Universidade de Aveiro
2019

Departamento de Eletrónica,
Telecomunicações e Informática

Ana Maria Ferreira Gameiro

Segurança em novos conceitos de Redes Definidas por Software

Security in new concepts of Software Defined Networks

Ana Maria Ferreira Gameiro

Segurança em novos conceitos de Redes Definidas por Software

Security in new concepts of Software Defined Networks

"*There is no failure except in no longer trying.*"

— Elbert Hubbard

Universidade de Aveiro
2019

Departamento de Eletrónica,
Telecomunicações e Informática

Ana Maria Ferreira Gameiro

Segurança em novos conceitos de Redes Definidas por Software

Security in new concepts of Software Defined Networks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor João Paulo Silva Barraca, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Daniel Nunes Corujo, Professor investigador doutorado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Agradecimento ao IT UID/EEA/50008/2019.

Agradecimento ao Projeto Smart EnterCom Nr. 021949 do COMPETE 2020 (POCI-01-0247-FEDER-021949).

o júri / the jury

presidente / president

Professor Doutor André Ventura da Cruz Marnoto Zúquete
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Professor Doutor Tiago José dos Santos Martins da Cruz
Professor Auxiliar da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Professor Doutor João Paulo Silva Barraca
Professor Auxiliar da Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço aos meus orientadores, Doutor João Paulo Barraca e Doutor Daniel Corujo, por toda a disponibilidade, colaboração e orientação, tornando possível a elaboração desta dissertação assim como a aprendizagem de novos conhecimentos. Agradeço também ao Vítor Cunha por todo o apoio e ajuda ao longo do desenvolvimento da dissertação, estando sempre disponível para o esclarecimento de dúvidas.
Agradeço também à minha família, aos meus amigos e aos meus colegas de curso, que sempre me apoiaram nesta jornada.

Palavras Chave SDN, honeypot, IDS, controlador SDN

Resumo

Atualmente, tudo é gravado e partilhado através de dispositivos eletrónicos. Contudo, a maioria das pessoas não põe em prática medidas de segurança para salvaguardar a sua informação. Este é um grande problema no ambiente empresarial visto que as pessoas vão trabalhar e ligam os seus dispositivos à rede da sua empresa. É apenas necessário um smartphone ou computador infectado para comprometer a empresa inteira. Por isso, segurança é um ponto crucial na gestão de uma empresa, especialmente em relação à sua rede. Mas isto não é algo fácil de se concretizar uma vez que o campo da segurança está constantemente a mudar e a evoluir para superar problemas e vulnerabilidades que são descobertas todos os dias.

O trabalho a ser apresentado é um dos que tenta superar alguns destes problemas. Este usa conceitos de Redes Definidas por Software (SDN) para melhorar a segurança de rede no ambiente empresarial.

Esta tese apresenta um sistema com vários elementos de segurança, cada um abordando um problema diferente. Fazendo uso da tecnologia de honeypots, o sistema desenvolvido não identifica apenas ataques mas consegue também criar chamarizes para atacantes de modo a extrair informação para melhorar as medidas de segurança já existentes.

Keywords SDN, honeypot, IDS, SDN controller

Abstract

Nowadays, everything is recorded and shared using electronic devices. However, most people don't apply security measures to safeguard their information. This is a major problem in the enterprise environment since people go to work and connect their devices to their company's network. It takes only one infected smartphone or computer to compromise the entire company. Therefore, security is a crucial point in a business' management, especially regarding its network. But this is not an easy thing to do since the security field is constantly changing and evolving to overcome problems and vulnerabilities that are discovered every day.

The work to be presented is one that attempts to overcome some of these problems. It uses Software Defined Networking (SDN) concepts to improve network security in the enterprise environment.

This thesis presents a system with several security elements, each one tackling a different problem. Making use of honeypots technology, the system developed not only identifies attacks but can also create decoys to attackers in order to extract information to improve the already existing security measures.

