machine creating a rule in the last set that denies all the packets that belong to the network layer and transport layer. The script generated by the java class is shown below.

```
#!/bin/sh
cmd="/sbin/ipfw␣-q"
${cmd} -f flush
${cmd} delete set 31
${cmd} add 65534 set 31 deny ip from any to any
```

Listing 7.7. Output of UC2 ipfirewall

The translation performed by the bpf-iptables class it is quite simple also in this use case. It is possible to see from the configuration below that the output of its related java class performs removal of all the rules on the available chains and sets the default action of each one to DROP, accordingly to the default action that characterizes the abstract module that defines the second use case.

```
#!/bin/sh
cmd="polycubectl␣pcn-iptables␣chain"
${cmd} INPUT rule del
${cmd} FORWARD rule del
${cmd} OUTPUT rule del
${cmd} INPUT set default=DROP
${cmd} FORWARD set default=DROP
${cmd} OUTPUT set default=DROP
```

Listing 7.8. Output of UC2 bpf-iptables

The configuration based on Open vSwitch in this second case can be summarized with the removing of all the flows that were available inside the virtual switch and a flow that blocks all the packets received using Ethernet interfaces. The output performed by the related java class is shown below.

```
#!/bin/sh
```

```
sudo ovs-ofctl del-flows b0
sudo ovs-ofctl add-flow b0 priority=0,dl_type=0x800,action=drop
```

Listing 7.9. Output of UC2 open vSwitch

As for the physical firewall, the configuration generated by the module will set only the operating mode of the device and a category for future configurations. It is possible to notice that in the file generated below there are no rules processed by the device because the default behavior of this particular packet filter is already in blacklisting mode, denying all the incoming and outgoing network traffic.

```
config system settings
       set opmode transparent
end
config firewall service category
       edit "custom_verefoo"
              set comment "new category for verefoo services"
       next
end
config firewall schedule recurring
       edit "always_custom"
              set day sunday monday tuesday wednesday thursday
                  friday saturday
       next
end
```

Listing 7.10. Output of UC2 fortinet

7.3 Use Case 3

The third section describes the implementation of a firewall in whitelisting mode with a blocking rule. The rule implements a netmask using the *"-1"* wildcard where it needs to implement the include that sets of addresses. Moreover, it is not specified the transport layer protocol, so will be enforced both tcp and udp.

The first configuration related to this use case is the Iptables one. It is possible to notice that after the previous policies deletion and the setting of the default action for all the chains, two rules are added instead of one, one to enforce the policy using tcp protocol and another enforcing udp. Moreover, the netmasks are correctly set as shown in the following output.

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

```
${cmd} -A FORWARD -p tcp -s 192.168.56.0/24 -d 192.168.57.4/32
   -j DROP
${cmd} -A FORWARD -p udp -s 192.168.56.0/24 -d 192.168.57.4/32
   -j DROP
```

Listing 7.11. Output of UC3 iptables

The configuration based on Ipfirewall for this use case presents a previous policy deletion rule, included the previously default action to perform for packets that do not match any rules and a rule that denies all the traffic having a source port and a destination port. in the end, there is a rule that will be defined by the default action. Also in this scenario, the netmasks are correctly set.

```
#!/bin/sh
cmd="/sbin/ipfw␣-q"
${cmd} -f flush
${cmd} delete set 31
${cmd} add 1 set 1 deny ip from 192.168.56.0/24 0-65535 to
   192.168.57.4/32 0-65535
${cmd} add 65534 set 31 allow ip from any to any
```

Listing 7.12. Output of UC3 ipfirewall

For the bpf-iptables scenario, in this use case, the module generates a configuration file that deletes all the previous rule chain by chain and sets the default action in the same way in ACCEPT state. Then two rules are added, one for each transport layer protocol. Also in this scenario, the netmask is correctly detected and enforced. The output of this configuration is shown below.

```
#!/bin/sh
cmd="polycubectl␣pcn-iptables␣chain"
${cmd} INPUT rule del
${cmd} FORWARD rule del
${cmd} OUTPUT rule del
${cmd} INPUT set default=ACCEPT
${cmd} FORWARD set default=ACCEPT
${cmd} OUTPUT set default=ACCEPT
${cmd} FORWARD insert id=0 src=192.168.56.0/24
   dst=192.168.57.4/32 l4proto=tcp action=DROP
${cmd} FORWARD append src=192.168.56.0/24 dst=192.168.57.4/32
   l4proto=udp action=DROP
```

Listing 7.13. Output of UC3 bpf-iptables

An example of firewall configuration using Open vSwitch for this particular use case defines at the beginning the previously deployed policy deletion, then sets two rules having the corresponding nw_proto equals to 6 for the tcp connection related rule and 17 for udp one. Moreover in the end there is a rule that sets the default action according to the model and the netmasks are correctly enforced.

```
#!/bin/sh
sudo ovs-ofctl del-flows b0
sudo ovs-ofctl add-flow b0 dl_type=0x800,
   nw_src=192.168.56.0/24,nw_dst=192.168.57.4/32,
   nw_proto=6,action=drop
sudo ovs-ofctl add-flow b0 dl_type=0x800,
   nw_src=192.168.56.0/24,nw_dst=192.168.57.4/32,
   nw_proto=17,action=drop
sudo ovs-ofctl add-flow b0 priority=0,dl_type=0x800,action=NORMAL
```

Listing 7.14. Output of UC3 open vSwitch

The last configuration presented for this use case is the one related to the physical firewall. It presents the settings of the device like the operative mode, a custom service category, and a custom schedule that will be triggered forever. Then there is the configuration of the addresses for source and destination that is correctly translated from the abstract configuration. There is also a service that is created having both tcp and udp without duplication of the rule. In the end, there is the composition of the policy using the groups previously defined. The same structure is defined for the policy created for default action since the behavior of this firewall is to block all the incoming connections.

```
config system settings
       set opmode transparent
end
config firewall service category
       edit "custom_verefoo"
              set comment "new category for verefoo services"
       next
end
config firewall schedule recurring
       edit "always_custom"
              set day sunday monday tuesday wednesday thursday
                  friday saturday
       next
end
config firewall address
       edit "src_1"
              set uuid 3989710a-74fe-4ca1-8d2a-39c5d20f5b5c
              set subnet 192.168.56.0 255.255.255.0
       next
end
config firewall address
       edit "dst_1"
              set uuid 5bf2f662-b4e6-40c2-9a8a-6ad42b601a71
              set subnet 192.168.57.4 255.255.255.255
       next
end
```

```
config firewall service custom
       edit "custom_service_1"
              set category "custom_verefoo"
              set comment "sample service"
              set tcp-portrange 0-65535:0-65535
              set udp-portrange 0-65535:0-65535

       next
end
config firewall policy
       edit 1
              set uuid 52da1cec-79a7-4de6-83e9-de6ff28f9897
              set srcintf "lan"
              set dstintf "lan"
              set srcaddr "src_1"
              set dstaddr "dst_1"
              set action deny
              set schedule "always_custom"
              set service "custom_service_1"
       next
end
config firewall address
       edit "src_2"
              set uuid aeeccbcb-2388-4f29-9931-6c2706cdac52
       next
end
config firewall address
       edit "dst_2"
              set uuid 0c68c105-e24d-4e74-9750-3575a2d0cbe8
       next
end
config firewall service custom
       edit "custom_service_2"
              set category "custom_verefoo"
              set comment "default action service"
              set tcp-portrange 0-65535:0-65535
              set udp-portrange 0-65535:0-65535

       next
end
config firewall policy
       edit 2
              set uuid 13925faf-d92c-4bb3-8cdb-0335b24cc21c
              set srcintf "lan"
              set dstintf "lan"
              set srcaddr "src_2"
              set dstaddr "dst_2"
              set action accept
```

```
                set schedule "always_custom"
                set service "custom_service_2"
        next
end
```

