Increasing Data Capturing Abilities of HoneyNets: a Proof of Concept

Diogo Miguel Marcos Ribeiro
Master in Network and Information Systems Engineering
Department of Computer Science
2023

Academic Supervisor
João Paulo da Silva Machado Garcia Vilela, Assistant Professor, FCUP

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

MSc

Sworn Statement

I, Diogo Miguel Marcos Ribeiro, enrolled in the Master Degree Network and Information Systems Engineering at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this dissertation reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this dissertation, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This dissertation does not include any content whose reproduction is protected by copyright laws.

I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

Diogo Miguel Marcos Ribeiro

07/03/2023

Abstract

As a technology, HoneyNets have been in use for some time now, with their development starting almost two decades ago. This tool was created with data gathering in mind, specifically, data about attacks launched by opponents against our legitimate systems.

There are several options one can take in an attempt of increasing the quality of data this tool captures, this can be done by increasing the quality of emulation used by the HoneyPots, having more attractive content in those HoneyPots or increasing the covertness of the entire HoneyNet.

In this dissertation we explore the third option. To do this we develop and implement an HoneyNet architecture with an active defence method, Moving Target Defence (MTD), that dynamically changes properties and configurations of the networks' hosts. With this implementation we are increasing the general covertness of our entire system by reducing its attack surface, as the attackers are unable to locate and focus on specific targets. This can help us decrease attacks with low information quality, like spam attacks or bot attacks, while still looking like an attractive target to attackers. For this implementation we make use of technologies related to Software Defined Networks (SDNs) and virtualization.

The contributions we aim to achieve with this dissertation include a survey on HoneyNet technology and SDNs and the development of an HoneyNet architecture that accomodates SDNs and MTD with high quality of virtualization, in our case, containerization.

Our results show a promising possibility of increased network covertness, while being able to maintain our HoneyNets primary objective, data collection. We implemented our solution as a proof of concept, using Internet Protocol (IP) address mutation as the main active defence.

Resumo

Como tecnologia, as HoneyNets estão em uso há já algum tempo, com o seu desenvolvimento a começar há quase duas décadas. Esta ferramenta foi criada com a recolha de dados em mente, especificamente, dados sobre ataques lançados por adversários contra os nossos sistemas legítimos.

Há várias opções que se podem tomar numa tentativa de aumentar a qualidade dos dados que esta ferramenta captura, isto pode ser feito aumentando a qualidade da emulação utilizada pelas HoneyPots, tendo um conteúdo mais atractivo nessas HoneyPots ou aumentando a dissimulação de toda a HoneyNet.

Nesta dissertação, exploramos a terceira opção. Para tal, desenvolvemos e implementamos uma arquitectura para uma HoneyNet com um método de defesa activa, Moving Target Defence (MTD), que altera dinâmicamente as propriedades e configurações dos anfitriões das redes. Com esta implementação estamos a aumentar a dissimulação geral de todo o nosso sistema, reduzindo a sua superfície de ataque, uma vez que os atacantes são incapazes de localizar e concentrar-se em alvos específicos. Isto pode ajudar-nos a diminuir ataques com baixa qualidade de informação, como ataques de spam ou ataques de bots, continuando a parecer um alvo atractivo para os atacantes. Para esta implementação, fazemos uso de tecnologias relacionadas com Redes Definidas por Software (SDNs) e virtualização.

As contribuições que pretendemos alcançar com esta dissertação incluem um estudo sobre a tecnologia HoneyNet e SDNs e o desenvolvimento de uma arquitectura HoneyNet que acomoda SDNs e MTD com alta qualidade de virtualização, no nosso caso, a containerization.

Os nossos resultados mostram uma possibilidade promissora de aumentar a dissimulação da rede, sendo ao mesmo tempo capaz de manter o objectivo principal da nossa HoneyNet, a recolha de dados. Implementamos a nossa solução como prova de conceito, utilizando a mutação de endereços Internet Protocol (IP) como principal defesa activa.

Agradecimentos

Obrigado ao André Duarte por toda a paciência e apoio ao longo do desenvolvimento desta dissertação. Um obrigado também ao professor João Vilela por todo o apoio e ajuda.

Dedico esta tese aos meus pais que me deram as possibilidades de o realizar. Um obrigado à minha mãe por sempre me ter incentivado a realizar este curso e por sempre me apoiar, e ao meu pai por despertar em mim um gosto por computadores. Obrigado também à minha irmã por estar sempre lá.

Finalmente, dedico também esta tese ao ano 2016/2017 de CC&Redes, SOMOS NÓS! Sem todos vocês não teria sido possível.

Contents

Abstract **i**

Resumo **iii**

Agradecimentos **v**

Contents **x**

List of Tables **xi**

List of Figures **xiv**

Listings **xv**

Acronyms **xvii**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Definition 2

1.3 Contributions 2

1.4 Document Outline 2

2 Background **3**

2.1 HoneyPots 3

2.1.1 Types of HoneyPots 3

2.1.2 Interaction Level 4

2.2 HoneyNet 5
2.2.1 HoneyNet Requirements 5
2.2.2 Data Control 6
2.2.3 Data Capture 6
2.2.4 Data Collection 7
2.3 Generational HoneyNet Architectures 7
2.3.1 Generational HoneyNets: Gen I 7
2.3.2 Generational HoneyNets: Gen II 8
2.4 Virtual Architectures 9
2.4.1 Self-Contained Architecture 10
2.5 Software Defined Networks 10

3 State of the Art 13

3.1 Moving Target Defense using Software Defined Networking 13
3.1.1 MTD Implementations Reviewed 14
3.1.2 OF-RHM Architecture and Protocol Walkthrough and Review 15
3.2 Virtualization for HoneyNets 18
3.2.1 Transparent Virtual Environments 18
3.2.2 Linux Containers as HoneyPots 18
3.2.3 Virtual Machine Detection 19
3.2.4 Container Detection 20
3.3 Summary 21

4 Preliminary Research 23

4.1 T-Pot - Technical Walk-Through 23
4.1.1 Services 24
4.1.2 Distributed Deployment 25
4.2 Experiment Setup 25
4.3 Experiment Results 25

4.3.1 Suricata Results 26
4.3.2 HoneyPots Results and Analysis 27

5 Implementation **31**
5.1 Architecture 31
5.2 Software Defined Networking 33
5.2.1 SDN Controllers 33
5.2.2 Controllers Comparison 36
5.3 Network Emulation Software 38
5.3.1 Mininet 38
5.3.2 Containernet 39
5.4 Open vSwitch 40
5.5 OpenFlow 41
5.6 Implementation 42
5.6.1 Network Topology 42
5.6.2 Ryu Script 43
5.6.3 STP Implementation 47
5.7 Summary 47

6 Results and Analysis **49**
6.1 Initial Tests 49
6.1.1 Reachability Test 50
6.1.2 iperf test 52
6.2 Link Failure 52
6.2.1 Link Failure with STP implementation 54
6.2.2 3 Switches Configuration 55
6.3 Summary 57

7 Conclusion **59**

7.1 Future Work 59

Bibliography **61**

List of Tables

5.1 Controller Comparison 37

List of Figures

2.1 Gen I Architecture example 8

2.2 Gen II Architecture example 9

3.1 APOD 14

3.2 DYNAT 15

3.3 OF-RHM architecture 16

3.4 Communication by DNS, as seen in [26] 17

3.5 Communication by IP, as seen in [26] 17

3.6 interaction capabilities vs system resource expense 19

3.7 Results of CPU Information Test 20

3.8 Results Arithmetic Test 20

3.9 Namespace and Permissions test 21

4.1 T-Pot Architecture 24

4.2 Suricata Number of Events 26

4.3 Suricata Events Histogram 26

4.4 Suricata Destination Ports Histogram 26

4.5 Suricata Detection of a Network Scan Histogram 27

4.6 Suricata Alert Signature - Top 10 27

4.7 Top 5 HoneyPot Attacks 28

4.8 DDOSPot Total Attacks and Unique SRC IPs 28

4.9 Attacker Source IP Reputation 28

5.1 In-depth Architecture 32

5.2 RYU Framework Architecture 36

5.3 OpenvSwitch Architecture 41

5.4 Initial Topology 43

6.1 Links and nodes 49

6.2 Reachability Test Fail 50

6.3 Packets of Reachability Test Failure 50

6.4 Address Translation List 50

6.5 Packets of Reachability Test Success 51

6.6 Internal Interface Packets 51

6.7 External Interface Packets 51

6.8 iperf TCP Test Results 52

6.9 iperf TCP Test Packets 52

6.10 iperf UDP Test Results 53

6.11 iperf UDP Test Packets 53

6.12 Link Failure Example 53

6.13 Link Failure Teste Hosts and Links 54

6.14 Redundant Link Failure Example 54

6.15 Redundant Link Failure Packets 54

6.16 STP Protocol Packets 55

6.17 STP Protocol Switches Joining 56

6.18 STP Protocol Ports is Block State 56

6.19 STP Protocol Ports is Learn State 56

6.21 STP Protocol ARP Packets 56

6.20 STP Protocol Ports is Final State 57

6.22 STP Protocol Ping Packets 57

6.23 STP Protocol Ping Test Successfull 57

Listings

3.1 OF-RHM Algorithm, as seen in [26] 17
5.1 Global Variables 44
5.2 updateAddrs 45
5.3 swicth features handler function 45
5.4 packet in handler header 46
5.5 packet in handler header 46

Acronyms

API	Application Programming Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPS	Intrusion Prevention System
IoT	Internet of Things
LAN	Local Area Network
MTD	Moving Target Defence
NASR	Network Address Space Randomization
NFV	Network Function Virtualization
NIDS	Network Intrusion Detection System
NIPS	Network Intrusion Prevention System
NTP	Network Time Protocol
ODL	Open Day Light
OF-RHM	Openflow Random Host Mutation
ONF	Open Networking Foundation
ONOS	Open Networking Operating System
OS	Operating System
OSI	Open Systems Interconnection
OSINT	Open Source Intelligence
OVS	Open vSwitch
SND	Software Defined Network
SSDP	Simple Service Discovery Protocol
TCP	Transmission Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol
VM	Virtual Machine

Chapter 1

Introduction

HoneyNets are a security tool used to obtain information and data about the methods and attacks used by opponents, its use as an information-gathering tool dates back decades to when the Internet started to be widely used. Defensive operations in information security are mainly about securing our assets and making sure an attacker does not have access to them or does not know they exist. HoneyNets brought a change to the defence paradigm, we can now go on the offensive and attract hackers with decoy systems specifically designed to gather as much information as possible. Due to the decoy nature of this tool, it is imperative to make sure an attacker believes he is dealing with a real system so that we can capture as much information as possible.

This thesis was done under the supervision and in conjunction with Ubiwhere, an IT company with a focus in Smart Cities and Telecom.

1.1 Motivation

We started this work by studying the technologies related to HoneyNets and data gathering in general; we quickly realised, however, that since HoneyNets are a tool whose conception happened some time ago, some of the technology applied to them is either outdated or not in use anymore. New concepts and technologies had since appeared that completely changed the paradigm that existed when HoneyNets were first implemented; motivation for this work came from applying some of these new concepts to HoneyNets to increase their data capturing and gathering capabilities. While researching state of the art, we quickly found a gap in the integration of HoneyNets with SDNs, for example.

1.2 Problem Definition

As we further explore in the chapter "Preliminary Research", there currently is a wide use of spam and automated attacks against targets on the open Internet. Therefore, HoneyNets, as a tool focused on capturing attack data, need a way to improve the quality of the data they are capturing.

To do this, we attempt to block the automated attacks from reaching our HoneyNet, allowing only higher-skilled attacks to reach it. Through doing this, we expect the information captured to have higher levels of quality, from which we can learn more to protect our own systems better.

1.3 Contributions

The contributions we aim to achieve in this thesis follow:

- A survey on the state of the art of HoneyNet technology;
- A survey on the state of the art concerning SDN's, its controllers and adjacent technologies;
- The development of an architecture which aims to improve HoneyNets covertness through the implementation of a MTD technology;

1.4 Document Outline

This document is organized the following way. Chapter 2 is dedicated to providing a solid background on the main topics of this thesis, we will approach the topics of HoneyPots, HoneyNets and a brief introduction to Software Defined Networks. In chapter 3 we will present the State of the Art on topics related to our work, we analyze it and show our findings, along with the gaps that we could fill with our work. This chapter approaches the topics of Moving Target Defence and Virtualization, topics that are not directly connected with HoneyNets, but can help us improve our HoneyNet performance indirectly. Chapter 4 is dedicated to presenting our initial analysis into this topic, we describe the experiment done and its results. In this chapter we demonstrated our initial hypothesis that automated and spam attacks are the main source of attack information collected by HoneyNets. Chapter 5 is dedicated to our solution of the problem presented in the previous chapter, we start by presenting our architecture, followed by an analysis of the technologies used and their alternatives. We follow up by describing our implementation, showing code snippets as support, here we present arguments to sustain our methods. Chapter 6 is where we present the results of our architecture in action, we show how it works and its peculiarities. Finally, in chapter 6 we conclude by doing an introspection of the work done, summarizing everything and discussing the possibilities of future work in this topic.

Chapter 2

Background

In this chapter we will provide some initial context for the work done in this dissertation. We provide definitions and standards that are critical to understand when discussing the topics of HoneyNets and computer security. We also provide an introduction to the topic of Software Defined Networking, which will be explored in higher detail further ahead in this document.

2.1 HoneyPots

Honeypots are one of the components of HoneyNets, this is what will essentially "lure" in the attackers, allowing us to perform the data collection we need for further analysis. There are several definitions of what a HoneyPot is, some experts consider it a tool of deception, while others think of it as a tool to lure in attackers or simply as yet another intrusion detection tool.

The most accepted definition is that a HoneyPot is "a security resource whose value lies in being probed, attacked, or compromised." [37], with this broader definition, we assume that when we design a HoneyPot, we expect that the system will be probed, attacked and potentially exploited. From this definition, we can see that HoneyPots can take on many manifestations, unlike other security tools that are designed to address specific problems. HoneyPots are not limited to solving a single specific problem, they are a highly flexible tool that can be used and applied to a wide range of situations, this is why the most common definitions seem so vague at first. HoneyPots have a variety of applications, these range from being used to deter attacks, detect attacks or capture and analyze automated attacks [37].

2.1.1 Types of HoneyPots

When deploying this type of system, we must first establish what we want it to do; only after that we can choose our configuration. As we previously said, HoneyPots are a highly flexible tool and can take on many different roles with different objectives in our architectures. Depending on the use we want our HoneyPot to have, we can choose between two configurations, Production and Research. In summary, Production HoneyPots protect an organization, while Research HoneyPots are used to learn about the methods and behaviours of attackers. The concept of these categories originally comes from Martin Roesch, developer of Snort [37].

- Production HoneyPots are what we typically think of when discussing HoneyPots; they can directly add value to our organization and help us mitigate risk. These systems are implemented because they can help us secure our environment by, for example, preemptively detecting attacks. Production HoneyPots are a lot simpler than their counterparts; this simplicity, however, is a double-edged sword; on the one hand, they are a lot easier to build and deploy, on the other hand, they will have less functionality [41].

  This lower degree of functionality implies that these systems have higher simplicity, with this simplicity also comes less risk; it is harder to use a Production HoneyPot to harm other systems. However, this lower functionality also means that it is a lot harder to obtain information about who is attacking us, their motives and practices.

- Research Honeypots are, in a few words, designed to gain information about the black hat community, which means that these systems will not directly add value to an organization. Their primary objective is to help us get a clearer view of whom we are dealing with, what tools they use, and their methods [36].

  We can think of these systems as a counterintelligence measure in that they, unlike Production HoneyPots, will help us get information on the "bad guys". Using this information, we can fortify our systems and protect ourselves against the threats we observed in this controlled environment.

  Research HoneyPots differ significantly from Production HoneyPots because they usually require a higher level of interaction. Since we are trying to lure hackers into our HoneyPot, we want them to believe they are dealing with a real production system; therefore, they must be able to interact with it. To do this, we need to provide them with real operating systems and applications which they must be able to attack [41].

  With this increased functionality also comes increased risk, Research Honeypots are far more complex and require higher maintenance time. Since they allow more freedom to the attacker, there is also a risk of losing control of the system, allowing the hacker to freely attack any target on the Internet through the HoneyPot.

2.1.2 Interaction Level

After deciding what type of HoneyPot we require, we need to establish how we want it to operate. HoneyPots can be separated by the level of interaction they provide, Lance Spitzner classified three levels of interaction: low, medium and high. His book describes the differences between the interaction levels, each having several advantages and disadvantages [37].

Generally speaking, the higher levels of interaction require higher levels of maintenance and more complexity when configuring them.

- Low Interaction HoneyPots: typically only emulate operating systems and services, so the attacker will be limited in its actions; this limit is defined by the quality of the emulation used. This type of HoneyPot normally has a minimal amount of commands available for the attacker to use, meaning that there is little to no risk that a successful attack (through the HoneyPot) will damage existing production systems.

- Medium Interaction: This HoneyPot sits between high interaction and low interaction. They are more complex than Low Interaction but do not provide any real operating system or services (like in the high interaction variant) for the attacker to interact with; however, the emulation quality will be higher, allowing the attacker to have more freedom of actions than in low interaction HoneyPots.

- High Interaction HoneyPots: These HoneyPots aim to obtain the maximum amount of information possible about the attacker and his methods, therefore, they will be the most complex type. This system will allow the attacker a much higher degree of freedom than the previous ones, allowing itself to be used, tampered it, and even damaged. To achieve this, high-interaction HoneyPots are usually a clone of an existing production system that is in use; remember that we need to make the attacker think he is dealing with a real system; therefore, the best way to achieve this is to use a clone of an "actual" system used in production. We can even go a step further and use a real operating system with real applications, which is implemented in real hardware, this implies that there will be no emulation [36].

2.2 HoneyNet

After understanding what HoneyPots are, we can move to HoneyNets. HoneyNets can be described succinctly as a network of HoneyPots, where all the network's components cooperate to capture information about attacks and exploits used against it. It is important to note that the term HoneyNet does not directly refer to any given specific tool or implementation; instead, we should think of HoneyNets as architectures which can take on many different manifestations according to their intended use [37].

2.2.1 HoneyNet Requirements

The HoneyNet Project, a group of individuals focused on HoneyNet development strategies, defined several definitions and requirements for HoneyNet's architecture and operation, which are as valid today as when they were defined [32].

As per the authors, to create the highly controlled environment needed for a HoneyNet the following requirements must be met:

- Data Control
- Data Capture
- Data Collection

2.2.2 Data Control

Data control defines the functionality that allows us to restrict malicious activity from taking over our HoneyNet, activity spreading outside means that Data Control failed. It is an attempt to minimize an attacker's risk of using our HoneyNet system to launch attacks against other non-HoneyNet systems. Outbound attacks must be limited in order to prevent the possible spread of malicious activity through the HoneyNet to our real production systems or even the Internet. However, we must perform this task in a way the attacker cannot detect; if our adversary realizes this, he will know that he is dealing with a decoy system [35].

