POLITECNICO
DI TORINO

Master's Degree in Computer Engineering

Master Degree Thesis

Automating the Deployment of Security Functions in Virtualized Networks

Supervisors
Prof. Riccardo SISTO
Prof. Fulvio VALENZA
Prof. Daniele BRINGHENTI

Candidate
Yasser HOBBALLAH

Academic Year 2021-2022

Summary

New frameworks that are developed daily must adhere to a specification, meaning that the framework should ensure certain expectations are satisfied.Usually, a developer goes through a process to establish the correct operation of a particular framework. This process, verification and validation, is typical in software to ensure that the software acts as expected. There are many techniques for verification and validation, one of which is software testing. Others may include testing at the logical level or even deploying the framework/software to be tested in specific conditions that replicate the conditions of a production real environment. However, regarding virtual environment creation, there is no standard process in the literature to create such an environment, and it largely depends on where the testing is performed. Some organizations may reproduce the whole physical environment (thus paying a lot of costs) to test a framework. Other organizations with limited resources may need to find alternative solutions such as virtual machines or containers. Building a virtual environment comes with many challenges, and a complexity that is difficult to manage. This thesis will target this challenges and try to solve them.

This thesis contributes to developing and testing VEREFOO (VErified REFinement and Optimized Orchestration), a framework that aims to provide a Security Automation approach. Currently, the VEREFOO framework supports mainly the firewall feature of security automation. The thesis focuses primarily on the following aspects, testing the low-level configurations of firewalls produced by the framework, then developing a testing environment that is cost-effective, dynamic, and can be deployed with minimal resources to test such framework correctly. Although the designed virtual environment is explicitly tailored to test firewall configurations produced by the framework, it can be further extended to test other frameworks. Furthermore, we combine the different stand-alone parts (GUI-virtual environment-VEREFOO firewall output...) of the VEREFOO framework into a single process through a demo demonstration.

This thesis will show how framework testing in virtual environments can be improved, extended, and better classified concerning automation aspects. The main contribution lays in the investigation of, and the improvement in, issues related to achieving a high level of automation, which will be evident at the end of this thesis

by presenting a developed translator algorithm customized for creating a virtual environment exploiting automation to the maximum.

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 Daniele Bringhenti, which of whom have provided patient advice and guidance throughout the research process. Thank you all for your unwavering support.

Yasser Hobballah

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing has evolved in recent years from manual testing to automated testing. There are several testing tools, techniques, and frameworks available today that are crucial to the error-free development of software. Software testing is simply the skill of examining software to make sure that the quality being tested is consistent with the functional requirements and flaws in the system are identified effectively. New technologies are constantly changing the way we perform our testing, one of these technologies is virtualization. Virtualization simplified a lot of the functions, rather than spending high costs on physical infrastructure to perform specific software testing, we can create a virtual environment that mimics a particular real physical environment. With the conventional method, specific hardware had to be installed in order to provide services like load balancing and web caching. However, with the advent of these cutting-edge networking technologies, virtualizing network operations may soon become commonplace. Actually, virtual functions can be installed dynamically on servers and started or stopped as needed.

The problem which arises, is that currently an automatic process to build the virtual environment functions to deploy in order to satisfy conditions for testing software does not exist. Given the need for precise testing of software in a network environment and the unwillingness to bear its costs, the thesis discovers the different possible methods and proposes an approach for testing that exploits the capabilities of virtualization technology to build a virtual network environment which is similar to the real network environment and automate such process achieving high levels of automation. Furthermore, this thesis focuses on proposing a methodology using white box testing to find best test cases to achieve an error-free software.

Although there is no standard approach of testing, different methodologies are present but it all depends on the availability of resources and technologies. However, what is certain, is the need to achieve a high degree of verification and validation of software. White box testing, which is the first technique employed in this dissertation, is the state-of-the-art testing approach that the tester must choose

from when determining which are the best technique to use. This thesis offers a possible methodology to employ whit box testing in an effective way and provide a good test bed to effectively ensure high degree of verification and validation. After performing the first phase of testing that is software testing of code using test cases, then follows testing in virtual environment.

Tester must run the software under specific conditions that mimic the environment that the software will be deployed on. This requires building such environment which largely depend on the availability of physical resources. Due to high cost of physical replication of resources in most cases, the last resort would be virtualization. In fact, it largely depend on the type of software to test. Building an automated virtual environment requires a good analysis of the purpose of the environment. Depending on the purpose, the complexity of building such environment may increase or decrease. In the context of virtualized networks, the exact replication of the physical environment is very complex due to the network variables that may exist in the real environment but may not be present in a contained virtual environment. In addition to the difficulty of virtual network management, there is the issue of automation. Automating the virtual network creation and functioning may require a lot of effort in different aspects, most of the automation is done to avoid human errors and repetitive tasks. A high degree of automation, ensures a better working environment that is more robust and behaves as expected. Until now, in the literature there is no standard approach to automation of such environment, this dissertation will offer a possible approach for building and automating such virtualized network using different possible virtualization techniques.

In this thesis, the testing is done in two phases; first phase is standard software testing of code and second is testing in a virtual environment correct functioning of software. The above mentioned methodologies are applied to the already exiting framework VEREFOO (VErified REFinement and Optimized Orchestration) which is developed in order to achieve a policy refinement in Network Function Virtualization environment. This thesis will focus on the testing of the already developed features of the framework. Moreover, it will automate the testing in a virtual environment.

Chapter 2

Background: VEREFOO Framework

2.1 Introduction

New emerging technologies like Network Functions Virtualization (NFV) and Software-Defined Networking (SDN) are intended to improve networking management and add possible features to networking flexibility [1] [2] [3]. NFV provides virtualized network functionalities placed on standard general purpose servers in the cloud, and SDN permits the defining of the paths that traffic flows must cover. Through the use of Service Graphs (SGs), which represent the related service functions and their connections, these capabilities enable service designers to specify the planned network services. The SG have enabled a new level of abstraction that leads to decoupling of the computing and physical infrastructures. It is a virtual network's logical representation that is separate from the physical infrastructure. In other words, there is no need to include the low level details of the physical infrastructure when performing analysis as the logical level, this opens new possibilities.

These ideas can also be used for network security functions (NSFs), such as the packet filter firewall and the intrusion detection system (IDS) exploiting abstractions. In fact, these NSFs are best deployed as virtual services on-demand to enable a quicker response to incoming attacks. Additionally, to relieve the service designer's workload, their configuration could be automated through the use of a distribution process, whose job it is to develop the necessary policy rules to abide by the Network Security Requirements (NSRs), which stand for the security requirements that must be met, such as the requirement for isolation between a server and a group of hosts. The focus of the following methodology is on packet filters, which represent the most common firewall technology and the most frequently exploited

security defense in networks.

In the context of packet filters, How to enforce the Network Security Requirements (NSRs) that an SG should meet becomes a challenge in this situation. Traditionally, Network Security Functions (NSFs) configuration is frequently done manually. This method not only requires a lengthier response time when network attacks are discovered, but it is also vulnerable to human error, which can result in the creation of vulnerabilities. For instance, the security manager can forget to define one of the hundreds of rules necessary to apply a specific isolation policy, rendering isolation ineffective. In this context, the articles [4] [5] [6] [7] proposes a new methodology that addresses the open issues. The goal is to provide an automatic way to allocate packet filters in a SG defined by the service designer, and to create firewall rules automatically, so as to satisfy the specified security requirements. The approach is founded on a formal model that offers confidence that the ultimate solution truly satisfies the security requirements. Another goal of this approach is: minimizing the number of firewalls to be allocated and the number of rules to be configured in each one of them increases the performance of the overall architecture, while reducing its cost in terms of employed resources.

2.2 VEREFOO Objectives

The VEREFOO (VErified REFinement and Optimized Orchestrator) is a framework developed to offer an automatic method of allocating packet filters—the most popular and traditional firewall technology—in a Service Graph defined by the service designer, as well as an auto-configuration technique to create firewall rules with respect to the specified security requirements.

The previous section describes the main purpose of VEREFOO framework which is discussed in articles [8] [5] [6] [7], that is implementing security automation. Currently, the framework mainly supports firewall deployment. The previous defined methodology to achieve automation in allocating firewalls and there respective rules, satisfying the required NSRs, is achieved using VEREFOO framework [9]. This framework intends to offer automatic network function deployment inside a specified group of nodes. In order to ensure proper enforcement of the nodes, it additionally optimizes the positions of the various nodes by using z3 as a solver for the MaxSMT problem [10] [5] [6]. It may generate the appropriate number of firewalls required and retrieve a configuration made up of abstract rules defined using the Medium Security Policy Language [1] (which is the second layer of policy abstraction) syntax given a set of isolation and reach-ability requirements for a Virtual Network system.

The issue addressed by the framework is how to automatically determine the best allocation strategy and configuration of packet filtering firewalls in an SG

while satisfying a set of NSRs. This was discussed previously in [5] [11]. More precisely, the idea is to formulate the problem as a partial weighted Maximum Satisfiability Modulo Theories (MaxSMT) problem and solve it accordingly [11]. Solving the MaxSMT will analyze the service graph (or Allocation Graph) and with mathematical calculations that offer formal correctness, it will find the optimal solution to allocate the firewalls with respective firewall rules. The next section defines the high level operation of VEREFOO.

2.3 VEREFOO High-Level Operation

The Service Graph (SG) is the a high level abstraction of the definition of the network topology, along with the Network Security Requirements (NSRs) which are the security constraints that the network behavior must implement. The technique focuses on connection requirements, that is the definition of which traffic flows must be allowed (or disallowed) between any pair of end points in the SG. These security limitations serve as the framework's second input (first input is the SG) and are distinguished by two features: A set of specific Network Security Requirements (NSRs), each of which specifies whether a traffic flow must be allowed (reachability requirement) or must be blocked by a firewall; and A general behavior representing the default rule applied to traffic flows for which the user does not specify any further indication (isolation requirement).

Both the SG and the NSRs are used as input the VEREFOO Framework as an XML file. The framework solves a huge issue in security configuration of firewalls, that traditionally is a time consuming and error prone security task that could be made easier by automating the creation of firewall rules and deploying there positions in the network. The following Figure summarizes the steps that the framework undergo.

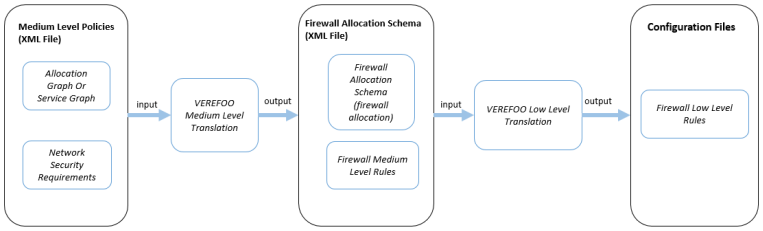

Figure 2.1: Verefoo Process

In Figure 2.1. First, there is the input Service Graph (SG) XML file. The Service Graph defines the network topology in an XML file manner with different XML elements. The SG could also be an Allocation Graph, which contains AP

(Allocation points; that are possible positions in the network graph where firewalls could be deployed). If the SG was chosen as input the framework will automatically convert it to Allocation Graph with Allocation points. Also the Network Security Requirements (NSRs) are defined. Translation from medium-level policies (MLP), will output a Firewall Allocation Schema (FAS) along with Firewall Medium Level Rules (FMLR). Then, after the FAS generation, the output XML file is used as input to the second part of the framework, which is responsible for translating the medium-level firewall rules to low-level rules specific to a firewall type (tables - openvswitch ...). The output files are Configuration files ready to be deployed on firewalls.

Moreover, the translation that is done from Medium Level Polices (AG and NSRs) to the Firewall Allocation Schema is the output of the calculations done by the solving the MaxSMT problems, this ensures the correctness of the allocated firewalls and there respective policies written in XML. However, for the low level translation there is the usage of trans-compiling to turn the medium level policies into low level configuration files, this is discussed in [1]. This summarizes the high level operation of VEREFOO.

The Firewall Allocation Schema output of the framework is already verified and validated with extensive testing in the framework. but the low level rules translation is not, this thesis will concentrate on testing of this translation by various methods like white box testing or testing in virtual environment to validate the correct operation of the firewalls.

2.4 Limitations of the VEREFOO

As specified previously, some parts of the framework are not tested yet. Specifically the translation from medium level policies to low level configuration files is not verified and validated yet at different levels. In addition to that currently the framework is a collection of stand-alone parts. This thesis solves this issues related to verification and validation of the low level configurations of the framework and demonstration of the different stand-alone parts of the framework.

Currently the framework produces low level firewall configuration files in the form of scripts ready to be deployed on firewalls. However, there is nothing to guarantee the correctness of the these configuration files, previous testing that was done was limited and did not achieve high coverage. Furthermore, the previous testing was done only at the software level, there was no creation of environment to test firewalls in.

To overcome these limitations and solve them, a dedicated analysis should be done. Verification and validation of this part of the framework requires understanding well what are the best methods to choose in order to achieve best results. Of

course, taking into consideration the available resources.

Chapter 3

Background: Virtual Environment

3.1 Virtual Environment For Testing

In general, before deploying any software or framework into production, the framework must be extensively tested by using the well know verification and validation approach. Usually the well-known white box testing approach used in software testing is adopted. But in some cases this is not enough, the framework may behave as expected in terms of software testing but when subjecting it to real production conditions results may vary due to many external factors. For that reason, in addition to white box testing approach, it is suggested to deploy frameworks in virtual environments and extensively test the framework in such environment before deploying it to production.

According to NIST [12] quoting, "Testing should be completed on a test network without connectivity to the production network. This test network should attempt to replicate the production network as faithfully as possible, including the network topology and network traffic that would travel through the firewall". So the virtual environment should be able to replicate the real physical environment with real conditions as faithfully as possible. Obviously, the best way is to replicate the whole physical environment, but that requires alot of cost which is not feasible in most cases. The alternatives are using virtual machine or containers to replicate the environment which are more reasonable in terms of cost.

The next section will briefly introduce the possible virtualization technologies used in this thesis and how they are useful for building a virtual environment.

3.2 Virtualization Technologies Key Concepts

The most well-known virtualization technologies in this era are virtual machines and containers. They are used for different cases, each technology have its own advantages and disadvantages. Based on these technologies characteristics, they could be used in different scenarios.

In order to run applications and run programs, a virtual machine (VM) uses software as opposed to a physical computer. On a physical "host" machine, one or more virtual "guest" machines are active. Even when they are all running on the same host, each virtual machine has its own operating system and operates independently of the others. This implies that a physical PC may, for instance, host a virtual Linux system or even a windows operating system. As mentioned in Figure 3.1, using a hypervisor that runs on top of the infrastructure or host OS, it is possible to create multiple virtual machines, each with its own operating system. Each virtual machine operating system is called guest OS.

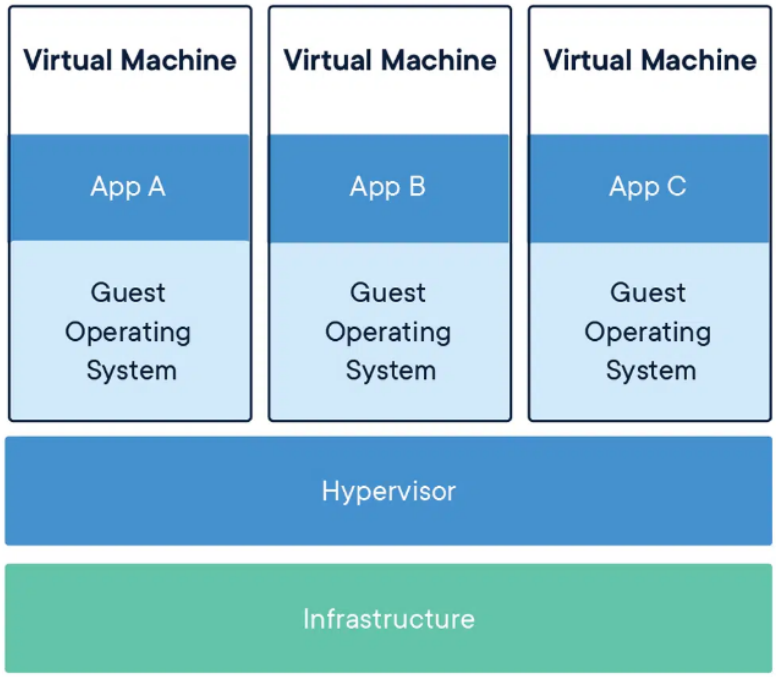

Figure 3.1: Virtual Machines

There are several application cases for virtual machine technologies in both on-premises and cloud contexts. In order to provide more cost-effective and flexible computation, public cloud services have more recently started employing virtual machines to deliver virtual application resources to multiple users simultaneously.

Virtual machines are known to be:

- Flexible where each vm has its own OS so that allows for multiple OS type support
- Can spawn in matters of seconds to minutes
- Can store data without losing them when creating a virtual disk
- flexible can support different flavors of operating systems
- isolation where each virtual machine is isolated from other machines running on same host also good from security point of view
- performance is not native and not fast as an operating system running on bare metal

The virtual machine technology is largely used in enterprises for deploying applications or creating test environment for certain products before deploying in production.

On the other hand, Docker is a platform for launching applications in software containers. Docker is a tool that can package an application and its dependencies into an isolated containers, which can be run on any server or anywhere. Docker removes repetitive tasks and is used throughout the development life cycle for easy and portable application development. So it is directed toward application and software development.

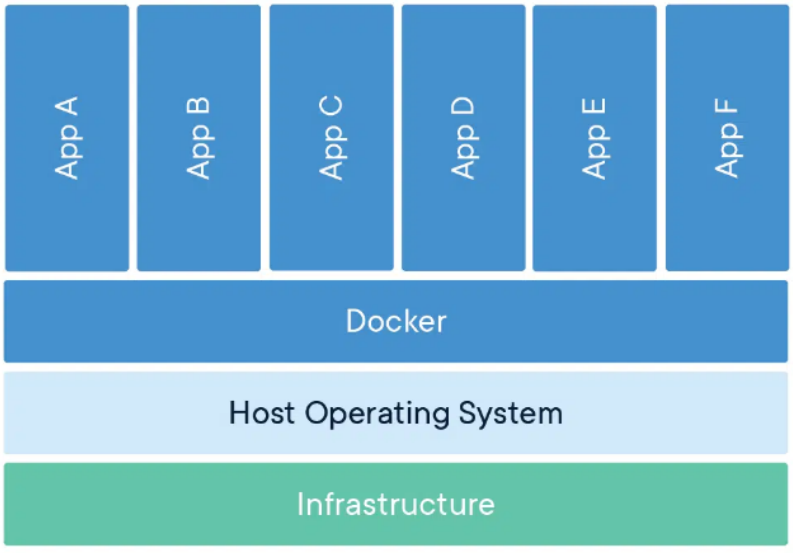

Figure 3.2: Container on host operating system

As specified by the docker documentation [13] and shown in Figure 3.2 which is taken from the documentation. An isolated docker container runs on top of a operating system with lightweight virtualization layer. All the containers share the same host operating system kernel but have different file systems. This allows each container to run an application with certain configurations. Containers are known to be:

- Very lightweight (consume much less resources than virtual machines)
- Fast to boot where it can boot up in matters of seconds
- easily created and destroyed, as they are highly volatile
- flexible can support different flavors of Linux operating systems

An application can be deployed using collection of containers where each container does a specific part of the application. The huge advantages of docker containers is the ability to run anywhere, since if it works on one machine it will work on every machine, this is what allows portability and containment.

Docker is tailored toward application development, but can also be used in order to create a virtual environment. The advantages it offers in terms of limited resource consumption makes it a good candidate for replicating the real environment using containers. In case of VEREFOO framework, a test environment would include the replication of network node functionalities like firewalls, routers, endpoints... Although docker isn't specifically built to emulate networks, it is possibly to use it in building a network environment based on containers. This is only possible because of the flexibility that docker containers offer and the ability to emulate different network topology using Linux operating systems.

3.3 Comparing Virtualization Technologies

Creating a virtual environment requires specifying the testing environment's requirements to correctly reproduce the actual environment in which the firewall will be present. Following the NIST recommendation reproducing a network similar to the real-production network is not easy, as it requires a lot of resources. The environment is created in its most simple possible scenario yet sufficient one that satisfies our needs.

In general, when it comes to building a contained virtual environment, a lot of aspects must be considered. The most relevant ones:

1. Where the virtual environment will be reproduced and the availability of resources: one of the essential things to consider is resources availability, if

the environment will be deployed on a cluster of servers or the cloud, or even locally on a single PC. This determines the number of resources available that can be reserved for a particular environment building.

2. target platform: The platform where the environment will be built determines if certain features are available to the environment. For example, in some cases the virtual environment could be deployed on a host Linux OS.

3. Flexibility and scalability: Flexibility of the virtual environment is of importance and the ability to manipulate it quickly to make changes to it without changing too much in code. For scalability, a topology may contain 12 nodes, But what if the testing extended toward a more ramified network with much more nodes (30+ nodes) in the environment to be simulated, Keeping that in mind helps us choose an environment that is capable of scaling up becoming more ramified for more complex testing. But other limitations for scalability exist, like routing (of course, static routing becomes too much for a large complex network).