Listing 7.15. Output of UC3 fortinet

7.4 Use Case 4

This section describes the implementation of a firewall in whitelisting mode with a blocking rule that blocks only the tcp transport layer traffic on port 8080. Moreover, it is implemented the directional flag is used for applying the policy in both dimensions.

The first configuration related to this use case is the Iptables one. It is possible to notice that after the previous policies deletion and the setting of the default action for all the chains, two rules are added instead of one, one to enforce the policy using tcp protocol having the source port set and another enforcing the same protocol but having the destination port set. Moreover, the source and destination addresses are also exchanged as shown in the following output.

```
#!/bin/sh
cmd="sudo␣iptables"
${cmd} -F
${cmd} -P INPUT ACCEPT
${cmd} -P FORWARD ACCEPT
${cmd} -P OUTPUT ACCEPT
${cmd} -A FORWARD -p tcp -s 192.168.56.3/32 -d 192.168.57.4/32
   --dport 8080 -j DROP
${cmd} -A FORWARD -p tcp -s 192.168.57.4/32 -d 192.168.57.4/32
   --sport 8080 -j DROP
```

Listing 7.16. Output of UC4 iptables

The configuration based on Ipfirewall for this use case presents a previous policy deletion rule, included the previously default action that is mandatory to perform for packets that do not match any rules. Then two rules are enforcing the same protocol, the transport layer in this case, but with all the fields related to the source and destination exchanged, like the address and the transport layer port. In the end, there is a rule that will be defined by the default action allowing all the other packets to pass.

```
#!/bin/sh
cmd="/sbin/ipfw␣-q"
${cmd} -f flush
${cmd} delete set 31
```

```
${cmd} add 1 set 1 deny tcp from 192.168.56.3/32 0-65535 to
   192.168.57.4/32 8080-8080
${cmd} add 1 set 1 deny tcp from 192.168.57.4/32 8080-8080 to
   192.168.56.3/32 0-65535
${cmd} add 65534 set 31 allow ip from any to any
```

Listing 7.17. Output of UC4 ipfirewall

For the bpf-iptables scenario, in this use case, the module generates a configuration file that deletes all the previous rule chain by chain and sets the default action to ACCEPT in order to allow all the incoming packets. Then two rules are added, one for each direction having addresses and ports exchanged. Even if it is only one policy divided into two rules the first is inserted in the chain and the other is appended because one already exists. The output of this configuration is shown below.

```
#!/bin/sh
cmd="polycubectl␣pcn-iptables␣chain"
${cmd} INPUT rule del
${cmd} FORWARD rule del
${cmd} OUTPUT rule del
${cmd} INPUT set default=ACCEPT
${cmd} FORWARD set default=ACCEPT
${cmd} OUTPUT set default=ACCEPT
${cmd} FORWARD insert id=0 src=192.168.56.3/32
   dst=192.168.57.4/32 l4proto=tcp dport=8080 action=DROP
${cmd} FORWARD append src=192.168.57.4/32 dst=192.168.56.3/32
   l4proto=tcp sport=8080 action=DROP
```

Listing 7.18. Output of UC4 bpf-iptables

An example of firewall configuration using Open vSwitch for this particular use case defines the previously deployed policy deletion, then sets two rules that have the same network protocol, but with opposed addresses and ports. It is possible to notice that the ports are correctly translated because the hexadecimal value of 0x1f90 is the integer number 8080.

```
#!/bin/sh
sudo ovs-ofctl del-flows b0
sudo ovs-ofctl add-flow b0 dl_type=0x800,
   nw_src=192.168.57.4/32,nw_dst=192.168.56.3/32,
   nw_proto=6,tp_src=0x1f90,action=drop
sudo ovs-ofctl add-flow b0 dl_type=0x800,
   nw_src=192.168.56.3/32,nw_dst=192.168.57.4/32,
   nw_proto=6,tp_dst=0x1f90,action=drop
sudo ovs-ofctl add-flow b0 priority=0,dl_type=0x800,action=NORMAL
```

Listing 7.19. Output of UC4 open vSwitch

The last configuration presented for this use case is the one related to the physical firewall. In this scenario, it is first set the device operative mode, the custom

service category, and the custom schedule that will be triggered forever. Then there is the configuration of the addresses for source and destination. There are two services created having exchanged ports but the same transport layer protocol. In the end, there is the composition of the two policies using the groups previously defined and reusing the two addresses created but used in the opposite fields. After the two policies, there is the last one created for default action since the normal behavior of this firewall is to block all the incoming connections.

```
config system settings
        set opmode transparent
end
config firewall service category
        edit "custom_verefoo"
                set comment "new category for verefoo services"
        next
end
config firewall schedule recurring
        edit "always_custom"
                set day sunday monday tuesday wednesday thursday
                    friday saturday
        next
end
config firewall address
        edit "src_1"
                set uuid 7a6127c6-6421-480a-8599-06edb3fe4b1b
                set subnet 192.168.56.3 255.255.255.255
        next
end
config firewall address
        edit "dst_1"
                set uuid b500a731-0c57-4610-a596-027cdcf9eb04
                set subnet 192.168.57.4 255.255.255.255
        next
end
config firewall service custom
        edit "custom_service_1"
                set category "custom_verefoo"
                set comment "sample service"
                set tcp-portrange 8080-8080:0-65535

        next
end
config firewall policy
        edit 1
                set uuid 6d2528c9-ac17-4553-b930-c78a1609eb22
                set srcintf "lan"
                set dstintf "lan"
```

```
                set srcaddr "src_1"
                set dstaddr "dst_1"
                set action deny
                set schedule "always_custom"
                set service "custom_service_1"
        next
end
config firewall service custom
        edit "custom_service_2"
                set category "custom_verefoo"
                set comment "sample service"
                set tcp-portrange 0-65535:8080-8080
config firewall policy
        edit 2
                set uuid 7c0876c2-4c8f-4961-a31d-c0105372a599
                set srcintf "lan"
                set dstintf "lan"
                set srcaddr "dst_1"
                set dstaddr "src_1"
                set action deny
                set schedule "always_custom"
                set service "custom_service_2"
        next
end
config firewall address
        edit "src_3"
                set uuid b056ac01-5ca3-4dfe-aac8-ac41a770cb75
        next
end
config firewall address
        edit "dst_3"
                set uuid 8c2371b4-096c-4973-83da-b104edfc61e4
        next
end
config firewall service custom
        edit "custom_service_3"
                set category "custom_verefoo"
                set comment "default action service"
                set tcp-portrange 0-65535:0-65535
                set udp-portrange 0-65535:0-65535

        next
end
config firewall policy
        edit 3
                set uuid 54035cec-f6ac-4ea6-a3ed-f1809b921ee6
                set srcintf "lan"
                set dstintf "lan"
```

```
                set srcaddr "src_3"
                set dstaddr "dst_3"
                set action accept
                set schedule "always_custom"
                set service "custom_service_3"
        next
end
```