Concerning Data Control, the group describes methods that help prevent an attacker from taking too much control of the HoneyNet and using it to harm other systems [32]. These methods include connection counting and Network Intrusion Preventing System (NIPS). With connection counting, we can limit the number of outbound connections a HoneyPot can initiate (connections to the Internet); this functionality was implemented with iptables. With NIPS, we can block or disable known attacks launched from the HoneyNet to the Internet; the group used snort-inline to implement this functionality.

2.2.3 Data Capture

The purpose of data capture is to gather data with the highest quality of information possible; that is, the capture of malicious activity must be done covertly so that the attacker does not suspect it. The group established some data points that are critical to obtain, user activity, firewall logs, network traffic (inbound and outbound) and system activity.

With regards to firewall logs, by implementing the IPTablesFirewall script, all outbound and inbound traffic is logged. To capture network traffic, the authors used a snort process to capture all IP traffic. To capture system activity, the authors chose to use the Sebek tool. Sebek is a hidden kernel module capable of logging attackers' activity. It does not store the information it gathers on the HoneyPot, it sends it via User Datagram Protocol (UDP) to a sniffing machine, such as the HoneyWall or a remote logging system [43].

2.2.4 Data Collection

Data collection applies mainly to distributed HoneyNet architectures. It is defined as the aggregation of all data captured by multiple HoneyNets. If we are using several distributed HoneyPots data collection also applies, since we typically aggregate the data in a single contained system where it can be logged and studied [37].

2.3 Generational HoneyNet Architectures

The Honeynet Project was one of the first groups to develop architectures for HoneyNets, as previously said; they developed guidelines that define this tool's necessary requirements and objectives. This group was responsible for the gen I and II architectures, which we will explore further ahead. The first generation attempted to create a system that is hard to detect and is used to collect various pieces of information about attacks and methods used by attackers.

The second generation evolved from the first generation, improving technologies and methodologies that can now be utilised to fulfil the requirements detailed above. With the second generation, we can now have a system that is easier to deploy and harder to detect [32].

Nevertheless, technologies concerning security are an ever-evolving topic, with new ways of exploiting systems constantly appearing, new security measures being implemented, and methodologies changing. Therefore, the architectures presented in this section are not suitable for deployment nowadays; they provide, however, a good ground plan for how to proceed with this work.

2.3.1 Generational HoneyNets: Gen I

On 10 May 2005, The HoneyNet Project published the paper "Know Your Enemy: Honeynets", describing one of the primary tools they used to capture information, the Gen I HoneyNet. This paper is an initial introduction to the concepts and issues of HoneyNets. The authors described the concept of a HoneyWall, a gateway device that separates the HoneyNet from the rest of the Internet. They also presented several requirements for this device: data control, data capture, data analysis and data collection.

The architecture of a Gen I HoneyNet, seen in image 2.1, is very straightforward; we create an isolated network that sits entirely behind a network access control device [4], like a firewall. This will funnel all traffic through this device. Any traffic that will then pass through this device must be logged. The positioning of this network as an isolated network is done to ensure that attackers cannot access anything that is not part of the HoneyNet, reducing the risk of our real production systems getting compromised [35].

In this paper, the authors also introduce the concept of a HoneyWall, a gateway device that separates the HoneyPots from the Internet. As described above, it essentially acts as a network access control device operating in layer 2 of the Open Systems Interconnection (OSI) model. The fact that this device operates in layer 2 makes it invisible to anyone interacting with the HoneyPots.

This architecture excels at capturing automated attacks or beginner attacks; however, little information can be gathered from them; therefore, its value is questionable. If we require a system that can help us gather information about advanced attacks, this generation is not a good choice as there are much better options, which we will discuss further.

Gen I HoneyNets were quickly found to have several limitations, a significant constraint is their ability to control and contain attackers after they gain access to the HoneyNet [32]. This generation was also substituted due to its inability to act as a dissimulated system for higher-skilled attackers. In other words, if we are dealing with an advanced attacker, he will be able to know that he is dealing with a HoneyNet. As Lance Spitzner showed [37], this is due to two reasons, with the first being the ease with which an attacker can fingerprint this system, specifically with signatures specific to data control. Second, this generation of HoneyNets provides little to no value in terms of information gathering because, in reality, they are not an attractive target to attackers.

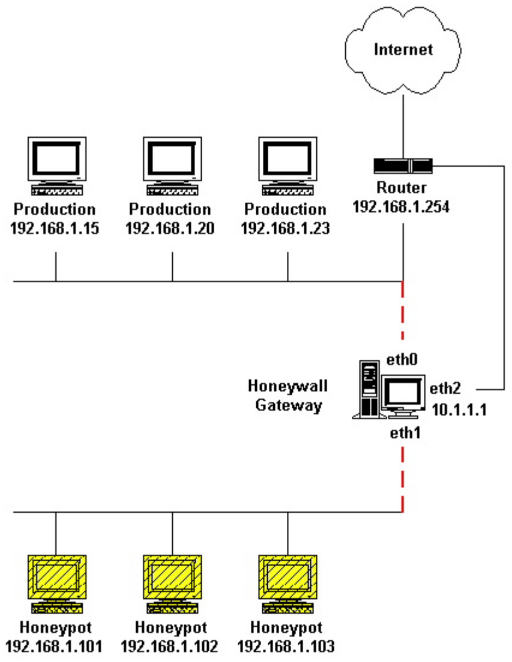


Figure 2.1: Gen I Architecture example, as seen in [4]

2.3.2 Generational HoneyNets: Gen II

On 12 May 2005, The HoneyNet Project published the paper "Know Your Enemy: GenII Honeynets", where they describe the second generation of HoneyNet technology. Generation II was developed mainly because Gen I HoneyNets were easily detected. Since Gen II HoneyNets are harder to detect, an attacker will remain in them for more time, allowing for more information to be collected. In this paper, the authors go into further detail on the HoneyWall concept, this device allows for smarter Data Control and Capture since both are implemented on the HoneyWall [43].

We can see an example of a gen II HoneyNet implementation in figure 2.2; we should note the gateway acting as a Honeywall, which is the main difference between Gen I and Gen II. Since all inbound and outbound traffic to the HoneyNet must pass through the gateway, it is the ideal place to implement our data control and capture mechanisms. Gen II HoneyNets offer improvements in two requirements: data control and data capture.

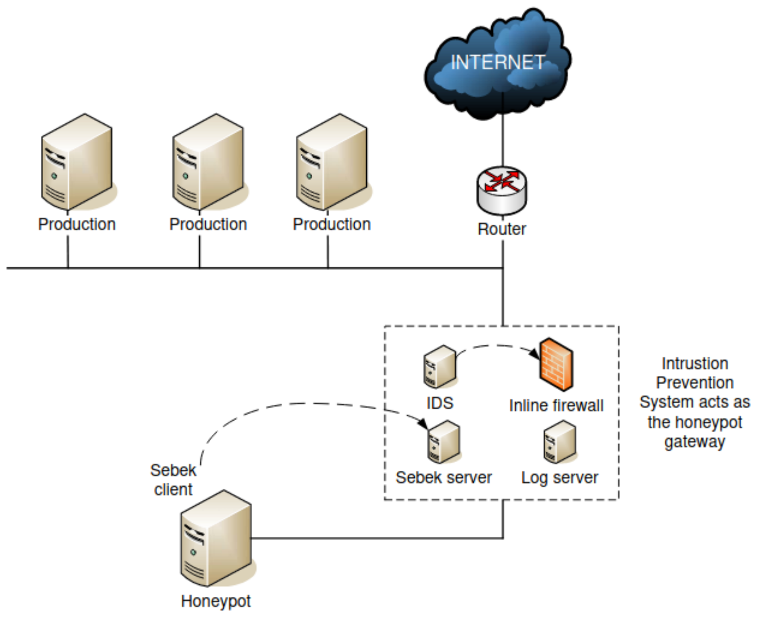


Figure 2.2: Gen II Architecture example, as seen in [43]

Data Control in gen II consists of a Network Intrusion Prevention System, which is essentially an inline firewall and a Network Intrusion Detection System (NIDS). Being implemented at layer two, the firewall does not change packets when they are processed, for example, decreasing the Time To Live (TTL); they are also invisible to an attacker.

2.4 Virtual Architectures

As the name implies, a virtual HoneyNet relies on an architecture that allows the deployment of the entire network on a single physical machine. To do this, we employ virtualization technologies, which allow us to run multiple HoneyPots and data collection tools on the same computer, sharing its resources and network connection [35].

As we will further explore in greater detail, virtualization dominates in today's implementations of HoneyPots and HoneyNets. With access to virtualization, we also get the option of virtualized network components, allowing for an entire network to be implemented inside a single contained system [39].

2.4.1 Self-Contained Architecture

As the name implies, the entire HoneyNet network will be hosted on a single machine in this architecture. This machine will run the host operating system along with data control and data capture tools. The host operating system will then resort to virtualization technology to run guest operating systems, which will host the HoneyPots.

This architecture provides the advantage of being very cost-efficient since there is no need to have multiple machines for our HoneyPots or specialized hardware for data control and data capture. In fact, since all the HoneyPots are running on the same system, data collection will be easier since there is no need to use encrypted tunnels to collect data; we can instead use logging technology on the virtual HoneyPots [22].

This architecture has some disadvantages when compared to physical ones. Since all the HoneyPots are running on the same system, if we encounter a problem with that system, then the entire HoneyNet will fail. System performance is another possible complication; if our machine does not have enough resources to deal with all HoneyPots and services, we will run into the risk of the attacker detecting the virtual HoneyNet environment.

2.5 Software Defined Networks

Software Defined Networking is a paradigm that uses software-based components to communicate with the underlying infrastructure and direct traffic or manage network policies, enabling an organization to program network control directly. In a traditional environment, a network component like a switch or router is only aware of the status of network devices adjacent to it. With SDN, we can centralize the "intelligence" (in the controller) and control everything. In addition, SDNs allow us to essentially decouple the forwarding plane from the data plane [16].

Software-defined networking enables a new way of controlling packet routing through a centralized server. It allows organizations to increase control over the network with greater speed and flexibility by passing the control over to a central authority and decoupling network tasks from vendor-specific software. Instead, administrators have more flexibility in choosing the networking equipment as long as it supports the protocol used to communicate with that hardware.

SDNs also allow administrators to easily customize their networking infrastructure and allocate resources where they are needed in real-time. Security is also improved with this technology; since we have visibility over the entire network, we have a better view of security threats. We can quickly and easily quarantine a specific compromised device so that it cannot infect the rest of the network.

Previously we said that SDNs allow us to decouple the forwarding plane from the control plane. To expand on this concept, we can think of the key difference between traditional networking and SDNs as the infrastructure. With SDNs, we have a software-based approach, while in traditional networking, we have a hardware-based approach. This software-based approach allows us to control the network through software; these modifications are, in turn, applied by the hardware. Essentially, we are giving instructions through software that the hardware will apply.

Additionally, as many of today's services and applications move to the cloud, SDNs become even more relevant. Cloud infrastructure is especially reliant on SDNs; without SDN operations in the cloud would become a lot harder to implement, if not impossible. This is due to SDN's ability to move workloads around a network quickly and its ability to scale up or down easily [16].

Chapter 3

State of the Art

HoneyNet-related technology has evolved a lot since its initial development nearly two decades ago; advances in architectures have brought us virtualized self-contained HoneyNets, while advances in virtualization have allowed us to move to fully containerized HoneyPots. In this chapter, we will discuss novel methods that are not directly related to HoneyNets but indirectly through the possible implementation of new defence mechanisms, with which we hope to increase our data-capturing abilities. Therefore, this chapter is dedicated to studying methods and techniques to increase our HoneyPot/Net performance.

During the development of this chapter, we noticed that there is a gap when it comes to HoneyNets and active defence methods. As we will explore in greater detail further ahead, it is of value to combine these two technologies in order to increase the potency of our HoneyNets. Therefore, in this chapter, we will introduce the active defence method known as Moving Target Defense, reviewing the implementation of such a method and sharing our thoughts on why this is a potentially beneficial technology to implement alongside HoneyNets [30]. We will then follow up with a discussion on the virtualization theme, an important one when discussing HoneyPots. Finally, comparisons will be shown on virtual machine vs container detection, demonstrating which best suits our requirement of being as covert as possible to fool a possible attacker.

3.1 Moving Target Defense using Software Defined Networking

Moving Target Defense (MTD) is a collection of techniques to improve computer network systems' security and resilience. It tries to achieve this by increasing the diversity of software and network paths [10].

MTD aims to increase protection against adversary reconnaissance efforts on static networks. By randomly changing values, network paths, and positioning in the network and system settings, we are creating a more challenging environment for the attacker to proliferate his exploits and laterally move inside the network [29].

MTD is classified as an active defence in the field of cybersecurity, meaning that, instead of other passive methods like firewalls or Intrusion Detection System (IDS), we are actively trying to prevent an intruder from scanning, gaining knowledge or penetrating our network.

Software Defined Networks are a technology that allows us to implement several moving target defences; due to their programmatic nature, we are given all the necessary tools to implement MTD strategies successfully. For example, Jafar Haadi et al. demonstrated the power of SDN when combined with MTD by developing a strategy that transparently implements an MTD method in SDN for frequently changing hosts' IP addresses [26].

3.1.1 MTD Implementations Reviewed

The creators of Openflow Random Host Mutation (OF-RHM) method describe a Moving Target Defense architecture implemented with OpenFlow that transparently mutates IP addresses. The authors noted that static network configurations greatly benefit the attacker, allowing him to quickly discover network targets and subsequently launch attacks against those targets.

Changing a host's IP address is a proactive moving target defence that helps hide network hosts and assets from internal and external scanners. Using the OpenFlow Protocol, a communications protocol used to access the forwarding plane of network devices, the authors leveraged the power of Software Defined Networking to implement their architecture transparently. A controller was used to assign each host a random virtual IP address that is translated to and from the actual IP of the host. Doing the translation this way assures that it will be entirely transparent for the end-hosts in our network [26].

The authors were motivated by the security issues with a static assignment of IP addresses, claiming that these configurations give a significant advantage to adversaries who attempt to remotely scan a network, such as scanning tools and worms. These tools usually work by sending probes to random IP addresses in the network to identify their targets; if any host responds to this probe, it can quickly be identified and attacked.

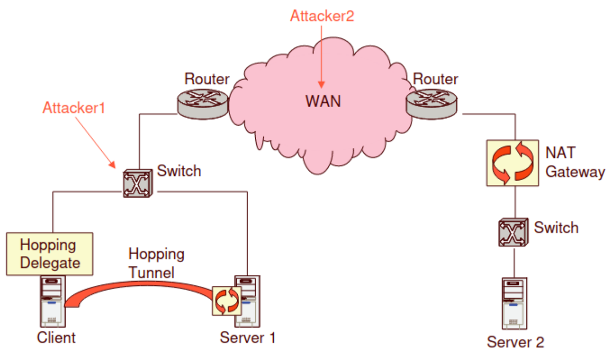


Figure 3.1: Applications That Participate in Their Own Defense scheme, as seen in [14]

Previous work on this topic was discussed, with the authors mentioning the APOD (Applications That Participate in Their Own Defense) scheme [14], represented in image 3.1, which relies on hopping tunnels based on address and port randomization; however, the authors claim that this scheme fails to be transparent, as both the client and the server need to cooperate with one another during the mutation process.

DyNAT, represented in figure 3.2, is another MTD implementation which provides transparent mutation by translating the IP addresses before packets enter our "core" or public network. This method is efficient in hiding the addresses from man-in-the-middle sniffing attacks; however, it does not work when the attacker relies on probe responses for discovering end hosts, as the hosts will respond to that probe giving away their identity [28].

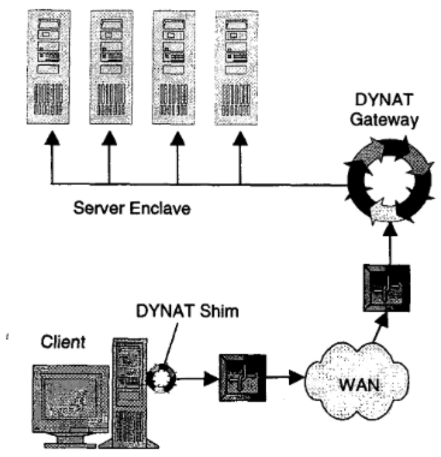


Figure 3.2: DYNAT Architecture, as seen in [28]

NASR (network address space randomization) is a Local Area Network (LAN) oriented scheme for address randomization based on Dynamic Host Configuration Protocol (DHCP) updates, making NASR not transparent; if an attacker has a connection with a host, he will see its session disrupted; this is because DHCP update changes are made to the end-host, which in turn results in the disruption of active connections during the transition itself. NASR also lacks unpredictability and speed in the address mutation; since we are using DHCP, we are limited to our LAN address space, with a minimum time for each mutation of 15 minutes [12].

3.1.2 OF-RHM Architecture and Protocol Walkthrough and Review

As we have noted, none of the previous approaches provides a transparent mechanism for IP translation that can defend against internal and external threats; with OF-RHM, the authors are exploiting the power of SDNs to implement IP translation with unpredictability and speed.

OF-RHM was implemented with Mininet, a network emulator, and a NOX controller, a type of SDN controller that is used for OpenFlow management. This architecture can be seen in figure 3.3. The controller is the central authority in IP translation management, installing flow rules in

Open Virtual Switches (OVS) and DNS responses, this controller will be executing the algorithm seen in listing 3.1, we point out the way the authors do the translation and install flow rules is reliant on DNS, something we would like to avoid. In addition, the authors deal with scalability issues by including several controllers, each responsible for a segment of the network [26].

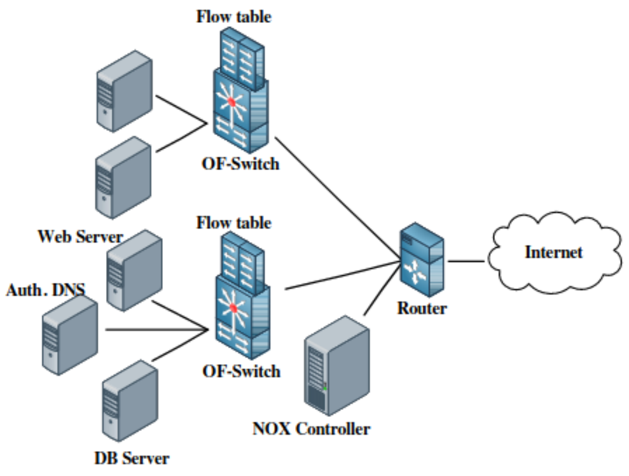


Figure 3.3: OF-RHM architecture, as seen in [26]

When the switch does not have a match for a received packet in its flow table (where we keep a record of flows which will then be used to perform packet lookup and forwarding decisions), it will encapsulate it and send it to the controller, who is responsible for processing it and determining the type of connection. The necessary flows are then installed in all switches in the path. We should note that each connection is associated with a unique flow because the rIP (real IP) to vIP (virtual IP) translation changes for each connection.