4. Effort for producing environment and Fidelity of the environment: Finding a virtual network environment that replicates the real physical one as much as possible. The well-known building blocks to reproduce a virtual environment are:

   (a) Physical topology reproduced
   (b) Network emulation software (GNS3. . .)
   (c) Virtual machines
   (d) Containers

   To pick out of the four options, different factors must be considered. The most important one is the Environment purpose. Consider that the virtual network Environment is needed for testing firewalls, so most of the work will be at the firewall level. The firewall nodes will do most of the computation and be our primary focus. Endpoints, on the other hand, are just there for basically two things, Receiving an IP address and pinging another endpoint for firewall testing these two functionalities are minimal. There is no need for the application to be run on endpoints. Same for routers that only require route packets in the topology to reach the destination. At least for our topology, routers also are minimal. In the future, routers may become more complex since many of their network functionalities are discarded here (no dynamic routing, no Access lists at the router level, no NAT . . .).

   First, Network emulation software are discarded as they are not considered an option in this work. As for the three other options, they are classified as follows.

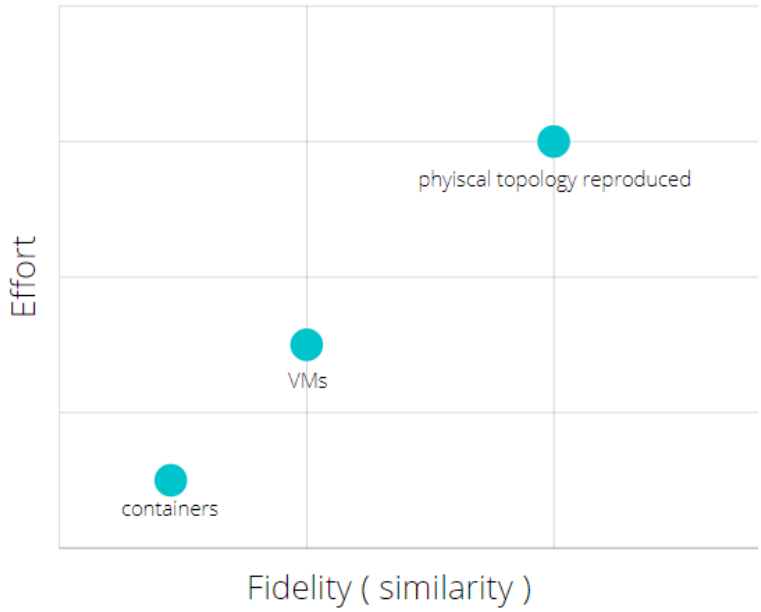

Figure 3.3: Compare different building blocks for virtual network

As seen in Figure 3.3, showing the comparison of three possible mechanisms in terms of effort and fidelity, where effort is the resources and time needed to reproduce the environment using a specific mechanism. And fidelity represents to what extent the virtual environment replicates the actual physical topology. As stated in the figure, physical topology reproduced, of course, has the highest fidelity as it mimics the actual physical topology, but it also requires the most effort. Then comes VMs, which replicate the actual topology in most of the functionalities and require much less effort to be reproduced. Finally, there is containers requiring the slightest effort to reproduce, which is a huge advantage but also has the lowest fidelity as it represents the actual topology with minimal similarities.

After specifying the different aspects to be considered when deciding on a suitable tool for building the virtual network, the next section displays the different tools that can be used in for virtualization with their advantages/disadvantages.

3.4 Tools

This section explains how previous technologies are used, and what tools allows us to use them effectively for the purposes of building a virtual network environment. The tools to be used for virtualization should manage the:

1. The target of the testing environment
2. Limited amount resources allocated for the environment
3. Automation and reproducibility, so that the environment is reproduced automatically each time the testing needs to be done

Satisfying the previous conditions requires to overcome many challenges that one must face in order to ensure the correct creation of an error free virtual network environment. Ensuring that the environment is error free is important, since the testing environment should not affect the test results that are produced when testing a particular framework or product in this environment. In order to build such environment, it is possible to state the open-source and free tools advantages and disadvantages:

1. First there is the Network Emulation Software (e.g. GNS3 ..) which are widely used in the market. The most important thing is that they cover a lot of features and uses Network operating systems that allows the usage of more functionalities. Emulation softwares are characterized by:
   - + build specifically for network emulation
   - + free versions are available
   - + has user interface (drag and drop)
   - + has dedicated network operating systems on nodes like routers and switches that facilitate complex network operations to be done in other environment
   - − heavy to open and run
   - − harder to automate using CLI
   - • usage of network emulation softwares is not considered in our case
2. Second there is Virtual Machine Orchestrators (e.g. OpenStack ..) and Orchestrators are used to launch a couple of virtual machines together rather than configuring each one and launching it. They are used to facilitate the process of launching virtual machines also they can be couple with different play books to define VMs using declarative language. They are characterized by :
   - + Are common to use in medium and large enterprises since they are proven to replicate real environment with high fidelity.
   - + resembles real-topology
   - + each vm has its own OS so that allows for multiple OS type support
   - + using the right OS can limit the amount of to be consumed

+ no problem with data persistence in case the environment was to be saved
− performance is slower than the native physical nodes or containers
− Consume a lot of resources compared to other more lightweight solutions
− Not purpose built for network topology virtualization, so many unnecessary functionalities may be present on the nodes.
• boot time of VMs is higher than containers but can be over looked for now

3. Finally there is Containers (e.g. docker or docker compose for launching couple of containers together), they are characterized by :

+ lightweight, can spawn many nodes on a single machine without any performance issues
+ flexible and scalable in terms of resources, can handle virtualization of a more complex and ramified network without worrying about resourcese running out
+ can be automated using CLI
+ fast to deploy and remove (boots in seconds)
− problem with data persistence in case the environment was to be saved
− Not purpose built for network emulation, so alot of network functionalities would be hard to emulate using containers
− No user interface, must of work done through CL and terminals
− doesn't support variety of OS, since all containers share the same kernel

The reasonable technologies are VMs and containers (since the usage of network emulator softwares is not free and not always open-sources), and we will see the available methods to automate and create the virtual environment. The next sub-sections will discuss some of the different open source tools and their way of automating the building of a virtual environment. Nevertheless, in our case, the virtual environment is built using one of them. Note that the development environment is Linux as other platforms offer many limitations to development.

3.4.1 Vbox Manage

The first tool that comes to our mind when using VMs is the oracle virtual box, but the oracle virtual box uses a GUI to deploy VMs, so how to automate the VM deployment. In fact, Oracle's VM Virtual Box has all the needed features in the back end of the virtualization server: the automated creation of VMs with scripting and easy networking of virtual machines. Can easily be done using Vbox

Manage, the Command line part of oracle Vbox. Vbox manage can automate Vms deployment by Using a bash script with a series of commands that are used to build the environment from scratch to run the environment. After running it for the first time, the VMS will be spawned. The environment will be persistent (i.e. after turning off the VMS no need to re-run the build of the environment script), or it can be deleted by another bash script that deletes the newly created testing environment (deletes all VMS and network created).

Next there is an overview of the automated building of a virtual network environment using Vbox Manage. Step-by-step procedure is followed for creating a VM using vbox manage and deleting it. After specifying the commands for creating one VM and attaching it to one internal network that is also created using the command line, the whole process can be automated by putting a series of commands to be executed inside a bash script to create VMS and networks then attach VMS to specifically created networks.

Pros and Cons:

+ automated deployment

+ persistant, once created no need to re-create it unless environment is deleted

+ each vm has its own OS so that allows for multiple OS type support

+ can support firewall types iptables- bpf_firewall-ipfirewall-openvswitch

− environment creation is slow as alot of nodes need to be created and deployed each with its own os

− needs high resources compared to containers so not suitable for large number of nodes, if the virtual environment is produced on single machine

− no UI of how nodes connections are done

− not flexible in terms of networking as we have to manually attach vms to networks using CLs

− too much Command line usage for automation so it is hard to trace when developing a larger and more ramified environment with many nodes.

− even when environment is created re-launching environment for testing require time, as each node has its own OS to boot unlike containers where launch time can reach milli seconds.

− flexibility is limited, since adding a node and removing a node requires entering alot of commands to ensure all the environment creation is done without conflicts

Example Deployment of one VM using VBox Manage for more information refer to the manual [14]. The following commands configure the VM settings. Note that Vbox Manage installation is required.

```
$ VBoxManage createvm --name ubuntutest --ostype Ubuntu_64 --register # create and register the vm

$ VBoxManage createmedium --filename /home/mibeyki/VirtualBox VMs/ubuntutest/ubuntutest.vdi --size 10240 # create a storage medium for VM

$ VBoxManage storagectl ubuntutest --name SATA --add SATA --controller IntelAhci # add a SATA controller to VM

$ VBoxManage storageattach ubuntutest --storagectl SATA --port 0 --device 0 --type hdd --medium /home/mibeyki/VirtualBox VMs/ubuntutest/ubuntutest.vd # add a SATA controller to VM

$ VBoxManage storagectl ubuntutest --name IDE --add ide # define storage for IDE ATA support CD-ROM

$ VBoxManage storageattach ubuntutest --storagectl IDE --port 0 --device 0 --type dvddrive --medium /home/mibeyki/Downloads/iso/Debian/ubuntu-18.04-server-amd64.iso # attach iso image

$ VBoxManage modifyvm ubuntutest --memory 1024 --vram 16 # define VM resources

$ VBoxManage modifyvm ubuntutest --nic1 bridged --bridgeadapter1 wlan0 --nic2 nat # assign VM two NICs, NAT and Bridged for NIC1 and NIC2 respectively
# Now we the VM is ready to be auto installed and launched
```

We can define alot of other VM settings, but the configuration requires time and is more complex and less flexible than other solutions that use declarative approach like yaml file. Basically we are replicating the steps to create a VM in GUI using a command line approach and then adding automation. Furthermore, for each node functionality we create a different configuration for the virtual machine, for example a virtual machine acting as firewall will have different configuration (for example have IP tables installed) than virtual machine acting as router. This is not the adopted solution in this work.

3.4.2 OpenStack Automation with Ansible

OpenStack is a free, open standard cloud computing platform. theoretically it have the same advantages and disadvantages of Vbox Manage, but management of VMs become easier using a VM orchestator like openStack, also there is possibility to automate VM deployment using Ansible for example. where Ansible uses a declarative language to describe a desired system configuration. Here are some Possible advantages and disadvantages:

+ automated deployment
+ persistent, once created no need to re-create it unless environment is deleted
+ each vm has its own OS so that allows for multiple OS type support
+ can support firewall types iptables- bpf_firewall-ipfirewall-openvswitch
+ there is a dashboard and GUI
+ less command line scripting if we use Ansible
+ more flexible as adding nodes and removing can be done using the playbook
+ using an orchestrator ensures that the desired state is achieved, and this is advantageous with respect to other solutions that do not gaurantee a correctly running environment
− environment creation is slow as alot of nodes need to be created and deployed each with its own os
− needs high resources compared to containers so not suitable for large number of nodes, if the virtual environment is produced on single machine
− even when environment is created re-launching environment for testing require time, as each node has its own OS to boot unlike containers where launch time can reach milli seconds.

Using the OpenStack-ansible documentation [15], a virtual machine connected to a network could be deployed. After deploying one VM and one network and attaching the VM to it using Ansible, It is easy to replicate the work in order to automate the deployment of a whole collection of virtual machines in order to construct the environment. This approach is not adopted, due to the limited resources for deploying 10+ VMs even with lightweight operating system, it is too much compared to what is really needed for representing nodes in a virtual network. Below is an example of launching one virtual machine using Ansible playbook, this could be replicated in order to build the whole environment. For each node

functionality the configurations of the virtual machine changes. When the whole yaml file is created, the virtual environment could easily be launched by executing the yaml file. The following is a basic example is just for running a virtual machine instance using declarative Ansible approach.

```
- name: Launch a compute instance
hosts: localhost
tasks:
- name: Launch a VM
os_server:
image: Ubuntu 20.04 LTS x64
name: myvm
key_name: mykeypair
availability_zone: nova
flavor: 22
state: present>
network: floatingIPv4
```

To maintain the desired state of the designated VM, the Ansible module os server executes commands on your local machine. Thus, the play book is applied on the localhost. The os server module makes sure the provided VM is present or not (depending on the state parameter to ensure correct enforcement of desired state). As stated in the playbook above, we can provide the specifics of the VM we wish to boot. There, we specified that the new virtual machine's name will be myvm, that it will boot from the Ubuntu 20.04 image in the nova availability zone with the keypair mykeypair, flavor cloudcompute.s, and an IP address assigned from the floatingIPv4 network, and that it will have these configurations. After this definition, we could execute the playbook and check dashboard to ensure the virtual machine is running. Replicating this declaration with different configurations (or even mount configurations to virtual machines) according to the network node functionality to be emulated, then the virtual network will be built.

Moreover, the best advantage of declaring virtual machine using playbook is the flexibility (where we can create virtual machine that emulate different network node functionalities) and robustness that adds a level of confidence that the virtual network environment will be up and running even if an error occur openstack will ensure its correct functioning.

3.4.3 Docker Container CLI

Docker is relatively much easier to create containers and deploy a virtual environment than using other CL approaches for creating VMs. Docker CLI allows to easily create a container and spawn it in less than a second with a specific image.

Docker offers high flexibility of what images to deploy, and the different operating system flavors. The most important advantage is that Docker containers are highly lightweight and require very minimal resources to operate, making them suitable for our case. Docker CLI is good for creating a container using the command line. It is possible to implement the virtual network with docker CLI using scripting. The following is a basic example of creating a docker container and a network, then attaching the docker to the network.

```
docker network create –driver=bridge –subnet=192.168.2.0/24 –
gateway=192.168.2.10 mynetwork
docker run –network=mynetwork –ip 192.168.2.2 -itd -v Config-
Files:/mnt –name=webclient alpine
```

The above commands first creates a network called mynetwork with a IP address range 192.168.2.0/24 and a docker gateway 192.168.2.10. Then launches a container called webclient with an alpine image, attaches the container to mynetwork assigned to it the IP address 192.168.2.2 and mounts a volume ConfigFiles to the container. The volume mounted is used to configure the containers when they start, for example the mounted volume could a configuration files for firewalls, so container executes them and it behaves like a firewall.

However, what if there is a need to build a new image that contains all the packages required to emulate a certain network node. In this case, Docker file becomes useful. Docker file is used in order to build images in a flexible way, which allows to design images according to the needs of the network nodes. For example the following Docker file builds an image that contains iptables that is the module used to emulate an iptables firewall.

```
FROM alpine:latest
# tcpdump is only for debugging purposes
RUN apk update
RUN apk add iptables sudo
RUN apk add tcpdump
RUN apk add bash
RUN apk add –no-cache quagga
&& touch /etc/quagga/zebra.conf
&& touch /etc/quagga/vtysh.conf
&& touch /etc/quagga/ripd.conf
```

Starting from alpine base image, the docker files contains sequence of commands to build a new image that represent the image of an iptable firewall, since alpine base image doesn't contain native iptables module, it must be installed in order to use it, the other commands are useful in the image for different purposes like debugging or using dynamic routing ...

3.4.4 Docker Compose

Docker-compose is a container orchestration technology intended to run several containers on a single host machine. Docker-compose is excellent for development environments. But the problem that arise from using Docker compose to virtualize an environment is docker-compose is not a good fit for that purpose, in fact it is not a technology that is tailored to create virtual networks. Docker compose is used to create applications not virtual networks, this will present challenges that need to be managed. Below is the pros and cons of Docker compose.

+ automated deployment
+ flexible, environment can be deployed and removed in seconds
+ can support firewall types iptables- bpf_firewall-openvswitch
+ declarative approach for creating network topology
+ very light weight so it is possible to extend created network to a more ramified network easily

• each time testing must be done the environment must be recreated from scratch (but it takes seconds)

– FreeBSD OS not supported so ip firewall type can not be tested
– No GUI
– Not built-specifically for purpose of testing network topologies

The following example shows a container acting as firewall defined in a declarative manner using docker compose yaml file.

```
firewall1: # iptables firewall
container_name: firewall1
hostname: firewall1
image: router_firewall_rip
cap_add:
- NET_ADMIN
volumes:
- ./RouterFirewallConfig:/mnt:ro
command: sh -c "mnt/staticroutes/fw1routes && tail -F anything "
networks:
lb-fw1:
ipv4_address: 20.0.0.2
fw1-cache:
ipv4_address: 20.0.1.1
```

In the above declaration, a container called firewall1 is declared with image router_firewall_rip that is specific image built using docker file based on alpine flavor. The container will mound at start time the RouterFirewallConfig files and executes the required scripts at startup. Finally the container is attached to two predefined network called lb-fw1 and fw1-cache.

Using this approach it is relatively easy to declare a couple of containers in a simple manner with less errors and more flexibility. Docker compose is usually used for development purposes and not for production environments, in the latter case kuberneutes (which is an orchestration tool that manages containers and applications in a more flexible and robust manner) is usually used.

3.4.5 Kuberneutes and Kompose

Kubernetes runs containers in the data center over a cluster of servers. But is specified here to open the possibility for future migration from docker-compose to Kubernetes. In case our reference topology was to expand to hundreds of nodes there is then a need to deploy the virtual environment into a cluster of servers using Kubernetes. For a that we keep migration from docker-compose to Kubernetes as a possibility, and fortunately, this is another advantage of using docker-compose, If in the future the topology developed using docker-compose became huge and required to be deployed over a set of servers. Using Kompose we can easily convert docker-compose files into other container orchestrators such as Kubernetes or openshift. Although again kuberneutes is not built for the purpose of network topology virtualization so limitations exist.

3.5 Firewall Testing Tools

Since the focus will be on testing firewalls configuration. In this section, several open-source security tools and libraries are introduced that contribute to firewall testing. Software packages that are suitable to form the basis of our testing tool are defined. The programs of interest can be divided into either packet generation and injection tools or sniffing and logging tools for debugging purposes.

3.5.1 Injection Tools

There are a lot of packet generation and injection tools that are open sources and free.

1. Nemesis: Nemesis is a command line-based. It provides a possibility to craft packets type according to our needs and sends them to a target destination. It can craft packet types like ARP, Ethernet, ICMP, IP, TCP, and UDP. In

addition to that, the packet parameters are adaptable, and user can change them as he likes. Unfortunately, nemesis has no sniffing functionality, so we will choose another tool to handle capturing packets.

2. Hping: Hping is a command-line tool. The interface is inspired by the ping command in Unix, but hping is not only able to send ICMP echo requests. It supports TCP, UDP, ICMP, and raw IP protocols, has a traceroute mode, and many other features. Hping can be used for port scanning, network testing, remote OS fingerprinting, and firewall testing. As hping provides the functionality we are looking for, it is a notable tool that can be converted into a more specific firewall testing tool.

3. Nmap: Is a free, open-source tool for network scanning. It is mainly used to scan large networks. Nmap uses IP packets to determine what hosts are available on the network, what services those hosts are offering, what operating systems they are running, what type of packet filters or firewalls are in use, and many other characteristics. The difference between Nmap and hping is that hping is an all-round tool sending user-defined packets to user-specified hosts. In contrast, Nmap provides a list of scanning techniques (TCP connect(), TCP SYN (half-open)), and advanced features (such as remote OS-detection, stealth scanning or TCP/IP fingerprinting) that allow the user to run sophisticated attacks against a specified network or host. In other words, Nmap provides a complete and handsome list of scanning techniques, but the user loses the facility to craft and sends self-made packets.

3.5.2 Sniffing Tools

1. Libpcap: The packet capture library provides a high-level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this interface. Libpcap is a de-facto standard in packet capture programming. We leave the possibility to use libcap in our situation, but it is unnecessary. A simple solution like tcpdump is enough.

2. Tcpdump: dumps the traffic on a network. It is related to libpcap in that they are maintained by the same group, and tcpdump heavily relies on libpcap. Tcpdump is the most common way to visualize the packets libpcap captures by printing them directly to the console or logging them in a file. When tcpdump finishes packet capture, it reports how many packets have been received, dropped, and processed. Tcpdump is appropriate for packet capture.

After observing the different tools out there, for example, hping3 could be used for injection since it is more effective for firewall testing. For sniffing, tcpdump

could be used on routers and firewalls to check the type of packets arriving at each node. Fortunately, there was the possibility of a container that checks all the traffic passing in the virtual environment.

Chapter 4

Thesis Objective

In view of motivations to verify and validate the low level translation of VEREFOO framework, in this thesis we propose testing based on virtualized networks to provide validation to the complete translation from medium-level language to low-level in the VEREFOO framework. The objective of this thesis is decomposed into three different aspects explained in the following sections.

4.1 Exhaustive Testing

First part of this thesis aims at the exhaustive testing of the low level translation in the VEREFOO framework. As mentioned before, this part of VEREFOO is not fully validated and verified. Chapter 5 will discuss the best methodology to test this part of the framework using the well-known white box approach. Furthermore, after exhaustively testing this part, the result firewall configuration files will be deployed in a created virtual network to additionally verify the correctness of such configurations in a virtual environment. This chapter will go through the details of developing a virtual network environment that faithfully replicates the actual physical network environment, this effort requires high level of automation and assurance that the environment does not contain any errors. The testing focuses on verifying the correctness of the VEREFOO module capable of performing a multi-language translation among several packet filters that are available in the market.