Listing 7.20. Output of UC4 fortinet

7.5 Use Case 5

This last section describes the implementation of a firewall in whitelisting mode with a blocking rule that blocks only the tcp transport layer traffic on ports between 45 to 56. Then there is an allowing policy having the same parameters except from the destination port that is only 50. Moreover, it is implemented the priority field that is used to apply the allowing policy before blocking one if one packet arrives.

The first configuration related to this complex use case is the Iptables one. It is possible to notice that after the previous policies deletion and the setting of the default action for all the chains, one rule is added, the one that allows connection on port 50 using tcp traffic. Then is added the blocking rule over tcp on the range between 45 to 56. This translation is the opposite as configured in the abstract model, but it follows the policy increasing number for the deployment.

```
#!/bin/sh
cmd="sudo␣iptables"
${cmd} -F
${cmd} -P INPUT ACCEPT
${cmd} -P FORWARD ACCEPT
${cmd} -P OUTPUT ACCEPT
${cmd} -A FORWARD -p tcp -s 192.168.56.3/32 -d 192.168.57.4/32
   --dport 50 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 192.168.56.3/32 -d 192.168.57.4/32
   --dport 45:56 -j DROP
```

Listing 7.21. Output of UC5 iptables

The configuration based on Ipfirewall for this use case presents a previous policy deletion rule, included the previously default action that is mandatory to perform to be sure to clean properly the environment. Then two rules are enforcing the same protocol, the transport layer, the same source and destination addresses, *192.168.56.3* and *192.168.57.4* respectively, but different action and destination port. The first rule blocks the traffic over ports 45-56 with a *rulenumber* equals to the priority, while the second allows it over port 50 and is characterized by a minor *rulenumber*. Even if they are added like showed in the model, the policy will be enforced based on the *rulenumber*. In the end, there is a rule that will be defined

by the default action allowing all the other packets to pass.

```
#!/bin/sh
cmd="/sbin/ipfw␣-q"
${cmd} -f flush
${cmd} delete set 31
${cmd} add 10 set 1 deny tcp from 192.168.56.3/32 0-65535 to
   192.168.57.4/32 45-56
${cmd} add 5 set 1 allow tcp from 192.168.56.3/32 0-65535 to
   192.168.57.4/32 50-50
${cmd} add 65534 set 31 allow ip from any to any
```

Listing 7.22. Output of UC5 ipfirewall

For the bpf-iptables scenario, in this use case, the module generates a configuration file that deletes all the previous rule chain by chain and sets the default action to ACCEPT in order to allow all the incoming packets. Then a rule is added based on allowing policy with higher priority, then are added several blocking rules generated by the blocking policy of the model, because this firewall doe not exploit port range as an argument. The output of this configuration is shown below to better understand how the algorithm for ranged port works.

```
#!/bin/sh
cmd="polycubectl␣pcn-iptables␣chain"
${cmd} INPUT rule del
${cmd} FORWARD rule del
${cmd} OUTPUT rule del
${cmd} INPUT set default=ACCEPT
${cmd} FORWARD set default=ACCEPT
${cmd} OUTPUT set default=ACCEPT
${cmd} FORWARD insert id=0 src=192.168.56.3/32
   dst=192.168.57.4/32 l4proto=tcp dport=50 action=ACCEPT
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=45 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=46 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=47 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=48 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=49 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=50 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=51 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=52 action=DROP
```

```
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=53 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=54 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=55 action=DROP
${cmd} FORWARD append src=192.168.56.3/32 dst=192.168.57.4/32
   l4proto=tcp dport=56 action=DROP
```

Listing 7.23. Output of UC5 bpf-iptables

An example of firewall configuration using Open vSwitch for this particular use case defines the previously deployed policy deletion, then sets multiple rules that having the same action, the same priority but different ports, since the device is only capable to understand the values in hexadecimal notation and the conversion generates multiple rules instead of the only one presented in the abstract configuration. Then is added the allowing policy over port 50 with higher priority and in the end, the policy encapsulates the default action.

```
#!/bin/sh
sudo ovs-ofctl del-flows b0
sudo ovs-ofctl add-flow b0 priority=65525,dl_type=0x800,
   nw_src=192.168.56.3/32,nw_dst=192.168.57.4/32,
   nw_proto=6,tp_dst=0x2e/0xfffe,action=drop
sudo ovs-ofctl add-flow b0 priority=65525,dl_type=0x800
   ,nw_src=192.168.56.3/32,nw_dst=192.168.57.4/32,
   nw_proto=6,tp_dst=0x30/0xfff8,action=drop
sudo ovs-ofctl add-flow b0 priority=65525,dl_type=0x800,
   nw_src=192.168.56.3/32,nw_dst=192.168.57.4/32,
   nw_proto=6,tp_dst=0x38,action=drop
sudo ovs-ofctl add-flow b0 priority=65525,dl_type=0x800,
   nw_src=192.168.56.3/32,nw_dst=192.168.57.4/32,
   nw_proto=6,tp_dst=0x2d,action=drop
sudo ovs-ofctl add-flow b0 priority=65530,dl_type=0x800,
   nw_src=192.168.56.3/32,nw_dst=192.168.57.4/32,
   nw_proto=6,tp_dst=0x32,action=NORMAL
sudo ovs-ofctl add-flow b0 priority=0,dl_type=0x800,action=NORMAL
```