The authors define two ways to communicate with hosts, using their real IP address, as seen in figure 3.4, or hostname, as seen in figure 3.5. When we receive a DNS query to resolve a host's name, the controller updates the DNS response to replace the rIP with the vIP; the enquiring machine can then initiate a connection using the vIP of the destination. Switches that do not take a direct part in the rIP-vIP translation are only configured with flows to route the traffic based on vIP (no actions are installed in these "traffic-only" switches. Future packets are directly matched in the flow table, so there is no need to send them to the controller. The final vIP-rIP translation is applied to packets by the switches.

Communication by real IP is reserved for authorized users. When one of these users initiates a connection with a destination host using its rIP, the switch will, again, not be able to match it and will send it to the controller, who will authorize the request and install the necessary flows.

This implementation also involves using weights for weighted mutation; the controller counts the number of times a particular vIP has been used and scanned in a given time frame. This value affects the new mutations by using probabilities with weights. Therefore, an address that has already been scanned will have a smaller probability of being chosen again.

```
determine unused ranges
determine range-to-subnet assignments
for all packets p from OF-switches do
    if p is Type-A DNS response for host hi then
        set DNS addr to current vIP(hi), TTL=~0
    else if p is a TCP-SYN or UDP from hi to hj then
        if p.src is internal then
            install in flow in src OF-switch with
                action srcIP(p):=vIP(hi)
            install out flow in src OF-switch with
                action dstIP(p):=rIP(hi)
        end if
        if p.dst is rIP then
            if hi access to hj is authorized then
                install in and out flows in dst OF-switch
            end if
        else[p.dst is vIP]
            install in flow in dst OF-switch with
                action dstIP(p):=rIP(hj)
            install out flow in dst OF-switch with
                action srcIP(p):=vIP(hj)
        end if
    end if
    for all mutation of each host hi do
        set vIP(hi) to a new vIP
    end for
end for
```

Listing 3.1: OF-RHM Algorithm, as seen in [26]

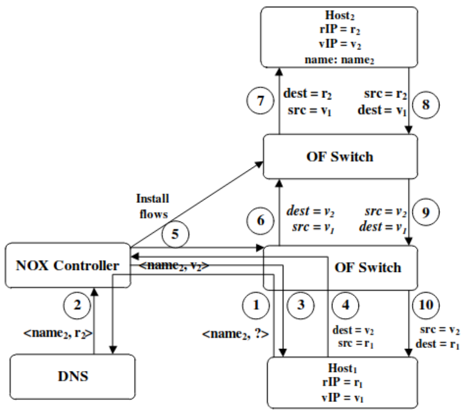


Figure 3.4: Communication by DNS, as seen in [26]

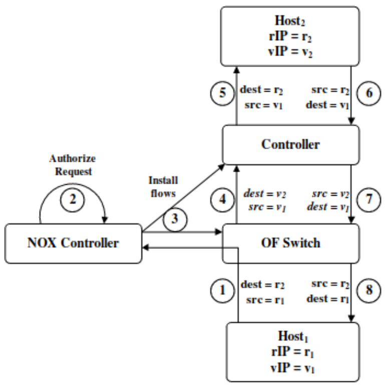


Figure 3.5: Communication by IP, as seen in [26]

3.2 Virtualization for HoneyNets

The authors of [27] analyze the feasibility of using Linux containers as a basis for HoneyPot implementations. There is an obvious comparison made between the use of virtual machines, containers and bare metal machines. Containers are a technology that has been replacing virtual machines in many areas due to their lower cost of operation and ease of scalability.

This paper makes comparisons in performance and security between these three technologies, presenting three key findings:

- The first finding confirms our initial assumption that virtual machines are easily fingerprinted. The authors particularly note the vulnerability of hardware-level environment detection like Central Processing Unit (CPU) clock sampling, reported CPU information and instruction execution time. In contrast, due to having a direct interface to the host kernel, a container returns a similar profile to bare metal systems.
- Investigations indicate containers are promising in defeating environment detection methods, which is attributed to containers' use of kernel namespaces for isolation [27].
- The authors claim that containers are vulnerable to methods of identification that exploit the tools used by containers, such as namespaces and permissions.

3.2.1 Transparent Virtual Environments

Attempts have been made to create a fully transparent virtual machine environment; the authors cite the examples of Cobra [40] and Ether [25], noticing that both fail to meet their goals. These implementations reveal their presence through hardware artefacts like inaccurate CPU semantics or timing tests. Garfinkel et al. even go as further as to claim that a fully transparent virtual machine is impossible to implement due to the nature of its workings [38].

As we explore further, virtual machines do not meet our transparency requirement; therefore, another method must be used to implement our honeypots. In addition, container technology, unlike Virtual Machines (VM), is much harder to detect and exploit.

3.2.2 Linux Containers as HoneyPots

Linux containers, are a product of joining together two Linux technologies, namespaces and cgroups, that provide isolation and containment for processes. This technology allows us to run multiple Linux systems on the same host, with them only sharing the system kernel.

Thanks to containers' reduced overhead, a new intersection between honeypot interaction level and system resource expense is possible. This allows us to implement high-interaction honeypots in low-power devices, like Internet of Things (IoT) devices.

This technology, however, carries different security concerns than virtual machines. Since the code inside a container is being run on the same kernel as the host machines, we need to put mechanisms in place that attempt to minimize the impact a container may have on the performance of the host machine.

Containers are directly managed by the system kernel, meaning they do not duplicate those kernel's functions. An isolated experiment was run in which code executing on LXC only added 6 per cent execution time, whereas it added 41 per cent when executed in KVM [27].

Honeypots executing within containers pose different security concerns than honeypots in virtual environments, which is a downside of sharing the host's OS kernel. This can also present an issue when using certain types of container implementations [31]. Docker, for example, requires root privileges on the host machine to execute, which can increase exposure to malware, even though it separates the OS from the services running on it.

Containerization has shown to be a good option to implement HoneyPots, allowing for facilitated data collection and increased security. The authors of HoneyKube note the higher number of attack surfaces that can be monitored with containers; which, combined with the multiple observation points inside the cluster allow for higher quantities of data collected [24].

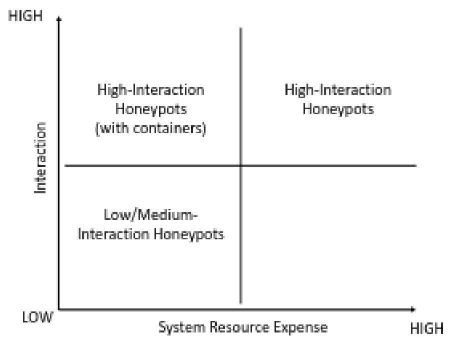

HIGH
Interaction
High-Interaction Honeypots (with containers)
High-Interaction Honeypots
Low/Medium-Interaction Honeypots
LOW
System Resource Expense
HIGH

Figure 3.6: interaction capabilities vs system resource expense, as seen in [27]

3.2.3 Virtual Machine Detection

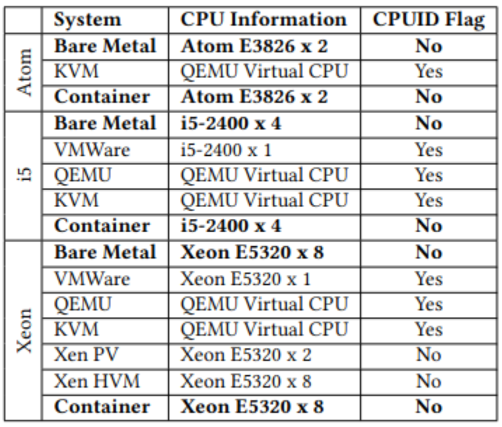
A comparison was made by Alexander et al. on the ease of identifying the type of environment used [27]. The authors chose several hardware chipsets and virtual environment software so as to avoid bias against particular hardware architectures and virtual environments.

Three commonly challenging detection methods were chosen to test the identifiability of the three test "subjects": bare metal implementations, virtual machines and Linux containers (LXC).

- Variability and execution time in CPU clock sampling
- Reported CPU information

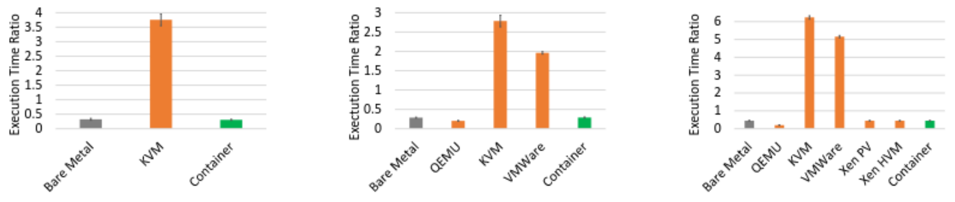
- Instruction execution time

The results presented below show that, in fact, containers are much harder to identify than virtual machines; this is notably visible in the reported CPU information and arithmetic operation performance tables.



	System	CPU Information	CPUID Flag
Atom	**Bare Metal**	**Atom E3826 x 2**	**No**
	KVM	QEMU Virtual CPU	Yes
	Container	**Atom E3826 x 2**	**No**
i5	**Bare Metal**	**i5-2400 x 4**	**No**
	VMWare	i5-2400 x 1	Yes
	QEMU	QEMU Virtual CPU	Yes
	KVM	QEMU Virtual CPU	Yes
	Container	**i5-2400 x 4**	**No**
Xeon	**Bare Metal**	**Xeon E5320 x 8**	**No**
	VMWare	Xeon E5320 x 1	Yes
	QEMU	QEMU Virtual CPU	Yes
	KVM	QEMU Virtual CPU	Yes
	Xen PV	Xeon E5320 x 2	No
	Xen HVM	Xeon E5320 x 8	No
	Container	**Xeon E5320 x 8**	**No**

Figure 3.7: Results of CPU Information Test, as seen in [27]



(a) Intel Atom (b) Intel Core i5 (c) Intel Xeon

Figure 3.8: Results Arithmetic Test, as seen in [27]

Detection of the VM environment can usually be done by capturing operating system (OS) and hardware artefacts.

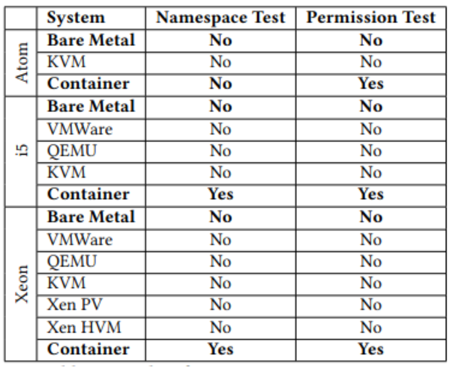
3.2.4 Container Detection

In this section, we will analyze the results Alexander et al. obtained when testing if container environments are as easily identifiable as virtual machines [27].

These tests focus on identifiable traits or characteristics that would differentiate the execution of code inside containers versus bare metal. Therefore the tests must be different from the ones used to identify virtual machines due to the different attack surfaces presented. Remember that containers make direct use of the host's kernel and execute directly on hardware.

The first test focuses on detection using namespaces. As previously said, containers make use of namespaces, used to enforce the segregation of resources. The authors mention the example of the VLANY LD_PRELOAD rootkit malware, which will compare the number of processes listed by the PS command versus the number of processes provided by SYSINFO. This vulnerability exemplifies the risks of detection if isolation is not severely taken into account.

The second test involves detection through permissions. With containers, we can have a peculiar situation in which a user may have root privileges inside a container when the container itself, executing in the host system, does not possess them. This can create the expectation in a user inside the container that he can perform any command desired; however, since that user is not root on the host system, some commands will be denied from execution. As shown in the results, permission tests identify containers with perfect accuracy. The authors admit this is a known problem, but it is nonetheless hard to solve, with a possible presented solution including granting unprivileged containers read-only permissions.



	System	Namespace Test	Permission Test
Atom	**Bare Metal**	**No**	**No**
	KVM	No	No
	Container	**No**	**Yes**
i5	**Bare Metal**	**No**	**No**
	VMWare	No	No
	QEMU	No	No
	KVM	No	No
	Container	**Yes**	**Yes**
Xeon	**Bare Metal**	**No**	**No**
	VMWare	No	No
	QEMU	No	No
	KVM	No	No
	Xen PV	No	No
	Xen HVM	No	No
	Container	**Yes**	**Yes**

Figure 3.9: Namespace and Permissions test, as seen in [27]

In sum, even though containers are not a perfect transparent environment [34], they can provide enough transparency that only certain types of attack can detect them. This was one of the reason containers were chosen for the implementation of our work [17].

3.3 Summary

As discussed in the previous sections, MTD is a favourable active defence when implemented with SDNs. It gives us flexibility and the ability to diversify our covert operations. However, when working on the state-of-the-art, we noticed that even though there are potentially favorable technologies to implement with HoneyNets, there have been few to none experiments with implementing active defence methods in HoneyNets. Active defence methods are a potential benefit to HoneyNets because they, MTD specifically, increase the potential of our HoneyNet data-capturing abilities by making our HoneyPots more obfuscated. As we claimed before, one of the essential things when deploying a HoneyNet is making sure an attacker does not suspect he is being investigated; therefore, by improving our HoneyPot's covertness, we are potentially increasing the quality of captured Data. This is a possibility we will study further ahead in this document.

Containers further improve our HoneyNets covertness; in the previous section, we demonstrated that containers are more challenging to detect than virtual machines. This, combined with lower operation costs, lower overhead, ease of scalability and increased security, made containers an obvious choice for HoneyPot deployment. Containers also provide us with the flexibility of being able to choose from hundreds of already established containerized HoneyPots. These images, freely available on docker hub, for example, have a wide variety of vulnerabilities to explore, with some more focused on general protocol exploits to more focused application-specific vulnerabilities. In sum, containers provide us with the flexibility to choose or develop HoneyPots for many different situations.

What we stand to gain by merging these two technologies is increasing what they do best by themselves, covertness.

Chapter 4

Preliminary Research

This chapter will discuss and explain our initial analysis of HoneyNets and HoneyPots; when starting our analysis of this theme, we first had to choose which tool we wanted our research to be based in; therefore we had to look into several HoneyNet implementations to find the one that best fits our work.

As discussed in the previous section, containerization is an integral part of covert operations, so we had to limit our search to tools that could accommodate that technology. We found that there exist multiple implementations of stand-alone HoneyPots; however, we focused on finding an implementation that conjoined several HoneyPots into a single architecture while supporting containerization. As we explore further ahead, we also require containerization due to the easability of merging this technology with SDNs.

All of this, combined with the ease of installation and data collection, led us to T-Pot [39]. This chapter will, therefore, present our findings with this tool. We would be remiss not to mention that are several other HoneyNet/Pot tools available which have as much or more functionality than T-Pot, however, few match its ease of installation and data analysis.

We start with a brief introduction of the T-Pot tool. T-Pot is a self-contained HoneyPot platform; it supports various types of HoneyPots while providing several visualization dashboards. It relies on containerization for its tools, logging and honeypots. From our research into the topic of HoneyNet implementations, we concluded that this is the tool most suitable to perform our initial analysis; T-pot uses containerization to launch the HoneyPots, a feature we previously identified as critical.

4.1 T-Pot - Technical Walk-Through

Data control on T-Pot relies on Suricata, an open-source IDS and Intrusion Prevention System (IPS). Data capture is the responsibility of each individual HoneyPot; since T-Pot is a collection of several HoneyPot systems, there is no single mechanism for data capture. Data collection is implemented with the aforementioned Elastic Stack.

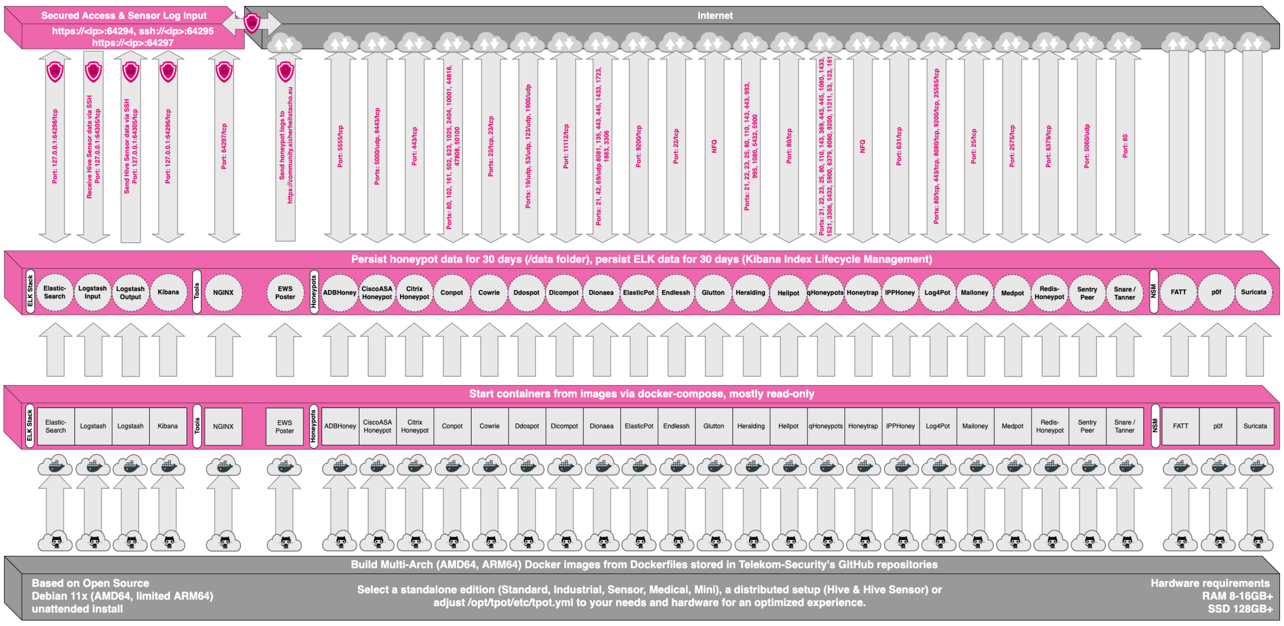

Secured Access & Sensor Log Input
https://<ip>:64294, ssh://<ip>:64295
https://<ip>:64297
Internet
Persist honeypot data for 30 days (/data folder), persist ELK data for 30 days (Kibana Index Lifecycle Management)
ELK Stack
Elastic-Search
Logstash Input
Logstash Output
Kibana
Tools
NGINX
EWS Poster
Honeypots
ADBHoney
CiscoASA Honeypot
Citrix Honeypot
Conpot
Cowrie
Ddospot
Dicompot
Dionaea
ElasticPot
Endlessh
Glutton
Heralding
Hellpot
qHoneypots
Honeytrap
IPPHoney
Log4Pot
Mailoney
Medpot
Redis-Honeypot
Sentry Peer
Snare / Tanner
NSM
FATT
p0f
Suricata
Start containers from images via docker-compose, mostly read-only
Build Multi-Arch (AMD64, ARM64) Docker images from Dockerfiles stored in Telekom-Security's GitHub repositories
Based on Open Source
Debian 11x (AMD64, limited ARM64)
unattended install
Select a standalone edition (Standard, Industrial, Sensor, Medical, Mini), a distributed setup (Hive & Hive Sensor) or
adjust /opt/tpot/etc/tpot.yml to your needs and hardware for an optimized experience.
Hardware requirements
RAM 8-16GB+
SSD 128GB+

Figure 4.1: T-Pot Architecture

w

4.1.1 Services

T-Pot offers a variety of services which can be divided into five groups:

- System Services: These are the services that the OS natively provides; they include, for example, SSH and Cockpit;
- Elastic Stack: The Elastic Stack is used for data visualization and aggregation; this stack is made up of 3 components, Elastic-Search, Logstash and Kibana. Elasticsearch is used for storing events; it is an analytics engine for all types of data. Logstash is responsible for ingesting, receiving and sending events to Elasticsearch, a server-side data processing pipeline. Finally, Kibana is a tool for data visualization; it receives events and displays them on dashboards;
- T-Pot also offers a variety of tools that are not directly connected to HoneyPots or malicious activity capture. These are, however, useful for data analysis or general OSINT (Open-source intelligence) tasks;
- HoneyPots: T-Pot offers a wide variety of containerized HoneyPots for data capture; all are open-source and freely available. Some of these HoneyPots are based on known vulnerabilities, and they simulate those vulnerabilities in order to attract attackers. Others simulate an environment in which the attacker has a high degree of freedom, allowing him to perform specific actions prohibited in real production systems;

- T-Pot also provides tools for network security monitoring; Fatt is used for extracting data from pcap files and network traffic. P0f is a passive fingerprinting tool, and Suricata is used for IDS/IPS.