Contents

List of Figures

List of Tables

List of Acronyms

AMI	Advanced Metering Infrastructure
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BA	Binding-Aware
BGP	Border Gateway Protocol
BI	Binding-Independent
CAPEC	Common Attack Pattern Enumeration and Classification
CAPEX	Capital Expenditure
CLI	Command Line Interface
CM	Captor Manager
CPU	Central Processing Unit
CRUD	Create, Read, Update, and Delete
CUDA	Computer Unified Device Architecture
DHCP	Dynamic Host Configuration Protocol
DM	Decoy Manager
DNS	Domain Name System
DOM	Document Object Model
DoS	Denial of Service
DDoS	Distributed Denial of Service
GPU	Graphics Processing Unit
GRE	Generic Routing Encapsulation
GUI	Graphical User Interface
HIDS	Host-based Intrusion Detection System
HIH	High Interaction Honeypot
HOT	Heat Orchestration Template
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IANA	Internet Assigned Numbers Authority
ID	Identifier/Identification
IDPS	Intrusion Detection and Prevention System
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
IPS	Intrusion Prevention System
IPv4	IP Version 4
ISP	Internet Service Provider
IT	Information Technology
KVM	Kernel-based Virtual Machine
LBL	Lawrence Berkeley National Laboratory
LIH	Low Interaction Honeypot
LXC	LinuX Containers
LXD	Linux Container Daemon
MAC	Media Access Control
MD-SAL	Model-Driven Service Abstraction Layer
MDSE	Model-Driven Software Engineering
MIH	Medium Interaction Honeypot
MITM	Man-in-the-Middle
MTD	Moving Target Defense
NAT	Network Address Translation
NBI	Northbound Interface
NETCONF	Network Configuration Protocol
NFV	Network Function Virtualization
NIDS	Network-based Intrusion Detection System
NOS	Network Operating System
OC	Orchestrator Core
ODL	OpenDaylight
OISF	Open Information Security Foundation
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OS	Operating System
OSGi	Open Services Gateway Initiative

OVS	Open vSwitch
PaaS	Platform-as-a-Service
PCEP	Path Computation Element Configuration Protocol
POP	Post Office Protocol
QoS	Quality of Service
REST	Representational State Transfer
RESTCONF	Representational State Transfer Configuration Protocol
RPC	Remote Procedure Call
SAL	Service Abstraction Layer
SBI	Southbound Interface
SDN	Software Defined Networking
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
VLAN	Virtual Local Area Network
VM	Virtual Machine
VxLAN	Virtual Extensible Local Area Network
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
YANG	Yet Another Next Generation

CHAPTER 1

Introduction

Currently, we are in the digital era. The technology is so evolved that the world is ruled by the digital contact instead of the physical one. Every day we are bombarded with new technologies and with new functions and improvements of existing technologies. This translates into an economy laid on digital tracks. That is, our homes, our jobs, our lives are recorded and they leave very clear tracks in every digital component with whom they contact, either being networks, databases, web pages, servers, or others.

The security field is always a crucial point to take into account in the digital context, especially in networks, since these are the ones who carry data everywhere. Without at least a few security components in a network, this is not much use, because no data that goes through this network will be protected.

As already mentioned, the current technology is quite developed, being that there are several solutions developed for the several problems found in the different technologies.

The SDN concept is fairly recent, and there already are quite some developed works about it. Some of these works have the goal to improve the concept and develop it, and others take the concept as a means to deepen existing works and to develop solutions in other areas.

1.1 Motivation

There already are several works based on the improvement of the SDN concept, mainly relative to its security. However, there aren't that many works that are based on taking advantage of SDN as a means to develop solutions in the security area. Since this area is so wide and is in constant development, it is believed that the advantages of SDN can be translated into the discovery of new security solutions, which can be more agile and dynamic.

SDN separates the network in several planes, splitting the control and forwarding logic into two different ones. This allows the existence of a central control unit responsible for the decision process, as well as the implementation of the results in devices scattered across a network. These features provide network administrators tools for easier and improved ways to manage networks, particularly, large-scale ones.