Listing 7.24. Output of UC5 open vSwitch

The last configuration presented for this use case is the one related to the physical firewall. In this scenario, it is first set the device operative mode, the custom service category, and the custom schedule that will be triggered forever. Then there is the configuration of the addresses for source and destination and the service for each policy of the model but implemented using the priority field. Since the execution it is based on the order of insertion, it is created firstly the rule having an allowing action and then the other one where the action is denied. After the two policies, there is the last one created for default action since the normal behavior

of this firewall is to block all the incoming connections.

```
config system settings
       set opmode transparent
end
config firewall service category
       edit "custom_verefoo"
              set comment "new category for verefoo services"
       next
end
config firewall schedule recurring
       edit "always_custom"
              set day sunday monday tuesday wednesday thursday
                 friday saturday
       next
end
config firewall address
       edit "src_1"
              set uuid 5d4f700a-1f25-4245-be0c-940270f8400c
              set subnet 192.168.56.3 255.255.255.255
       next
end
config firewall address
       edit "dst_1"
              set uuid 55031bd8-8d96-4a3f-ad9a-f65fc8b3c1b7
              set subnet 192.168.57.4 255.255.255.255
       next
end
config firewall service custom
       edit "custom_service_1"
              set category "custom_verefoo"
              set comment "sample service"
              set tcp-portrange 50-50:0-65535

       next
end
config firewall policy
       edit 1
              set uuid 65f4772a-53b4-47f0-a43f-20756100547c
              set srcintf "lan"
              set dstintf "lan"
              set srcaddr "src_1"
              set dstaddr "dst_1"
              set action accept
              set schedule "always_custom"
              set service "custom_service_1"
       next
```

```
end
config firewall address
       edit "src_2"
              set uuid 69e96499-aa99-4de4-a2ac-041387ba5f61
              set subnet 192.168.56.3 255.255.255.255
       next
end
config firewall address
       edit "dst_2"
              set uuid fd471c25-5907-45ff-b81b-c48596cdbc50
              set subnet 192.168.57.4 255.255.255.255
       next
end
config firewall service custom
       edit "custom_service_2"
              set category "custom_verefoo"
              set comment "sample service"
              set tcp-portrange 45-56:0-65535

       next
end
config firewall policy
       edit 2
              set uuid 6a75a64b-8ec1-424d-993f-671b06a98f4d
              set srcintf "lan"
              set dstintf "lan"
              set srcaddr "src_2"
              set dstaddr "dst_2"
              set action deny
              set schedule "always_custom"
              set service "custom_service_2"
       next
end
config firewall address
       edit "src_3"
              set uuid 634dd48c-c832-47b5-852c-c3d5c8e48e27
       next
end
config firewall address
       edit "dst_3"
              set uuid 329c624d-eccc-4164-ab48-23937aed708d
       next
end
config firewall service custom
       edit "custom_service_3"
              set category "custom_verefoo"
              set comment "default action service"
              set tcp-portrange 0-65535:0-65535
```

```
                set udp-portrange 0-65535:0-65535

        next
    end
    config firewall policy
        edit 3
                set uuid 2ba02398-c3d6-4858-829e-e5504ba6c446
                set srcintf "lan"
                set dstintf "lan"
                set srcaddr "src_3"
                set dstaddr "dst_3"
                set action accept
                set schedule "always_custom"
                set service "custom_service_3"
        next
    end
```

Listing 7.25. Output of UC5 fortinet

Chapter 8

Conclusions

8.1 Achieved Objectives

This research is intended to perform an analysis of the current firewall available in the market and integrate them in the automation context of *Network function Virtualization* and *Software Defined Networking*. These two concepts are new paradigms that are having a lot of resonance in this years. In particular, exploiting the policy-language configuration this research was able to develop a service capable of performing the last step of policy translation and it can dynamically change the translated language according to the environment where the node will be deployed.

This result is achieved thanks to preliminary work done studying in literature different data models that in the end, it is an extension of the model proposed as *Medium Security Policy Language* inside VEREFOO. Besides, several types of research were performed on the technology to exploit in this work that in the end is a packet filter capable of use eBPF, another one is an open-source so compliant to the main concepts of *Software Defined Networking* and another one capable of performing a filtering action over Open Flow protocol.

Moreover were performed several tests in normal network scenarios obtaining very good results for most of the firewall developed during this research. In particular, the best results were achieved with Iptables, Ipfirewall, and bpf-iptables, while the worst performance was achieved with the one based on Open vSwitch.

However, during this research, there were several limitations. The first one was to apply the same abstract language to all the devices and overcome the differences in terms of configuration management. Another was the efficiency of the firewall chosen to develop and related tests. Especially the choice of Open vSwitch as a firewall does not achieve the same performance as the other configuration mostly because the use of packet filter for this architecture is not the role thought by its developers.

8.2 Future works

As for future development related to this research, it is possible to choose different directions. One of them is to implement other modules for firewall exploiting other technologies or discover new firewalls that achieve better performance than the one used in this work, especially for Open Flow protocol-based firewall. Another one is to develop a machine learning mechanism capable of choosing the firewall suitable for the condition of the *Network Function* to deploy, taking into consideration the environment where it should be deployed and the computational resources needed by the machine.

Bibliography

[1] M. S. Bonfim, K. L. Dias, and S. F. L. Fernandes, "Integrated NFV/SDN architectures," *ACM Computing Surveys*, vol. 51, no. 6, pp. 1–39, Feb. 2019. [Online]. Available: https://doi.org/10.1145/3172866

[2] V. Jain, V. Yatri, Kanchan, and C. Kapoor, "Software defined networking: State-of-the-art," *Journal of High Speed Networks*, vol. 25, no. 1, p. 1?40, Feb. 2019. [Online]. Available: https://doi.org/10.3233/JHS-190601

[3] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016. [Online]. Available: https://doi.org/10.1109/comst.2015.2477041

[4] F. Valenza and A. Lioy, "User-oriented network security policy specification," *J. Internet Serv. Inf. Secur.*, vol. 8, no. 2, pp. 33–47, 2018. [Online]. Available: https://doi.org/10.22667/JISIS.2018.05.31.033

[5] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Towards a fully automated and optimized network security functions orchestration," in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE, Oct. 2019. [Online]. Available: https://doi.org/10.1109/cccs.2019.8888130

[6] S. Miano, M. Bertrone, M. V. Bernal, F. Risso, Y. Lu, and J. Pi, "Securing linux with a faster and scalable iptables," 2018. [Online]. Available: https://mbertrone.github.io/documents/21-Securing_Linux_with_a_Faster_and_Scalable_Iptables.pdf

[7] K. Greene, "Mit tech review 10 breakthrough technologies: Software-defined networking." [Online]. Available: http://www2.technologyreview.com/news/412194/tr10-software-defined-networking/

[8] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, "Terminology for policy-based management," Tech. Rep., Nov. 2001. [Online]. Available: https://doi.org/10.17487/rfc3198

[9] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model – version 1 specification," Tech. Rep., Feb. 2001. [Online]. Available: https://doi.org/10.17487/rfc3060

[10] C. Bodei, P. Degano, R. Focardi, L. Galletta, and M. Tempesta, "Transcompiling firewalls," in *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 303–324. [Online]. Available: https://doi.org/10.1007/978-3-319-89722-6_13

[11] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized

networks," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110402

[12] ——, "Introducing programmability and automation in the synthesis of virtual firewall rules," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, Jun. 2020. [Online]. Available: https://doi.org/10.1109/netsoft48620.2020.9165434

[13] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, "Improving the formal verification of reachability policies in virtualized networks," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2020. [Online]. Available: https://doi.org/10.1109/tnsm.2020.3045781

[14] "M0n0walll documentation," https://m0n0.ch/wall/index.php.