4.1.2 Distributed Deployment

T-Pot provides several deployment options, with a distributed deployment in which we have two types of systems, the HIVE and HIVESENSOR. The HIVE system hosts the Elastic Stack and other general T-Pot tools. The HIVESensor will host the honeypots and transmit the log data to the HIVE's Elastic Stack [39].

If we take advantage of this deployment, we can make T-Pot a distributed HoneyPot, essentially transforming it from a single self-contained HoneyPot system into a HoneyNet system where we can distribute our Sensors across a network, which will then collect data into a single system.

4.2 Experiment Setup

This system was installed on a cloud provider, vultr. The developers of T-Pot recommend a minimum of 8GB of RAM and 128GB of SSD storage [7]. The machine used has 2 vCPUs, 16GB of RAM and 100GB of SSD storage. Installation was ISO based, meaning we had to use the Debian installer. For this experiment, the standalone option was chosen.

4.3 Experiment Results

With this experiment, we were able to investigate how honeypots are interacted with in the "open" Internet. We should note that since this system was installed in a cloud provider, it is expected that the attack frequency will be higher; this is due to the fact that the Ip address ranges used by the cloud provider are already known to attackers. This means there is a higher likelihood of attackers launching scans or attack attempts against our system.

However, our results far exceeded our expectations regarding attack frequency and amount. Just within the first hours of the HoneyPot being up and running, we had already detected thousands of potential attacks.

As previously said, T-Pot provides data visualization tools that we used to present the following data.

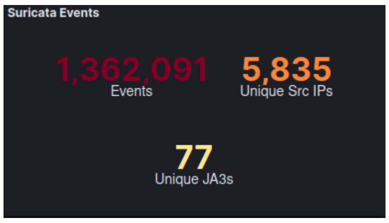


Figure 4.2: Suricata Number of Events

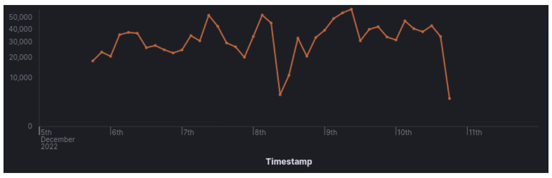


Figure 4.3: Suricata Events Histogram

4.3.1 Suricata Results

We will start this analysis by looking at the Suricata logs. As previously said, Suricata is a signature-based IDS/IPS, with applications for network security monitoring.

As we can see in figure 4.2, the number of events Suricata detects is exceptionally high, with a total of >1.3 million events in a time span of 5 days. Furthermore, these events were generated from a total of almost 6000 unique IP addresses.

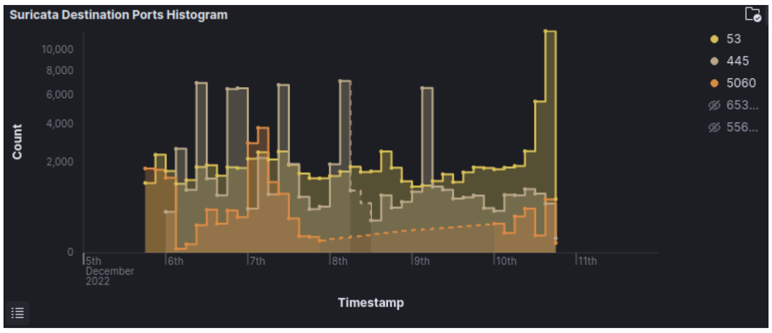


Figure 4.4: Suricata Destination Ports Histogram

Looking at the most affected ports, we can again verify that the attack attempt frequency is very high, with a big focus on ports 53, 445 and 5060. These ports were the highest affected, and when we check their purpose we can get a better picture of why that is.

- Port 53 (tpc/udp) is mostly used for DNS (Domain Name System). DNS servers listen on port 53 for queries from DNS clients.
- Port 445 is used by Microsoft Directory Services for Active Directory (AD) and for the Server Message Block (SMB) protocol over Transmission Control Protocol (TCP)/IP.
- Port 5060 is used by the Session Initiation Protocol (SIP). It is an Application Layer control protocol that creates, modifies, and terminates sessions with one or more participants. SIP clients usually use TCP or UDP on port numbers 5060 or 5061 to connect to SIP servers and other SIP endpoints.

Suricata can detect certain types of network scans; we consider this information valuable since an attacker usually starts his attack procedure by first scanning the target network in hopes he will find any information that may assist him later.

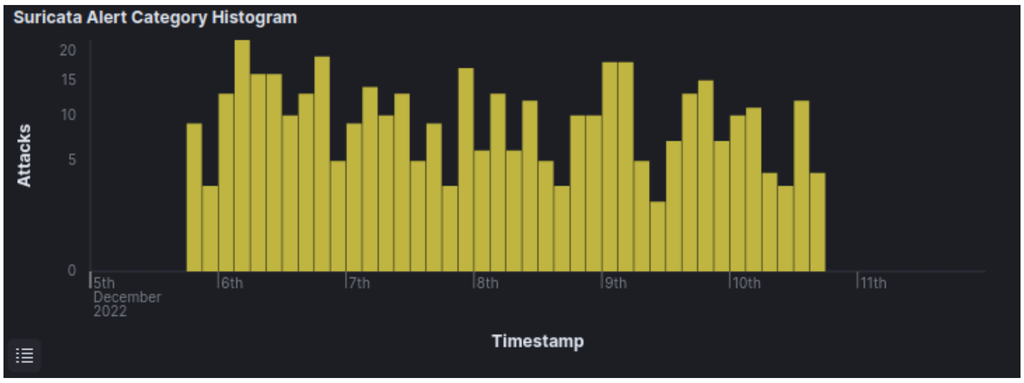
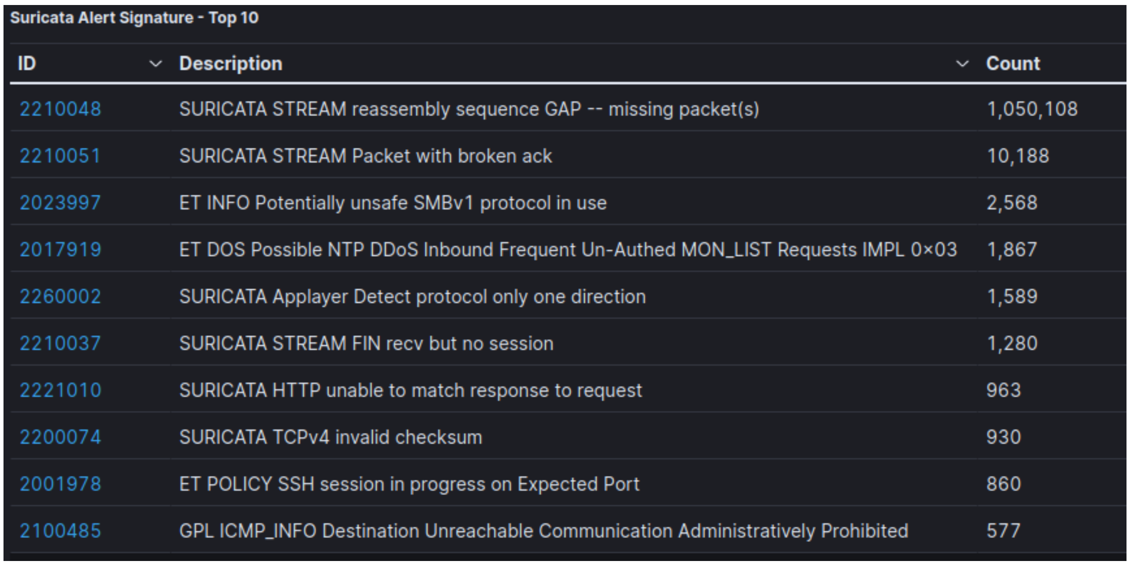


Figure 4.5: Suricata Detection of a Network Scan Histogram

This is an important statistic; our work presented in the following sections aims to attempt to decrease network scans' efficiency, reducing their value to an attacker. This information confirms our initial suspicion that HonetNets can be especially vulnerable to network scans.

The top 10 alert signatures are presented in fig. 4.6, where we can see the high number of occurences for each of those signatures.


Suricata Alert Signature - Top 10

ID	Description	Count
2210048	SURICATA STREAM reassembly sequence GAP -- missing packet(s)	1,050,108
2210051	SURICATA STREAM Packet with broken ack	10,188
2023997	ET INFO Potentially unsafe SMBv1 protocol in use	2,568
2017919	ET DOS Possible NTP DDoS Inbound Frequent Un-Authed MON_LIST Requests IMPL 0x03	1,867
2260002	SURICATA Applayer Detect protocol only one direction	1,589
2210037	SURICATA STREAM FIN recv but no session	1,280
2221010	SURICATA HTTP unable to match response to request	963
2200074	SURICATA TCPv4 invalid checksum	930
2001978	ET POLICY SSH session in progress on Expected Port	860
2100485	GPL ICMP_INFO Destination Unreachable Communication Administratively Prohibited	577

Figure 4.6: Suricata Alert Signature - Top 10

4.3.2 HoneyPots Results and Analysis

As previously said, T-Pot integrates several containeraized HoneyPots; from analysing their logs and results, we aim to demonstrate the most widely used attack techniques and the most exploited vulnerabilities.

Below we present our findings after three days of HoneyPot deployment. We start by showing the top 5 attacked honeypots and a simple explanation of their purpose.

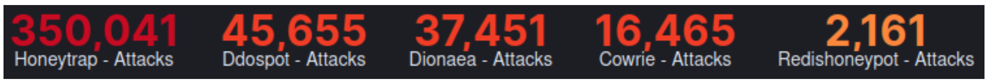


Figure 4.7: Top 5 HoneyPot Attacks

- HoneyTrap is and extensible system for running, managing and monitoring HoneyPots. With this tool, developers can costumize which services are to be used and with which modes they should run. Modes include sensor, low and high interaction. We can also deploy multiple agents connected to a central HoneyTrap server.
- Dionaea is a low-interaction honeypot designed to emulate environments and services that are known to be exploited by worms to spread across networks. It's goal is to capture the attack payloads and malware.
- DDOSPot is a honeypot for tracking and monitoring DDOS (Distributed Denial of Service) attacks. It supports various types of services/servers, like Network Time Protocol (NTP), DNS, Simple Service Discovery Protoco (SSDP), CHARGEN [9].
- Cowrie is a honeypot that can emulate SSH activity and interaction with the shell [21].
- RedisHoneyPot is a high interaction honeypot solution for the Redis protocol.

An essential piece of information is the high number of unique source IP addresses interacting with our HoneyPots. Many addresses have poor reputation, indicating that they are most likely bots and automated scripts. The information gathered by such attacks shows low quality since most of them are mass attacks or spam attempts. Most of the attacks we recorded on our HoneyNet came from IPs with low reputations, as seen in the following graphic.

With the development of our MTD method, we hope to reduce the ease with which we suffer mass attacks from spammers and bots. By implementing this active defensive mechanism we are regularly changing characteristics of our HoneyPot hosts, making them harder to identify and track. This, combined with the HoneyPots' integrated data control should help increase

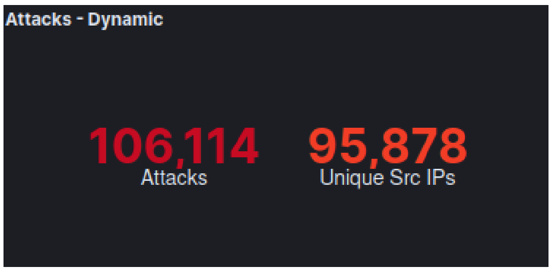


Figure 4.8: DDOSPot Total Attacks and Unique SRC IPs

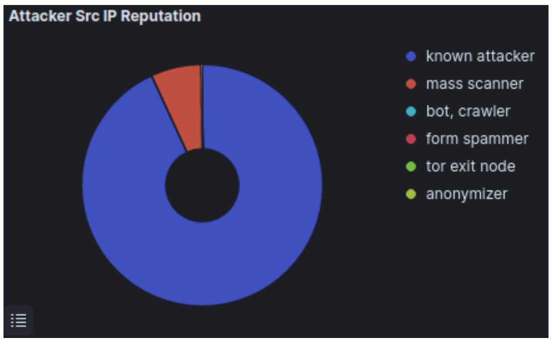


Figure 4.9: Attacker Source IP Reputation

the quality of information we capture. This is not a direct consequence, but an indirect one, by implementing this methods we are making it harder for automated attacks to reach our network, allowing only for higher skilled opponents with higher skilled methods to attack our system.

Chapter 5

Implementation

In this chapter, we go into detail on our solution for the issues presented above. We demonstrated in the State of the Art that there is a gap in HoneyNet technology when it comes to their integration with active defence methods. Nevertheless, it is valuable to implement active defences in HoneyNets due to the increase over the years of automated attacks by hackers, as demonstrated in the previous chapter, and the possible increase in the quality of captured data.

For these reasons, we chose moving target defence as the active defence method to implement. As previously stated, MTD has high synergy with software defined networks; this technology allows us to implement this defence from the perspective of the network.

We will start by describing our architecture in detail, followed by a technology analysis where we explore the tools and technologies used in our solution. After, we go in-depth on our implementation, showing the final result.

5.1 Architecture

As we previously discussed, the contributions we set out to achieve from the beginning of this work involve an architecture for HoneyNet deployment that implements an active defence method, MTD. In figure 5.1, we have a representation of our final architecture, with its main components present.

We can observe the RYU framework on the top, responsible for the SDN aspect of our solution. This framework will ingest our rules script through its northbound Application Programming Interface (API), using the Ryu API to achieve it. It will then make use of its southbound interface and, with the OpenFlow protocol, will relay these instructions to the OVS switch manager, ovs-vswitchd.

Looking at the bottom of the figure, we can observe our DockerFile. This DockerFile can represent any HoneyPot container image, which is one of the reasons we chose containerization, its flexibility. This file is then used to generate the container image, which will give us our

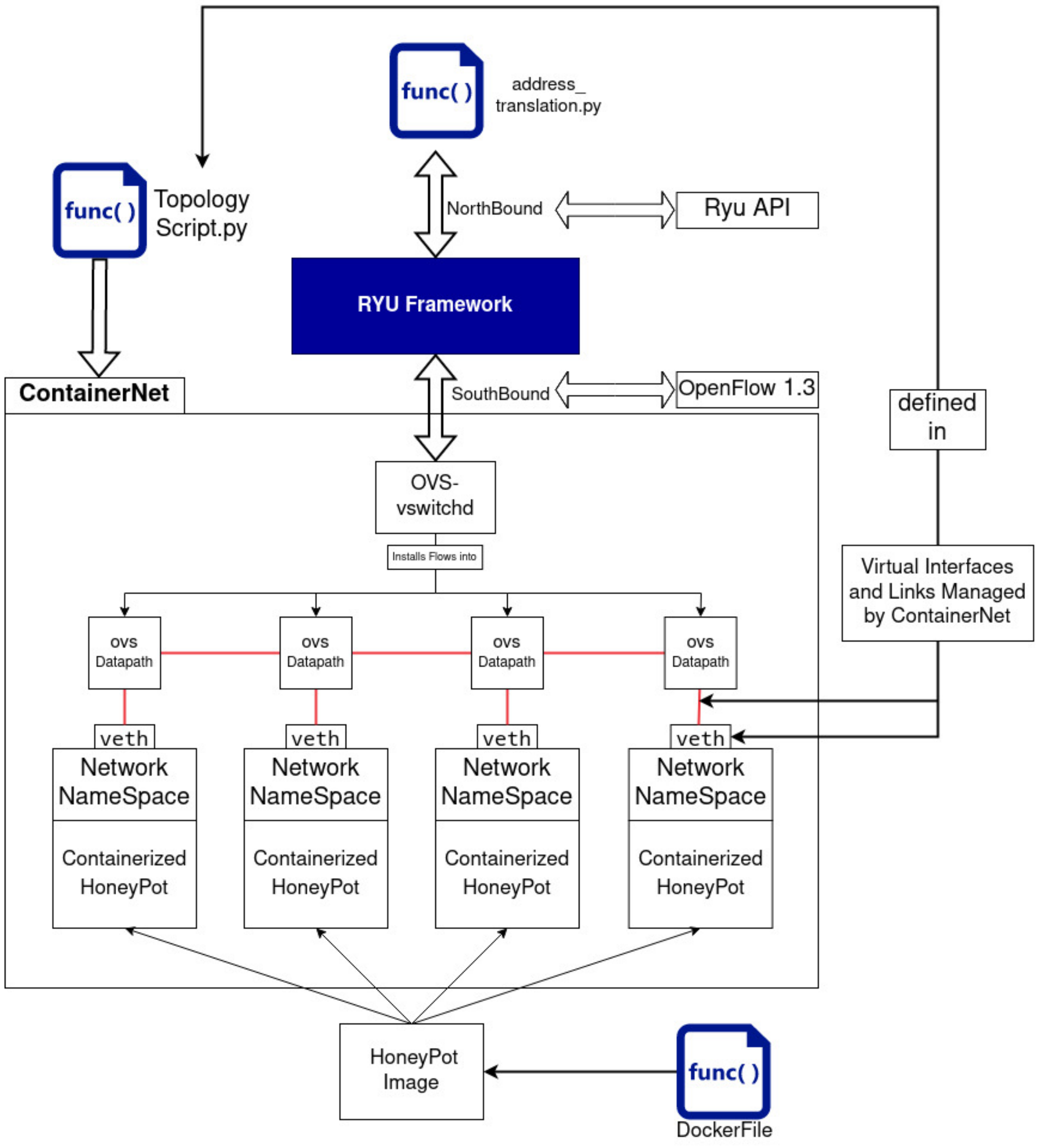


Figure 5.1: In-depth Architecture

HoneyPot containers when run. We should note the "veth" on every running container; each container will have its associated network namespace, which will, in turn, have a virtual ethernet interface.