Testing will include two phases. Phase one is to ensure the correctness of the translation code; phase two is testing in a pre-produced virtual environment which will be produced to verify not only the correctness of the translation code but the satisfaction of the Network Security Requirements defined at the input of the VEREFOO framework. The testing in a virtual environment allows to detect issues that may be hidden and solve them.

4.2 Demo Presentation And Translator Algorithm

Another goal of this thesis is to demonstrate the complete functioning of VEREFOO, combining the different stand-alone parts of the framework and presenting them as a single process to the user.

Then finally, to achieve maximum automation, a demonstration will be done of a simple algorithm that does the translation of the output Firewall Allocation Schema of the VEREFOO framework to configuration files to start the virtual environment corresponding to the FAS. This algorithm could be used in the future to facilitate the repetitive process of creating virtual networks for testing. Each time a new Firewall Allocation Schema is produced rather than building the network environment from scratch, it is possible to input the FAS XML file in to the translator and immediately the algorithm creates the required configuration files to start the environment for testing.

Chapter 5

Verification and Validation of low level configuration output

5.1 Testing Purpose

The focus is to test the translation of medium-level firewall rules to low-level configurations of the already developed portion of VEREFOO [1]. So it is expected that the MLPs (Medium Level Policies) generated by the VEREFOO framework are correct. There is a need to achieve certain confidence about the code correctness that translates from the medium-level policy in XML format to low-level configuration produced as bash scripts ready to be deployed on firewalls. The optimal set of test cases that covers possible scenarios of a firewall deployment are created to ensure the correctness of translation.

The generation of the test cases should not be random but rather more specific and extensive to find possible issues in the translation code. The more accurate the test cases are in finding errors, the more robust the framework will be. And be able to produce an error-free low-level translation. Next is explained the methodology followed to create test cases.

5.2 Testing Approach

To start generating test cases, the right testing approach must be found. In particular, according to [16] , there is 3 general approaches for firewall testing,*penetration testing*, *testing of firewall implementation*, and *testing of firewall rules*. In summary, *Penetration testing* is performed to check the firewall for potential breaches of

security that can be exploited using various tools, which is not of interest here. *Testing of the firewall implementation* focuses on the firewall Configuration details. Different firewall types have different configuration languages. The firewall implementation testing approach evaluates if the firewall rules correspond to the action the firewall performs (for example, a firewall policy blocks a packet but the firewall allows it, or the configuration file of a specific firewall produces a rule that is different from the policy expressed in the MLPs in our case). *Testing of the firewall rules* checks if the security policy is correctly enforced by the firewall rules or not. Each approach corresponds to a different methodology for generating the test cases. For example, generating test cases and tools used for penetration testing differs from test cases and tools used to test firewall implementation.

The focus here is testing the correctness of the translation of a security policy (presented in XML format) into firewall rules written in a specific vendor language. The output configuration must be compared to the input security policies to determine if they match. Otherwise, an error is obtained in the translation process. This means *Testing the firewall implementation* approach must be performed, and generate the test bed accordingly. As shown in section 5.3 .

The firewall implementation testing is achieved through defining test cases and translating test cases to MLP using XML. Then the resulting XML is used as input to the translation code. Furthermore, the output configuration files are checked for correctness through static verification, and later testing is done in a virtual network environment.

5.3 Finding Test Cases Abstractions

It is easy to lose focus on the objective of testing. The testing should primarily verify translation to low-level configurations. Therefore, the testing is Not directed toward finding policies that are dis-joint, good, or verified... this is done at the medium level policy generation that VEREFOO generates. The MLPs should be correct, verified, and follow the guidelines of building policies based on a particular formal model (ex: FIREWALL POLICY MODELLING like in [17] where Ehab S. Al-Shaer and Hazem H. Hamed define a specific model to detect anomalies in firewall policies). Test cases generated have policies that may not be disjoint nor perfect since they are directed toward the translation part of VEREFOO. Test cases focus more on the syntax than the logical part of firewall policies. Then subsequently, the proper enforcement of policies into firewalls in the virtual environment is tested.

5.3.1 Finding firewall Parameters

For generating test cases, an academic approach is followed. As in the literature, there has been no alternative method developed yet. First, an abstraction of the

TCs will be created, to be populated later by the correct values for testing the output of the TCs on a virtual environment. According to [18], in his software engineering book, Pressman states that test cases must guarantee that all independent paths within a module are executed at least once, execute all possible logical decisions, execute all loops at their boundaries, all of that can be defined using white box testing methods (in context of software engineering). The same principles can be applied in our case to ensure all possible cases are covered. So white-box testing approach is used to test all possible cases, which in our case is feasible due to the limited number of parameters to be changed.

Using this academic approach, start from the possible parameters that the medium-level policy can have, as shown in Figure 5.1 is an XML element schema representing a policy to be enforced with all its possible parameters that can be used as input to the translation code.

```
<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>
```

Figure 5.1: XML element schema

So the parameters with there possible values in parenthesis are:

1. Action (deny – allow)
2. Source IP (specific IP – range IP)
3. Destination IP (specific IP – range IP)
4. Protocol (TCP – UDP – ANY)
5. Source PORT (specific port – range port - *)
6. Destination PORT (specific port – range port - *)

7. Priority (number - * - not present)

8. Directional (true – false – not present)

In addition to the default action parameter of the firewall (default deny or default allow), the total is nine parameters. Then the test cases are created by manipulating the parameters to cover all possible combinations. It is worth noting that here the testing is directed to the function that creates the firewall rule implementation at the software code level.

This type of testing is possible since the number of parameters of a policy firewall is limited (8 parameters) in the NFV element so that it can be extensively tested, and test cases can be generated manually. Nevertheless, in the future, if the number of parameters to generate test cases increases, maybe this is not the best way to go, but it is a base for an automated test generation. (give input, and perhaps in an automated way, test cases are generated by shuffling all parameters). Related work has already developed [16] a firewall tool to automate firewall implementation testing by packet injection.

Test Cases will cover the correct translation of specific elements in a policy, each test case target a particular set of values taken by a parameter to check its correct translation. So generally there are:

- Default action
- Policy action
- Specific source IP to specific destination IP
- Specific source IP to range destination IP (and vice versa)
- Range source IP to range destination IP
- Protocols TCP, UDP, ANY.
- Specific source PORT to specific destination PORT
- Specific source PORT to range destination PORT
- Range source PORT to range destination PORT
- Directional
- Priority

5.3.2 Test Case Abstractions

We could generate a TC for each of the previous values. However, it will be repetitive and time-consuming, so it is better if each TC targets a couple of values of a particular parameter which would be a more reasonable number of TCs. for example, TC1 could cover testing of the IP parameter with all of its possible values (Specific src IP to specific dest IP-Specific src IP to range dest IP (and vice versa)- Range src IP to range dest IP), and same for the rest of TCs. This is possible since the translation of each parameter at the code level is done independently from other parameters. For example, the values of source IP and destination IP do not affect the translation of port numbers. To organize TCs in order to effectively test and target the correct translation of each parameter alone, the generation of test cases must be done in with each parameter targeted alone, following this approach:

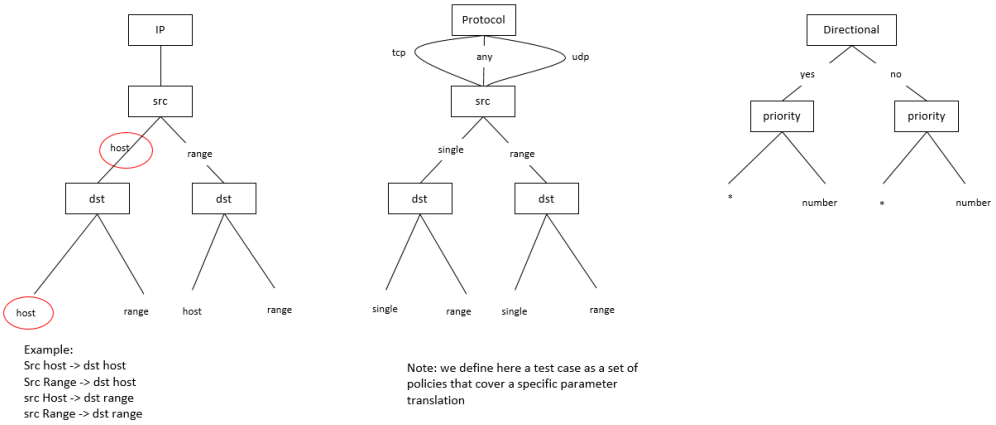

Figure 5.2: Policy Trees

In Figure 5.2 we can target each parameter alone to be used to generate the test cases. For example, first focus on IP alone and ignore the rest of the parameters (fixing them with general values). then change the IP parameter alone to check the correct translation of each possible policy. It is worth noting that there is no need to do input validation since we assume that the firewall medium-level rules are semantically correct and verified at a higher abstraction of VEREFOO (source IP is always expected to be in format xx.xx.xx.xx/xx, it is not possible to have an input like 1000.2322.112.2 from medium level rules as the validation is done at a higher level). Finally, we end up with these seven TCs:

1. **Test Case 1** in black list mode (default = allow): TC1 is the simplest one, as it targets only the IP parameter. This TC will focus on changing the IP source and destination to cover all possible values format. The rest of the

parameters are omitted (priority and directional) or put to ANY (protocol type and ports). TC1 covers:

- specific (host) IP to specific IP

- specific IP to range IP

- range IP to range IP

Example:

Action	Src IP	Dst IP
deny	10.0.1.1	30.0.5.1
deny	10.0.0.1	10.0.2.0/24
deny	10.0.2.0/24	10.0.0.1
deny	10.0.1.0/24	10.0.2.0/24

2. **Test Case 2** in black list mode (default = allow): TC2 extends TC1 and cover the the PORT/protocol parameter, it focuses on manipulating values of protocols with the source ports and destination ports. The priority and directional parameters are omitted here. TC2 covers: - TCP, UDP, ANY protocols - specific PORT to specific PORT - specific PORT to range PORT - range PORT to range PORT

   Example: Src IP and Dst IP can be of any values

Action	Src IP	Dst IP	Protocol	Src Port	Dst Port
deny	*	*	TCP	2000	80
deny	*	*	TCP	2000	500-600
deny	*	*	UDP	400-500	500-600
deny	*	*	ANY	*	*

3. **Test Case 3** in black list mode (default = allow): Finally TC3 extends TC2 to cover the Directional and Priority parameters. TC3 covers:

   - Directional

   - Priority

   Example: IP and PORTs can be of any value

Action	Directional	Priority
deny	yes	5
deny	yes	10
deny	no	*
allow	yes	*

Note: The same work flow is followed in case of black list mode where the default action is deny and policy actions are mostly allow. This leave results with the following:

4. **Test Case 4** in white list mode (default = deny) covers:

   - specific IP to specific IP

   - specific IP to range IP

   - range IP to range IP

5. **Test Case 5** in white list mode (default = deny) covers:

   - TCP,UDP,ANY protocols

   - specific PORT to specific PORT

   - specific PORT to range PORT

   - range PORT to range PORT

6. **Test Case 6** in white list mode (default = deny) covers:

   - Directional

   - Priority

7. **Test Case 7**:

   Is just an additional test case used to simulate real-world policies for checking the correct translation of all of the parameters together.

Now that this abstraction is created for the Test Cases, next populate these TCs with values to input them into the framework and check the resulted output. To do that, there is two options. First, populate TCs with random but correct values and translate TCs to an NFV element as an input to the Firewall Serializer class. Then check statically by scanning the output configuration files and comparing the output of the files with the corresponding input policies. The translation is correct if the functionality of the commands in the configuration file corresponds to that of the input policy. The second option, create TCs with their values based on a reference network topology, which will then be deployed in a testing environment to run live tests on the firewalls configured by the produced output. In our case, we will build a virtual testing environment, so the second option will be adopted. The firewall policies will be generated based on a reference network topology shown in the next section.

5.4 Test Case Generation based on Reference Topology

This section will show the choice of the right topology to be taken as reference. Then thanks to the test case abstractions found before, we can easily generate the test cases to be used as input to the framework low-level translator.

5.4.1 Finding the Right Topology

In order to find the reference topology, start searching for the optimal one. Due to a lack of physical machines and resources, the virtual environment will be reproduced and deployed locally. The optimum Network topology must:

1. Covers all possible test cases
2. be lightweight, so it can be reproduced locally in terms of resources, So we need to have a sufficient and limited number of nodes in the topology.
3. simulates a real-world network topology pattern

So this limits how many test cases are produced and their coverage. The problem here is to find a network topology that is ramified enough to cover all test cases yet small enough to consume as few resources as possible. The best way to continue is to start from the most common pattern of a firewall network topology. According to NIST [12], "Organizations should use firewalls wherever their internal networks and systems interface with external networks and systems, and where security requirements vary among their internal network" adding "firewalls should be placed at the edge of logical network boundaries", then explained that the most common way to protect a network of a business offering services to the internet is using a DMZ concept, where there is an external network, internal network and a DMZ where servers should be positioned (as in Figure 5.3).

A demilitarized zone network is a perimeter network that protects and adds an extra layer of security to an organization's internal local-area network from external, not trusted traffic. A common DMZ is a network that sits between the external and internal networks in a Company. The DMZ is usually used for servers (like a Web server and a Mail server) that should be accessible both from the outside (Internet) and from the internal network (intranet) and thus are governed by a more relaxed policy than the internal network that should be protected from external threats.

What is interesting in Figure 5.3 is the three different areas in the topology, the internal, external, and the DMZ areas. Build a topology similar to Figure 5.3 small-business topology, where we add a couple of endpoints inside the internal network,

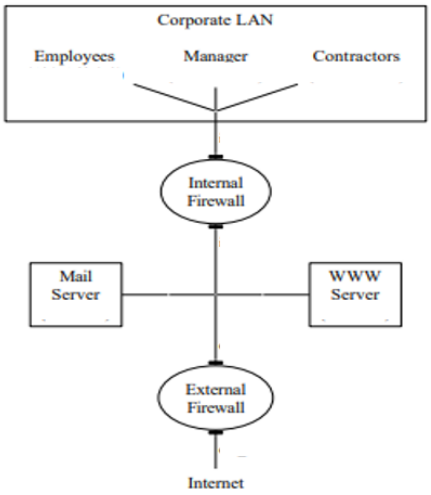

Figure 5.3: small-business network-topology figure depicted from

two firewalls also acting as routers, and a couple of servers in the DMZ network. Also, for the external network, we add two endpoints, each with a different network, to have various external clients coming from different IP subnets. Therefore we could set up our network topology to look like Figure 5.4.

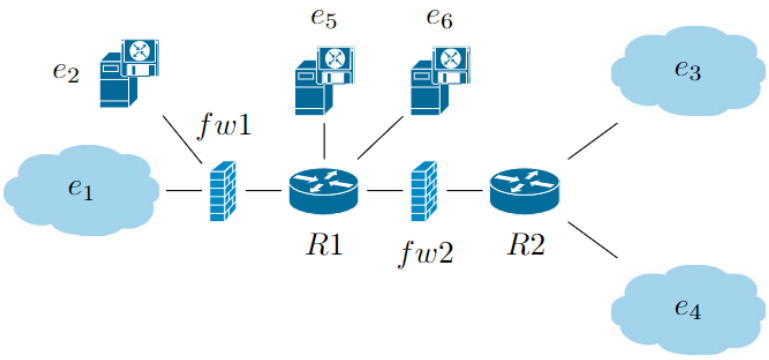

Figure 5.4: Reference Network Topology

For now, layer 2 connectivity is not of interest. Lets focus on network assignment and position of firewalls/routers. first start by the Assigning to the topology the following networks:

- External Network:

1. Network 10.0.0.0/24 (e3): represents external web clients that need access to web server or email server
2. Network 10.0.1.0/24 (e4): another external network that require access to servers

- DMZ 30.0.5.0/24 (e5 and e6): network where all servers that require public access are located.
- Network 10.0.2.0/24 (e1 and e2): internal Network for employees connected to the internal network of the company.

Both external networks are connected to Router 2, which is the entry point to the company network protected by a first firewall (External Firewall 1). Router 1 is considered the point of routing external and internal traffic. It is connected to Firewall 1 and Firewall 2 in addition to the DMZ 30.0.5.0/24, as shown in Figure 5.4. We can also observe the positions of the firewalls where enforcement of policies takes place. To simplify the example, we considered that external web clients have private IPs (10.0.0.0/24 or 10.0.1.0/24), and the point-to-point links between routers and firewalls are of sub-net /24. of course, in a real scenario they are of /30 sub-net that is assigned to point to point (p2p) links. In any case, this could be changed for other types of testing.

Now that we have found our simple network topology, it will be used as a reference for all the work that follows, from generating test cases to building the virtual environment for testing. Using the test cases abstraction done in section 5.3, we can easily populate each test case from TC1 to TC7 with policies created using the reference topology respecting each test case rule defined earlier. The policies to be generated will target firewall 1 (external firewall) and firewall 2 (internal firewall), So each TC is expected to have policies related to firewalls 1 and 2. In other words, we will obtain in each test case; TC1-Ex (test case 1 policies for external firewall), and TC1-In (test case policies for internal firewall) similarly to the rest of the test cases.

5.4.2 Test Case Generation

Now that we have the reference topology lets Start by creating the first TC1, note that its creation will be done to correspond with the work flow defined in subsection 5.3.2.

■ **TC1**

The first test case covers the parameter IP in white list mode with its different values (IP - IP , IP - range , range - range). The tables represent the firewall policies of external and internal firewall according to our reference topology.

Ex = External firewall policy; In = Internal Firewall policy. Default Action = Allow

Firewall	Action	Src IP	Dst IP	Protocol	Src PORT	Dst PORT	Priority	Directional
1 External	deny	10.0.1.1	30.0.5.1	ANY	*	*	*	no
2 External	deny	30.0.5.1	10.0.1.1	ANY	*	*	*	no
3 External	deny	10.0.1.1	30.0.5.2	ANY	*	*	*	no
4 External	deny	30.0.5.2	10.0.1.1	ANY	*	*	*	no
5 External	deny	10.0.0.1	10.0.2.0/24	ANY	*	*	*	no
6 External	deny	10.0.2.0/24	10.0.0.1	ANY	*	*	*	no
7 External	deny	10.0.1.0/24	10.0.2.0/24	ANY	*	*	*	no
8 External	deny	10.0.2.0/24	10.0.1.0/24	ANY	*	*	*	no
9 Internal	deny	10.0.2.1	30.0.5.1	ANY	*	*	*	no
10 Internal	deny	30.0.5.1	10.0.2.1	ANY	*	*	*	no
11 Internal	deny	10.0.0.1	10.0.2.0/24	ANY	*	*	*	no
12 Internal	deny	10.0.2.0/24	10.0.0.1	ANY	*	*	*	no
13 Internal	deny	10.0.2.0/24	10.0.1.0/24	ANY	*	*	*	no

Table 5.1: TC1 policies

- 1-2-3-4-9-10 tests IP to IP translation
- 5-6-11-12 tests IP to range translation
- 7-8-13 tests range to range translation

■ **TC2**

The Second test case covers the protocol and PORT parameters in white list mode with its different values. The tables represent the firewall policies of external and internal firewall according to our reference topology. Ex = External firewall policy; In = Internal Firewall policy. Default Action = Allow

Firewall	Action	Src IP	Dst IP	Protocol	Src PORT	Dst PORT	Priority	Directional
1 External	deny	10.0.0.1	30.0.5.1	TCP	2000	80	*	no
2 External	deny	30.0.5.1	10.0.0.1	TCP	80	2000	*	no
3 External	deny	10.0.0.1	30.0.5.2	TCP	2000	500-600	*	no
4 External	deny	30.0.5.2	10.0.0.1	TCP	500-600	2000	*	no
5 External	deny	10.0.1.0/24	30.0.5.1	TCP	*	80	*	no
6 External	deny	30.0.5.1	10.0.1.0/24	TCP	80	*	*	no
7 External	deny	10.0.1.0/24	30.0.5.1	TCP	*	400-500	*	no
8 External	deny	30.0.5.1	10.0.1.0/24	TCP	400-500	*	*	no
9 Internal	deny	10.0.2.1	30.0.5.2	TCP	2000-3000	0-1000	*	no
10 Internal	deny	30.0.5.2	10.0.2.1	TCP	0-1000	2000-3000	*	no
11 Internal	deny	10.0.2.2	30.0.5.2	UDP	*	*	*	no
12 Internal	deny	30.0.5.2	10.0.2.2	UDP	*	*	*	no
13 Internal	deny	10.0.2.3	30.0.5.2	ANY	*	*	*	no
14 Internal	deny	30.0.5.2	10.0.2.3	ANY	*	*	*	no

Table 5.2: TC2 policies

As we can see here the main focus was testing the PORT/protocol parameter. The policies number:

- 1-2 tests PORT to PORT translation
- 3-4-5-6 tests PORT to range translation
- 7-8-9-10 tests range to range translation
- all tests verify the translation of TCP/UDP/ANY protocols

■ **TC3**

The Third test case covers the directional and priority parameters in white list mode with its different values. The tables represent the firewall policies of external and internal firewall according to our reference topology. Ex = External firewall policy; In = Internal Firewall policy. Default Action = Allow