[15] "OPNsense documentation," https://docs.opnsense.org/intro.html.

[16] K. M. Goertzel, *Information Assurance Tools Report ? Firewalls*, 7th ed., Herndon, VA, 2011.

[17] S. Kaur, K. Kaur, and V. Gupta, "Implementing openflow based distributed firewall," in *2016 International Conference on Information Technology (InCITe) - The Next Generation IT Summit on the Theme - Internet of Things: Connect your Worlds*. IEEE, Oct. 2016. [Online]. Available: https://doi.org/10.1109/incite.2016.7857611

[18] "Open vSwitchl documentation," https://docs.openvswitch.org/en/latest/.

[19] F. X. Wibowo, M. A. Gregory, K. Ahmed, and K. M. Gomez, "Multi-domain software defined networking: Research status and challenges," *Journal of Network and Computer Applications*, vol. 87, pp. 32–45, Jun. 2017. [Online]. Available: https://doi.org/10.1016/j.jnca.2017.03.004

[20] "FortiGate 50E datasheet," https://www.fortinet.com/content/dam/fortinet/assets/data-sheets/FortiGate_FortiWiFi_50E_Series.pdf.

[21] P. Krongbaramee and Y. Somchit, "Implementation of SDN stateful firewall on data plane using open vSwitch," in *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, Jul. 2018. [Online]. Available: https://doi.org/10.1109/jcsse.2018.8457354

[22] J. Postel, "User datagram protocol," Tech. Rep., Aug. 1980. [Online]. Available: https://doi.org/10.17487/rfc0768

[23] ——, "Transmission control protocol," Tech. Rep., Sep. 1981. [Online]. Available: https://doi.org/10.17487/rfc0793

[24] "Firewall cli fortinet," https://docs.fortinet.com/document/fortigate/6.0.0/cli-reference/923548/firewall.

[25] F. N. Nife and Z. Kotulski, "Application-aware firewall mechanism for software defined networks," *Journal of Network and Systems Management*, vol. 28, no. 3, pp. 605–626, Mar. 2020. [Online]. Available: https://doi.org/10.1007/s10922-020-09518-z

[26] S. Kim, S. Yoon, J. Narantuya, and H. Lim, "Secure collecting, optimizing, and deploying of firewall rules in software-defined networks," *IEEE Access*, vol. 8, pp. 15 166–15 177, 2020. [Online]. Available: https://doi.org/10.1109/access.2020.2967503

[27] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software defined networking: State of the art and research challenges," *Computer*

Networks, vol. 72, pp. 74–98, Oct. 2014. [Online]. Available: https://doi.org/10.1016/j.comnet.2014.07.004

[28] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating linux security with eBPF iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM, Aug. 2018. [Online]. Available: https://doi.org/10.1145/3234200.3234228

[29] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. Boston, MA: Pearson, 2016.

[30] J. N. Bakker, I. Welch, and W. K. Seah, "Network-wide virtual firewall using SDN/OpenFlow," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2016. [Online]. Available: https://doi.org/10.1109/nfv-sdn.2016.7919477

[31] C. Basile, F. Valenza, A. Lioy, D. R. Lopez, and A. P. Perales, "Adding support for automatic enforcement of security policies in NFV networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 2, pp. 707–720, Apr. 2019. [Online]. Available: https://doi.org/10.1109/tnet.2019.2895278

[32] C. Basile, A. Lioy, C. Pitscheider, F. Valenza, and M. Vallini, "A novel approach for integrating security policy enforcement with dynamic network virtualization," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, Apr. 2015. [Online]. Available: https://doi.org/10.1109/netsoft.2015.7116152

[33] Fortinet, *FortiOS 6.2.0 Cookbook*, 6th ed., 2019.

[34] "IpFirewall user manual," https://docs.freebsd.org/en_US.ISO8859-1/books/handbook/firewalls-ipfw.html.

[35] "bpf-iptables git repository," https://github.com/mbertrone/bpf-iptables.

[36] "Iptables documentation," https://linux.die.net/man/8/iptables.

[37] "VErified REFinement and Optimized Orchestrator repository," https://github.com/netgroup-polito/verefoo.

[38] S. Davy, K. Barrett, B. Jennings, and S. van der Meer, "On the use of policy based management for pervasive m-government services," *in Proc. of the 1st Euro Conf. on Mobile Government , Euro mGov*, pp. 110–121, 01 2005.

Appendices

Appendix A

REST API

A.1 Design

In this section it is provided the REST APIs implementation in order to fully use this module with others in Software Defined Networking Environment. Now are presented all the resources that are possible to obtain through a request. The following methods can be summarized in the figure A.1:

- */fwd/nodes/addnfv*
  - ☐ POST
    This method accepts a *Network Function Virtualization* element and creates a set of nodes, the one compatible with a firewall node. If success the reply is the set of the current nodes available for the deployment with status code 200, otherwise a 400 status code will be emitted if some problem occurs.
- */fwd/nodes/addgraph*
  - ☐ POST
    This method accepts a *Graph* element and creates a set of nodes that have the capability to be deployed as firewalls. If success the reply is the set of the current nodes available for the deployment with status code 200, otherwise a 400 status code will be emitted if some problem occurs.
- */fwd/nodes*
  - ☐ POST
    This method accepts a *node* element that will be inserted in the list of available nodes. If success the reply is the id of the current node created having status code 200. If a node with the same id already exists a status code 409 will be emitted, otherwise, a 400 status code will be emitted if another problem related to the server occurs.
  - ☐ GET
    This method accepts two integers as a request parameter and displays

all the nodes between them with a status code 200. If the parameters are wrong, a 400 status code is emitted, while if there are no nodes between that parameters, a 404 status code is emitted.

- ☐ DELETE
  This method deletes all the nodes inside the application emitting the status code 204 if the deletion occurs without problems. If the list of nodes is empty, a 404 status code is emitted.

- */fwd/nodes/{nid}*

  - ☐ PUT
    This method accepts a *node* element and a *nid* and replaces the existing node with the new one sent. If it succeeds, the reply has status code 204, otherwise, a 404 status code will be emitted.
  - ☐ GET
    This method accepts a *nid* as a request parameter and retrieves the node with that particular identification number. The reply shows the node with status code set equals 200, otherwise a 404 status code will be emitted if the resource is not found.
  - ☐ DELETE
    This method accept a *nid* as request parameter and delete the node with that particular identification number. The reply is the status code equals to 204 if the operation is performed correctly, otherwise a 404 status code will be emitted.