The interface on each running container is launched and managed by containernet, which has its characteristics defined in the containernet script, topology_script.py. This script is also responsible for launching and managing the virtual links that connect all components of our architecture. This is where we will essentially define the topology of our network and its characteristics; it is also where we define which containers should be launched within the network.

5.2 Software Defined Networking

In this section, we will revisit the topic of software defined networks from a more practical standpoint. As we previously explained in greater detail, SDNs attempt to centralize network intelligence in a network component by decoupling the forwarding process from the routing process. In other words, we are separating the control plane from the data plane.

To do this, we need a central authority from which we can have a global view of the network and issue control messages. This central authority of the control plane consists of controllers that can be considered the central intelligence of the network.

When discussing SDNs it is inevitable to stumble into the term OpenFlow; it is essential to reinforce the notion that OpenFlow is merely a communications protocol, except this protocol has the peculiarity of giving us access to the forwarding plane of a network switch or router. OpenFlow allows our controller to install flow rules into, for example, switches.

When researching controllers, we looked into several options, below we will present our findings and discuss their meaning.

5.2.1 SDN Controllers

As previously stated, the controller is the component responsible for the intelligence part of an SDN. Its task is, therefore, an important one that will affect the entire performance and operation of the network it is working on. There are many different options when choosing a controller for a network, some open source and others proprietary, each with its features and specific details [13].

We will analyze several controllers to find the one that best suits our needs. We will look into the following controllers:

- Open Networking Operating System (ONOS);
- Faucet;
- OpenDayLight;
- Ryu

5.2.1.1 ONOS: Open Networking Operating System

ONOS provides the control plane for an SDN, which includes managing network components like switches or links. As the name suggests, ONOS is analogous to server operating systems, providing some of the same types of functionality.

ONOS was designed by its creators with the purpose of being a distributed system, an SDN controller designed with scalability and high availability in mind. This controller attempts to achieve this with the separation of control and data plane for wide area networks and service provider networks. A significant advantage of this platform is the big amount of documentation available, since it is a project of the Open Networking Foundation (ONF) and open-source [42].

The architecture of this controller is based on a three-tier system, southbound, northbound and distributed core [6].

- The southbound interface is a high-level API that allows the core to interact with the network environment using protocol-specific adapters to perform these interactions. Examples of protocols used by this layer are OpenFlow, Netconf and OVSDB.
- On the north-facing layer, we have a set of abstractions exposed to applications and additional network services. These abstractions allow us to access various network informations, such as devices, links and hosts. They also allow us to affect the network state via flow objective and intent-based programming (more on intent programming further ahead).
- The distributed core allows the separation of data and control functions. It is responsible for presenting a centralized view of the network state and logically centralized access to network control functions.

This controller uses an intent-based framework; instead of a network administrator configuring a network by manual processes, he can define the expected outcome or objective (the intent). The network's software is responsible for figuring out how to achieve that goal. ONOS achieves this by abstracting a network service into a set of rules a flow should meet, which will then be implemented internally by the generation of the underlying OpenFlow configuration.

ONOS can connect or disconnect to network components while it is still running; this is thanks to mechanisms built in to provide the high flexibility that ONOS gives us.

On the topic of scalability, ONOS allows users to create clusters of multiple machines that can act together as a unified distributed system, this is what the controller creators call a cluster. Cluster configuration is made simple, with the added flexibility of new controllers being able to join and leave dynamically. This controller is specifically designed to scale horizontally while taking performance into account [6].

ONOS can provide fault tolerance as long as it runs an odd number of controllers. If a master node fails, an election will choose the next node responsible for taking control of the network [13].

5.2.1.2 OpenDayLight

OpenDayLight (ODL) is a controller developed for integration with java virtual machine software, giving it the flexibility to run on any system as long as its hardware supports java. It also uses the following tools [2]:

- Maven for build automation
- OSGi (Open Service Gateway Initiave) is a framework that is used for the back-end of this controller. It allows for dynamically loading bundles and packages
- Java interfaces are used for event listening, specifications and forming patterns
- REST APIs are used for the northbound APIs like topology manager, host tracker, flow programmer, etc

This controller is based on the Model-Driven Service Abstraction Layer (MD-SAL); essentially, underlying network devices and applications are represented as models whose interactions are processed in the SAL. Broadly speaking, Service Abstraction is a design principle in which the information given in a service contract is limited to what is required to utilize that service. [8]

ODL models are defined by their roles in a given interaction; they can be either "producer" or "consumer" models. A "producer" model implements an API and provides its data, while a "consumer" uses the API and consumes its data.

Security is a big focus of ODL; its platform provides frameworks for authentication and authorization and automatic discovery and securing of network devices and controllers. ODL has similar fault tolerance to ONOS, requiring an odd number of controllers to achieve it [42].

5.2.1.3 Ryu

Ryu is described by its creators as a component-based software-defined networking framework; it provides software components with well-defined APIs that allow developers to create network management and control applications easily. Ryu supports multiple protocols for managing network devices, like Netconf, OF-config and OpenFlow versions 1.0 to 1.5. Ryu is available as an open-source license, giving it the benefits of open-source software [4].

We can think of Ryu as a toolbox, with which an SDN controller can be functionality built. Therefore, this controller presents itself as a different proposition from the other options available.

In image 5.1, we can see the components that compose a Ryu SDN controller. Ryu has many components that can be dynamically used to suit our needs; these components can vary, operationally speaking, we have for example, a component for app_management, the central management of Ryu applications, or a component of_proto_v1_0_parser, used for decoding and encoding implementations of OpenFlow 1.0 [11].

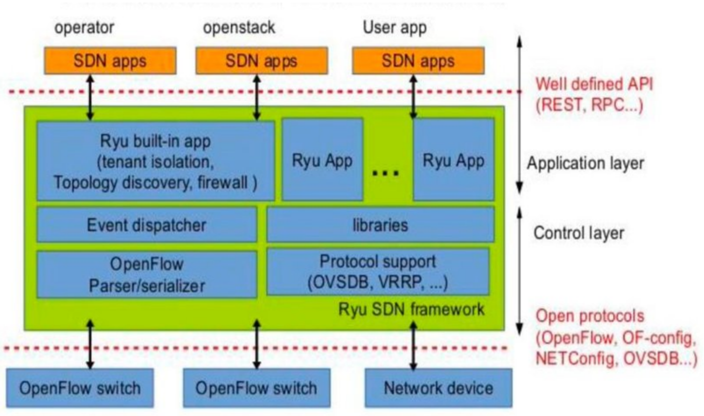

operator
openstack
User app
SDN apps
SDN apps
SDN apps
Well defined API (REST, RPC...)
Ryu built-in app (tenant isolation, Topology discovery, firewall)
Ryu App
...
Ryu App
Application layer
Event dispatcher
libraries
Control layer
OpenFlow Parser/serializer
Protocol support (OVSDB, VRRP, ...)
Ryu SDN framework
Open protocols (OpenFlow, OF-config, NETConfig, OVSDB...)
OpenFlow switch
OpenFlow switch
Network device

Figure 5.2: RYU Framework Architecture , as seen in [11]

Ryu has strong flexibility thanks to how its supporting infrastructure is built; users can write code to utilise those modules as required.

In terms of interfaces, Ryu supports the aforementioned protocols for southbound management. On its northbound interface, Ryu relies on RESTful APIs, which are more limited when compared to ONOS and ODL.

5.2.1.4 Faucet

Built on top of Ryu, Faucet is described by its creators as a compact open-source OpenFlow controller; it moves away from vendor-specific software solutions, allowing the use of open-source software for network control functions.

Faucet does not support clustering with other controllers; each instance is completely idempotent and only deals with what it is configured to control. This leads to poor architectural scalability.

Concerning its interfaces, faucet's southbound supports OpenFlow 1.3 as its protocol and has support for VLANs, IPv4, IPv6, static and BGP routing, policy based forwarding and ACLs matching. On the northbound, YAML configuration files are used to track the intended system state instead of relying on API calls. The use of these YAML files requires external tools for dynamically applying configurations; it does, however, also give us the benefit of administrating the SDN by CI/CD pipelines and testing tools [1].

5.2.2 Controllers Comparison

With a clear view of the purpose and differences between the SDN controllers presented, we can now move to a stage where we choose the one we will use for our work. As seen in the previous subsections, each controller has specific characteristics which, in a way, make them different from each other. From the options presented, ONOS and ODL are what we classify as "higher-level",

	ONOS	ODL	Faucet	Ryu
OF Support	yes	yes	no (only 1 version supported)	yes
Scalability	yes	yes	no	yes
Documentation and Resources	yes	yes	no	yes
Programming Language	Java	Java	Python	Python
Extensibility	yes	yes	yes	yes

Table 5.1: Controller Comparison

that is, they provide many functionalities that are more focused on high-scale projects.

Our solution, however, doesn't require the high levels of complexity that some of the controllers presented have. Therefore, we should try to choose the one that best fits our needs without adding unnecessary complexity. To expand on this notion, ONOS, for example, is a "higher-level" controller with a fully-fledged operating system designed with networking operations in mind. Even though it is very versatile in its applications, stable and scalable, we chose not to go its route. We have to remember that higher complexity can often bring issues that would not occur in simpler applications; this is a notion that is applied to many fields, essentially, use only what is strictly necessary, nothing more and nothing less [33].

We note that if we were looking for the most featured controller, our choice would be more inclined towards ONOS or ODL, which also present high modularity and scalability.

Ryu, as previously discussed, presents itself as a good option for modest-sized implementations of SDNs or research applications [11]. Due to its component-based infrastructure and its development language being python, we can easily create a script that will meet our needs without compromising performance. We also found that this framework is very well documented, unlike others which lack the documentation and community that an open-source project attracts. Therefore, even though Ryu is not the best performer in many areas, like scalability or modularity, it is the best choice for our work due to the reasons presented in this document.

5.3 Network Emulation Software

While choosing a controller is an integral part of this work, we also need to decide where we want our controller to run. We could have chosen to implement our solution in real hardware, using real machines to implement either network devices or hosts. However, we chose to implement our SDN solution in a virtualized environment; for this, we had to choose from a wide variety of network emulation software.

Before we begin our discussion of this topic, it is essential to clarify some common misunderstandings. Simulation and emulation are terms often used interchangeably when talking about network software; however, there is a difference between network emulation and network simulation. Network simulators are software solutions used primarily as testing tools in the design and development of a network; they are helpful tools to evaluate a network's theoretical model, including aspects like topology and application flow. On the other hand, network emulators are used to test the performance of a real network; they can also serve purposes like quality assurance, proof of concept or troubleshooting. Emulators allow network architects to accurately measure an application's responsiveness and throughput prior to applying changes to the real system [18].

Taking this into account, we chose to go the emulator route. Network emulation better fits this work's purposes, especially when considering the SDN aspect involved in our solution. Since we are looking for a software solution that could accommodate SDN controllers, virtual switches and dynamic configurations, emulators were found to be the best fit.

5.3.1 Mininet

Mininet is a network emulator that creates a network with virtual hosts, switches, controllers and links. The hosts created by mininet run the standard Linux network software, assuring compatibility with real operating systems. The switches support OpenFlow and, therefore, SDN.

Mininet is an ideal tool for developing and testing OpenFlow applications; it is easy to set up and inexpensive, requiring low resources for the potential it provides. The creators claim that they have successfully booted networks with up to 4096 hosts, demonstrating the efficiency this tool provides [20].

Mininet uses process-based virtualization to run its hosts and switches on a single kernel. Network namespaces are also used to provide the virtualization needed; they provide individual processes with separate network interfaces, routing tables and arp tables. This gives us the isolation needed.

A mininet network can be tought of as consisting of core 3 components:

- Isolated hosts: essentially consist of user-level processes moved into a network namespace, that will provide exclusive access and ownership of interfaces, ports and routing tables.

- Emulated links: using Linux traffic control we can control the data rate of each link to shape traffic to our needs on a given route.
- Emulated switches: can either be the default Linux bridge or Open vSwitch running in kernel mode, used to swicth packets across interfaces.

A significant advantage of Mininet is its ease of use. Using python, a developer can create simple scripts to create highly complex networks. Due to its performance, an entire network can be created and torn down in a matter of seconds, allowing for flexibility in its use.

Comparing Mininet to full system virtualization, we can conclude that Mininet excels in many parameters, like boot time which is exceedingly lower for Mininet. Scalability is also a factor where Mininet takes advantage; it can scale to hundreds or thousands of hosts, as opposed to single-digit hosts for virtualized options. It is also extremely easier to install and run; it can be installed and running in a matter of minutes.

Hardware testbeds also fail short when compared to Mininet; hardware has the limitation of being expensive and requiring more time to set up, and it is also much harder to reconfigure to test new setups.

5.3.1.1 Mininet Operation

Mininet's operation can be broken down into two components: the CLI (command line interface) and scripting. Upon installation, a user can start Mininet by either invoking a specific script which will be immediately run or by calling its startup option through the command line. Starting a network through the command line usually gives us less configurability; the available options are especially limited, and the default networks are harder to edit when using this option.

The Python API gives Mininet its highly configurable options, allowing users to create specific topologies easily, specify link characteristics and choose from the available controllers and switches, among many other options.

5.3.2 Containernet

Containernet is a fork of the Mininet emulator; therefore, it is based on it. What sets it apart from the vanilla emulator is the possibility of using Docker containers as hosts in the emulated network topologies. This opens many options for testing and research in the fields of cloud computing, fog computing, Network Function Virtualization (NFV) and Multi-access Edge Computing (MEC) [19].

Containernet offers many other features apart from using containers as hosts. For example, it allows for dynamic topology changes, like adding/removing containers to a running network topology and resource limitation of these same containers, allowing a developer to impose limitations to CPU and memory consumption for containers, giving him an improved real-world scenario in which tests can be more accurately run. There is also the option of exposing container ports and setting their environment variables through the Python API.

Containernet has requirements that must be met in order for containers to be able to run as hosts in our emulated network. Docker containers must have "bash" installed so a command line can be presented inside the container and the "IP" utility to provide networking functionalities, lastly "ping" utility must also be present in the container so that we can use the Mininet's built-in ping test.

Due to the nature of our work and the assumptions made previously, container support is critical when considering the choice of emulator used. We, therefore, chose to use Mininet's fork so that our solution could be successfully implemented.

5.4 Open vSwitch

A virtual switch allows virtual machines to interact with one another and with physical networks. They are also used to connect virtual and physical networks. Virtual switches give us the flexibility of easily changing their properties through software changes, as opposed to physical switches, in which changes can be harder to execute and even require changing certain parts of the hardware. They also allow us to configure specific actions we want that switch to make, turning a "dumb" switch that only forwards traffic into an "intelligent" switch that can perform more complex tasks according to our needs [15].

Open vSwitch, a distributed virtual multilayer switch, was created to help with multi-server virtualization deployments, in which we have multiple individual servers running virtualization technology that require the ability to communicate between themselves and the outside world. Due to the highly dynamic nature of these systems, we need a way to programmatically instruct the network on how to behave in specific scenarios, which OVS allows us to do. For this, OVS can use the OpenFlow protocol while still supporting standard management interfaces and protocols, like NetFlow, sFlow, IPFIX, etc [3].

The OVS implementation is split between eight core components: ovs-vswitchd, Linux kernel module, ovsdb-server, ovs-dpctl, ovs-vsctl, ovs-appctl, ovs-ofctl, and ovs-pki.

Here follows a list of the more relevant components and a brief description of their purpose:

- Ovs-switchd is a daemon used for management and control of OVS switches on the machine.
- The ovs-appctl provides a way to invoke commands at runtime to control the behavior and query settings from OVS daemons.
- The ovs-dpctl program is used to modify or delete OVS datapaths, multiple datapaths can

be present on the same machine. Datapath is used to refer to the forwarding plane of OVS.

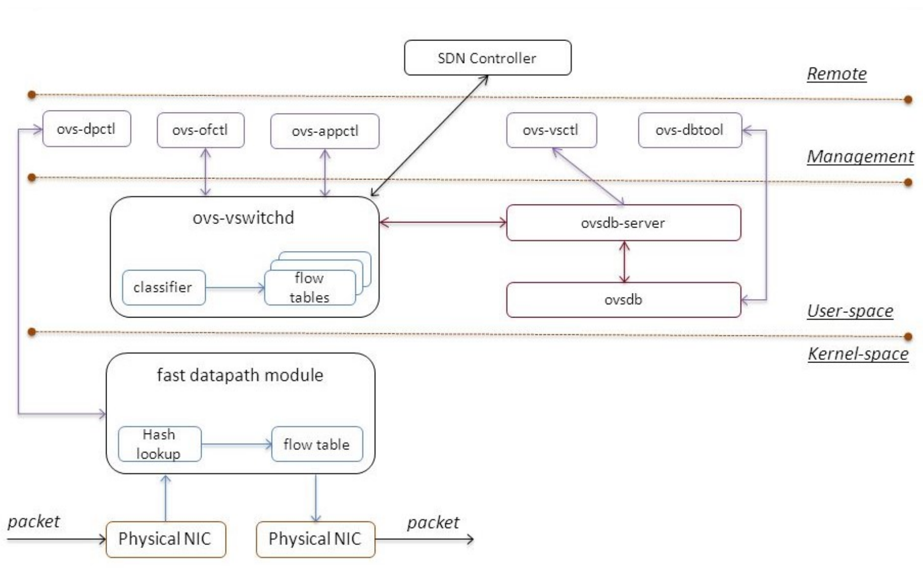


Figure 5.3: OpenvSwitch Architecture, as seen in [3]

5.5 OpenFlow

OpenFlow is a communications protocol with the peculiarity of allowing us to access the forwarding plane of a network device remotely. Essentially, this protocol allows us to install flow rules into the switches, which will redirect packets to the appropriate place when one that matches that flow arrives [23].

We also previously mentioned the term southbound interface. To expand on this context, a southbound interface (or protocol) allows our SDN controller to communicate and send updates to switches. There are multiple southbound protocols; however, OpenFlow is the most well-known, thanks to its continuous development. Another advantage is that since its inception, it has been managed by the Open Networking Foundation, a user-led organization that focuses on developing and promoting SDN solutions.