Firewall	Action	Src IP	Dst IP	Protocol	Src PORT	Dst PORT	Priority	Directional
1 External	deny	10.0.0.1	30.0.5.1	TCP	2000	80	*	yes
2 External	deny	10.0.0.1	30.0.5.2	TCP	2000	500-600	*	yes
3 External	allow	10.0.1.0/24	30.0.5.1	TCP	*	443	5	yes
4 External	deny	10.0.1.0/24	30.0.5.1	TCP	*	400-500	10	yes
5 Internal	deny	10.0.2.1	30.0.5.2	TCP	2000-3000	0-1000	*	yes
6 Internal	deny	10.0.2.2	30.0.5.2	UDP	*	*	*	yes
7 Internal	deny	10.0.2.3	30.0.5.2	ANY	*	*	5	yes
8 Internal	allow	10.0.2.3	30.0.5.2	ANY	1000	400	1	yes

Table 5.3: TC3 policies

As we can see here the main focus was testing the Directional/Priorirty parameters with different IP and PORT numbers. The policies number:

- all policies cover directional parameter
- 3-4-7-8 covers priority with different PORT numbers

■ **TC4**

The fourth test case covers the parameter IP in black list mode with its different values (IP - IP , IP - range , range - range). The tables represent the firewall policies of external and internal firewall according to our reference topology. Ex = External firewall policy; In = Internal Firewall policy. Default Action = deny

Firewall	Action	Src IP	Dst IP	Protocol	Src PORT	Dst PORT	Priority	Directional
1 External	allow	10.0.1.1	30.0.5.1	ANY	*	*	*	no
2 External	allow	30.0.5.1	10.0.1.1	ANY	*	*	*	no
3 External	allow	10.0.1.1	30.0.5.2	ANY	*	*	*	no
4 External	allow	30.0.5.2	10.0.1.1	ANY	*	*	*	no
5 External	allow	10.0.0.1	10.0.2.0/24	ANY	*	*	*	no
6 External	allow	10.0.2.0/24	10.0.0.0/24	ANY	*	*	*	no
8 Internal	allow	10.0.2.0/24	10.0.0.1	ANY	*	*	*	no
9 Internal	allow	10.0.0/24	10.0.2.0/24	ANY	*	*	*	no
10 Internal	allow	10.0.0.2.0/24	30.0.5.0/24	ANY	*	*	*	no
11 Internal	allow	30.0.5.0/24	10.0.2.0/24	ANY	*	*	*	no

Table 5.4: TC4 policies

As we can see here the main focus was testing the Src/Dst IP parameter. The policies number:

- 1-2-3-4 tests IP to IP translation
- 5-7 tests IP to range translation
- 6-9-10-11 tests range to range translation

■ **TC5**

The fifth test case covers the protocol and PORT parameters in black list mode with its different values. The tables represent the firewall policies of external and internal firewall according to our reference topology. Ex = External firewall policy; In = Internal Firewall policy. Default Action = deny

Firewall	Action	Src IP	Dst IP	Protocol	Src PORT	Dst PORT	Priority	Directional
1 External	allow	10.0.0.1	30.0.5.1	TCP	*	80	*	no
2 External	allow	30.0.5.1	10.0.0.1	TCP	80	*	*	no
3 External	allow	10.0.0.1	30.0.5.2	TCP	*	500-600	*	no
4 External	allow	30.0.5.2	10.0.0.1	TCP	500-600	*	*	no
5 External	allow	10.0.1.0/24	30.0.5.1	TCP	*	80	*	no
6 External	allow	30.0.5.1	10.0.1.0/24	TCP	80	*	*	no
7 External	allow	10.0.1.0/24	30.0.5.1	TCP	*	400-500	*	no
8 External	allow	30.0.5.1	10.0.1.0/24	TCP	400-500	*	*	no
9 Internal	allow	10.0.2.1	30.0.5.2	TCP	2000-3000	0-1000	*	no
10 Internal	allow	30.0.5.2	10.0.2.1	TCP	0-1000	2000-3000	*	no
11 Internal	allow	10.0.2.2	30.0.5.2	UDP	*	*	*	no
12 Internal	allow	30.0.5.2	10.0.2.2	UDP	*	*	*	no
13 Internal	allow	10.0.2.3	30.0.5.2	ANY	*	*	*	no
14 Internal	allow	30.0.5.2	10.0.2.3	ANY	*	*	*	no

Table 5.5: TC5 policies

As we can see here the main focus was testing the PORT/protocol parameter. The policies number:

- 1-2-5-6 tests PORT to PORT translation
- 3-4-7-8 tests PORT to range translation
- 9-10 tests range to range translation
- all tests verify the translation of TCP/UDP/ANY protocols

■ TC6

The sixth test case covers the directional and priority parameters in black list mode with its different values. The tables represent the firewall policies of external and internal firewall according to our reference topology. Ex = External firewall policy; In = Internal Firewall policy. Default Action = deny

Firewall	Action	Src IP	Dst IP	Protocol	Src PORT	Dst PORT	Priority	Directional
1 External	allow	10.0.0.1	30.0.5.1	TCP	2000	80	*	yes
2 External	allow	10.0.0.1	30.0.5.2	TCP	2000	500-600	*	yes
3 External	deny	10.0.1.0/24	30.0.5.1	TCP	*	443	5	yes
4 External	allow	10.0.1.0/24	30.0.5.1	TCP	*	400-500	10	yes
5 Internal	allow	10.0.2.1	30.0.5.2	TCP	2000-3000	0-1000	*	yes
6 Internal	allow	10.0.2.2	30.0.5.2	UDP	*	*	*	yes
7 Internal	allow	10.0.2.3	30.0.5.2	ANY	*	*	5	yes
8 Internal	deny	10.0.2.3	30.0.5.2	ANY	1000	400	1	yes

Table 5.6: TC6 policies

As we can see here the main focus was testing the Directional/Priority parameters with different IP and PORT numbers. The policies number:

- all policies cover directional parameter
- 3-4-7-8 covers priority with different PORT numbers

■ TC7

The seventh test case represents a possible real-world configuration of firewalls with certain common practices performed commonly, like blocking a specific IP from external malicious client to access we b server, or allowing a specific port from external to reach internal client, this TC is here to ensure the correct translation of everything. The tables represent the firewall policies of external and internal firewall according to our reference topology. Ex = External firewall policy; In = Internal Firewall policy. Default Action = deny

Firewall	Action	Src IP	Dst IP	Protocol	Src PORT	Dst PORT	Priority	Directional
1 External	allow	*	30.0.5.1	TCP	*	80	10	yes
2 External	allow	*	30.0.5.1	TCP	*	443	10	yes
3 External	allow	10.0.0.0/24	30.0.5.2	TCP	*	587	*	yes
4 External	deny	10.0.1.1	30.0.5.1	TCP	*	80	5	yes
5 External	deny	10.0.1.1	30.0.5.1	TCP	*	443	5	yes
6 External	allow	10.0.0.1	10.0.2.4	TCP	50000	53	*	yes
7 External	allow	10.0.0.1	10.0.2.4	UDP	50001	53	*	yes
8 External	allow	10.0.0.1	10.0.2.1	ANY	*	*	*	yes
9 Internal	allow	10.0.0.1	10.0.2.4	TCP	50000	53	*	yes
10 Internal	allow	10.0.0.1	10.0.2.4	UDP	50001	53	*	yes
11 Internal	allow	10.0.2.1	10.0.0.1	ANY	*	*	*	yes
12 Internal	allow	10.0.2.1	30.0.5.0/24	ANY	*	*	*	yes
13 Internal	allow	10.0.2.2	30.0.5.0/24	ANY	*	*	10	yes
14 Internal	deny	10.0.2.2	30.0.5.1	ANY	*	*	5	yes

Table 5.7: TC7 policies

Finally this TC provides an additional assurance that the code will produce an error-free translation. TC7 validates all previous TCs.

5.5 Correction of translation Code

To test the code, first translate the policies in TCs produced into an XML format that the code accepts as an input. So the file TestCases.xml is generated, written in it all the TCs. It is located in /verefoo/testfile/Others. Also, another file called TestCases-bpf is created which include test cases specific to bpf firewall. It is basically the same as TestCases.xml but with reduced PORT ranges. For example, a TC with a policy having source PORT 2000-3000 will be converted to a policy with source PORT range 2-3 omitting all the zeros, so that the PORT range becomes small, and so on for the rest (400-600 becomes 4-6). This is done just in case of bpf firewall since according to their documentation [19], although it is an iptables-like syntax, they didn't specify a PORT range command that allows covering a whole range of PORTs [1]. and in VEREFOO code the translation is done according to an algorithm that takes the PORT range and translates each PORT in the range into a single command. So a PORT range 2000-3000 would produce 1000 lines of commands, each line configures a single PORT, which will be too much in our case, so just for the case of bpf firewall, the PORT range is reduced as mentioned earlier.

Starting from the TestCases.xml file, we input the file to the translation code and check the results. having 7 test cases, each TC has an external and internal firewall configuration so for each firewall type we should expect an output of 14

configuration files. For example, Iptables translation code produced 14 configuration files, named as follows:

- "Firewall Type"_"TC Number"-"Ex or In"_"File Number" where Ex stands for External firewall configuration and In stand for Internal firewall configuration firewall.

The generated files for each firewall type were checked statically for errors, and the code was fixed accordingly. At this point the correction was done by comparing input policies with produced commands to check syntax correctness. but still the code isn't fully correct until it is tested practically in virtual environment to check additional possible problems. Now that the code is corrected statically, there is a little bit of confidence that the framework will produce error free translation. There is still a need to test in a virtual environment in order to be sure of the frameworks output.

5.6 Building of Test Environment

Now that the test cases are generated, it is time to create a testing environment. Due to the lack of physical machines to perform the testing. We will build our virtual network environment exploiting automation as much as possible so that the environment can be reproduced easily each time the testing needs to be done without needing to rebuild it again. Many factors contribute to the choice of the environment and the ability of our firewall types to be reproduced in such an environment. Building the virtual environment begins with defining the tools that will be required. To achieve that, the possibility of virtualizing network typologies is explored using several methods. As there is no standard way to virtualize a testing environment for networks, many possibilities exist, from using VMs to containers or software emulators. Different frameworks were developed to facilitate the deployment of a testing environment, but until now, there is no standard way or general approach to building a virtual environment. As specified in the background chapters, the best way to go when using virtualization is docker containers, specifically when there is a lack of resources (in case there was more resources working with virtual machines may be better). Again the topology is shown in Figure 5.5.

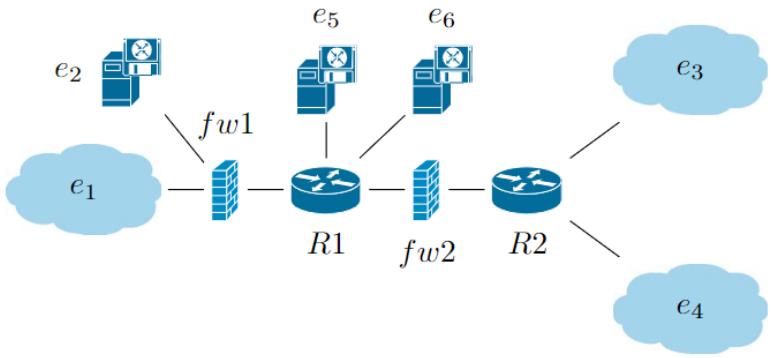

Figure 5.5: Reference Network Topology

As seen in Figure 5.5, It is necessary to count how many nodes our virtual environment will have. As the primary goal is firewall testing, we will assume that endpoints and servers have the same configuration since there is no special configuration needed for servers. The outcome of our environment is simply a couple of endpoints pinging each other or pinging servers. So the complexity occurs at the firewall/router level, specifically at the firewall level where different firewall types should be deployed, so we have to manage the compatibility of the firewall type that are being deploying with the node that represents the firewall (for example, ipfirewall needs a FREEBSD OS that Can not run inside containers). In the topology, We distinguish three kinds of nodes:

1. endpoint and server nodes: need to perform simple tasks, just as pinging using UDP and TCP flags. so their functionality is quite simple and doesn't need a complex node setup.

2. routers are expected to be a little more complex than endpoints as they need to run an additional routing functionality and must be configured accordingly to router packets correctly inside the network. Also, here we discard the router's complex functionality like NAT or having access lists or other complex networking functionalities. we just consider the routing part, as the others are not important right now. (maybe these functionalities could be extended in the future, for example dynamic routing to facilitate the routing configuration inside the virtual network environment)

3. firewalls act as routers with firewall rules. They filter incoming traffic before routing them to the next hop. So they should implement the functionality of routing and filtering. The complexity of deploying firewalls in our environment depends on the type of firewall used.

Layer two connectivity is ignored as they are not of interest right now (we assume that they are up and working ex: bridge mechanism), but for the openVswitch firewall, there is a need to dive deeper into L2 connectivity to apply such filtering at L2 level for that the reference topology may have minimal changes according to the used firewall type. In terms of the number of nodes, there is 12 nodes to be deployed (consider in each endpoint sub network there is two endpoints that must be emulated). This must be taken into considerations, in order to manage resources to be reserved for the virtual environment.

5.6.1 Implementation of Virtual Environment

After deciding that the environment is built using containers, specifically Docker-Compose. As discussed earlier, the nodes in the virtual network environment are required to do minimal network functionalities and are not complex. In order to maximize efficiency and decrease as much as possible resources the following OS are chosen for the deployment of each of the nodes:

- Servers and endpoints containers will have an image based on an alpine image that is a very stripped down OS (only 6 MB). The alpine image contains most of the functionalities needed for endpoints/servers but misses just the tool hping3 that will be used for testing (explained later). The tool is installed and a new image is created using the docker file.

- For routers again, for the sake of resources optimization, we use the base alpine image, which is sufficient for the router to perform routing. We install an addition of tcpdump package on routers for debugging purposes.

- For firewalls, the image depends on the type of the firewall. also the implementation of firewall differs. Lets go through each firewall type:

    1. iptables-firewall: is the easiest to deploy, since all it requires is an alpine image with iptables installed on it. This makes the iptables virtual network environment the most light weight and consumes incredibility low resources.
    2. bpf-firewall: is a bit more complex than deploying iptables since it requires a larger image and consumes more resources. We could use an ubuntu base image to build the bpf-firewall on base metal but this will generate an image of 2.7 GB large. This happens because bpf-firewall requires alot of dependencies before launching the container. Fortunately there is a possibility to run bpf-firewall as a container by just pulling "polycubenets/polycube" image that is more light weight (270 MB). However, bpf-firewall container requires linux kernel headers in order to

launch correctly the container thats why we mount the Linux header at container startup. Also, the "polycubenets/polycube" image had some functionalities that were installed using a docker file in order to make the container complete.

3. openvswitch: openvswitch works at the level L2, which makes it a little bit more tricky since the L2 connectivity is managed by docker bridges, so we have to modify the bridges at the host OS level in order to change the normal L2 bridges to openvswitches. This is done in startScript file, where it setup the L2 openvswitch as shown Figure 5.6.

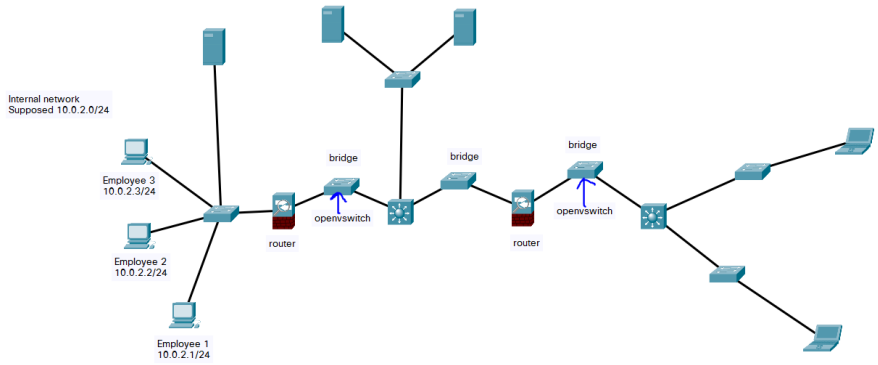

Figure 5.6: Open Virtual Switch Topology

The L2 connectivity is now manipulated at the host OS level, but to preserve the network topology, we left the external and internal firewalls now they just act as regular routers and forward traffic.

Now that it is decided to deploy our network using Docker Compose The next step will be to implement and run the environment. All the commands specified are extracted from the manual of docker compose [13].

To run the virtual environment, it is required to enter the install docker engine and Docker-Compose for that refer to Docker Documentation. Now that the requirements are installed to run the environment, start from the Folder "vnetwork" which contains all the code developed in order to reproduce the environment. Let us explore the different files inside the project:

1. startScript: This file is what runs the whole environment simply by executing the script ./startScript in the working directory of the project (/vnetwork). Then the whole environment will be reproduced but before the user must

choose which type of firewall is required to reproduce in the environment. iptables, bpf-firewall, or openvswitch. Moreover, the user also specify the corresponding TestCase to run, since everything is packaged inside the project, the environment will be reproduced and it only takes seconds before it is ready for testing.

2. endScript: After finishing the testing, stops and removes all created containers and networks. user just execute ./endScript.

3. docker-compose-bpf.yml: file responsible for creating the virtual network topology of our reference network having bpf-firewall as firewall type. It has a declarative syntax, we can easily remove/add a container/network and attach the newly added container to a network.

4. docker-compose-iptables.yml: file responsible for creating the virtual network topology of our reference network having iptables-firewall as firewall type. It has a declarative syntax. developer can easily remove/add a container/network and attach the newly added container to a network.

5. docker-compose-openvswitch.yml: file responsible for creating the virtual network topology of our reference network having openvswitch-firewall as firewall type. It has a declarative syntax, developer can easily remove/add a container/network and attach the newly added container to a network.

6. endpoints_build: contains the Docker file for creating the image of endpoints/servers in the environment with required packages.

7. firewall_build_bpf: contains the Docker file responsible for building the bpf firewall image with required packages.

8. routerfirewall_build_iptables: container Docker firewall that generates an image for routers and iptables firewall with required packages.

9. RouterFirewallConfig: contains the configurations of routers (static routes) and scripts of test cases.

10. startwireshark: starts a container that runs a wireshark image used for debugging purposes. It opens on the local host.

11. Hping Tests: The directory Contains a couple of tests to be done in the virtual environment.

12. README.md: contains some useful docker commands.

Example of Starting iptables virtual environment:

- Make sure docker service is up and running by

```
$ service docker status
$ service docker start # if the service is down
```

- Enter the vnetwork Directory and execute ./startScript, then Input firewall type "iptables"
- Input the TestCase number to be deployed from '1' to '7'
- if it is the first time running the environment, it will take some time to pull all required images for the containers and build the Docker Files.
- Now when all containers are created and started, terminals will automatically open corresponding to different nodes of the environment, and now the environment is ready for the testing.
- You can use for testing the commands specified in HpingTests directory, or create your own tests according to the firewall policies. Also tcpdump can be used on firewalls and routers in order to sniff packets for debugging.
- when the testing is done, simply execute the ./endScript to shutdown all of the environment and delete all containers/networks.

Now that the virtual environment is built for each firewall type, it is ready to be started and tested.

5.6.2 Testing using Hping3

According to [20], "Although injection based firewall testing is accepted as an inefficient way of testing firewall implementations in the literature. There has been no alternative method developed yet". Manual injection of packets is adopted to test the firewall implementation in the virtual environment. Now that the virtual environment up and running let's start the testing using Hping3 commands. Here is specified the general commands that are useful for testing. Leaving the possibility to manipulate these commands as desired for testing.

```
1 $ hping3 destinationIP -s sourcePORT -p destinationPORT -k -c
    numberOfPackets  # This command sends TCP type packets
2
3 where -s: specify source PORT, -p: specify destination PORT,
4 -k: specify to keep source port as it is for each ping, -c: specify
    number of packets to be send.
```

```
5
6 $ hping3 -2 destinationIP -s sourcePORT -p destinationPORT -k -c
    numberOfPackets # This command sends UDP type packets
7
8 where -s: specify source PORT, -p: specify destination PORT,
9 -k: specify to keep source port as it is for each ping, -c: specify
    number of packets to be send.
10
11 Examples:
12 $ hping3 30.0.5.2 -s 2000 -p 80 -k -c 7 # Send 7 TCP packets to
    destination IP 30.0.5.2 with port source 2000 and destination port
     80 keeping the source port as it is always.
13
14 $ hping3 -2 30.0.5.2 -s 2000 -p 80 -k -c 7 # Send 7 UDP packets to
    destination IP 30.0.5.2 with port source 2000 and destination port
     80 keeping the source port as it is always.
```

Using these commands we can generate all packets of type TCP or UDP that are of interest to us in our testing. After the firewall is deployed, we test it in each virtual environment, verifying that the firewall configurations are correct, and imposing policies flawlessly. The testing is done through terminal, where each terminal represent a specific container and the possible outcome of the testing is either success or fail of ping request, from this simple result we could ensure the correctness of firewall configurations. There is another possibility to do tests and inject packets rather than simple packet sending manually and that is discussed in [20] which is testing done toward implementation of already developed firewall rules.