- */fwd/nodes/{nid}/configuration*

  - ☐ POST
    This method accepts a *nid* as a request parameter and a *firewall* element and creates the firewall configuration for that node that is a set of policies. The reply is the list of policies of that node having a status code set to 201, while a 409 status code will be emitted if there is a conflict inside the resource or a generic 400 otherwise.
  - ☐ PUT
    This method accept a *nid* as request parameter and change the configuration of that node. The reply has status code setted to 204 if the content is changed correctly, otherwise a 404 status code will be emitted.

- */fwd/nodes/{nid}/configuration/firewall*

  - ☐ POST
    This method accept a *nid* as request parameter and a *element* element and creates a policy for that node. The reply is the policy id for that node having status code setted to 201, while a 409 status code will be emitted if there is a conflict inside the resource or a generic 400 otherwise.

- */fwd/nodes/{nid}/configuration/firewall/{eid}*

- ☐ PUT
  This method accepts an *element* element and retrieves *nid* and *eid* from the request parameter and replaces the existing policy of a node with the new one sent. If it succeeds, the reply has status code 204, otherwise, a 404 status code will be emitted if the resource is not found or 409 if the node does not exist.
- ☐ GET
  This method accept a *nid* and *eid* as request parameter and retrieve the policy with that particular identification node. The reply shows the policy with status code set equals 200, otherwise, a 404 status code will be emitted if the resource is not found or 409 if there is a problem retrieving the node.
- ☐ DELETE
  This method accept a *nid* and *eid* as request parameter and delete the policy with that particular node. The reply is the status code equals to 204 if the operation is performed correctly, otherwise a 404 status code will be emitted.

- */fwd/deploy/getIptables/*{*nid*}
  - ☐ GET
    This method accept a *nid* as request parameter and translate the policies contained in that node for an Iptables firewall configuration. The reply is the configuration file ready for the download with status code setted equals to 200, otherwise a 404 status code will be emitted if the resource it is not found.
- */fwd/deploy/getIpfw/*{*nid*}
  - ☐ GET
    This method accept a *nid* as request parameter and translate the policies contained in that node for an Ipfirewall firewall configuration. The reply is the configuration file ready for the download with status code setted equals to 200, otherwise a 404 status code will be emitted if the resource it is not found.
- */fwd/deploy/getBpfFirewall/*{*nid*}
  - ☐ GET
    This method accept a *nid* as request parameter and translate the policies contained in that node for a bpf-iptables firewall configuration. The reply is the configuration file ready for the download with status code setted equals to 200, otherwise a 404 status code will be emitted if the resource it is not found.
- */fwd/deploy/getOpenVswitch/*{*nid*}
  - ☐ GET
    This method accept a *nid* as request parameter and translate the policies contained in that node for an Open vSwitch firewall configuration. The

reply is the configuration file ready for the download with status code setted equals to 200, otherwise a 404 status code will be emitted if the resource it is not found.

- */fwd/deploy/getFortinet/{nid}*
  - ☐ GET
    This method accept a *nid* as request parameter and translate the policies contained in that node for a Fortinet firewall configuration. The reply is the configuration file ready for the download with status code setted equals to 200, otherwise a 404 status code will be emitted if the resource it is not found.

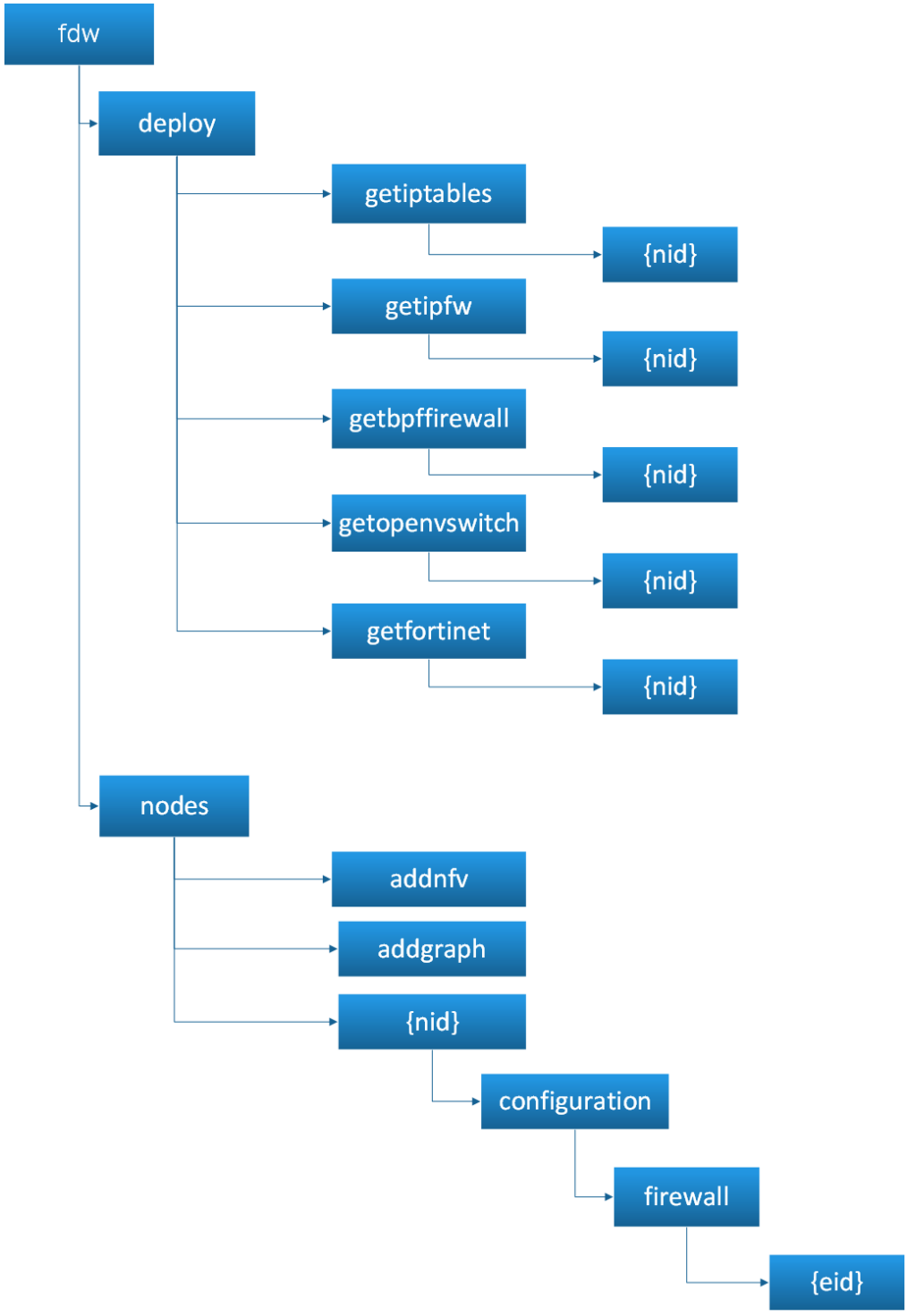

Figure A.1. Resource design

Appendix B

Test Replication

In this appendix will be explained all the techniques used to set up the environment for replicate the tests. They were performed on a machine having:

- Ubuntu 20.04.1 LTS as 64-bit operating system.
- Hard disk of 1T.
- Intel Core i7-4700MQ as processor.
- RAM installed equal to 11 GB.
- VirtualBox version 6.1.19 installed.