For our implementation, we will use Openflow version 1.3, released in April 2012. Even though more recent versions are available, we concluded that this one is suitable for our work due to the additional advantages over previous versions, such as better flow table matching, enhanced security and better performance. It also has great compatibility and it provides all the necessary components to implement our solution while having already proved its stability and reliability.

5.6 Implementation

As discussed in the previous sections, our implementation will be executed using a containernet network emulator running OVS switches and a Ryu controller.

For our implementation, some assumptions were made that could be untrue in a "real world" scenario. First, we assume that every host has a switch directly connected to it and that connection is safe. This is because in our implementation of MTD, an address will only be virtual after the directly connected switch. This means that the first hop after a packet leaves a host will present the actual IP address of that host, only being changed to a virtual address after that packet passes through the first switch. So, implicitly, each switch is responsible for the address translation to its directly connected host.

5.6.1 Network Topology

We developed the topology.py script, which handles network topology. This is a script specific for containernet in which we specify the topology of our network.

As previously stated, we rely on directly connected switches to perform address translation to the host they are connected with. Therefore, for each host, we must have a switch. Regarding links, each switch must maintain a minimum of one connection to other switches, with most maintaining two or more connections. The first topology we developed is represented in figure 5.4.

As previously mentioned, containernet allows us to use containers as hosts in our emulated network; for this, we need to specify which docker images should be used in our topology script. Through this script, we can customize the IP address assigned to our container, as well as docker-specific configurations like volumes to be used and resource limitations. The use of volumes is, of course, very advantageous in this use case since we obviously want our data to be permanently stored instead of being volatile like the containers they are collected in. For now, we will be using the standard ubuntu image, "ubuntu:trusty", for demonstration purposes.

We should note that thanks to the dynamic nature of the technologies we are using, this topology can be easily changed to suit our changing needs; however, we first chose to start with a static topology like the one represented to have a better testing scenario for our MTD method and flow analysis.

After our initial analysis, we concluded that this architecture is subject to failure if any of the links fail. If, for example, we were to lose the link between switch two and switch three, all connectivity would be lost between their assigned hosts, host two and host three.

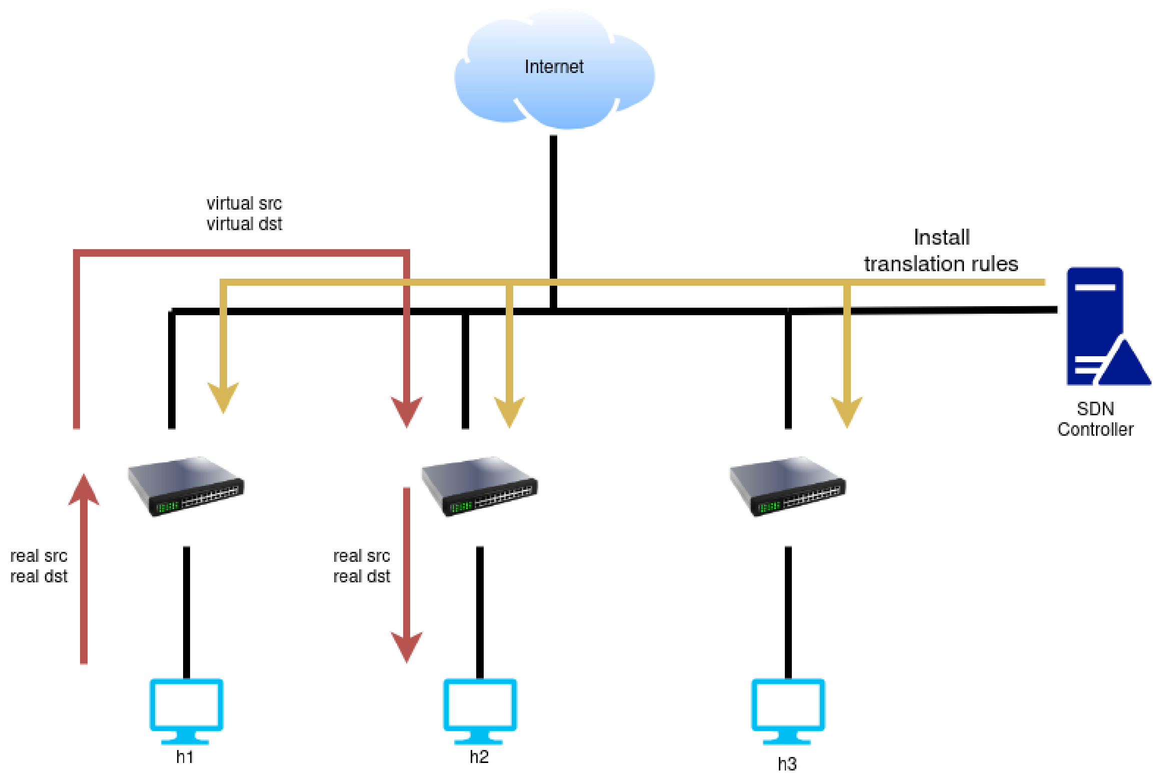


Figure 5.4: Initial Topology

5.6.2 Ryu Script

This script, address_translation.py, will essentially give our entire network its functionality by instructing the switches on what to do when a packet arrives. This allows our architecture to have switching capabilities and address translation between switches. This script was adapted and modified by us from [4] to better suit our needs. Functions were added and modified for increased functionality.

As previously stated, one of the reasons Ryu was the chosen controller for the implementation of this work is its ease of usability without compromising either functionality or features. In addition, Ryu uses python as its scripting language, allowing us to write and understand new scripts with new functionalities easily.

This script is split into several functions that work together to provide the functionalities mentioned above; its main functions are:

- switch_features_handler
- add_flow
- packet_in_handler
- update_addrs

- isDirectContact
- isRealIp
- isVirtualIp

From here on out we will refer to a datapath as an instance of an OpenFlow switch.

5.6.2.1 Global Variables

We need to define some global variables to implement address translation; we need two global lists where our addresses will be stored. Specifically, we define one for real addresses and another for virtual addresses. We also require several dictionaries, which, in python, act as key-value stores. One dictionary stores real-to-virtual translations, other stores virtual-to-real translations and the third is used to store the real addresses of hosts directly connected to switches.

```
Real_Addrs = ["10.0.0.1", "10.0.0.2", "10.0.0.3", "10.0.0.4", "10.0.0.5",
    "10.0.0.6", "10.0.0.7", "10.0.0.8"]

Virtual_Addrs = []

Real_to_Virtual = {"10.0.0.1": , "10.0.0.2": , "10.0.0.3": , "10.0.0.4": ,
    "10.0.0.5": , "10.0.0.6": , "10.0.0.7": , "10.0.0.8": }

Virtual_to_Real = {: "10.0.0.1", : "10.0.0.2", : "10.0.0.3", : "10.0.0.4",
    : "10.0.0.5", : "10.0.0.6", : "10.0.0.7", : "10.0.0.8"}

Real_Switch_Connections = {"10.0.0.1": 1, "10.0.0.2": 2, "10.0.0.3": 3,
    "10.0.0.4": 4, "10.0.0.5": 5, "10.0.0.6": 6, "10.0.0.7": 7, "10.0.0.8":
    8}

Virtual_Switch_Connections = {}
```

Listing 5.1: Global Variables

We should note that the virtual address related entries will start in an empty state, and will be populated as virtual addresses are generated and assigned to hosts.

5.6.2.2 update_addrs

The update_addrs function is responsible for generating new addresses to be assigned in the next translation phase; it makes use of the "Faker" python library to generate random addresses and change the hosts' entry in the Virtual_to_Real list.

The updateAddrs function is triggered by the TimerEventGen, which runs after a set amount of time, in this case, 5 seconds. TimerEventGen clears the Virtual_Switch_Connections dictionary

```
def updateAddrs(self):
    self.Real_to_Virtual =
        self.Real_to_Virtual.fromkeys(self.Real_to_Virtual, 0)
    faker = Faker()
    lista = []
    for i in self.Real_to_Virtual.keys():
        temp = faker.ipv4()
        self.Real_to_Virtual[i] = temp
        lista.append(temp)
        self.Virtual_Addrs.append(temp)
    self.Virtual_to_Real =
        dict(zip(lista,list(self.Virtual_to_Real.values())))
```

Listing 5.2: updateAddrs

of its values before calling the updateAddrs function, which will populate that same dictionary with new values. The timer relies on the hub spawned in the start() function and is set by hub.sleep().

5.6.2.3 switch_features_handler

The switch_features_handler function is responsible for setting new flow entries to datapaths. An "EventOFPSwitchFeatures" event triggers it and requires an event as input, from which it can extract information about the message and the datapath that triggered it. This function creates a list of actions that the add_flow function will ingest.

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
        ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)
```

Listing 5.3: swicth features handler function

We should note the use of decorators (@set_ev_cls) in this function. In python, a decorator is used to add functionality to an existing object without modifying its structure. In Ryu, this particular decorator is used to declare an event handler, so this decorated method will, therefore, become a handler for these events, meaning they trigger it.

5.6.2.4 add_flow

The add_flow function will receive the previously created list of actions and additional information, like the corresponding datapath and the priority level of flow entry, along with others. This function will then use the Ryu parser to build a message to be sent to the appropriate datapath, which is done via the send_msg() method.

5.6.2.5 packet_in_handler

The packet_in_handler is responsible for the core operation of the network; it is responsible for handling packet_in events for switches that are unable to match those same packets with an appropriate response.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
```

Listing 5.4: packet in handler header

We start this function by deconstructing the received event, getting its origin datapath, corresponding parser and actual packet. We can extract the protocol being used from this packet, which is helpful for differentiating arp and ipv4 packets.

```
    msg = ev.msg
    datapath = msg.datapath
    dpid = datapath.id
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)

    arp_pkt = pkt.get_protocol(arp.arp)
    icmp_pkt = pkt.get_protocol(ipv4.ipv4)
```

Listing 5.5: packet in handler header

We then have conditions to check if the protocol being used is either arp or ipv4; this is a necessary check since we first need to populate the routing tables using arp before being able to switch ipv4 packets.

If the packet is an arp packet, we start by checking if the source IP is a real address; if true, we check if it is directly connected to the switch that issued the message to the controller. This essentially checks if this switch is the one "responsible" for the address translation to this host. After that, we change the real address to the virtual address so the packet can transverse the network in an obfuscated state.

Actions installed in datapaths are stored on the actions list, including actions to change specific fields in packets, like source or destination address. These actions are then parsed into a packet containing all necessary information and sent to the appropriate datapath via the send_msg() method.

In this function, we analyse the received packet, and the address translation is done according to its origin host and switch. If the host is not directly connected to the switch that sent this message, the packet gets routed to the appropriate switch.

In other words, switches are responsible for address translation only for the host they are directly connected with.

5.6.3 STP Implementation

Spanning Tree is a protocol implemented in layer 2 to prevent loops in a network topology. Loops in a network can cause traffic to be exchanged indefinitely if not adequately controlled. For example, a loop can occur when redundant paths connect different network segments. However, redundancy is important when designing a network: it helps keep the network functional even if certain links stop working. With STP, we analyze a network for redundant paths and disable the appropriate ports that can lead to unnecessary loops.

We implemented STP in our network with the Ryu stplib library, which provides all needed methods for proper implementation of this protocol. We will explore this characteristic of our system in further detail in the next chapter [4].

5.7 Summary

This chapter was dedicated to the design and implementation of our architecture, we justified our decisions and showed why the implementation was performed this way. With our architecture fully designed and implemented in practice, we can now move on to a stage where we test its capabilities and functionalities. The next chapter will explore our implementation in further detail, showing the results we got while testing it.

Chapter 6

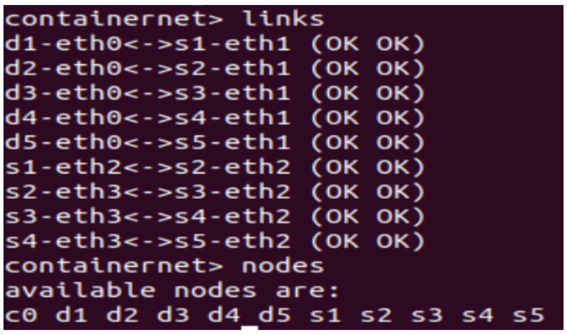
Results and Analysis

6.1 Initial Tests

In this section, we will present our results of experiments with the basic topology.py, where we only have hosts (containers running Ubuntu) directly connected to OVS switches. Host N is directly connected with switch N, which is connected to switches N-1 and N+1 (except for the first and last).

As previously discussed, we are using Ryu as our controller, and it is running the address_translation.py script. This script performs the IP translation from real to virtual. The real IP's are only exposed in the connection from the switch to the source and destination host; for the rest of the path the virtual IP will be used. Hosts communicate between themselves using real IPs; therefore, a host must be previously familiar with the real IP of the host it is trying to contact. The list of hosts, switches and their associated links is represented in figure 6.1.

This topology is graphically represented in figure 5.4.

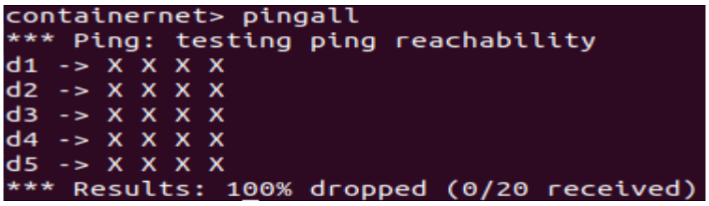


```
containernet> links
d1-eth0<->s1-eth1 (OK OK)
d2-eth0<->s2-eth1 (OK OK)
d3-eth0<->s3-eth1 (OK OK)
d4-eth0<->s4-eth1 (OK OK)
d5-eth0<->s5-eth1 (OK OK)
s1-eth2<->s2-eth2 (OK OK)
s2-eth3<->s3-eth2 (OK OK)
s3-eth3<->s4-eth2 (OK OK)
s4-eth3<->s5-eth2 (OK OK)
containernet> nodes
available nodes are:
c0 d1 d2 d3 d4 d5 s1 s2 s3 s4 s5
```

Figure 6.1: Links and nodes

6.1.1 Reachability Test

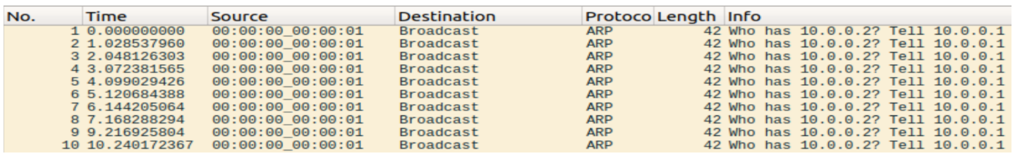
Before we connect the controller hosts will not be able to communicate between themselves, this is obvious since the switches do not yet know where to forward those packets to.



```
containernet> pingall
*** Ping: testing ping reachability
d1 -> X X X X
d2 -> X X X X
d3 -> X X X X
d4 -> X X X X
d5 -> X X X X
*** Results: 100% dropped (0/20 received)
```

Figure 6.2: Reachability Test Fail

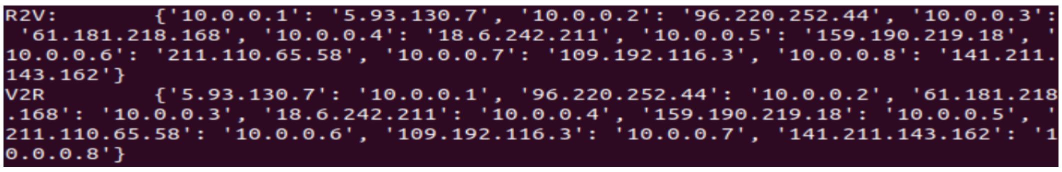
If we attempt a ping between d1 and d2 and check the directly connected switch's interface (s1-eth1) we can see that arp requests does reach it, however the switch will drop the packet since it does not have flow rules installed. This failure is represented in figure 6.2. The ARP packets associated with this failure are represented in figure 6.3



```
No.     Time          Source              Destination   Protoco Length Info
      1 0.000000000   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      2 1.028537960   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      3 2.048126303   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      4 3.072381565   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      5 4.099029426   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      6 5.120684388   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      7 6.144205064   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      8 7.168288294   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
      9 9.216925804   00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
     10 10.240172367  00:00:00_00:00:01   Broadcast     ARP         42 Who has 10.0.0.2? Tell 10.0.0.1
```

Figure 6.3: Packets of Reachability Test Failure

When we start the controller script address_translation.py the translation rules will be preemptively installed on the switches. 2 key-value lists are kept, translation from real to virtual and translation from virtual to real. Virtual IP addresses are randomly created using the faker python library. In figure 6.4 we have a representation of a translation list.

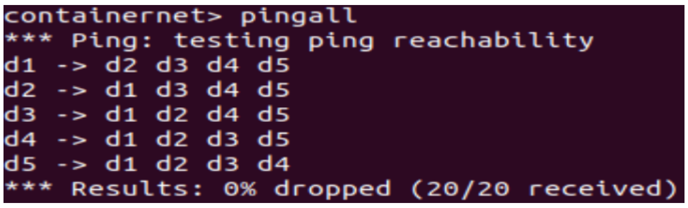


```
R2V:        {'10.0.0.1': '5.93.130.7', '10.0.0.2': '96.220.252.44', '10.0.0.3':
 '61.181.218.168', '10.0.0.4': '18.6.242.211', '10.0.0.5': '159.190.219.18', '
10.0.0.6': '211.110.65.58', '10.0.0.7': '109.192.116.3', '10.0.0.8': '141.211.
143.162'}
V2R         {'5.93.130.7': '10.0.0.1', '96.220.252.44': '10.0.0.2', '61.181.218
.168': '10.0.0.3', '18.6.242.211': '10.0.0.4', '159.190.219.18': '10.0.0.5', '
211.110.65.58': '10.0.0.6', '109.192.116.3': '10.0.0.7', '141.211.143.162': '1
0.0.0.8'}
```

Figure 6.4: Address Translation List

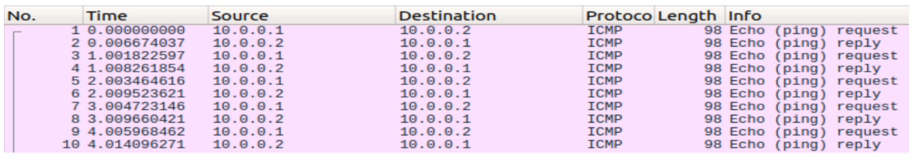
After starting the controller we retry to ping to check reachability between hosts. A sucessfull transaction of pings is represented in figure 6.5.