5.6.3 Tests evaluation

According to [16]. This is how to perform firewall testing: (1) identify appropriate test cases, (2) derive the test packets, (3) send the test packets to the firewall and (4) evaluate the reaction of the firewall. and this is what is done so far. If the firewall does not react as intended, one of the following failures occurred:

1. The test case is faulty and predicts a wrong firewall reaction (e.g. the security policy specifies to block a packet but the test case indicates that the firewall forwards the packet).

2. The firewall rules do not implement the security policy (e.g. the security policy specifies to block a packet but the firewall rules let it pass).

3. The firewall implementation is erroneous and the rules do not correspond to the actions of the firewall (e.g. the firewall rule specifies to block the packet but the firewall forwards the packet).

4. The test environment has bugs or limitations. This is sometimes due to the low fidelity ratio between the real physical topology and the virtual one. where the virtual topology misses some characteristics that are relevant to the testing, for example, dynamic routing is not present in the virtual environment this reduces fidelity ratio (since probably in a real physical scenario dynamic routing will be present) but does not affect the outcome of testing of firewalls as static routing is enough, unlike other cases where if one functionality is missing that reduces the fidelity, this leads to wrong test results.

5.6.4 Environment Fidelity Effect

So testing fails not only because the rules in the firewall are not implemented or at fault, but sometimes tests fail because the virtual environment created does not perfectly replicate the real physical scenario and has some limitations due to fidelity (check Figure 3.3). Like in our case, one test that fails. If we send a UDP packet to a server specified in the environment, which is allowed by the firewall, the expected response from the server is a UDP packet. Rather an ICMP packet is sent, which is blocked by the firewall. To solve this issue, the environment must be manipulated by finding alternative ways to represent a UDP server that accepts UDP connections. For the UDP case, it is suggested to follow this approach: When sending a UDP packet from an endpoint to a server, the UDP packet is allowed to pass, but the response is an ICMP packet from the server container that the firewall drops, so it appears as if the test fails. Proposed Solution: use "nc" tool to create a client-server model and allow send of UDP and receive of UDP packets. EX: - "nc -u -l -p 1000 -s 30.0.5.1" -> server listening in UDP mode on port 1000 at address 30.0.5.1.

- "nc -u 30.0.5.1 1000" -> client. connect to server 30.0.5.1 at port 1000.

Using this proposed solution, the virtual network topology replicates the real one more faithfully, adding a higher degree of fidelity (similarity between virtual and real environment). The more relevant functionalities to the virtual environment, the more fidelity ratio increases. Optimally, we would like to reach a fidelity ratio near 1 (virtual environment replicates real environment exactly), but that is unrealistic and also not needed. We need to maximize the relevant fidelity ratio, that is, the functionalities that must be added to the virtual environment that directly affect firewall testing. If these functionalities are not present, this may lead to misleading test results. Other functionalities like dynamic routing do not affect the outcome of test results. Note that this is a qualitative approach, and not quantitative. In case quantitative results are needed a study must be done to see the set of all functionalities of physical environment and see how much is replicated exactly in the virtual environment,

To fully understand the concept of fidelity and relative fidelity the following

example is demonstrated:

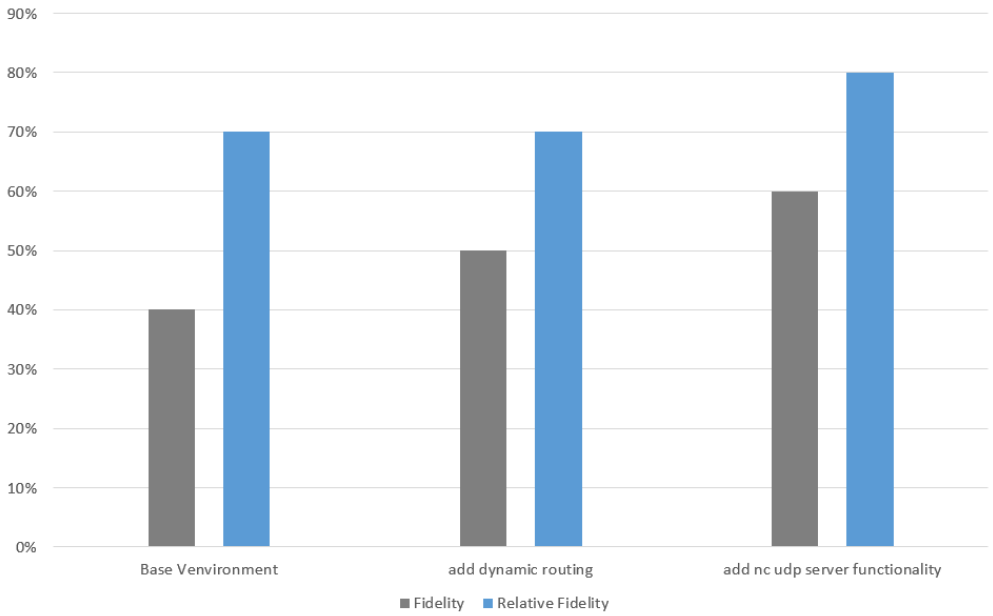

Figure 5.7: Fidelity Vs Relative Fidelity

In Figure 6.6, suppose that fidelity is measured in percentage, and the values here are just for the sake of the example. Let us break down each one:

- The base environment (without adding dynamic routing and nc functionality as discussed before) has the fidelity of nearly 40% which means that the virtual environment replicates 40% of the real physical environment with all its conditions (dynamic routing-physical characteristics of real nodes-functionalities of each node-applications mounted on each node-real type of operating systems on each node...) A virtual environment can never have 100% replicas of the physical one. While Relative fidelity, which is higher, corresponds to the similarity of the virtual environment with respect to an ideal virtual environment where firewall test conditions are the same as the real environment. Obviously, this percentage is higher since the functionalities that need to be supported by the virtual environment need to be affected directly by firewall testing, which is much lower than all functionalities supported by a real environment. (for example, dynamic routing presence does not affect the testing of firewall results, while having NAT functionality does affect since there is a change in packet headers).
- when adding to the base virtual environment dynamic routing, notice that the fidelity increases. However, the relative fidelity stays the same since this

functionality is not in the space of required functionalities for proper firewall testing.

- when adding to the base virtual environment the previous nc UDP server functionality (server node acting as a real server listening on UDP port), notice that the fidelity increases, as well as relative fidelity.

Finally, the goal is to maximize the relative fidelity to minimize the errors that can occur from testing in a fault virtual environment. The same methodology suggested here could be applied in other cases, where first, we define the space of minimal functionalities that are sufficient for performing testing in a virtual environment. We create the environment trying to achieve maximized relative fidelity.

5.7 Routing in Virtual Environment (Enhancement)

As seen in previous sections, the testing environment was built using docker containers. Exploiting their main advantage, which is a lightweight environment and flexibility. However, the environment was pre-produced from scratch, and everything was configured statically. Main focus here is on routing. From a routing point of view, in the previous network topology, the routes were built statically, which means the configuration files were written from scratch that configure static routes on each node in the network, which is time-consuming and error-prone. In a larger test topology with more than 20 nodes, writing static routing configuration files is not the best way to go. For that, this section will explore the possibility of using dynamic routing in a docker container environment and its advantages and disadvantages.

5.7.1 Routing Functionalities in Containers

Complex and redundant networks require different routing policies from those typically found in small networks. Ideally, the routers will know all the paths that lead to the target, but configuring them manually can quickly become confusing and lead to mistakes. In containers, the layer two networks are managed by docker, so no need to worry about it. Focus here on the layer three routing possibilities. Inherently in docker, there is no possibility for dynamic routing, nor is there a way to declare dynamic routing in docker-compose files or generally using docker management interfaces. So the only for us to establish dynamic routing was using the containers themselves. In other words, the containers now should act as routers more broadly. Previously, containers were already acting to some extent as routers.

They were forwarding packets, routing packets, and filtering packets according to some rules (firewalls). However, the containers' functionality are to be extended to resemble more routers. Of course, routers having network operating systems are more complex and offer a broad set of functionalities that containers with Linux operating systems can mimic to a particular limitation. Shown in the following tables, some of the functionalities that router offers and what Linux containers acting as routers have offered until now in our test environment.

Functionality	Router	Linux Container
Forwarding packets	yes	yes
multiple interfaces	yes	yes
Packet filtering	yes	yes
Dynamic Routing	yes	no
Nat Forwarding	yes	no
load balancing	yes	no
DHCP module	yes	no
QOS	yes	no
Bandwidth control	yes	no
...	...	...

Table 5.8: Router and Container Functionalities

Moreover, as the list of router functionalities goes on, one can easily see that our containers that 'act' as a router have minimal router functionalities. Nevertheless, that is the point; there is no need for all these router functionalities for our type of test environment, we focus on the minimal router functionalities that can be emulated as a Linux container to satisfy our need in such a testing environment, and like this, resources consumption is minimized of our environment.

As you have noticed in Table 5.8, the dynamic routing functionality is not yet supported by Containers in our case. To add dynamic routing, first find the right tools to be installed on containers to allow this possibility. Furthermore, the right dynamic routing algorithm will be chosen.

5.7.2 Dynamic Routing Algorithm and Tool

The solution to static routing is to distribute dynamically changing route information automatically. The web world has developed special dynamic routing protocols for this, usually only found on routers by Cisco or Juniper. The most famous tool used to add such functionality to Linux machines/servers is called Quagga, which is the one used.

Quoting [21], "Quagga gives IT administrators the option of participating in the world's largest group of routers with a Linux computer. The Quagga project

originated with the Zebra Routing Daemon by Japanese developer Kunihiro Ishiguro. The software is included in all popular Linux distributions". Quagga does not handle the routing since it is still the domain of the underlying operating system kernel. However, it does provide several routing protocols – Routing Information Protocol (RIP), RIPng, Open Shortest Path First (OSPF) for IPv4 and IPv6, Border Gateway Protocol (BGP), and Intermediate System to Intermediate System (IS-IS) – and it modifies the kernel routing table on the routes it learns. So Quagga is the software that installs on Linux that allows the possibility of dynamic routing on a Linux machine or, in our case, a Linux container. Quagga uses Zebra daemon to connect to the daemon. vtysh is used to connect through Unix sockets.

Note that Quagga includes several daemons: One service exists for each routing protocol. The various routing daemons are managed by a master daemon – known as Zebra. Every daemon has its configuration file and can be configured on a separate port. The Zebra control daemon controls and coordinates the whole thing. Using Alpine containers, Quagga is installed using the Docker file, which creates a new image with Quagga pre-installed. After installing Quagga, containers are configured to start the Zebra daemon with the corresponding Routing protocol immediately when the container starts. In such a way, the routing protocol with the running daemon is started automatically when the virtual environment starts, and routes are generated immediately.

To Start the daemon with the routing protocol, At run time, zebraStart.sh configuration file is mounted to containers that is executed when the latter starts. The zebraStart.sh file is written as follows without starting the routing protocol.

```
#!/bin/ash
echo "Starting Daemon"
/usr/sbin/zebra -d -f /etc/quagga/zebra.conf
for name in $@
do
    if [ $name = "zebra" ]
    then
        echo "Daemon already running"
    elif [ -s "/usr/sbin/$name" ]
    then
        if [ ! -f "/etc/quagga/$name.conf" ]
        the
            echo "Creating empty config for $name daemon..."
            touch /etc/quagga/$name.conf
        fi
        echo "Starting $name daemon..."
       /usr/sbin/$name -d -f /etc/quagga/$name.conf
    else
        echo "Unknown daemon: $name"
    fi
```

```
21 done
22 zebra -d
```

This configuration file starts the zebra daemon in an alpine container. Now that the Zebra daemon has started, we have to choose the routing protocol that we need to start on every container acting as a router to allow the correct functioning of the routing protocol.

5.7.3 RIP Routing Protocol

Quagga offers the following dynamic routing protocol:

1. Routing Information Protocol (RIP): The Routing Information Protocol (RIP) is an old distance vector routing protocol that uses the hop count as the routing metric. The largest number of hops allowed for RIP is 15, limiting the size of networks that RIP can support. In most networking environments, RIP is not the preferred choice of routing protocol, as its time to converge and scalability are poor compared to other protocols. However, it is easy to configure because RIP does not require any parameters, unlike other protocols.

2. Open Shortest Path First (OSPF): Is a routing protocol for Internet Protocol (IP) networks. It uses a link-state routing (LSR) algorithm and falls into the group of interior gateway protocols (IGPs), operating within a single autonomous system (AS). OSPF gathers link state information from available routers and constructs a topology map of the network. OSPF is widely used in large enterprise networks. IS-IS, another LSR-based protocol, is more common in large service provider networks.

3. Border Gateway Protocol (BGP): Is a standardized exterior gateway protocol designed to exchange routing and reachability information among autonomous systems (AS) on the Internet. It is not interesting in our case.

After checking some of the routing protocols offered by Quagga, we can see clearly that the best one is either RIP or OSPF (with their different versions). OSPF is more widely used in real networks as it is more scalable but requires more configuration effort than RIP. Using OSPF in our case will be too much in terms of configuration for just a relatively small network topologies. For our case, RIP is used. Although it is not used much now in networks, it is the best choice in our virtual environment since it is easy to configure, and RIP simplicity is sufficient for relatively small networks with 20 to 60 nodes. Note that RIP uses the User Datagram Protocol (UDP) as its transport protocol and is assigned the reserved port number 520 as we need to always open UDP port 520 on firewalls to allow the routing protocol to function correctly.

Now, after choosing the best routing protocol, containers are configured to act as routers with the rip protocol at the startup of the virtual environment. To do that, zebraStart.sh configuration file is modified to add rip protocol configuration where the configuration of rip is taken from the official documentation of cisco rip [22] . So that file could become as follows.

```
1
2  #!/bin/ash
3  echo "Starting zebra daemon..."
4  /usr/sbin/zebra -d -f /etc/quagga/zebra.conf
5  for name in $@
6  do
7      if [ $name = "zebra" ]
8      then
9          echo "zebra daemon is already started -> skip"
10     elif [ -s "/usr/sbin/$name" ]
11     then
12         if [ ! -f "/etc/quagga/$name.conf" ]
13         then
14             echo "Creating empty config for $name daemon..."
15             touch /etc/quagga/$name.conf
16         fi
17         echo "Starting $name daemon..."
18        /usr/sbin/$name -d -f /etc/quagga/$name.conf
19     else
20         echo "Unknown daemon: $name"
21     fi
22 done
23 zebra -d
24 ripd -d
25 hs=`hostname`
26 if [[ $hs == *"router1"* ]]; then
27 vtysh -c "configure terminal" -c "router rip" -c "network
       20.0.0.0/24" -c "network 20.0.1.0/24" -c "network 30.0.5.0/24" # -
       c "timers basic 30 1200 1200"
28 fi
29 if [[ $hs == *"router2"* ]]; then
30 vtysh -c "configure terminal" -c "router rip" -c "network
       20.0.2.0/24" -c "network 10.0.0.0/24" -c "network 10.0.1.0/24" # -
       c "timers basic 30 1200 1200"
31 fi
32 if [[ $hs == *"firewall1"* ]]; then
33 vtysh -c "configure terminal" -c "router rip" -c "network
       20.0.2.0/24" -c "network 20.0.1.0/24" # -c "timers basic 30 1200
       1200"
34 fi
35 if [[ $hs == *"firewall2"* ]]; then
```

```
vtysh -c "configure terminal" -c "router rip" -c "network
    10.0.2.0/24" -c "network 20.0.0.0/24" # -c "timers basic 30 1200
    1200"
fi
```

As is evident in the previous we manually are detecting the container of our reference topology (in chapter 2) and configure the rip protocol on each interface. This is a sufficient approach to achieve dynamic routing for a pre-produced network topology, where we know all nodes and the environment is static (doesn't change) which is our case. But it could be made more automated, for example, the configuration file could detect automatically the network interfaces attached to the container and their respective IP addresses that need to be added to the rip protocol. this automation allows configuring rip protocol for whatever topology, no need for nodes to be known apriori. For now, this advantage may seem not quite useful. But this advantage will be evident in the next chapters to follow. The zebraStart_automated.sh becomes as follows.

```
#!/bin/ash
echo "Starting zebra daemon..."
/usr/sbin/zebra -d -f /etc/quagga/zebra.conf
for name in $@
do
    if [ $name = "zebra" ]
    then
        echo "zebra daemon is already started -> skip"
    elif [ -s "/usr/sbin/$name" ]
    then
        if [ ! -f "/etc/quagga/$name.conf" ]
        then
            echo "Creating empty config for $name daemon..."
            touch /etc/quagga/$name.conf
        fi
        echo "Starting $name daemon..."
       /usr/sbin/$name -d -f /etc/quagga/$name.conf
    else
        echo "Unknown daemon: $name"
    fi
done
zebra -d
ripd -d
ip_addresses=$(hostname -i)
ip_addresses=(${ip_addresses//" "/ })
for i in "${ip_addresses[@]}"; do
new=""
IFS='.' read -ra ADDR <<< "$i"
for (( j=0; j<3; j++ ));
```

```
30 do
31   new+="${ADDR[$j]}."
32 done
33 new+="0/24"
34 vtysh -c "configure terminal" -c "router rip" -c "network $new"
35 done
36 vtysh -c "configure terminal" -c "router rip" -c "timers basic 30
     1800 1800"
```

Now that everything is ready for deploying rip protocol, we will test this deployment of rip routing protocol into our virtual environment already developed in previous chapter.

5.7.4 Rip Protocol in Virtual Environment

After finding the tool and writing the configuration files, the virtual environment is ready to be deployed for testing that supports rip as a dynamic routing protocol. This section points out the advantages and disadvantages of using a dynamic routing protocol and more precisely using RIP protocol. First, let's recall the reference network topology with the containers used and their images.

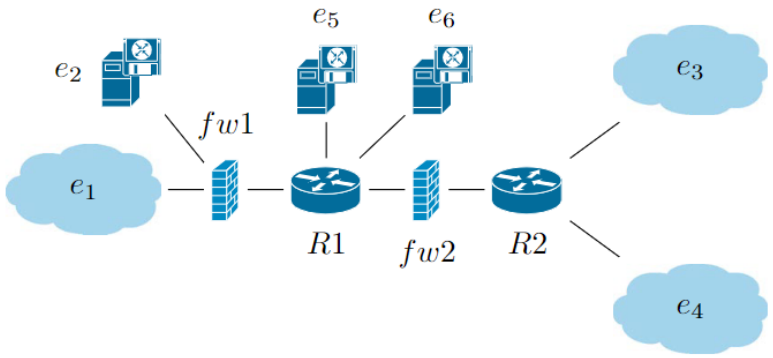

Figure 5.8: Network Topology

Again to summarize the containers purposes, recall three kinds of nodes: 1. endpoint and server nodes: Performs simple tasks like pinging. For that, docker image "endpoint" is used, a simple alpine image with an additional "hping3" package installed. 2. routers: used to apply forwarding of packets according to routes in routing tables, but now with additional functionality that is dynamic routing. For that, the docker image "router_firewall_rip" is used which is an Alpine image with some packages that include Quagga software installed. 3. firewalls: Also required to perform routing plus filtering, again now with additional functionality that

is dynamic routing, so the docker image "router_firewall_rip" is used that is an Alpine image with some packages that include Quagga software installed. But for firewalls, note that every firewall should have port 520 UDP open for the correct functioning of the rip protocol. Note Here. ONLY iptables is used as firewall types other firewall types are not yet supported.

Now start the virtual environment with the configuration file zebraStart.sh that is mounted and executed by routers and firewall containers at the start of the environment. As soon as the environment is up, checking the routing table of any firewall or router container, notice that the rip protocol has populated the routing tables of the containers by routes that are learned using the protocol. The routes are accurate and correct, and now there is no need to worry about routes and their correctness as is done before using static routes. In general, after testing the environment, we observe the advantages and disadvantages of using a dynamic routing protocol, specifically RIP:

+ Easier to create, rather than writing static routes every time a new environment is created.
+ less error prone, and saves time
+ rip is easy to configure and simpler than other routing protocol
+ can adapt to environmental changes in case a new container was attached or detached.
+ decouples virtual environment from routing process, so if a new network topology is created, there is no need to find static routes and write them.
− additional functionality that needs to be maintained.
− needs to open the UDP port on firewalls for the protocol to function.
− additional CPU consumption by environment as a result of deploying the rip protocol (but the additional overhead is minimal and not significant)

Finally, implementing new functionality in the virtual environment is a continuous process. A lot of functionalities can be added to make the environment more robust and error-free and to facilitate the creation of more nodes in a topology (scalability and flexibility). Each time a new functionality is added it should be tested and ensure that it doesn't interfere with other functionalities implemented in the environment.

Chapter 6

VEREFOO Demo And Translator Algorithm

The previous chapters focused on a specific component of VEREFOO. Precisely, on verifying the translation of Firewall Medium Level Rules (FMLR) to Firewall Low-Level Rules (FLLR). Then a sample network topology was created to test these firewalls in a virtual environment. But the first part of VEREFOO was ignored, the refinement of medium-level policies (MLP) to firewall allocation scheme (FAS) and firewall medium-level rules (FMLR). This chapter will demonstrate how VEREFOO operates as a single process from the Allocation Graph (AG) input to the Firewall Allocation Schema (FAS) output with the firewall rules and also the low-level configuration files. To do this demonstration, a Full Demo approach is used with a certain pre-defined network topology that will be more ramified and include more functionalities than the previous network reference topology.