The SDN concept is relatively new, which means that new approaches, implementations, and solutions using it are currently being developed, and, considering its advantages, it is being thoroughly explored.

1.2 Objectives

This thesis aims to develop a security solution that takes advantage of SDN and be capable of showing that the concept can be used by the networks' security area as a means to the discovery of new solutions.

The thesis also intends to show that SDN can leverage the development of solutions that automate a network's management and security, that is, to facilitate the work of network administrators.

1.3 Contribution

The work developed demonstrates that is possible the implementation of security strategies in a network in order to not only block attacks as also inspect new or unknown attacks with the goal to implement new security mechanism that can detect those.

A solution is proposed to allow a more automatic and dynamic manner to manage networks, as well as provide means to investigate attacks. This is achieved by using SDN and other existing tools to manipulate traffic and gather relevant information.

The solution was implemented with several components, some directly regarding security while others with more generic functions, with a SDN network connecting them all together. It includes a component responsible for classifying traffic and another responsible for implementing an isolated setting to study attacks. The solution allows traffic classification, whether to provide Quality of Service (QoS) measures or for security reasons, such as identifying attacks. An SDN controller is the central point of the network created, and also the component responsible for orchestration and rule implementation in the switches. The controller and the traffic classification component communicate with each other allowing the exchange of information to manage the network and to implement security measures. The controller also communicates with the component responsible for the implementation of an isolated setting, the location of honeypots that allow the deception of attackers and the subsequent gathering of information regarding attacks.

All of the features and components of the solution's implementation allow a network administrator to manage a network more automatically and dynamically, in addition, to provide means to improve security measures and prevent future attacks.

1.4 Structure

The structure of the remainder of the thesis is as followed:

- Chapter 2: State-of-the-Art. This chapter describes the state-of-the-art concerning the several themes encompassed by the presented problem and the respective solution.

- Chapter 3: Solution Architecture. In this chapter, the problem and its context are described, being presented the requirements, use cases, and its architecture, as well as the respective modules and functionalities.
- Chapter 4: Proof of Concept. This chapter encompasses the description of the implemented solution, thus being described in detail, namely, the technologies used, the architecture and the components developed.
- Chapter 5: Results and Evaluation. In this chapter, the results obtained by the solution and an evaluation of the same, in the context of the problem, are presented.
- Chapter 6: Conclusion and Future Work. This chapter presents a conclusion of the developed work and also some future works that can develop the solution proposed.

CHAPTER 2

State-of-the-art

The human being is, by nature, unsatisfied, which is why we are constantly trying to improve ourselves and what surrounds us. Regarding technology, we have seen major improvements over the years, as we try our best to achieve what was thought impossible. But progress (the discovery of something) is not the only necessity, we also want to bring these innovations to every corner of the world, and to accomplish this, technologies, particularly network-related, have to evolve.

2.1 Software Defined Networking

Network services and infrastructure have developed rapidly in recent years, to follow the constant demand. This growth is reflected in the need for applications capable of rapid real-time network provisioning, optimized traffic management and virtualization of shared resources, which prompted the conceptualization and adoption of new networking models. One of the leading networking paradigms (even though it is relatively new) is Software Defined Networking (SDN), which seeks to simplify network management by decoupling network control logic from the underlying hardware and introduces real-time network programmability, enabling innovation [1].

The explosion of mobile devices and content, server virtualization, and advent of cloud services were some of the reasons that drove the networking industry to reexamine traditional network architectures, since they started to be unsuitable for the requirements of enterprises, carriers, and end-users, such as dynamic computing and storage, which could not be fully achieved with a static architecture like one of conventional networks. Some of the trends that drove the need for a new network paradigm were the changing of traffic patterns (due to more communication between server machines, users connecting from anywhere at any time, and the use of utility computing model by enterprises, which results in additional traffic across the wide-area network), the "consumerization of Information Technology (IT)" (it has to accommodate personal devices in corporate networks, while protecting corporate data, intellectual property and meeting compliance mandates), the rise of cloud services (implying