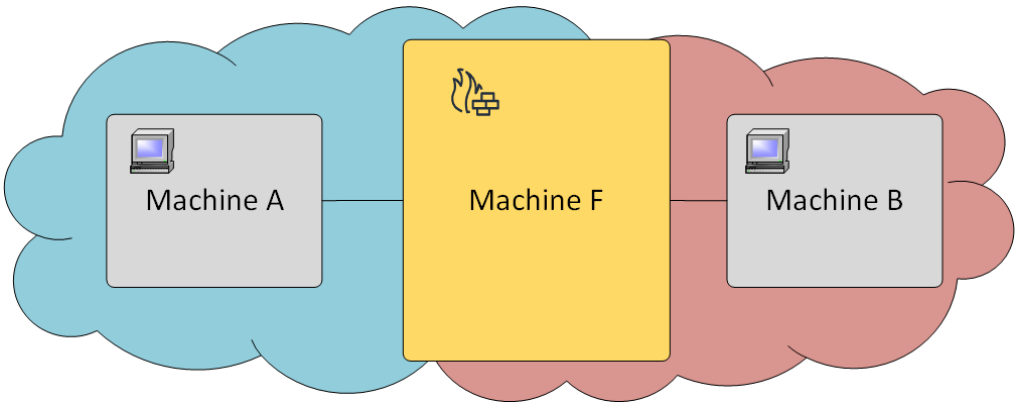

Figure B.1. Configuration used for test validation

Moreover the five use cases can be found inside the XML file *useCaseFirewallDeployment.xml* inside the folder *testfile/Others/* [37].

B.1 Iptables Scenario

In order to replicate this scenario it is needed to create three virtual machines:

- Machine A having Ubuntu 18.04.5 LTS, 1024 MB of RAM, and 10 GB of the virtual hard disk.
- Machine F having Ubuntu 18.04.5 LTS, 1024 MB of RAM, and 10 GB of the virtual hard disk.
- Machine B having Ubuntu 18.04.5 LTS, 1024 MB of RAM, and 10 GB of the virtual hard disk.

Now it is time to set up the network accessing to network management of VirtualBox and creating two subnets and editing them as follows:

Table B.1. Subnet set-up iptables scenario

Name	AddressIPv4	Netmask IPv4	Enable DHCP
vboxnet0	192.168.56.1	255.255.255.0	yes
vboxnet1	192.168.57.1	255.255.255.0	yes

Before assigning the subnet it is better to have Machine F in bridge mode and insert on terminal:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install net-tools
sudo sysctl net.ipv4.ip_forward=1
```

Listing B.1. Set-up Machine F iptables scenario

Now the machine should be shut down and before restarting it, Machine A and B should be set. It is possible to turn on both machines and install net-tools if it is not installed yet. After this operation, they have to be configured in the VirtualBox network panel.

- In Machine A will be enabled the first adapter with connection type " only hosts" selecting the subnet *vboxnet0*.
- In Machine B will be enabled the first adapter with connection type " only hosts" selecting the subnet *vboxnet1*.

Both machines should be restarted and check if the IP address is correctly configured using the command *ifconfig* on the terminal.

- Machine F network is configured on Virtualbox enabling two network adapter and setting both on "only hosts". The first adapter should be linked to *vboxnet0*, while the second should be linked to *vboxnet1*.

It is possible to restart also this machine and set up one configuration file produced by the module developed during this thesis. After checking the IP address given by the DHCP server using the command *ifconfig* on the terminal, it is possible to set up Machine A:

```
sudo ip route add 192.168.57.0/24 via 192.168.56.X dev enp0s3
   #substitute X with the value set by DHCP on machine F
```

Listing B.2. Enable Machine A routing to Machine B iptables scenario

And machine B:

```
sudo ip route add 192.168.56.0/24 via 192.168.57.X dev enp0s3
   #substitute X with the value set by DHCP on machine F
```

Listing B.3. Enable Machine B routing to Machine A iptables scenario

After running of the module developed during this thesis having as input file the one defined at the beginning of this appendix and IPTABLES as enum parameter, there are several files in the main directory of the project. Each file having the prefix *iptablesFirewall_* followed by the use case number plus *.sh* can be deployed on this machine F enforcing the related rules.

Then it is possible to check if the rule deployed are correctly enforced using ping command or sudo nc -l [*portNumber*] on server side and nc [*serverIpAddress*] [*portNumber*] on client side for transport layer simulation.

B.2 IpFirewall Scenario

In order to replicate this scenario it is needed to create three virtual machines:

- Machine A having Ubuntu 18.04.5 LTS, 1024 MB of RAM, and 10 GB of the virtual hard disk.
- Machine F having FreeBSD 12.1, 1024 MB of RAM, and 16 GB of the virtual hard disk.
- Machine B having Ubuntu 18.04.5 LTS, 1024 MB of RAM, and 10 GB of the virtual hard disk.

After running of the module developed during this thesis having as input file the one defined at the beginning of this appendix and IPFIREWALL as enum parameter, there are several files in the main directory of the project. Each file having the prefix *rc_* followed by the use case number plus *.rules* can be deployed on this machine F enforcing the related rules.

Now it is time to set up the network accessing to network management of VirtualBox and creating two subnets and editing them as follows:

Table B.2. Subnet set-up IpFirewall scenario

Name	AddressIPv4	Netmask IPv4	Enable DHCP
vboxnet0	192.168.56.1	255.255.255.0	yes
vboxnet1	192.168.57.1	255.255.255.0	yes

Now it is possible to set-up the environment of the F machine typing the following commands in the Command Line Interface:

```
sysrc firewall_enable=''YES''
sysrc firewall_script=''/etc/rc_1.rules'' #or the firewall rule
   name according to the use case
service ipfw start
```

Listing B.4. Set-up Machine F IpFirewall scenario

In order to enable packet forwarding, it is possible to edit the default file /etc/rc.conf changing *gateway_enable* = "*NO*" into *gateway_enable* = "*YES*" using a text editor.

Now the machine should be shut down and before restarting it, Machine A and B should be set. It is possible to turn on both machines and install net-tools if it is not installed yet. After this operation, they have to be configured in the VirtualBox network panel.

- In Machine A will be enabled the first adapter with connection type *" only hosts"* selecting the subnet *vboxnet0*.
- In Machine B will be enabled the first adapter with connection type *" only hosts"* selecting the subnet *vboxnet1*.

Both machines A and B should be restarted and check if the ip address is correctly configured using the command *ifconfig* on the terminal.

- Machine F network is configured on Virtualbox enabling two network adapter and setting both on "only hosts". The first adapter should be linked to *vboxnet0*, while the second should be linked to *vboxnet1*.

It is possible to restart also this machine and set up one configuration file produced by the module developed during this thesis. After checking the IP address given by the DHCP server using the command *ifconfig* on the terminal, it is possible to set up Machine A:

```
sudo ip route add 192.168.57.0/24 via 192.168.56.X dev enp0s3
   #substitute X with the value set by DHCP on machine F
```

Listing B.5. Enable Machine A routing to Machine B IpFirewall scenario

And machine B:

```
sudo ip route add 192.168.56.0/24 via 192.168.57.X dev enp0s3
   #substitute X with the value set by DHCP on machine F
```

Listing B.6. Enable Machine B routing to Machine A IpFirewall scenario

Then it is possible to check if the rule deployed are correctly enforced using ping command or sudo nc -l [*portNumber*] on server side and nc [*serverIpAddress*] [*portNumber*] on client side for transport layer simulation.

B.3 BPF-Iptables Scenario

In order to replicate this scenario it is needed to create three virtual machines:

- Machine A having Ubuntu 18.04.5 LTS, 1024 MB of RAM, and 10 GB of the virtual hard disk.
- Machine F having Ubuntu 18.04.5 LTS, 4096 MB of RAM, and 94 GB of the virtual hard disk.
- Machine B having Ubuntu 18.04.5 LTS, 1024 MB of RAM, and 10 GB of the virtual hard disk.

Now it is time to set up the network accessing to network management of VirtualBox and creating two subnets and editing them as follows:

Table B.3. Subnet set-up bpf-iptables scenario

Name	AddressIPv4	Netmask IPv4	Enable DHCP
vboxnet0	192.168.56.1	255.255.255.0	yes
vboxnet1	192.168.57.1	255.255.255.0	yes

Before the assignment of the subnet it is better to set-up Machine F in bare metal configuration inserting on the terminal the following commands:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install git
copy polycube
cd polycube/
sudo ./SCRIPTS/install.sh pcn-iptables
sudo polycubed --daemon
sudo pcn-iptables-init
sudo script
sudo apt-get install net-tools
sudo sysctl net.ipv4.ip_forward=1
```

Listing B.7. Set-up Machine F bpf-iptables scenario

Now the machine should be shut down and before restarting it, Machine A and B should be set. It is possible to turn on both machines and install net-tools if it is not installed yet. After this operation, they have to be configured in the VirtualBox network panel.

- In Machine A will be enabled the first adapter with connection type " only hosts" selecting the subnet *vboxnet0*.
- In Machine B will be enabled the first adapter with connection type " only hosts" selecting the subnet *vboxnet1*.

Both machines should be restarted and check if the IP address is correctly configured using the command *ifconfig* on the terminal.

- Machine F network is configured on Virtualbox enabling two network adapter and setting both on "only hosts". The first adapter should be linked to *vboxnet0*, while the second should be linked to *vboxnet1*.

It is possible to restart also this machine and set up one configuration file produced by the module developed during this thesis. After checking the IP address given by the DHCP server using the command *ifconfig* on the terminal, it is possible to set up Machine A:

```
sudo ip route add 192.168.57.0/24 via 192.168.56.X dev enp0s3
   #substitute X with the value set by DHCP on machine F
```

Listing B.8. Enable Machine A routing to Machine B bpf-iptables scenario

And machine B:

```
sudo ip route add 192.168.56.0/24 via 192.168.57.X dev enp0s3
   #substitute X with the value set by DHCP on machine F
```

Listing B.9. Enable Machine B routing to Machine A bpf-iptables scenario

After running of the module developed during this thesis having as input file the one defined at the beginning of this appendix and EBPF as enum parameter, there are several files in the main directory of the project. Each file having the prefix *bpfFirewall_* followed by the use case number plus *.sh* can be deployed on this machine F enforcing the related rules.

Then it is possible to check if the rule deployed are correctly enforced using ping command or sudo nc -l [*portNumber*] on server side and nc [*serverIpAddress*] [*portNumber*] on client side for transport layer simulation.

B.4 Open vSwitch Scenario

In order to replicate this scenario it is needed to create just one virtual machine with the following specifications:

- Machine F having Ubuntu 18.04.5 LTS, 4096 MB of RAM, and 40 GB of the virtual hard disk.

Then are needed two namespaces in order to test the firewall configuration of this scenario. In particular, are defined the namespace left and right in order to put machine F in the "middle". The first commands to insert in the terminal of machine F are:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install net-tools
sudo ip netns add left
sudo ip link add vethl0 type veth peer name vethl1 netns left
sudo ip netns add right
sudo ip link add vethr0 type veth peer name vethr1 netns right
sudo apt install openvswitch-switch
```

Listing B.10. Set-up F machine for Open vSwitch scenario

Now that the main points of the environment are set, it is possible to define the first namespace using the following commands:

```
sudo ip netns exec left sudo ip addr add 192.168.56.3/24 dev
   vethl1
sudo ip netns exec left sudo ip link set dev vethl1 up
sudo ip netns exec left sudo ip route add 192.168.57.0/24 dev
   vethl1
```

Listing B.11. Set-up left namespace

In order to achieve the configuration of the second namespace, the commands to use are:

```
sudo ip netns exec right sudo ip addr add 192.168.57.3/24 dev
   vethr1
sudo ip netns exec right sudo ip link set dev vethr1 up
sudo ip netns exec right sudo ip route add 192.168.56.0/24 dev
   vethl1
```

Listing B.12. Set-up right namespace

The last thing to do is to create the open vSwitch bridge that will be used as a point to deploy the different flows outlined by the configuration file. It can be done using the following commands:

```
sudo ovs-vsctl add-br b0
sudo ovs-vsctl add-port b0 vethl0
sudo ovs-vsctl add-port b0 vethr0
sudo ip link set vethr0 up
sudo ip link set vethl0 up
```

Listing B.13. Set-up Open vSwitch after namespace creation

Another step to do is to enable the left namespace loopback interface and the right namespace loopback interface using the following piece of code:

```
sudo ip netns exec left sudo ip link set lo up
```

Listing B.14. Last settings in left namespace

```
sudo ip netns exec right sudo ip link set lo up
```

Listing B.15. Last settings in right namespace

The environment is finally set and it is possible to install one of the configuration files produced by the module developed during this thesis for this scenario having as input file the one defined at the beginning of this appendix and OPENVSWITCH as enum parameter. Several files in the main directory of the project will be created having the prefix *ovsFirewall_* followed by the use case number plus *.sh* that can be deployed on this machine enforcing the related flows.