6.1 Background

The following Figure summarizes the steps that an input Allocation Graph XML follows until reaching the deployment of output configuration files in a virtual environment, as the whole process is described in detail in [5].

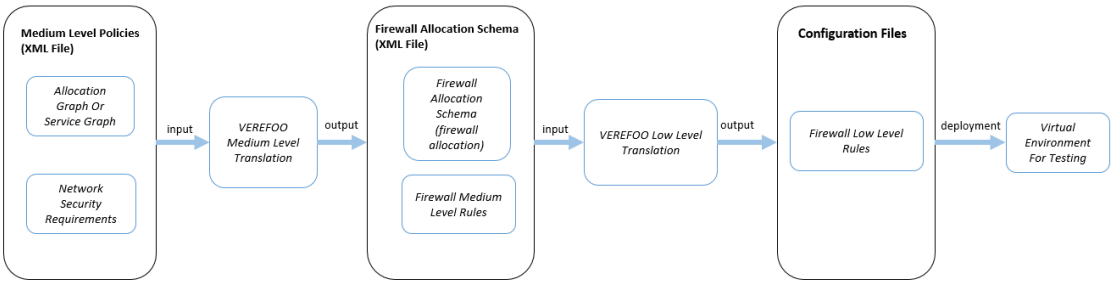

Figure 6.1: Verefoo Process

So let us break Figure 6.1. First, start from the Service Graph (SG) created by the network administrator. The Service Graph defines the network topology in an XML file manner with different XML elements. Along with the SG, the Network Security Requirements (NSRs) are also defined. When the SG is taken as input to the framework, Allocation Places will be automatically added to the Graph (APs are located at each point-to-point link where firewalls may be possibly allocated). Alternatively, AG and NSRs will be used immediately as input to the framework. The first part of the framework, which is responsible for translation from medium-level policies (MLP), will output a Firewall Allocation Schema (FAS) along with Firewall Medium Level Rules (FMLR). The FAS and FMLR are in the same output XML file, similar to the input one but with a firewall allocated along with their firewall rules. Then, after the FAS generation, it is used as input to the second part of the framework, which is responsible for translating the medium-level firewall rules to low-level rules specific to a firewall type (tables - openvswitch ...). The output files are Configuration files ready to be deployed on firewalls. Finally, After the configuration files are generated, the pre-produced virtual environment is launched based on containers for testing purposes.

From the brief description of the process, currently, the framework is decomposed into two parts, in addition to the third part, which is deploying in a virtual environment. What is done in the next sections is generate a Full Demo that allows the whole parts of the framework to be seen as one—automating the whole process where the user inputs the AG and NSRs and immediately gets the firewall configurations deployed in a virtual environment. There will be two versions of the demo, one version that contains an explanation of the whole process at each step and another demo where there are no instructions, and the final result is generated immediately.

6.2 Full Demo Topology

This section will show best topology to represent the Demo. For that, the topology already presented by the TDSC paper [11] is chosen that is also tested with the VEREFOO framework. The topology chosen is great for our demo since it already represents a well-ramified network and covers a set of additional functionalities with respect to chapter 2 reference topology to be emulated in the virtual environment to be developed (NAT - Loadbalancing). This is also a good example to test the scalability of docker containers in terms of resource consumption.

The following is the Allocation Graph topology depicted from the TDSC paper [11].

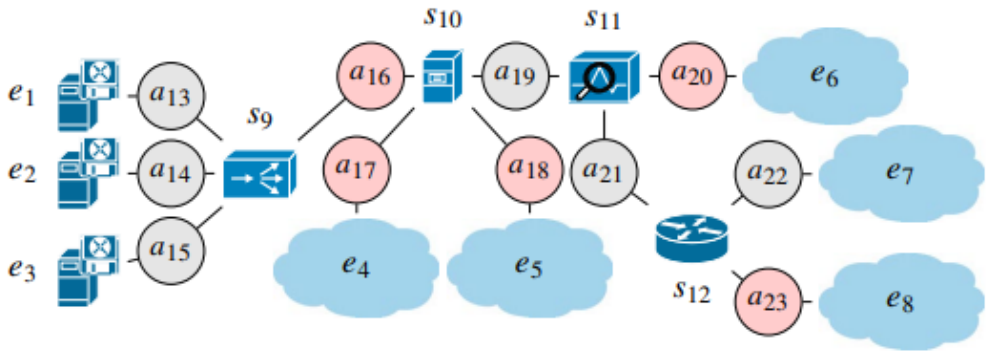

Figure 6.2: Allocation Graph

And the following are the functionalities of each node in the Allocation graph, note that aij is allocation place where a firewall could be possibly allocated.

Identifier	IP address	Function type / role
e_1	130.10.0.1	HTTP web server
e_2	130.10.0.2	HTTP web server
e_3	130.10.0.3	HTTP web server
e_4	40.40.41.∗	IT office of Company A
e_5	40.40.42.∗	Business office of Company A
e_6	88.80.84.∗	Company B
e_7	192.168.1.∗	IT office of Company C
e_8	192.168.2.∗	Business office of Company C
s_9	130.10.0.4	Load balancer
s_{10}	33.33.33.2	Web cache
s_{11}	33.33.33.3	Traffic monitor
s_{12}	220.124.30.1	NAT

Figure 6.3: Allocation Graph Functionalities

Finally the Network Security Requirements are defined, that are according to TDSC [11], set of the user-provided NSRs assumed to be anomaly-free and no conflicts between them.

Action	IPSrc	IPDst	pSrc	pDst	tProto
Allow	192.168.1.∗	192.168.2.∗	∗	∗	∗
Allow	192.168.2.∗	192.168.1.∗	∗	∗	∗
Allow	192.168.1.∗	130.10.0.∗	∗	80	TCP
Deny	192.168.1.∗	130.10.0.∗	∗	≠80	TCP
Deny	192.168.1.∗	130.10.0.∗	∗	∗	UDP
Deny	192.168.2.∗	130.10.0.∗	∗	∗	∗
Allow	130.10.0.∗	192.168.1.∗	∗	∗	∗
Allow	40.40.41.∗	130.10.0.∗	∗	80	TCP
Deny	40.40.41.∗	130.10.0.∗	∗	≠80	TCP
Deny	40.40.41.∗	130.10.0.∗	∗	∗	UDP
Deny	40.40.42.∗	130.10.0.∗	∗	∗	∗
Allow	130.10.0.∗	40.40.41.∗	∗	∗	∗
Allow	40.40.42.∗	40.40.41.∗	∗	∗	∗
Deny	40.40.41.∗	40.40.42.∗	∗	∗	∗
Allow	88.80.84.∗	40.40.∗.∗	∗	∗	∗
Deny	88.80.84.∗	130.10.0.∗	∗	∗	∗

Figure 6.4: Network Security Functionality

Also the example provided by the paper is already tested in VEREFOO, but limited to generation of Firewall Allocation Schema and firewall medium level rules. Which are shown in Figure 6.5 and Figure 6.6.

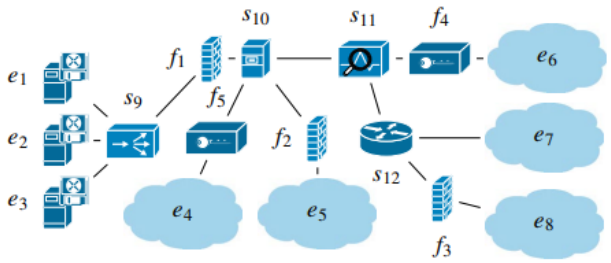

Figure 6.5: Firewall Allocation Schema

Firewall fw_1

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	220.124.30.1	130.10.0.4	*	80	TCP
2	Allow	40.40.41.*	130.10.0.4	*	80	TCP
3	Allow	130.10.0.4	*.*.*.*	*	*	*
D	Deny	*.*.*.*	*.*.*.*	*	*	*

Firewall fw_2

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	40.40.42.*	40.40.41.*	*	*	*
2	Allow	88.80.84.*	40.40.42.*	*	*	*
D	Deny	*.*.*.*	*.*.*.*	*	*	*

Firewall fw_3

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	*.*.*.*	192.168.*.*	*	*	*
D	Deny	*.*.*.*	*.*.*.*	*	*	*

Figure 6.6: Firewall Rules

but low-level configuration files weren't produced and are not yet tested in a virtual environment. Here the complete demo includes the whole process of verefoo from translation to FAS and FMLR to the deployment of configuration files in a virtual environment for testing. Ultimately, the input AG ready with the NSRs to be the used in the complete demo, next section will show the build of the demo using a script that automates the whole process for the user to test.

6.3 Full Demo Components

The Demo will be assumed to run on an Ubuntu Machine (20.04 LTS). The Main components of the Demo will be:

- startfull.sh : Script that runs the full demo with instructions at each step. This

type of demo shows using the terminal the whole process of interacting with VEREFOO framework using curl commands. This demo with the example provided in it is intended to explain the whole process in an easy and automated manner to users.

- GUI : Directory responsible for having the gui of the framework.
- vererfoo : Directory responsible for having the framework where jar file is to be generated.
- demo-vnetwork : Directory responsible for having the virtual environment of the FAS of the previous example that is already mentioned.
- testfiles-images: Where all testfile and images related to demo are saved

Let's dive into the details of each component and there functioning. First start with the "startfull.sh", it is decomposed mainly into 3 parts.

1. Demo visualization functions: Are the functions responsible for how commands are visualized and for demonstration purposes.
2. step functions: Are the functions that compose the demo itself, each step in the demo is demonstrated by one step function. This also include a function that check if required packages are installed before starting the demo.
3. Demo start: Which are series of commands to execute the demo in an ordered manner.

To run the demo successfully (i.e execute 'startfull.sh'), there is a number of packages that are required to be installed. In any case, the demo will fail to launch in case any requirement to run the demo is missing. The requirements are:

1. Linux Ubuntu 20.04 LTS Virtual Machine
2. Docker Engine for ubuntu
3. Docker-Compose for ubuntu
4. java openjdk-1.8 (for framework launching)
5. node.js (for GUI launching)
6. curl package
7. pv package

8. z3 library must be installed for ubuntu, and must be located in /home/z3 directory (to launch framework correctly). in addition to changing /etc/environment for library path.

After all the requirements are met, you are ready to run the demo to demonstrate the process of running VEREFOO and using GUI. The next component is the GUI developed explicitly for the VEREFOO framework. Using the GUI, The drag-and-drop approach can be used to create dynamic network topologies and allocation graphs. Once the topology has been created, the GUI could be used to specify the Network Security Requirements. Finally, sending the created topology with the NSRs to the framework is possible, and the returned FAS is visualized using the GUI. The third component is the framework itself, which takes as input AG and NSRs, and outputs FAS in addition to firewall rules and low-level configuration files. The following subsection will discuss the component of interest for this thesis work which is the Virtual Environment.

6.4 Full Virtual Environment

The Last component is the ‘demo-vnetwork‘ where the virtual environment is stored. The virtual environment will be based on a docker container and using docker-compose as the declarative language for running multiple containers in parallel. The Network topology built is shown in Figure 6.5 and Figure 6.6. However, the previous topologies miss the low details of a Network topology like point-to-point link network and IP addresses assignment, for that, the following topology is designed with all the missing details.

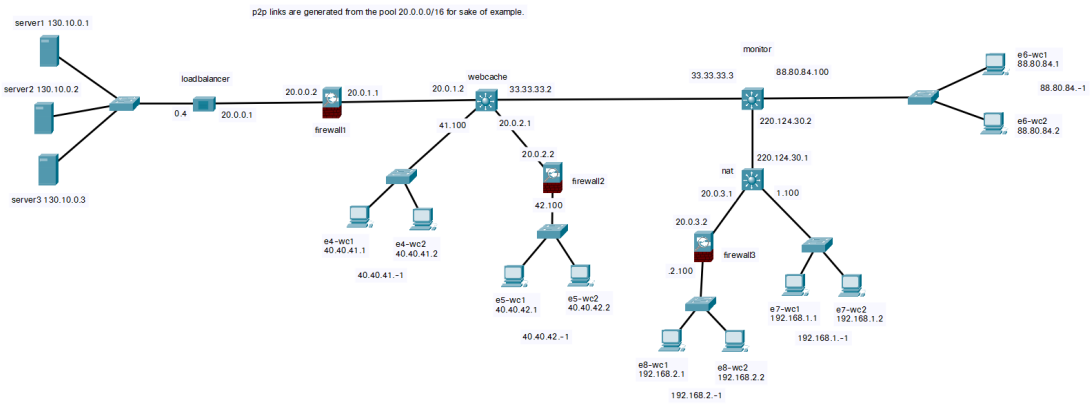

Figure 6.7: Firewall Network Topology

As shown in Figure 6.18, the endpoint networks were replaced by two hosts for

each network, just for the sake of representation. For example network 192.168.1.-1 is represented by host 192.168.1.1 and host 192.168.1.2.

As discussed earlier in Chapter 2 the nodes in the virtual network environment are required to do minimal network functionalities and aren't complex. It is already discussed some docker images used to run the environment, particularly in Chapter 2 environment the functionalities that are addressed were endpoints (web clients and web servers), containers acting as routers with limited functionalities, and firewalls (iptables - bpf - openvswitch in layer 2). But now the topology is more complex as two more functionalities were added which are the load balancer and NAT.

To create these two functionalities, recall Table 5.8. Currently our Linux container represents some of the real router functionalities, chapter 3 added the Dynamic routing functionality, and now we will add the NAT Forwarding and load balancing functionalities.

Functionality	Router	Linux Container
Forwarding packets	yes	yes
multiple interfaces	yes	yes
Packet filtering	yes	yes
Dynamic Routing	yes	yes
Nat Forwarding	yes	yes
load balancing	yes	yes
DHCP module	yes	no
QOS	yes	no
Bandwidth control	yes	no
...	...	...

Table 6.1: Comparison Router and Container Functionalities

NAT stands for network address translation, It's a way to map multiple local private addresses to a public one before transferring the information. So there is a need to translate the Private IP of our network to public IP which is at the interface of the NAT. Fortunately, using iptables it is possible to emulate NAT behavior in a container. So for that there is already have a docker image that is based on alpine and have iptables installed, that is the same image used to emulate iptables firewalls. For example, in our case we need to create a NAT that shadows networks 192.168.1.0/24 and 192.168.2.0/24, so any packet with an IP address from any of these networks must be translated to Nat's IP address, to do that simply write:

```
$ sudo iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -j SNAT - -to-source
220.124.30.1
$ sudo iptables -t nat -A POSTROUTING -s 192.168.2.0/24 -j SNAT - -to-source
220.124.30.1
```

like this we have the Nat container configured, to translate any IP address from 192.168.1.0/24 or 192.168.2.0/24 to 220.124.30.1 which is the interface IP of the Nat.

For the Load balancer functionality it is a bit more complex, as there is a couple of ways to emulate a load balancer, and according to [], "load balancing refers to the process of distributing a set of tasks over a set of resources, to make their overall processing more efficient", so in our case, distribute packets reaching 130.10.0.4 load balancer among the three servers available. For the sake of this example, Round Robin Algorithm will be implemented as in [23]. In summary, this algorithm takes two different parameters: every (n) and packet(p). The rule will be evaluated for every n packet starting at the packet p. To load balance between three different hosts you will need to create those three rules:

```
$sudo iptables -A PREROUTING -t nat -p tcp -d 130.10.0.4 - -dport 80 -m
statistic - -mode nth - -every 3 - -packet 0 -j DNAT - -to-destination 130.10.0.1:80
$sudo iptables -A PREROUTING -t nat -p tcp -d 130.10.0.4 - -dport 80 -m
statistic - -mode nth - -every 2 - -packet 0 -j DNAT - -to-destination 130.10.0.2:80
$sudo iptables -A PREROUTING -t nat -p tcp -d 130.10.0.4 - -dport 80 -j DNAT
- -to-destination 130.10.0.3:80
$sudo iptables -A POSTROUTING -t nat -p tcp -d 130.10.0.1 - -dport 80 -j SNAT
- -to-source 130.10.0.4
$sudo iptables -A POSTROUTING -t nat -p tcp -d 130.10.0.2 - -dport 80 -j SNAT
- -to-source 130.10.0.4
$sudo iptables -A POSTROUTING -t nat -p tcp -d 130.10.0.3 - -dport 80 -j SNAT
- -to-source 130.10.0.4
```

This will distribute traffic that reaches the load balancer at 130.10.0.4 into the three servers server1,server2, and server3 at addresses 130.10.0.1, 130.10.0.2, and 130.10.0.3 respectively. and the distribution will be done every nth packet. It is important for us to emulate these two functionalities (Nat and load balancing) since for testing the firewall, we need to ensure that any packet modification in terms of the route (load balancer) or packet headers (Nat) is represented correctly so the testing in the environment is closest to a real-life scenario.

Now that all functionalities are ready to be deployed let's review the docker images with their main packages used to deploy such an environment.

- end points (web client and web servers): Docker image is based on alpine with hping3 package installed for testing.
- firewalls: First for iptables, images is based on alpine with iptables installed. Then for openvswitch, since it acts at layer 2, the firewalls containers in this case act as routers only and firewall is deployed at layer 2. finally for bpf firewall, we use the image

- Forwarders (containers acting as routers this include webcache and monitor): For this we use simple docker image based on alpine with forwarding mechanisms enabled.

- NAT: simple alpine image with iptables.

- Load Balancer: simple alpine image with iptables.

And like this, all images built using Docker files are ready to run the environment using Compose file in a declarative manner. In addition to the docker-compose file there is all the other required configuration files that are mounted to containers at startup time and executed automatically, this includes firewall configuration files, routes configuration files (or dynamic routing configuration file, if it is enabled), Nat configuration file, Load Balancer configuration file, Docker files to build images with required packages, and finally startScript and endScript to start and shutdown environment. Like this the environment can be launched and as in Chapter 2 specified, we could start testing to verify that the NSRs are satisfied using various tools like hping3, tcpdump for debugging, iptables commands to check correct configuration deployment... Finally, it's good to note that the environment even with running 20+ containers each doing different functions, is still lightweight and can be easily run locally on a machine.

6.5 Running the Demo

Now it is time to combine all the components of the Demo to run. The goal of the demo is to demonstrate how user can use the framework starting from an example input Allocation Graph with the Network Security Requirements defined, reaching the goal of allocating firewall into a topology with their configuration files. A user can run the Demo and understand how to use the framework along with its GUI. Furthermore, the advantage of such a Stand-alone demo is the step-by-step approach that is followed that allows the user to understand the different elements of the process with the commands executed. Generally, the Demo has 6 steps:

1. Step 0: Display to the user the Allocation Graph along with the Network Security Requirements through images or xml file.

2. Step 1: Demonstrate to user how to generate the jar file that runs the framework, Although in our case the jar file is already generated for ease of usage of demo.

3. Step 2: Run the framework using SPRING.

4. Step 3: Demonstrate how to Interact using REST API with the framework to generate the firewall Allocation Schema and display the file generated along with image of the new topology.

5. Step 4: Demonstrate how to get the configuration files of a specific firewall type using REST APIs.

6. Step 5: Demonstrate how to launch the virtual environment, and do testing in it to verify that NSRs are satisfied.

7. Step 6: Final step requires user to interact with the GUI, and see the possibility of using GUI to create using drag and drop approach Allocation Graphs and defining NSRs, then sending them to the framework to get back the FAS.

After running the demo, the user is expected to understand the process of how VEREFOO works. Furthermore, the user can use such demo to integrate VEREFOO framework to his own environment.

6.6 Translator Algorithm

This section extends what's already done in previous chapters and particularly in previous section when building the Demo virtual network. In terms of building a contained virtual environment for firewall testing. By creating an algorithm, used to automatically create virtual environments.

6.6.1 Introduction

Previous chapter began with a simple common topology, for which there was internal/external networks and DMZ where servers are placed. For the first time, forwarding and filtering capabilities were introduced. Also, docker containers were chosen to represent the environment mainly due to resource limitations. Then in chapter 4, we developed a more complex topology with additional functionalities, and that requires more resources (running more than 20 containers). All of that gave us the basis for developing network topologies using containers. However, rather than writing the configuration files for the virtual environment (where firewalls must be tested) from scratch, the process could further automate the process of producing a virtual environment. An algorithm was created that takes the Firewall Allocation schema as input and produces the required configuration files to convert the Firewall allocation schema topology into a real network topology with containers. To achieve that, the Algorithm must be able to extract information from the FAS XML file and translate them into configuration files that are used to start up the environment. This comes with limitations, and the limitations emerge

from the fact that the FAS XML file contains very limited information about the real network topology it contains a very high-level abstraction of network topology with very little information about the real topology. The following sections will demonstrate the possibilities of creating such a translator algorithm. The following questions will be answered, is it possible, is it effective, what are the limitations, what are the benefits to the framework, and finally, what future possibilities could be gained from such a feature. Note that all the work done in this chapter will mainly focus on iptables firewall type.