the provisioning of resources à la carte) and the existence of "big data" (which requires more bandwidth to process and handling it). The attempts to apply this trends showed some limitations concerning the existent network technologies, which include complexity (which lead to stasis, since IT would seek to minimize the risk of service disruption), inconsistent policies (since there were thousands of devices and mechanisms in a network, to configure all of them with the same policies would be a very difficult task), inability to scale (because all of the devices are configured manually by network administrators, and companies networks are very vast and complex) and vendor dependence (the lack of standard results in open interfaces, which limits the ability of network operators to tailor the network to their individual environments). All of these limitations brought the industry to a critical moment, where the traditional network architectures could not possibly accommodate these changes, and so, an answer was found: the creation of Software-Defined Networking (SDN) architecture and the development of associated standards [2].

Although the fundamental requirement of introducing network programmability exists since the beginning of computer networks, the term "software-defined networking" was only first used in 2009 [3] to describe work done in developing a standard called OpenFlow, which would give network engineers access to flow tables in switches and routers from external computers, making it possible to change network layout and traffic flow in real-time. [1] From this moment forward, the term continued to be used and the underlying technology starts to evolve, resulting in a lot of different definitions, each of which with its own merits. [4] However, one that stands out is the one from the Open Networking Foundation (Open Networking Foundation (ONF)) [5], a non-profit industry consortium that is leading the advancement of SDN and standardizing critical elements of its architecture [2], which defines SDN as "the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices" [6]. Unlike before, where the network devices comprised the two planes mentioned, in this architecture the network control is directly programmable, which can abstract the underlying infrastructure for applications and networks services, making the network seems like a logical or virtual entity [2].

2.1.1 Architecture of Softwarized Networks

To better understand how SDN works and how it was an improvement over existing network models, its architecture has to be considered, along with some of the latter. To compare them is necessary to consider the existence of three planes: data, control and application planes, being SDN the one having all three in its architecture. The control plane, as the name states, controls the data plane, since it is responsible for its configuration and for determining the paths to be used for data flows. This information is passed to the hardware that constitutes the data plane, so they can forward the data using the paths [7].

Depending on the deployment of control and data planes, networking paradigms can be divided into three types: traditional, hybrid and SDN, all of which will be further explained and whose representation can be seen in Figure 2.1. Traditional networks are hardware-centric since the switches usually have their own control and data planes and support manufacturer-

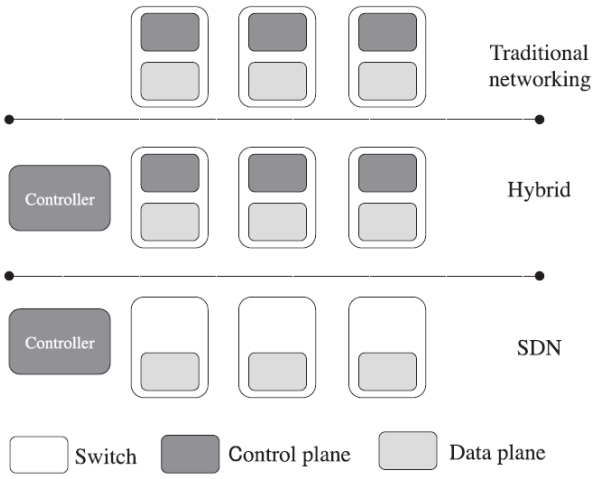


Figure 2.1: Networking Paradigms representation. Source: [8]

specific control interfaces; in other words, they are closed systems. Consequently, when a forwarding policy is defined, there is not much one can do to modify it, being the only way to change the configuration of the devices where it is. Because of this, the networks are very difficult to scale, since the deployment of new protocols and services, or of new versions of existing ones, requires the updating or replacement of all the switches in the network, which, in an enterprise environment, may mean thousands of equipment.
Regarding the SDN paradigm, it opposes the traditional given that it separates the control and data planes, being the former represented by a centralized controller and the latter comprising switches. This setting permits an easier deployment because the controller receives instructions and he is the one that programs the switches, being able to configure several of these simultaneously and making them just forwarding devices. Thereby, the centralized controller can optimize flow management and support service-user requirements of scalability and flexibility, due to the fact that he can gather information about the entire network and, therefore, knowing its topology.
At last, there can exist another paradigm, the hybrid one, which is a combination of the previous two. This model represents a network capable of supporting both distributed and centralized control because it allows the programming of switches through the controller or through their individual software (by using traditional operation procedures and protocols) [7], [8].