To check if the translation is being properly made we use wireshark to observe the different interfaces and capture the packets that transverse them. If we check an interface directly connected to a host we expect to see real IP addresses in the source and destination fields of packets transversing it, this is represented in figure 6.6.



```
containernet> pingall
*** Ping: testing ping reachability
d1 -> d2 d3 d4 d5
d2 -> d1 d3 d4 d5
d3 -> d1 d2 d4 d5
d4 -> d1 d2 d3 d5
d5 -> d1 d2 d3 d4
*** Results: 0% dropped (20/20 received)
```

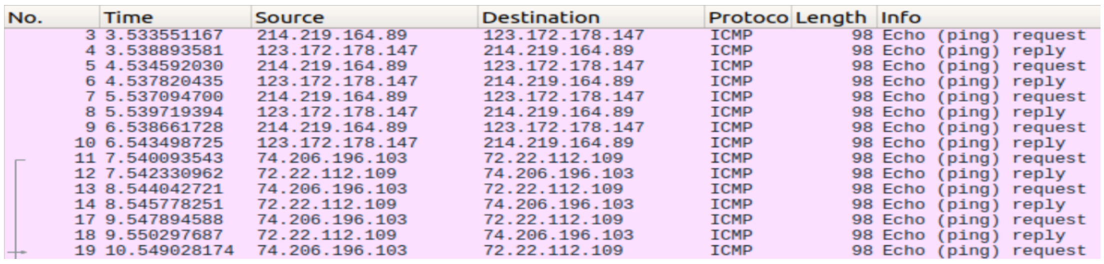
Figure 6.5: Reachability Test Success



No.	Time	Source	Destination	Protoco	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request
2	0.006674037	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply
3	1.001822597	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request
4	1.008261854	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply
5	2.003464616	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request
6	2.009523621	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply
7	3.004723146	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request
8	3.009660421	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply
9	4.005968462	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request
10	4.014096271	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply

Figure 6.6: Internal Interface Packets

In this specific case (figure 6.6) we are pinging d1 and d2 and are observing the s1-eth1 interface. As we can check this is the expected outcome, real IP addresses. However, if we now observe the outer interface of the same switch this is what we capture, represented in figure 6.7.



No.	Time	Source	Destination	Protoco	Length	Info
3	3.533551167	214.219.164.89	123.172.178.147	ICMP	98	Echo (ping) request
4	3.538893581	123.172.178.147	214.219.164.89	ICMP	98	Echo (ping) reply
5	4.534592030	214.219.164.89	123.172.178.147	ICMP	98	Echo (ping) request
6	4.537820435	123.172.178.147	214.219.164.89	ICMP	98	Echo (ping) reply
7	5.537094700	214.219.164.89	123.172.178.147	ICMP	98	Echo (ping) request
8	5.539719394	123.172.178.147	214.219.164.89	ICMP	98	Echo (ping) reply
9	6.538661728	214.219.164.89	123.172.178.147	ICMP	98	Echo (ping) request
10	6.543498725	123.172.178.147	214.219.164.89	ICMP	98	Echo (ping) reply
11	7.540093543	74.206.196.103	72.22.112.109	ICMP	98	Echo (ping) request
12	7.542330962	72.22.112.109	74.206.196.103	ICMP	98	Echo (ping) reply
13	8.544042721	74.206.196.103	72.22.112.109	ICMP	98	Echo (ping) request
14	8.545778251	72.22.112.109	74.206.196.103	ICMP	98	Echo (ping) reply
17	9.547894588	74.206.196.103	72.22.112.109	ICMP	98	Echo (ping) request
18	9.550297687	72.22.112.109	74.206.196.103	ICMP	98	Echo (ping) reply
19	10.549028174	74.206.196.103	72.22.112.109	ICMP	98	Echo (ping) request

Figure 6.7: External Interface Packets

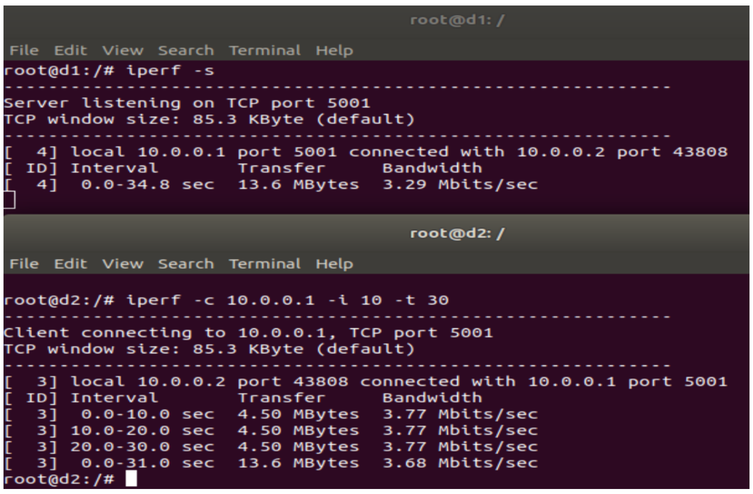
Notice the source and destination fields of this packets (figure 6.7), as we can see this IP addresses do not "belong" to any host on our network, yet the switches know where to forward them thanks to the translation rules. Notice also that initially pings were being made between 214.219.164.89 and 123.172.178.147 however after packet number 11 the addresses change to 74.206.196.103 and 72.22.112.109. This is another property of our translation technique, after a given amount of time, new virtual addresses are generated and new flow rules corresponding to this new translation installed on the switches.

So, in practice, a virtual IP has a finite life span and will be replaced with a new one after a given amount of time. To not have collisions between virtual addresses is a property related to the dynamism of the address assignment, and that we find important to implement since collisions would mean we are re-utilizing virtual IPs, something we need to avoid in order to inadvertently provide information to adversaries.

6.1.2 iperf test

iperf is a tool used to measure network performance, iperf can create TCP and UDP data streams and measure the throughput of the network carrying them. The ideia behind this test is assuring that the IP address translation is maintained throughout the entire stream duration. We need to avoid a situation in which our real IP address is exposed during a stream, for example when the controller updates the virtual addresses.

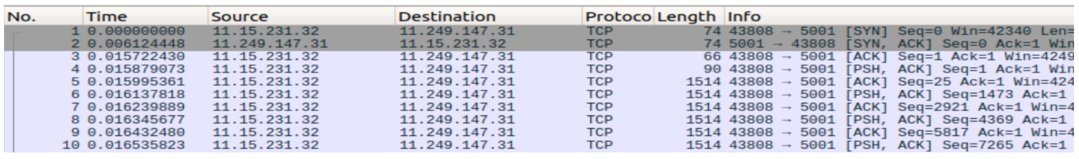
TCP data stream test is represented in figure 6.8, and the packets associated with this exchange are represented in figure 6.9. UDP data stream test is represented in figure 6.10, and the packets associated with this exchange are represented in figure 6.11.


root@d1: /
File Edit View Search Terminal Help
root@d2: /
File Edit View Search Terminal Help

```
root@d1:/# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 43808
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-34.8 sec  13.6 MBytes  3.29 Mbits/sec
```

```
root@d2:/# iperf -c 10.0.0.1 -i 10 -t 30
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.2 port 43808 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  4.50 MBytes  3.77 Mbits/sec
[  3] 10.0-20.0 sec  4.50 MBytes  3.77 Mbits/sec
[  3] 20.0-30.0 sec  4.50 MBytes  3.77 Mbits/sec
[  3]  0.0-31.0 sec  13.6 MBytes  3.68 Mbits/sec
root@d2:/#
```

Figure 6.8: iperf TPC Test Results

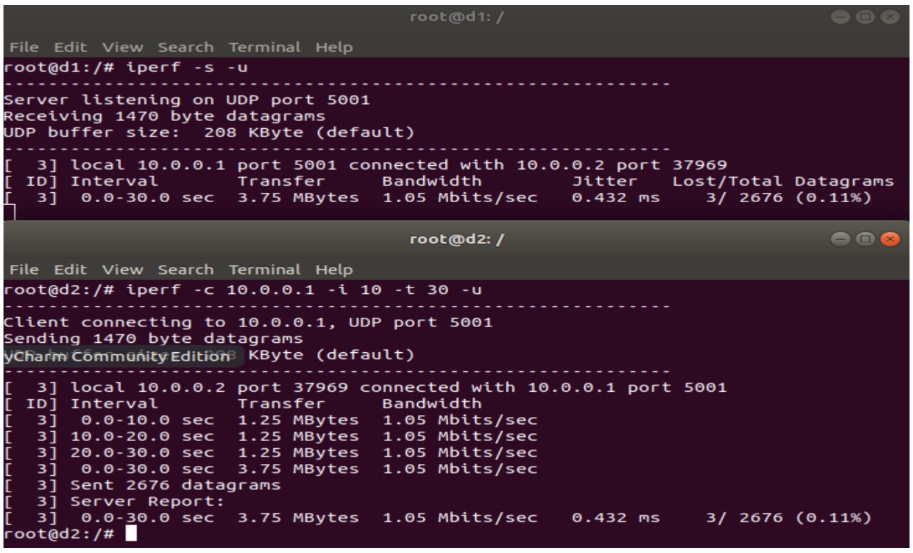


```
No.  Time         Source          Destination     Protoco Length Info
   1 0.000000000  11.15.231.32    11.249.147.31   TCP       74 43808 → 5001 [SYN] Seq=0 Win=42340 Len=
   2 0.006124448  11.249.147.31   11.15.231.32    TCP       74 5001 → 43808 [SYN, ACK] Seq=0 Ack=1 Win
   3 0.015722430  11.15.231.32    11.249.147.31   TCP       66 43808 → 5001 [ACK] Seq=1 Ack=1 Win=4249
   4 0.015879073  11.15.231.32    11.249.147.31   TCP       90 43808 → 5001 [PSH, ACK] Seq=1 Ack=1 Win
   5 0.015995361  11.15.231.32    11.249.147.31   TCP     1514 43808 → 5001 [ACK] Seq=25 Ack=1 Win=424
   6 0.016137818  11.15.231.32    11.249.147.31   TCP     1514 43808 → 5001 [PSH, ACK] Seq=1473 Ack=1
   7 0.016239889  11.15.231.32    11.249.147.31   TCP     1514 43808 → 5001 [ACK] Seq=2921 Ack=1 Win=4
   8 0.016345677  11.15.231.32    11.249.147.31   TCP     1514 43808 → 5001 [PSH, ACK] Seq=4369 Ack=1
   9 0.016432480  11.15.231.32    11.249.147.31   TCP     1514 43808 → 5001 [ACK] Seq=5817 Ack=1 Win=4
  10 0.016535823  11.15.231.32    11.249.147.31   TCP     1514 43808 → 5001 [PSH, ACK] Seq=7265 Ack=1
```

Figure 6.9: iperf TCP Test Packets

6.2 Link Failure

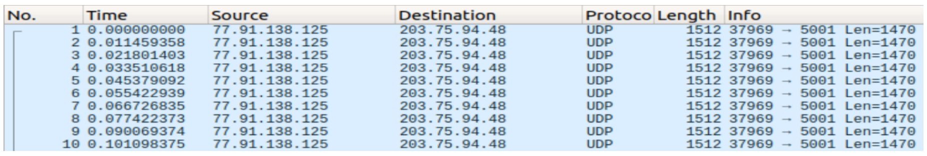
In this subsection we will discuss the possibility of link failures and how our script deals with them. Let's assume the scenario where we have 3 hosts (d1,d2,d3) and 3 corresponding switches (s1,s2,s3). Each host is directly connected to its corresponding switch and they are linked linearly between themselves. If the link between s2 and s3 would fail what should we expect to happen? Well, if this link fails d3 will be isolated in the network along with its switch, therefore connection will be lost between hosts d1/d3 and d2/d3. We can confirm our suspicions by running a reachability test on our network, represented in figure 6.12.



```
root@d1: /
File  Edit  View  Search  Terminal  Help
root@d1:/# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 37969
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-30.0 sec  3.75 MBytes  1.05 Mbits/sec   0.432 ms    3/ 2676 (0.11%)
```

```
root@d2: /
File  Edit  View  Search  Terminal  Help
root@d2:/# iperf -c 10.0.0.1 -i 10 -t 30 -u
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5001
Sending 1470 byte datagrams
yCharm Community Edition KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.2 port 37969 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] 10.0-20.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] 20.0-30.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3]  0.0-30.0 sec  3.75 MBytes  1.05 Mbits/sec
[  3] Sent 2676 datagrams
[  3] Server Report:
[  3]  0.0-30.0 sec  3.75 MBytes  1.05 Mbits/sec   0.432 ms    3/ 2676 (0.11%)
root@d2:/#
```

Figure 6.10: iperf UDP Test Results



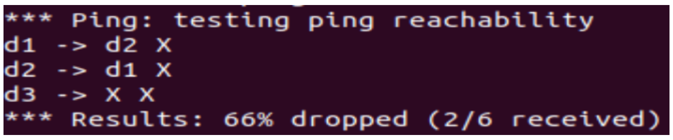
No.	Time	Source	Destination	Protoco	Length	Info
1	0.000000000	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
2	0.011459358	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
3	0.021801403	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
4	0.033510618	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
5	0.045379092	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
6	0.055422939	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
7	0.066726835	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
8	0.077422373	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
9	0.090069374	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470
10	0.101098375	77.91.138.125	203.75.94.48	UDP	1512	37969 → 5001 Len=1470

Figure 6.11: iperf UDP Test Packets

The topology described in the last section works for our script, however it is not resistant to link failures, links are expected to always be up and running, and if only one of those links were to fail the entire connectivity of the network would be compromised.

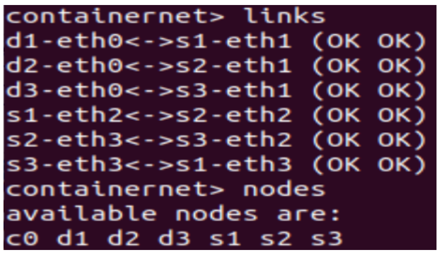
In an attempt to correct this and introduce resistance to link failure we attempt to add a link between s1 and s3, therefore we would have redundancy if one link were to fail. We are still assuming that the direct connection between a host and its corresponding switch is invulnerable. For this following test we will use the hosts and network topology represented in figure 6.13.

The result of a reachability test under this conditions is represented in figure 6.14. As we can see none of the hosts is able to communicate with other hosts. If we check wireshark on interface s1-eth3 and do a ping between d1 and d3 we get the following result, represented in figure 6.15.



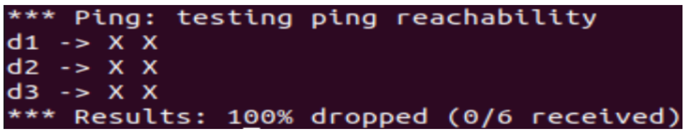
```
*** Ping: testing ping reachability
d1 -> d2 X
d2 -> d1 X
d3 -> X X
*** Results: 66% dropped (2/6 received)
```

Figure 6.12: Link Failure Example



```
containernet> links
d1-eth0<->s1-eth1 (OK OK)
d2-eth0<->s2-eth1 (OK OK)
d3-eth0<->s3-eth1 (OK OK)
s1-eth2<->s2-eth2 (OK OK)
s2-eth3<->s3-eth2 (OK OK)
s3-eth3<->s1-eth3 (OK OK)
containernet> nodes
available nodes are:
c0 d1 d2 d3 s1 s2 s3
```

Figure 6.13: Link Failure Teste Hosts and Links



```
*** Ping: testing ping reachability
d1 -> X X
d2 -> X X
d3 -> X X
*** Results: 100% dropped (0/6 received)
```

Figure 6.14: Redundant Link Failure Example

As we can see by the timestamp of messages, they are being sent at a very fast rate, in fact if we check bandwidth usage we will see that these messages are exhausting the links capacity.

What is happening here is what is known as a broadcast storm. Broadcast storms happen when there is an accumulation of broadcast traffic in a given network. These storms can consume enough resources to exhaust the network links.

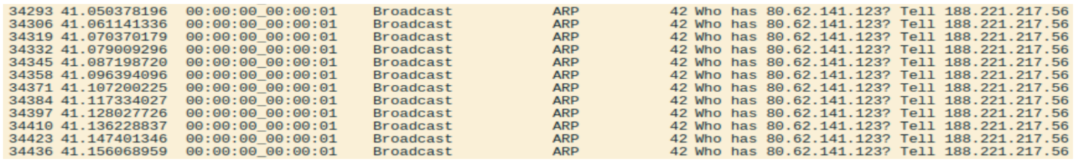
In this case we can see that while trying to make our network link failure resistant we inadvertently created a loop between our switches, leading to the duplication of broadcast traffic. This is related to how layer 2 switches deal with the reception of a broadcast type packet, when this happens the switch outputs that same packet on all ports except the one which received it.

There are several ways of fixing broadcast storms, but the objective when doing so is always to prevent link loops. We chose to implement Spanning Tree Protocol, this protocol is able to detect loops and "turn off" redundant ports.

6.2.1 Link Failure with STP implementation

In this subsection we will go into detail on the random ip translation with STP implementation for loop prevention. A secondary consequence of STP is that it is used as a means to secure network redundancy to automatically switch the path in case of a network failure.

STP works by blocking a port if a loop structure occurs, so that with that port blocked there



```
34293 41.050378196  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34306 41.061141336  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34319 41.070370179  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34332 41.079009296  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34345 41.087198720  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34358 41.096394096  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34371 41.107200225  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34384 41.117334027  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34397 41.128027726  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34410 41.136228837  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34423 41.147401346  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
34436 41.156068959  00:00:00_00:00:01    Broadcast          ARP          42 Who has 80.62.141.123? Tell 188.221.217.56
```

Figure 6.15: Redundant Link Failure Packets

is no loop. RYU takes advantage of the EventTopologyChange to detect changes in the networks topology, allowing it to react to network link failures by simply changing some ports status so that traffic can be re-routed.

We implemented this in final_stp.py script. For this we relied on the Ryu stplib library, which provides methods and classes necessary for STP implementation.

In Ryu the STP calculation is done by the following way:

1. Select root bridge

2. Decide the role of ports

3. Port state change

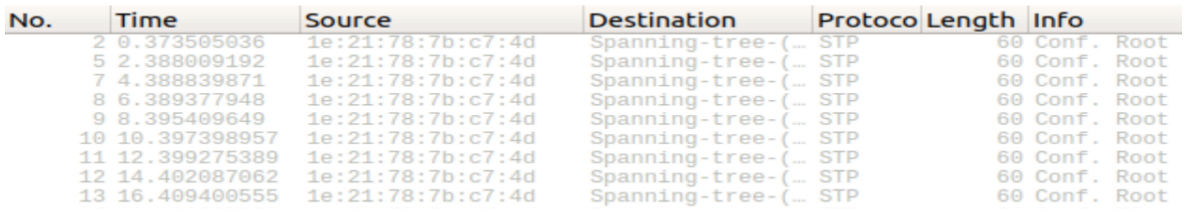
The two most critical functions implemented were topology_change_handler and port_state_change_handler. The first is responsible for deleting all flows (flushing the mac table) of a switch from which it receives the event EventTopolyChange. This is necessary since for every network topology change, the STP calculation needs to be remade. The second is responsible for handling port states, it is triggered by the EventPortStateChange.