6.6.2 Network Topology Knowledge

Before building the algorithm, first we define what intelligence can be extracted from the FAS XML file. How much does the FAS provide details about the real network topology, what are the missing pieces of information that FAS does not provide, and how to generate these missing information. To begin, let us look at what the firewall allocation schema provides.

```
<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>
```

Figure 6.8: XML node schema

In the Figure 6.8, A typical XML node contains the following general information:

1. name is usually an IP identifier, but this doesn't represent the node's IP. The reason is that if we have a router (FORWARDER) with a couple of IP

interfaces„ how do we know the IPs attached to each interface. This IP may be useful to identify endpoints as endpoints have only one IP to be attached to them.

2. id: the id of the node in the XML, but for our case, it isn't important since no information that helps in building topology could be extracted from it.

3. functional_type: this helps us to identify the functionality that the node must support in the real topology. For example, a FORWARDER could inform us that the container representing this node will simply act as a router with several interface and forwarding capabilities, Or a firewall/NAT/load balancer container will require certain configuration files. So with this, based on functional type each container will have a certain image. Furthermore, the topology connections will be based on nodes' functional types.

4. list of neighbors: each neighbor is represented by a name which is the IP identifier of the neighbor node. This is used to represent links in VEREFOO, but in network topology, the information provided by neighbor names is not useful. It is useful to build the topology in terms of nodes' position with respect to each other. But it doesn't provide real details about the link-to-link connectivity and the assignment of IP addresses to each container interface. Using the information provided by neighbour names and adding logic to it, the algorithm could assign IP addresses effectively to interfaces of a specific container to ensure point-to-point connectivity between two adjacent containers.

5. Configuration: This is useful only in the case of NAT and loads balancer containers. Using information about Nat source IP addresses that are shadowed by NAT or pools of addresses that load balancer use to load balancer traffic. This information is used to generate the configuration files of NAT and load balancer nodes.

Mainly those are the elements that could be present in a node. The challenges to overcome are:

- when the Allocation places (AP) are defined in the input Allocation Graph (AG), the VEREFOO framework will allocate firewalls in some of the AP, and the rest of the AP, FORWARDERS nodes will be defined, which are quite useless for the defining the real network topology. It is therefore necessary to find a way to get rid of unneeded nodes in the FAS. To solve this issue, either implement the removal of forwarders at the algorithm level or at the VEREFOO level, where the output FAS doesn't contain such unneeded forwarders.

- we need to define a methodology, to determine the IP addresses assigned to each interface of each container, and in case a container requires multiple interfaces, what are the IP addresses to be assigned to them.

- how to determine what IP addresses to be attached to containers based on node name and neighbor names. also, we need to find a way to define new network sub-nets for point-to-point links.

- Routing: How to decide statically the routes before knowing the topology apriori, this issue is solved thanks to the possibility of dynamic routing explained in chapter 3.

There are a lot of challenges to overcome, and building an algorithm that translates FAS to a virtual environment that handles the randomness and lack of information about real network topology in the FAS is somewhat complex and, to some extent, impossible to cover all possibilities. Even if an algorithm was build to cover all such scenarios, it would be relatively inefficient and unnecessarily complex. From here, there are two possibilities. First, we could manipulate the FAS that the VEREFOO framework outputs, so it provides more information about the real topology. For example, nodes can now provide a list of NIC addresses to be attached to containers. Such improvement will make the translator algorithm more uncomplicated and more straightforward. The idea here is that the more information injected into the FAS and the more standard the FAS XML file is (standard format), the simpler and error-free the translation algorithm would be. The second possibility is to leave FAS as it is and make the Algorithm more complex in a way that it can handle as many scenarios as possible with a high success rate in translation, and this is rather complex and isn't addressed here. This chapter is instead offers a compromise what will be discussed is to, force some limitations on the input FAS XML file, and make the algorithm a little bit complex. If those limitations are not met, the algorithm will fail the translation.

The main takeaway is the idea, that the more information we have in the FAS XML file, the better the translation is. We can not extract from FAS full real topology knowledge, so approximations must be made. This leads to a specific success rate.

6.6.3 Building The Algorithm

This sub section will demonstrate how the algorithm operates. But before, as anticipated previously, we will express the limitations that the FAS XML file must satisfy to achieve high confidence with the translation.

6.6.4 Firewall Allocation schema Input

While developing the algorithm, the same topology as chapter 4 is used, which is in Figure 6.18. The algorithm will be based on this topology. In other words, in the end, the algorithm should correctly translate the FAS XML file that represents Figure 6.5 and Figure 6.6, into Figure 6.18 built by containers with all IP addresses assigned correctly. So any topology that has a similar structure as this FAS example, the algorithm should be able to translate it correctly. Lets break down the FAS XML file, that represent Figure 6.5 and Figure 6.6. Mainly the input XML is decomposed into 6 main types of nodes that are distinguished during translation.

```
<node name="130.10.0.1" functional_type="WEBSERVER">
        <neighbor name="1.0.0.1"/>
        <configuration name="httpserver1" description="e1 ">
            <webserver>
                <name>130.10.0.1</name>
            </webserver>
        </configuration>
```

Figure 6.9: Web Server XML element

Starting from Web clients and Web servers, which are the endpoints of our topology, and they are easiest to translate since the name of the web client or web service XML node is the same IP address that will be attached to the container that will represent it. And there is only one neighbor name, which will represent the gateway for these endpoints. Endpoints have no limitations and they are not required to be in a specific format for the translation since they are simple. Note that the IP of endpoints in virtual environment must be the same as the XML input file for correct testing.

```
<node name="1.0.0.1" functional_type="FORWARDER">
        <neighbor name="130.10.0.1"/>
        <neighbor name="130.10.0.4"/>
        <configuration name="ForwardConf">
            <forwarder>
                <name>Forwarder</name>
            </forwarder>
        </configuration>
</node>
```

Figure 6.10: Allocation Place forwarder XML element

Second is the first type of FORWARDER, when the AG is inputted to the framework, as output in the FAS Allocation places is replaced by firewalls allocation, and the rest of the APs are replaced by forwarders at each point-to-point link, so these forwarders are at all point to point links. This makes the translation harder since such forwarders are useless and need to be removed, they just represent that traffic must be forwarded at the point-to-point links, but we won't create a container acting as a router for each forwarder, it will be too much and waste of resources. Such forwarders will be removed by the VEREFOO framework, so the input FAS XML will not have such forwarders anymore. This issue is solved at the FAS level not at the algorithm level, this leads to easier translation algorithm and more efficient.

```
<node name="33.33.33.2" functional_type="FORWARDER">
        <neighbor name="1.0.0.4"/>
        <neighbor name="1.0.0.5"/>
        <neighbor name="1.0.0.6"/>
        <neighbor name="1.0.0.7"/>
        <configuration name="ForwardConf">
            <forwarder>
                <name>Forwarder</name>
            </forwarder>
        </configuration>
</node>
```

Figure 6.11: forwarder XML element

Third is the Second type of FORWARDER (web cache - traffic monitor ...), these forwarders must be translated to containers acting as routers and they should forward traffic. Depending on neighbouring node, a point-to-point link address is attached to the forwarder container. Note here the name of the forwarder isn't taken to consideration, it is ignored to make the algorithm less depended on user input in terms of forwarder node name.

```
<node name="130.10.0.4" functional_type="LOADBALANCER">
        <neighbor name="1.0.0.1"/>
        <neighbor name="1.0.0.2"/>
        <neighbor name="1.0.0.3"/>
        <neighbor name="1.0.0.4"/>
        <configuration name="loadbalancer">
            <loadbalancer>
                <pool>130.10.0.1</pool>
                <pool>130.10.0.2</pool>
                <pool>130.10.0.3</pool>
            </loadbalancer>
        </configuration>
</node>
```

Figure 6.12: Load Balancer XML element

Fourth is the load balancer, the only difference between a load balancer and a container acting as a router, is that the load balancer must have a configuration file that load balances traffic between the servers. To generate the load balancer configuration file the pool elements are used which are the servers to distribute traffic.

```
<node name="220.124.20.1" functional_type="NAT">
        <neighbor name="1.0.0.9"/>
        <neighbor name="1.0.0.10"/>
        <neighbor name="1.0.0.11"/>
        <configuration name="nat">
            <nat>
                <source>192.168.1.-1</source>
                <source>192.168.1.-2</source>
            </nat>
        </configuration>
</node>
```

Figure 6.13: NAT XML element

Fifth is the NAT, the only difference between NAT and a container acting as a router, is that the NAT must have a configuration file that shadows the IP addresses of private networks. To generate the NAT configuration file we use the source elements which are the private addresses to shadow.

```
<node name="1.0.0.4" functional_type="FIREWALL">
        <neighbor name="130.10.0.4"/>
        <neighbor name="33.33.33.2"/>
        <configuration name="AutoConf">
            <firewall defaultAction="DENY" >
                <elements>...</elements>
                ....
            </firewall>
        </configuration>
</node>
```

Figure 6.14: firewall XML element

Finally is the firewall. The firewall will filter packets, and route unfiltered packets, the configuration files for the firewall are already generated by the framework, so the firewall configuration part of the XML node is ignored.

Finally, before going into algorithm implementation, let's summarize what is expected to be the input FAS:

1. Allocation places (firewalls) must have only two neighbours not more which is the usual case.
2. For simplicity all sub-net are considered /24. other sub-nets are not supported currently
3. In terms of links, the algorithm doesn't expect to have two adjacent NAT connected to each other or two adjacent load balancers for example, this would lead to errors. It is expected that the input FAS XML represent a usual network topology with no unexpected connections.

Such assumptions facilitate the algorithm's operation, in the future, the algorithm could be improved to handle more general cases of FAS, or if the FAS was made more standardized the algorithm could be improved accordingly.

6.6.5 Implementation of Translator

let's explain the concepts related to the algorithm, which are translatable XML and non-translatable XML. Simply a translatable XML is a series of XML nodes (or XML file), where the meaning of the name and neighbor name of a node are redefined internally in the algorithm, where the IP addresses of name/neighbor names are simply IP addresses of NICs to be attached to the container. Explicitly:

1. firewalls: neighbour names are NICs addresses to attach to container (ignore node name).

2. endpoints: name of node is NIC address to attach to container, and neighbour name is the gateway of the endpoint.

3. loadbalancer: name and neighbour names are addresses of NICs to attach to containers.

4. NAT: name and neighbour name are addresses of NICs to attach to containers.

5. forwarder: ignore node name and attach only neighbour names as NICs to containers.

While non-translatable XML is simply the input FAS, where node names and neighbor names are not related to the real network topology, they are just high-level definitions of nodes.

This distinction is offered for various advantages, first advantage is this makes the translation part (which translates XML to docker-compose configuration file) of the algorithm rather easy and direct. Furthermore, this offers decoupled stages of the algorithm, where the translation part of the algorithm is decoupled from the pre-processing part of the algorithm. In other words, the algorithm first takes as input a FAS XML and transforms it into a translatable XML and this is the pre-processing step, then the translatable XML is given as input to another part of the algorithm which formulates direct one-to-one translation with no complexity, just simple translation where each node written in XML format is translated into a container written in a docker-compose file using declarative language with all NICs to be attached to a container are pre-defined.

For example, let's look at the following figures, those figures represent the load balancer node and the web client node of Figure 6.18 Network topology. Let us see what is a non-translatable XML and a translatable one.

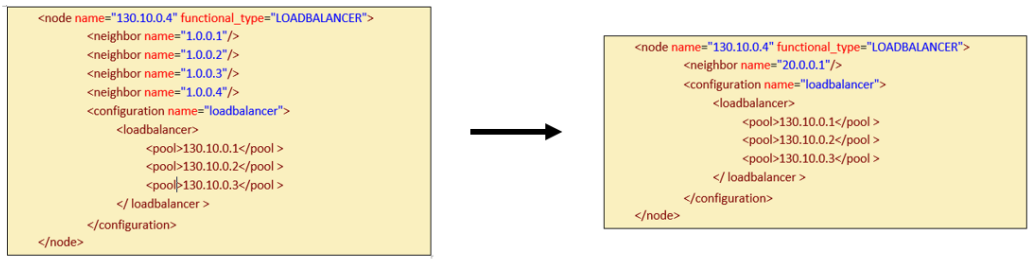

Figure 6.15: Translatable Load Balancer

In Figure 6.15, it is shown that the neighbours of the load balancer are all discarded, and a new neighbour name was assigned to it which is "20.0.0.1", this is the IP address of the NIC attached to the load balancer container connected on the same network as firewall1 as shown in Figure 6.18. After having the translatable load balancer node, it is possible now translate it directly to container written in docker-compose having IP addresses "130.10.0.4" and "20.0.0.1"

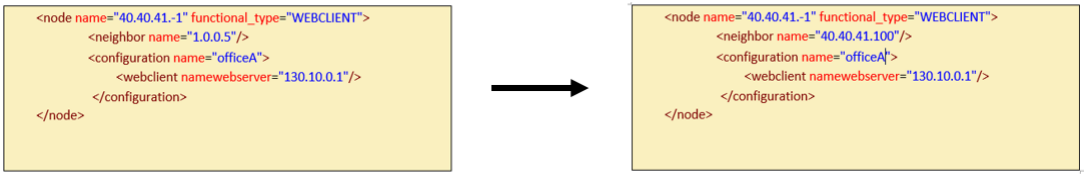

Figure 6.16: Translatable Web Client

In Figure 6.16, it is shown that the neighbor of the web client is discarded (since it is not important), and a new neighbor name was assigned to it which is "40.40.41.100", in the context of the web client and web servers this is the default gateway of the endpoint Or in case of a load balancer connected to web servers it is supposed that the load balancer node name is on same subnet as the web servers so there is no need to assign a "40.40.41.100", the neighbour name is kept as it is.

The Algorithm is mainly decomposed into two main java classes, "VnetwrokTranslator" and "IptablesVnetwork". First, the "VnetwrokTranslator" implements the logic behind understanding the input FAS and extracting knowledge that is useful in building the topology, then transforming the input XML file from a non-translatable file to a translatable file. After the "VnetwrokTranslator" outputs a translatable XML, it takes these XML nodes and inputs them to "IptablesVnetwork" which does the direct translation and produces the configuration files to build the virtual environment.

The logic implemented by "VnetwrokTranslator" is the pre-processing step to translation (i.e prepare the XML nodes for translation). The logic implemented is as follows (to be explained in the following sections):

Algorithm 1 Processing Algorithm

```
1: procedure First Scan(Assign neighbours to nodes)
2:     For each node in the Node list
3:         If node is web client OR web server AND node neighbour name is not
   on same sub-net then
4:         replace neighbour name with x.x.x.100
5:         Else If node is Forwarder or Load Balancer or NAT then
6:             For each neighbour name different from 20.0.x.x
7:                 if neighbour name is on same sub-net as node name then
8:                 remove neighbour name
9:                     else if neighbour name end with x.x.x.-1 then
10:                    replace x.x.x.-1 with x.x.x.100
11:                        else if neighbour name is firewall
12:                        else assign p2p addresses from 20.0.x.x pool of addresses
13: end procedure
```

The complexity of the algorithm comes from covering the different possibilities in a chain of if...else decisions, or yes...no choices. To demonstrate by example what the Algorithm in "VnetwrokTranslator" does. But note that before this step, there is a step of removing unwanted forwarders which is done at the VEREFOO level. The output FAS XML file of the VEREFOO framework will have already removed unwanted forwarders. As in Figure 6.17

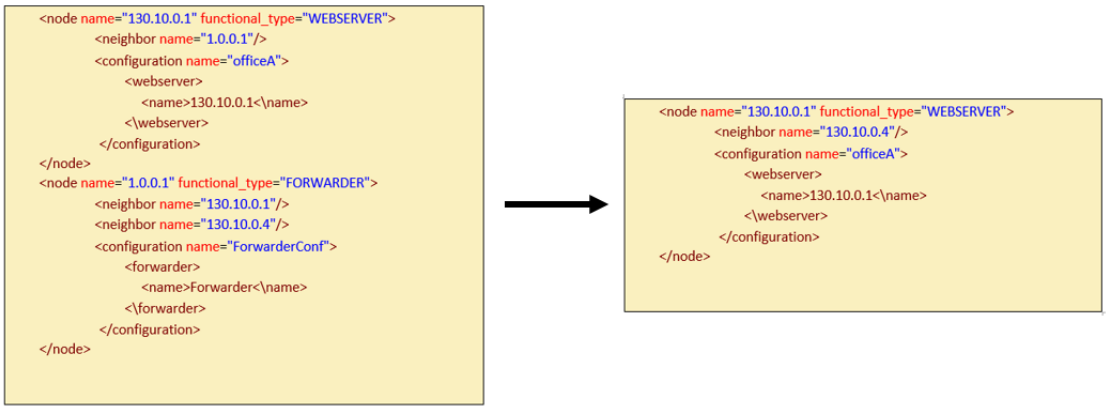

Figure 6.17: Remove Forwarder

and this is done for all the nodes, removing unwanted forwarders and replacing neighbor names accordingly.

For the scan, the algorithm starts analyzing the location of nodes with respect to each other and accordingly takes a decision whether to change a neighbor name,

add a point-to-point link from the 20.0.x.x pool of addresses, or remove a neighbor name. Taking into consideration that each neighbor's name will be a NIC address that will be attached to a container. Optimally the algorithm covers all simple scenarios, but for more complex scenarios, the algorithm needs to be improved. To make it simple the Scan in the algorithm is built to cover as many as possible scenarios. In other words, we list all possible decisions that the algorithm can face and ensure that there is an answer for every case. We will explicitly list all the cases that the algorithm takes into consideration according to the topology in Figure 6.18.

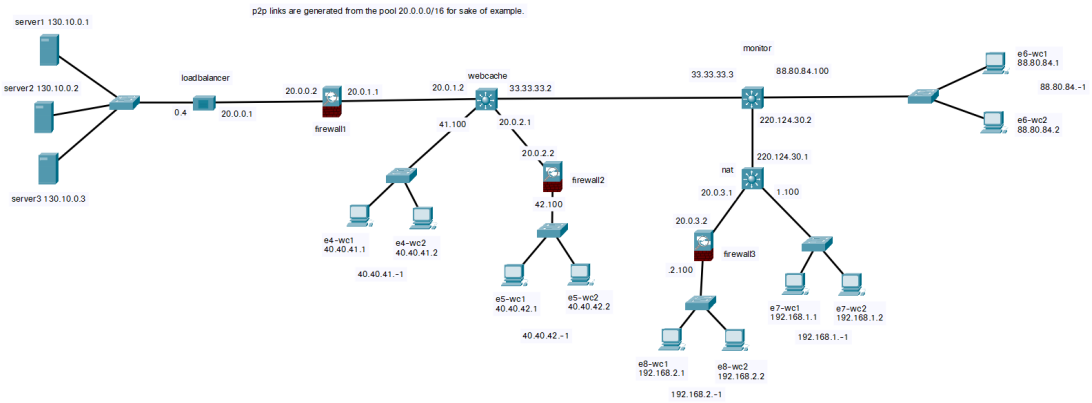

Figure 6.18: Firewall Network Topology

To simplify the understanding cases, the possible decisions are:

1. if node is endpoint (web server or web client), then possibilities are:

   - if neighbour is on same sub-net as the node name –> leave it as it is (it is the default gateway and this is the case when we have a load balancer and web servers).

   - if neighbour is on different sub-net –> change neighbour to sub-net of node (x.x.x.100) which will be default gateway.

2. if node is Forwarder, or load balancer then possibilities are:

   - if neighbour is on same sub-net as the node name –> remove neighbor from neighbor list (usually in load balancer only).

   - if neighbour is a firewall or forwarder –> replace neighbor name to an IP address from sub-net 20.0.x.x/24, and replace neighbor name of firewall to another IP address of same sub-net (to form p2p link)

   - if neighbour is a network of web clients (neighbor name end with x.x.x.-1) then replace neighbor name with x.x.x.100

3. if node is NAT, then:

   - For NAT, we expect that only one forwarder is connected to it and is the interface to the outer world. we just change the neighbor node IP to attach a NIC IP to it that is on same subnet as NAT node name.

4. if node is firewall, then possibilities are:

   - if neighbor is x.x.x.-1 –> change to x.x.x.100

   - ignore all other cases –> automatically managed by the logic above

And this is the logic implemented by the 2nd scan, it checks which condition is valid and acts accordingly, optimally it would cover all cases, which are limited and feasible in our example topology, but with more testing to be done cases would be modified or added.

Next, there is the "IptablesVnetwork" class, where the translation logic is developed, this translation is simple and doesn't need any complex logic. This class takes as input a list of nodes, which are expected to be translatable, and generates the different configuration files needed for the environment to be built. It generates the following scripts

1. Docker-compose: which contains the definition of containers and the respective networks, simply the translator performs the one-to-one translation. in other words, it scans a node and checks the Functional type, name of the node, and list of neighbor names of the node, Then it uses the functional type to determine the docker image with the required configuration files of the container, also it uses the name of the node and its neighbors to directly attach NICs to a container with there IP addresses.

2. NAT - Load Balancer Configuration files: If a node is of functional type NAT or Load Balancer the corresponding scripts to configure containers are generated

3. other scripts: the scripts startScript, endScript, and zebraStart Configuration files are static and don't change if the network topology changes. so they are generated the same always independent of the input list of nodes.