SDN is a network architecture with some basic principles. These will be described now, although some of the terms used here will only be explained later.
First of all, there is the decoupling of control and data planes, done by removing the control functionality from the network devices, which become simple (packet) forwarding elements. However, the control must necessarily be applied within data plane systems, the reason why the interface between the SDN controller and the network elements (data plane devices) should be defined so the former can delegate significant functionality to the latter while remaining aware of their state.

Another basic principle is logically centralized control. Besides separating the control logic from the network elements, this is moved to an external entity, named SDN controller or Network Operating System (NOS). Comparing to the traditional networks, where the control was attached to the data plane in the network elements, a centralized controller has a wider perspective of the resources under its control (through an abstract network view), and, therefore, can potentially make better decisions about how to deploy them. Both the decoupling of planes and the centralization of control improve the scalability of a network, allowing increasingly global (but less detailed) views of network resources.
The exposure fo abstract network resources and state to external applications can be considered another principle of SDN. Applications may exist at any level of abstraction or granularity, and, just like controllers, they may relate to each other as peers or as clients and servers. They run on top of the controller or NOS and interact with the underlying data plane devices to program the network.
There is another feature of SDN that can be considered as a part of its definition: the type of forwarding decisions. In traditional networks, the forwarding decisions are destination-based, hence all of the traffic's information is disregarded except its destination. Nevertheless, in SDN the forwarding decisions are flow-based. In the SDN/OpenFlow context, a flow is a sequence of packets between a source and a destination, but the term is broadly defined by a set of packet filed values acting as a match (filter) criterion and a set of actions (instructions). Thus, the use of flows to program forwarding decision allows unifying the behavior of different types of network devices, including routers, switches, firewalls, and middleboxes, and enables unprecedented flexibility, limited only to the capabilities of the implemented flow tables [9], [10].

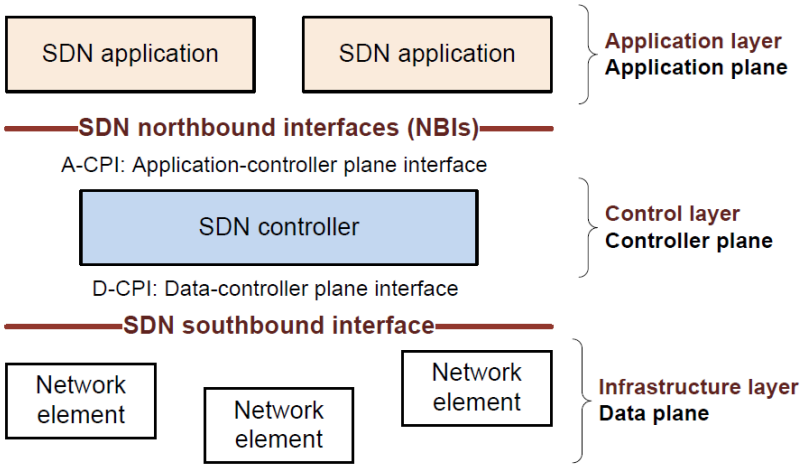


Figure 2.2: SDN Architecture. Source: [9]

The SDN framework has three main components: the data, control and application planes, as can be seen in figure 2.2, where the "controller plane" corresponds to the control plane. These are the most common terminologies used in literature, although the ONF [6] uses the terms infrastructure, control and application layer (as shown by the red text on the right

side of said figure). Each of the mentioned planes has well-defined boundaries, a specific role, and also relevant Application Programming Interfaces (APIs) to communicate with adjacent planes, all of which will be further explained [1].