The final_stp.py script was based on the implementation in [5] and mofied by us to accomodate for needed changes and functions.

6.2.2 3 Switches Configuration

Using the three switches configuration mininet script, we can see that the broadcast storm is avoided and hosts were able to communicate.

In the figure 6.16 we are listening to s1-eth2 and can see the inital STP configuration messages being exchanged for the STP calculations required.



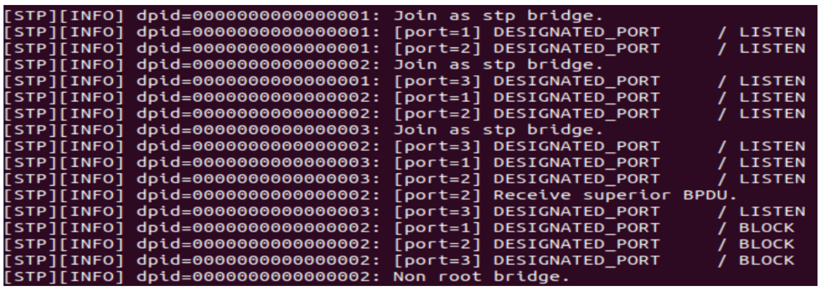
No.	Time	Source	Destination	Protoco	Length	Info
2	0.373505036	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
5	2.388009192	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
7	4.388839871	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
8	6.389377948	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
9	8.395409649	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
10	10.397398957	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
11	12.399275389	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
12	14.402087062	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root
13	16.409400555	1e:21:78:7b:c7:4d	Spanning-tree-(…	STP	60	Conf. Root

Figure 6.16: STP Protocol Packets

After the network is setup we start the Ryu controller script final_stp.py,the controller perspective is represented in figure 6.17

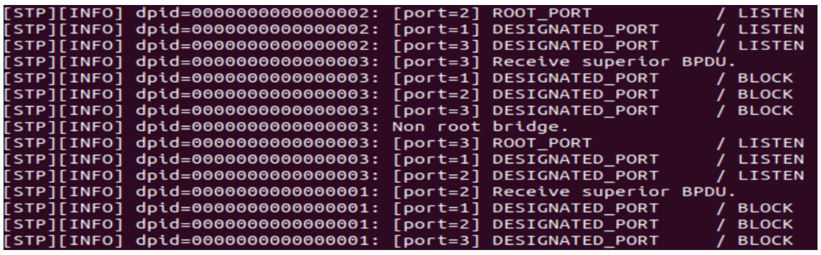
We can see the Open vSwitches joining the network in figure 6.17

We can see the port status changing as calculations are made in figure 6.18



```
[STP][INFO] dpid=0000000000000001: Join as stp bridge.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000002: Join as stp bridge.
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000003: Join as stp bridge.
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000002: Non root bridge.
```

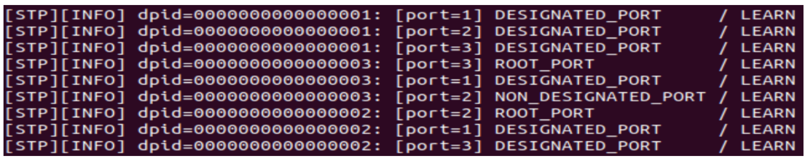
Figure 6.17: STP Protocol Switches Joining



```
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT           / LISTEN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] Receive superior BPDU.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000003: Non root bridge.
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT           / LISTEN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT     / BLOCK
```

Figure 6.18: STP Protocol Ports is Block State

After some time we can see the ports entering the learn state in figure 6.19.

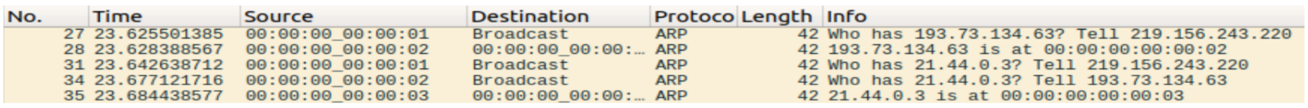


```
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT     / LEARN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT     / LEARN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT     / LEARN
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT           / LEARN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT     / LEARN
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / LEARN
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT           / LEARN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT     / LEARN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT     / LEARN
```

Figure 6.19: STP Protocol Ports in Learn State

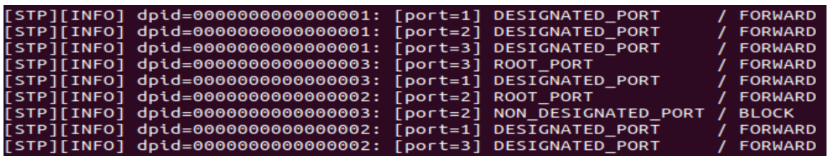
After some time ports enter their final status, until a new topologychangeevent, represented in figure 6.20.

We are now ready to perform a reachability test on our network. We will observe s1-eth2 in wireshark. Since we are observing from the perspective of a single switch there are some tests we can't "see". We start by showing the arp requests/replies in figure 6.21. And the ping request/replies, in figure 6.22. And from the perspective of mininet, represented in figure 6.23.



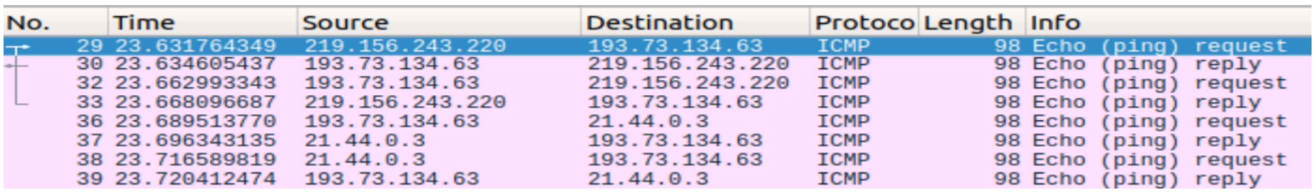
```
No.   Time          Source              Destination         Protoco Length Info
   27 23.625501385  00:00:00_00:00:01   Broadcast           ARP         42 Who has 193.73.134.63? Tell 219.156.243.220
   28 23.628388567  00:00:00_00:00:02   00:00:00_00:00:…    ARP         42 193.73.134.63 is at 00:00:00:00:00:02
   31 23.642638712  00:00:00_00:00:01   Broadcast           ARP         42 Who has 21.44.0.3? Tell 219.156.243.220
   34 23.677121716  00:00:00_00:00:02   Broadcast           ARP         42 Who has 21.44.0.3? Tell 193.73.134.63
   35 23.684438577  00:00:00_00:00:03   00:00:00_00:00:…    ARP         42 21.44.0.3 is at 00:00:00:00:00:03
```

Figure 6.21: STP Protocol ARP Packets



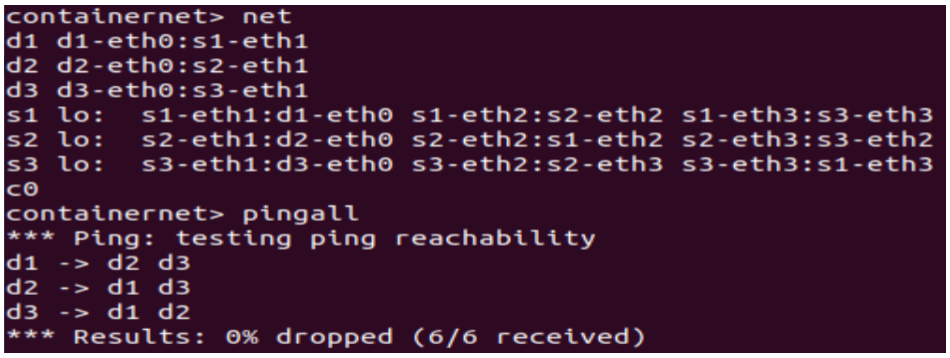
```
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT     / FORWARD
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT     / FORWARD
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT     / FORWARD
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT           / FORWARD
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT     / FORWARD
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT           / FORWARD
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / BLOCK
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT     / FORWARD
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT     / FORWARD
```

Figure 6.20: STP Protocol Ports is Final State



No.	Time	Source	Destination	Protoco	Length	Info
29	23.631764349	219.156.243.220	193.73.134.63	ICMP	98	Echo (ping) request
30	23.634605437	193.73.134.63	219.156.243.220	ICMP	98	Echo (ping) reply
32	23.662993343	193.73.134.63	219.156.243.220	ICMP	98	Echo (ping) request
33	23.668096687	219.156.243.220	193.73.134.63	ICMP	98	Echo (ping) reply
36	23.689513770	193.73.134.63	21.44.0.3	ICMP	98	Echo (ping) request
37	23.696343135	21.44.0.3	193.73.134.63	ICMP	98	Echo (ping) reply
38	23.716589819	21.44.0.3	193.73.134.63	ICMP	98	Echo (ping) request
39	23.720412474	193.73.134.63	21.44.0.3	ICMP	98	Echo (ping) reply

Figure 6.22: STP Protocol Ping Packets



```
containernet> net
d1 d1-eth0:s1-eth1
d2 d2-eth0:s2-eth1
d3 d3-eth0:s3-eth1
s1 lo:  s1-eth1:d1-eth0 s1-eth2:s2-eth2 s1-eth3:s3-eth3
s2 lo:  s2-eth1:d2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo:  s3-eth1:d3-eth0 s3-eth2:s2-eth3 s3-eth3:s1-eth3
c0
containernet> pingall
*** Ping: testing ping reachability
d1 -> d2 d3
d2 -> d1 d3
d3 -> d1 d2
*** Results: 0% dropped (6/6 received)
```

Figure 6.23: STP Protocol Ping Test Successfull

6.3 Summary

In this chapter we presented the results we obtained from the tests of our implementation. We were able to conclude that MTD does, in fact, improve our systems covertness by changing its properties. Our implementation is, however, only a proof of concept, therefore additional research most be conducted on its operation in real-world circumstances. Due to the nature of proof of concept deployments, there were some attributes we were unable to test, namely, we were unable to explicitly verify that the addition of a MTD increases the number or quality of attacks suffered by our HoneyNet.

This proof of concept does, however, show that such an implementation can be attained, with all the benefits that we described throughout this document.

Chapter 7

Conclusion

In this dissertation we planned and implemented a modular and flexible architecture for HoneyNets. What differs on our implementation when compared to already existing ones is the incorporation of an active defence method, MTD. After surveying the technologies and implementations of the latest HoneyNets, we concluded that they suffer from a lack of defensive mechanisms to prevent mass scanning and attacking.

During the development of this dissertation we had to explore various topics across multiple areas of knowledge. What first began as a study of HoneyNets and defensive mechanisms quickly developed into multiple separate researches of topics like networking, security and virtualization. Therefore, this thesis, naturally, did not focus on a particular issue or problem of computing, instead we had to research and learn about multiple topics, experiment and test with technologies and do various implementations in order to make sure our work was heading in the right direction.

Overall, from our multiple surveys and analysis into this various topics we learned a lot about the concept of software defined networking, we present a study we performed in which we classify various SDN controllers in order to choose the one most fit for our work. We also had to get related with security topics, due to the nature of this work security is an obvious concern, so we had to get as familiar to those concepts as possible. Finally, trough studying HoneyPots/Nets and defensive mechanisms we got familiar with concepts and tools more related to information "warfare", also a critical topic to take in consideration on our work.

7.1 Future Work

Work into this topic is still very much needed, the architecture we proposed intends to solve some of the issues presented throughout this dissertation, like the lack of covert host HoneyPots or the massification of spam attacks from opponents.

We should take note that our implementation only serves the purpose of being a proof of concept, as already noted; with this implementation we showed that it is possible, and of value, to explore the concepts discussed in this thesis in the context of HoneyNets.

Our suggested architecture was not implemented in the "real-world", this is what we classify as the most important thing to achieve as future work of this dissertation, to have a system that implements the characteristics we presented implemented in the "open Internet".

To achieve this we propose two solutions, the first involves the distributed deployment of T-Pot discussed in section 4.1.2. With this installation type we can have the HoneyPot hosts distributed across a cloud infrastructure, with the MTD scheme applied on top of this network. With this first solution we are eliminating possible issues with HoneyPot deployment, management and data collection since we are using an already tested and proven tool, T-Pot.

Our second suggestion differs from the first on that we not relying on third-party HoneyPot tools. This suggestion involves the implementation of a cloud native HoneyNet through the use Kubernetes and custom HoneyPot containers; this, however, brings increased levels of complexity and higher possibility of errors. Kubernetes would be used for container orchestration and deployment automation, with an MTD scheme being applied on top for increased covertness.

Finally, having an architecture that could be applied on the open Internet would allow us to prove (or disprove) our hypothesis that increased HoneyPot covertness brings higher levels of quality of captured data.

Bibliography

[1] Faucet documentation. URL: https://docs.faucet.nz/en/latest/, Last consulted in Dez 2021.

[2] Opendaylight controller documentation, . URL: https://docs.opendaylight.org/en/stable-sulfur/user-guide/opendaylight-controller-overview.html, Last consulted in Dez 2022.

[3] Ovs documentation, . URL: https://docs.openvswitch.org/en/latest/, Last consulted in Dez 2022.

[4] Ryu documentation. URL: https://ryu.readthedocs.io/en/latest/, Last consulted in Dez 2022.

[5] Ryu spanning tree documentation. URL: https://osrg.github.io/ryu-book/en/html/spanning_tree.html, Last consulted in Nov 2022.

[6] Onos documentation. URL: https://wiki.onosproject.org/display/ONOS/ONOS+Documentation, Last consulted in Nov 2022.

[7] T-pot honeypot framework installation " cyber-99, Jan 2020. URL: https://cyber-99.co.uk/t-pot-honeypot-framework-installation, Last consulted in Jul 2022.

[8] Odl platform overview, Mar 2021. https://www.opendaylight.org/about/platform-overview, Last consulted in Feb 2022.

[9] Aelth. Ddospot documentation. https://github.com/aelth/ddospot, Last consulted in Mar 2022.

[10] Ehab Al-Shaer, Qi Duan, and Jafar Haadi Jafarian. Random host mutation for moving target defense. In *International Conference on Security and Privacy in Communication Systems*, pages 310–327. Springer, 2013.

[11] Alaa Taima Albu-Salih. Performance evaluation of ryu controller in software defined networks. *Journal of Al-Qadisiyah for computer science and mathematics*, 14(1):Page–1, 2022.

[12] Spyros Antonatos, Periklis Akritidis, Evangelos P Markatos, and Kostas G Anagnostakis. Defending against hitlist worms using network address space randomization. In *Proceedings of the 2005 ACM workshop on Rapid malcode*, pages 30–40, 2005.

[13] Arun K Arahunashi, S Neethu, and HV Ravish Aradhya. Performance analysis of various sdn controllers in mininet emulator. In *2019 4th International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*, pages 752–756. IEEE, 2019.

[14] Michael Atighetchi, Partha Pal, Franklin Webber, and Christopher Jones. Adaptive use of network-centric mechanisms in cyber-defense. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003.*, pages 183–192. IEEE, 2003.

[15] Fouaz Bouguerra. Data centre networking: What is ovs?, Nov 2021.

[16] Fouaz Bouguerra. Data centre networking: What is sdn?, Oct 2021.

[17] Ian Buchanan. Containers vs virtual machines.

[18] ComputerNetworkingNotes. Differences between emulation and simulation, Jul 2022.

[19] Containernet Project Contributors. Containernet, .

[20] Mininet Project Contributors. Mininet, .

[21] Cowrie. Cowrie/cowrie: Cowrie ssh/telnet honeypot https://cowrie.readthedocs.io.

[22] Wenjun Fan, David Fernández, and Zhihui Du. Versatile virtual honeynet management framework. *IET Information Security*, 11(1):38–45, 2017.

[23] Paul Goransson and Chuck Black. The openflow specification. *Software Defined Networks*, page 81–118, 2014. doi:10.1016/b978-0-12-416675-2.00005-x.

[24] Chakshu Gupta. Honeykube: designing a honeypot using microservices-based architecture. Master's thesis, University of Twente, 2021.

[25] Boldizsár Bencsáth Gábor Pék and Levente Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. *European Workshop on System Security*, 14(1):Page–1, 2011.

[26] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132, 2012.

[27] Alexander Kedrowitsch, Danfeng Yao, Gang Wang, and Kirk Cameron. A first look: Using linux containers for deceptive honeypots. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, pages 15–22, 2017.

[28] Dorene Kewley, Russ Fink, John Lowry, and Mike Dean. Dynamic approaches to thwart adversary intelligence gathering. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 1, pages 176–185. IEEE, 2001.

[29] Changting Lin, Chunming Wu, Min Huang, Zhenyu Wen, and Qiumei Cheng. Adaptive ip mutation: A proactive approach for defending against worm propagation. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 61–66. IEEE, 2016.

[30] Douglas C MacFarland and Craig A Shue. The sdn shuffle: creating a moving-target defense using host-based software-defined networking. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 37–41, 2015.

[31] Nogol Memari, SJ Hashim, and K Samsudin. Container based virtual honeynet for increased network security. In *2015 5th National Symposium on Information Technology: Towards New Smart World (NSITNSW)*, pages 1–6. IEEE, 2015.

[32] Honeynet Project. Know your enemy: Genii honeynets.

[33] VR Sudarsana Raju. Sdn controllers comparison. In *Proceedings of science globe international conference*, 2018.

[34] Dubravko Sever and Tonimir Kisasondi. Efficiency and security of docker based honeypot systems. *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018. doi:10.23919/mipro.2018.8400212.

[35] Danny Velasco Silva and Glen D Rodríguez Rafael. A review of the current state of honeynet architectures and tools. *International Journal of Security and Networks*, 12(4):255–272, 2017.

[36] Lance Spitzner. The honeynet project: Trapping the hackers. *IEEE Security & Privacy*, 1 (2):15–23, 2003.

[37] Lance Spitzner. *Honeypots: tracking hackers*, volume 1. Addison-Wesley Reading, 2003.

[38] Andrew Warfield Jason Franklin Tal Garfinkel, Keith Adams. Compatibility is not transparency: Vmm detection myths and realities. *Stanford University*, 14(1):Page–1, 2007.

[39] Telekom-Security. Telekom-security/tpotce: t-pot - the all in one honeypot platform.

[40] Christopher Kruegel Thomas Raffetseder and Engin Kirda. Detecting system emulators. *ICISC*, 14(1):Page–1, 2007.

[41] Abhilash Verma. Production honeypots: An organization's view. *SANS Security Essentials*, 1:28, 2003.

[42] José Manuel Sanchez Vilchez and David Espinel Sarmiento. Fault tolerance comparison of onos and opendaylight sdn controllers. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 277–282. IEEE, 2018.

[43] Bojan Zdrnja. Second-generation (genii) honeypots. 2000.