The main idea of the whole algorithm implementation in such methodology, is to reduce the dependency of the translation on the user input of IP, and increase dependency on functional types of nodes. So using only functional types to build the topology while adding our own IP addresses rather than depending on user to provide them from node names which may lead to errors. Obviously, the algorithm must use IP of endpoints as provide in FAS and they must not be ignored, or the

testing of firewalls will be wrong. However, for the forwarders and firewalls they are ignored.

Next section will display the results of testing the algorithm on FAS represented in chapter 4, its advantages and disadvantages, and the possibilities of such features with respect to the VEREFOO framework.

6.6.6 Advantages And Limitations

The algorithm was tested on Figure 6.18 and was able to correctly translate the FAS into a virtual environment for iptables firewall type only, for now, other parts are ignored. In fact, the algorithm is able to translate correctly FASs that are similar to this example, if the FAS XML file has a similar format respecting the limitation of the input XML then the translation will be correct. However, the algorithm is general enough to cover most possibilities of a FAS, but there is limitations in case an unexpected input topology format is provided. For example, two neighboring NATs is not expected to be in such FAS, or even two neighboring load balancer. As long as the input network topology is reasonable the translation is correct, in case of unexpected format the program will rise an exception.

The Limitations of such translation algorithm are:

- Algorithm can correctly translate certain FAS format, with limitations already mentioned
- Algorithm expects the input FAS to describe a reasonable network topology and not random. otherwise exception will be raised.

For the advantages part, we differentiate between advantages of the current algorithm and advantages of future work based on this algorithm.

- Algorithm is rather fast, as will be shown in testing section, the algorithm can translate 40 node topology with less than 200 ms.
- automates the process of testing in a virtual environment, rather than every time a new FAS is created and testing need to be done, the virtual environment can created dynamically allowing the test of the output firewall configurations in a virtual environment.
- Since there is a decoupled translation, where pre processing the input XML is done at one class, and the translation is done at another class, there is a possibility for the user to manually create a translatable FAS and directly input it to the 2nd part of the algorithm which is the translation only.

- this algorithm adds a feature to the framework, that is the ability the test the output of the framework (what ever it is) before deploying it to real conditions or environment. Rather than manually re-creating everything in the virtual environment every time testing is needed before deployment.

- flexible as other firewall type could be added in the future due to its flexible organization.

Finally, this algorithm is just a possibility of what could be done regarding dynamic testing in the VEREFOO framework using containers. It could always be made better and improved. In any case, the algorithm needs to be maintained and improved to ensure its robustness. The main takeaway from this algorithm is something to be built upon in the future, as there are two ways to go from here, either make the algorithm more complex, or work on the input FAS XML to contain more information about actual network topology, thus making the algorithm more straightforward. The next section will check the test cases used to test the algorithm.

6.6.7 Translator Testing

After implementation the algorithm, we have to ensure its robustness by creating a couple of test cases that cover most scenarios that the algorithm will possibly translate in the future. The generation of test cases will be based on white box testing, where we try to cover as much cases as possible. Then finally we will perform one performance test which consist of translating 40 nodes to assess the algorithm performance. The test cases will be generated based on combination of possible functional types neighboring each other. We have six functional types (web client-web server-load balancer-firewall-NAT-forwarder), so the test cases to be generated must cover the following scenarios (we also take into consideration some not possible cases):

1. Web Client: test cases must cover translation of web client having possible neighbor node that are forwarder, firewall, load balancer, and nat. it is not possible to have another web client or web server as neighbor to web client

2. Web Server: test cases must cover translation of web server having possible neighbor node that are forwarder, firewall, load balancer, and nat. it is not possible to have another web client or web server as neighbor to web client

3. Forwarder: test cases must cover translation of forwarder having possible neighbor node that are another forwarder, firewall, load balancer, nat, web client, and web server.

4. Firewall: test cases must cover translation of firewall having possible neighbor node that are forwarder, load balancer, nat, web client, and web server. Two firewalls connected to each other is not allowed.

5. Load Balancer: test cases must cover translation of load balancer having possible neighbor node that are forwarder, firewall, and web server. It is not possible to have neighbor that is web server, or load balancer connected to load balancer.

6. NAT: test cases must cover translation of NAT having possible neighbor node that are forwarder, firewall, web client, web server, and load balancer. It is not possible to have two nats connected to each other.

As mentioned before these scenarios must be covered by the generated test cases to ensure translation correctness with high confidence that the translation algorithm will not produce any errors. Six test cases were generated, each covering a certain number of nodes. Specifically, each test case cover certain number of the above mentioned scenarios. The algorithm was able to translate all of the six test cases correctly. Here the main one is only mentioned that cover most scenarios.

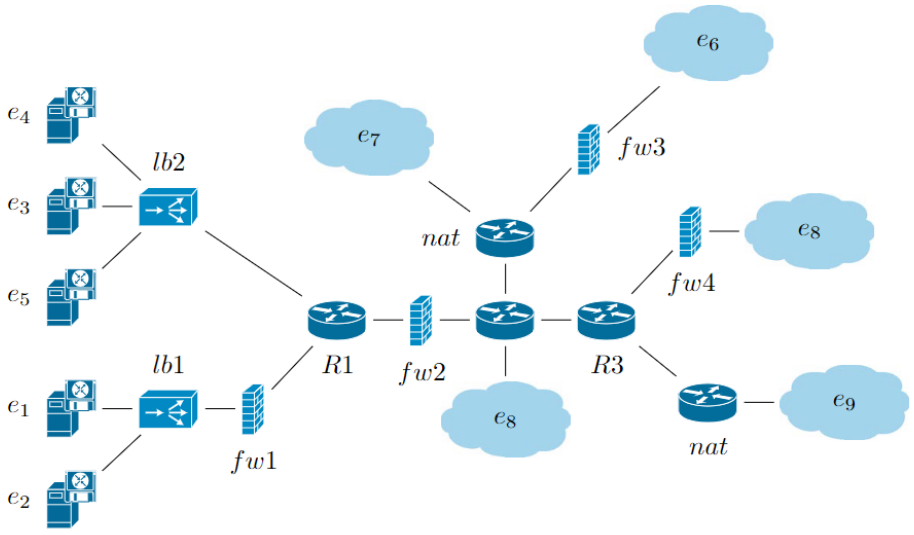

Figure 6.19: Test Case with 26 nodes to be translated

As is shown in Figure 6.19, the network in this test case is sufficiently ramified to cover most of the previous scenarios. When we input this topology as a FAS written in XML file to the translator, we immediately get all the configuration files required to run the virtual environment, but we only need to provide the

firewall configuration files in iptables directory in the produced directories. The other test cases also verified the correct functioning of the program. After testing, we are confident that any possible combination of network nodes similar to such test case the algorithm will correctly translate it. In terms of performance, the following graph indicate the efficiency of the algorithm in terms of execution time and number of nodes. Note that All the test instances have been solved on a machine with an Intel core i7-8557U CPU at 3.40 GHz, 20GB of RAM.

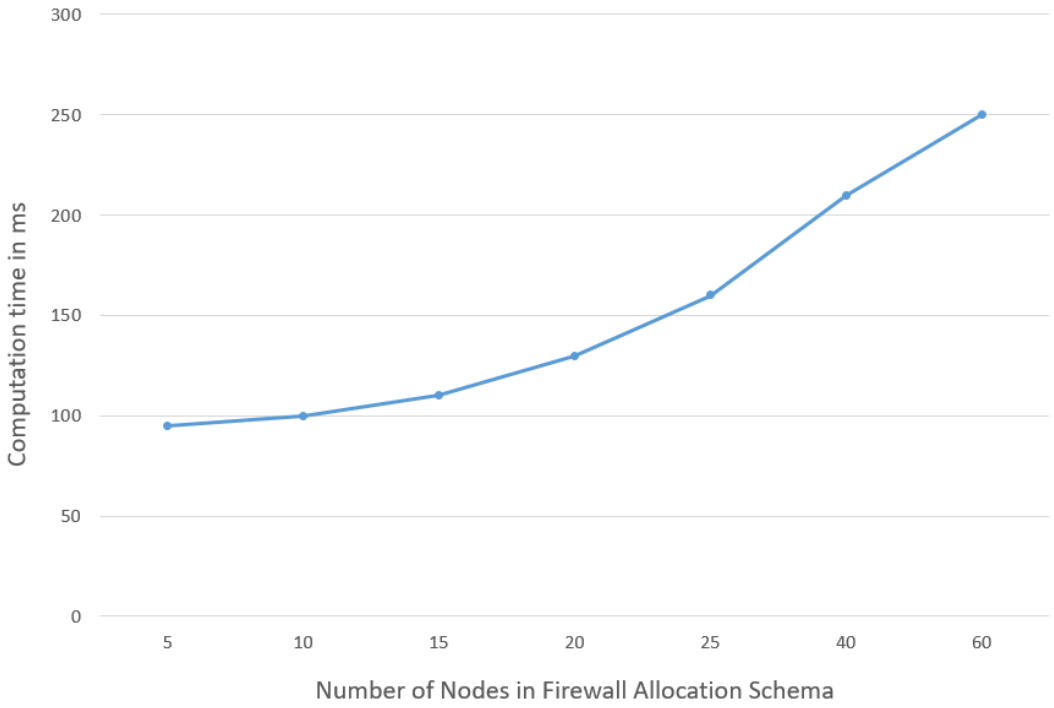

Figure 6.20: Scalability for increasing number of Nodes

The scalability of the approach has been evaluated varying the number of nodes in the input FAS XML file. As evident in the Graph 6.20, when increasing the number of nodes the computation time increases slightly, most of the time the algorithm will not translate more than 40 nodes due to limitations in virtualization of containers using a single machine, but in any case the algorithm have proven to work well for 60+ nodes for computation time less than 300 ms.

Chapter 7

Conclusions

7.1 Achieved Objectives

This research is intended to perform an analysis of the best method to verify and validate the VEREFOO framework in its different components and ensure there correctly functioning together. The focus was developing the right testing methodology to achieve confidence in the framework correct operation. Thanks to the preliminary work done studying in literature along with the results obtained in this thesis, the framework can now be assumed error-free in terms of low level translation of Medium Security Policy Language into firewall configuration written in specific vendor firewall type. The Verification and Validation that was done at different levels. First using software testing with white box approach generating the optimal test cases that ensures correct testing. Second level is testing in virtual environment, the work in this thesis was able to research the different tools and possibilities to develop the best virtual environment with the available resources. The work done in this thesis regarding developing virtual environment used for testing can be extended to different framework, where the idea behind building such environment is the same for every framework that need to be tested in a virtual environment with limited resources.

Moreover, after testing the low level translation of the framework, we focused in an incremental manner on increasing the automation in different aspects. For example, the pre-produced virtual environment where testing is done required minimal effort from the user to launch it. With a simple script all the environment is up and running and ready to start testing. Furthermore, to add a layer in automation for building the virtual environment we introduced dynamic routing which increases the efficiency of building more virtual environments for future testing. All of this work and results have accumulated to lead to the development of a translation algorithm. Each time a new network topology with new firewall

configurations needs to be tested, there is no need anymore for building the virtual environments from scratch and writing them. Simply from the Firewall Allocation Schema that VEREFOO outputs, we can immediately obtain a virtual environment that is automatically built by an algorithm that takes the Firewall Allocation Schema as an input.

Furthermore, we integrated the different parts of VEREFOO into a single process that is demonstrated to the user. Where the user can interact with the Full Standalone developed demo and understand how to interact with the VEREFOO framework at different levels, this allows users who are interested in using this framework in understanding how they could integrate to their own environment.

However, during this research, there were several limitations. The first one is the firewalls supported by the translation algorithm, currently only iptables firewall is supported. Another was the limitations of virtual environment and how much faithfully does the virtual environment replicates the real physical environment with minimal errors.

7.2 Future Work

As for future development related to this research, it is possible to choose different directions. One of the advantages of the methodology used to develop the virtual environment in its different aspects, is flexibility. The environment could be extended, manipulated, and improved in various ways to adapt to future needs. If new Security features were added to VEREFOO, the virtual environment could be extended also to test these features. Furthermore, in the future while integrating the framework with open sources orchestrator, the virtual environment could be handy in testing this integration. This thesis have gave the basis for building environment for testing that could be useful in future studies as the framework progresses.

Bibliography

[1] Riccardo Sisto, Fulvio Valenza, and Antonio Amoroso. «Automated Policy Enforcement in Software Defined Networking and Network Function Virtualization Environment». In: () (cit. on pp. 3, 4, 6, 27, 41).

[2] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. «Short paper: Automatic configuration for an optimal channel protection in virtualized networks». In: *Proceedings of the 2nd Workshop on Cyber-Security Arms Race*. 2020, pp. 25–30 (cit. on p. 3).

[3] Fulvio Valenza, Serena Spinoso, and Riccardo Sisto. «Formally specifying and checking policies and anomalies in service function chaining». In: *Journal of Network and Computer Applications* 146 (2019), p. 102419 (cit. on p. 3).

[4] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Towards a fully automated and optimized network security functions orchestration». In: *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE. 2019, pp. 1–7 (cit. on p. 4).

[5] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Automated optimal firewall orchestration and configuration in virtualized networks». In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2020, pp. 1–7 (cit. on pp. 4, 5, 59).

[6] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Introducing programmability and automation in the synthesis of virtual firewall rules». In: *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 473–478 (cit. on p. 4).

[7] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Serena Spinoso, Fulvio Valenza, and Jalolliddin Yusupov. «Improving the formal verification of reachability policies in virtualized networks». In: *IEEE Transactions on Network and Service Management* 18.1 (2020), pp. 713–728 (cit. on p. 4).

[8] Ignazio Pedone, Antonio Lioy, and Fulvio Valenza. «Towards an efficient management and orchestration framework for virtual network security functions». In: *Security and Communication Networks* 2019 (2019) (cit. on p. 4).

[9] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. «A novel approach for security function graph configuration and deployment». In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE. 2021, pp. 457–463 (cit. on p. 4).

[10] Guido Marchetto, Riccardo Sisto, Fulvio Valenza, Jalolliddin Yusupov, and Adlen Ksentini. «A formal approach to verify connectivity and optimize VNF placement in industrial networks». In: *IEEE Transactions on Industrial Informatics* 17.2 (2020), pp. 1515–1525 (cit. on p. 4).

[11] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Automated firewall configuration in virtual networks». In: *IEEE Transactions on Dependable and Secure Computing* (2022) (cit. on pp. 5, 61, 62).

[12] John Wack, Ken Cutler, and Jamie Pole. *Guidelines on firewalls and firewall policy*. Tech. rep. BOOZ-ALLEN and HAMILTON INC MCLEAN VA, 2002 (cit. on pp. 8, 34).

[13] Docker Documentation. *Introduction to Docker*. URL: `https://docs.docker.com/reference/` (cit. on pp. 11, 45, 91).

[14] oracle. *oracle*. URL: `https://www.virtualbox.org/manual/ch08.html` (cit. on p. 17).

[15] openstack. *openstack*. URL: `https://docs.openstack.org/openstack-ansible/latest/` (cit. on p. 18).

[16] Gerry Zaugg. «Firewall testing». In: *ETH Zurich* (2005) (cit. on pp. 27, 30, 48).

[17] Ehab S Al-Shaer and Hazem H Hamed. «Discovery of policy anomalies in distributed firewalls». In: *Ieee Infocom 2004*. Vol. 4. IEEE. 2004, pp. 2605–2616 (cit. on p. 28).

[18] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005 (cit. on p. 29).

[19] Polycube Documentation. *Introduction to Polycube*. URL: `https://polycube-network.readthedocs.io/en/latest/intro.html` (cit. on p. 41).

[20] Tugkan Tuglular. «Test case generation for firewall implementation testing using software testing techniques». In: *Proceedings of the International Conference on Security of Inform. and Networks*. 2008, pp. 196–203 (cit. on pp. 47, 48).

[21] Quagga-Routing. *Quagga-Routing*. URL: `https://www.admin-magazine.com/Articles/Routing-with-Quagga` (cit. on p. 52).

[22] rip-cisco. *rip-cisco*. URL: `https://www.cisco.com/c/en/us/td/docs/ios/iproute_rip/command/reference/irr_book/irr_rip.html` (cit. on p. 55).

[23] Scalingo. *Scalingo*. URL: `https://scalingo.com/blog/iptables` (cit. on p. 67).

Appendix A

Open Issues

In this appendix will be explained all the techniques used to set up the environment for replicate the tests. In addition, we will specify some open issues that could be improved in the framework.

A.1 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 virtual machine having:

- Ubuntu 20.04.1 LTS as 64-bit operating system
- Hard disk of 30GB
- Intel Core i7-8550 as processor.
- RAM installed equal to 8 GB.

The virtual machine used for development is vritualized using virtual box. In terms of software, in order to run the virtual environment on the virtual machine, it is required to install docker and docker compose. In fact, there is three setups for running the virtual environment to do testing:

1. Running the standard virtual environment (https://gitlab.com/negroup/thesis/2022-hobballah/-/tree/vnetwork): In this case, we run the virtual environment that is used to test the low level translations from medium level policies to low level configuration files. For this case we use the previously mentioned configurations (Ubuntu virtual machine), in addition to that we install on the virtual machine Docker and Docker compose (see [13].). Then after setting up this environment all that is required to do is to run ./startScript.sh and automatically the environment will be up and running.

2. Full Demo (https://gitlab.com/negroup/thesis/2022-hobballah/-/tree/fullDemo): For the demo that integrates VEREFOO as a single process, the set up environment was also done on Ubuntu following the previous specification. However, in this case more resources are needed to run different aspects of the demo (VEREFOO framework - virtual environment - GUI ...). To correctly run the demo we require the following to be installed:

   (a) Docker Engine and Docker Compose

   (b) java openjdk-1.8 (for framework launching)

   (c) node.js (for GUI operation)

   (d) curl and pv packages are required

   (e) make sure that you download z3-glibc library from z3 library newer versions after 4.8.15 may have some problems with the framework. After download it, extract it and then copy the z3 directory to /home in the machine (this is important for correct execution of the framework).

3. Translation Algorithm Operation : To setup the environment for using the translator, we could use the virtual machine already specified or we could use windows operating system with Docker Desktop that has a Dash Board useful to see all containers up and running. All that is need to be done is input Firewall Allocation Schema XML file and the translator will generate directory call "vnetwork_" where all configuration files required to run the virtual environment are present. Note that in order to correctly launch the environment it is required to manually copy the firewall configuration file into the directory "./vnetwork_/FirewallConfig/iptables/" and the format must be "iptablesFirewall_1_1".

The initial virtual environment specified in Chapter 2 and the Full Demo can be run only in Ubuntu, Since of the firewall types OpenvSwitch and BPF firewall types that require Linux kernel headers for containers to run. While iptables can be run cross platform in Ubuntu Linux Or Windows OS.

NOTE: When cloning the virtual environment repositories, please give execution permissions in case of errors. (does not always happen)

A.2 Open Issues

We discuss some of issues that could be added in the framework in future work. It is worth noting that the test bed that was carried in chapter two was done targeting the features produced by configuration files are limited to the input of the nfv element that the framework supports. The TCs focuses strictly on the

features supported by the input nfv element (8 parameters specified previously) only. So there is always possibility to add more features like:

1. Adding a protocol other than TCP/UDP, like ICMP since the input nfv element right now supports only TCP-UDP-ANY values, and doesn't have ICMP support. and this will be more evident when testing in the virtual environment where the PING test will always work and bypass the firewalls in case of white list mode. Since if the default of a firewall is Allow, we need to specifically deny the ICMP traffic by mentioning a deny policy which is not supported by VEREFOO at this moment. in case of black list mode the default action is deny will deny all traffic unless explicitly specified and ICMP will be dropped by default but in case we need to allow ICMP for any reason by a rule, there should be support for ICMP element.

2. In more specified firewall configurations there could be more specific cases where protocols other than TCP-UDP-ICMP be added (telnet -smpt ...), we leave this possiblity open for now.

3. The Firewall serializer class accepts as input only one firewall type at a time, so it isn't possible to have 2 firewall types in the same graph.

4. the translation code targets the **Forward Chain** the other chains follow the same configuration as the default action, this limitation is offered since there is no way to specify in input NFV element the target chain (FORWARD-INPUT-OUTPUT) to implement in it a certain policy in a firewall.

5. SSH to firewall is not configured, since for SSH to be enabled in black list mode the firewall should explicitly allow SSH traffic in the INPUT chain of the firewall which is not currently supported.

6. issue: other features that could be added are vendor specific, for example (dropping packets based on logs can be found in one firewall type but not in another)

7. Test cases of UDP: the policy that allows UDP packets through firewall in black listing mode, face a problem while tested in virtual environment. Since when sending a udp packet from an endpoint to server the udp packet is allowed to pass but the response is an ICMP packet from the server container that is dropped by the firewall, so it appears as if the test fails.

   Proposed Solution: use "nc" tool to create a client-server model and allow send of udp and receive of udp packets. EX:

   - "nc -u -l -p 1000 -s 30.0.5.1" -> server listening in udp mode on port 1000 at address 30.0.5.1.

- "nc -u 30.0.5.1 1000" -> client. connect to server 30.0.5.1 at port 1000. The list isn't extensive and additional stuff could be found.