The data plane comprises network elements, namely switching devices (which can be switches, routers, virtual networking equipment, and others), whose sole purpose is to forward network traffic as efficiently as possible, based on a certain set of forwarding rules instructed by the control plane. For the network elements to receive instructions, they have to expose their capabilities via interfaces, which are called Southbound Interfaces (SBIs). These interfaces, located between the control plane and the data plane, are APIs whose responsibility is the communication between the planes. Several vendors, as well as the ONF, choose and support the OpenFlow protocol as the southbound communication protocol used between the referred planes.

Located between the data and application planes, is the control plane, which bridges the former two planes, via its interfaces, one of which was previously mentioned (SBI). The other interface is called Northbound Interface (NBI), and is located between the control and application planes. While the SBI specifies functions for controllers to access switching devices, the NBI provides service points (through APIs) for applications to interact with the control plane. An SDN controller, placed in the control plane, has the task to translate the applications' requirements and to exert control over the network elements while providing relevant information up to the SDN applications. In other words, the control plane is responsible for making decisions regarding the traffic in the network, based on end-user application requirements, and for communicating the resulting network policies to the data plane. It is considered that the control plane is constituted by, at least, one SDN controller, but, based on the size of the network, there can be multiple controllers. In this case, they will have two additional interfaces, east and west (also called, eastbound and westbound), used for communication between the controllers, since they have to share network information and coordinate their decision-making processes. Given the described features, one may conclude that the SDN controller is the central component of a control plane, which provides a clear and centralized view of the underlying network. This centralized management of network elements gives additional leverage to administrators providing them vital statistics of existing network conditions to adapt service quality and customize network topology as needed, making it a powerful network management tool to fine-tune performance.

Finally, the application plane, composed by, as the name explains, SDN applications (network-specific and business ones), which are designed to fulfill user requirements. Via an abstract view of the underlying network presented by the controller through the NBI APIs, the applications can access network status information reported from the switching devices (data plane), make system tuning decisions, and carry out these decision by communicating their network requirements to the controller to do so, via the same interface (NBI) [1], [4], [9].

2.1.2 Data Plane: Forwarding Devices

Traditional networks, as well as SDN ones, are composed of a set of networking equipment (switches, routers, and middlebox appliances). However, in SDN the network intelligence is stripped from the network devices and relocated to a logically-centralized control system, making a separation between the data and control planes. This separation results in a data plane composed simply by forwarding devices, since they do not have embedded control or software to make autonomous decisions, making them specialized in packet forwarding. In SDN terminology, these forwarding devices are commonly referred to simply as "switches". The switches can be hardware or software elements, with an open interface which will be used to connect with the control plane (controller), so they can receive rules for packet forwarding and send their network status, for decision-making. Thereby, SDN switching devices are simpler and should be easier to manufacture, since the reduced complexity (from devices with full intelligence and decision-making capabilities to simply forwarding devices) should lead to a low-cost solution [4], [10], [11].

As said before, the forwarding devices receive instructions from the control plane (controller) through its southbound interface, composed by APIs. These southbound APIs enable the SDN controller to dynamically make network changes required in real-time, providing network control. To establish this communication, there are some protocols that can be used, namely Cisco OpFlex [12], Extensible Messaging and Presence Protocol (XMPP) [13] and OpenFlow [14].

Cisco OpFlex has gained momentum among southbound APIs, facilitating the communication between the control and data planes and aiming to become a standard policy implementing language across physical and virtual environments. Cisco OpFlex protocol focuses more on the implementation and definition of policies, unlike the OpenFlow protocol which centralizes all the network control functions using the SDN controller [12]. The framework focuses on policies to avoid network bottleneck, which can happen by controller scalability and control channel communication, and also to push some level of intelligence to the devices, using legacy protocols. Cisco OpFlex allows the definition of policies within a logical, centralized repository, located in the SDN controller, and which then will be communicated and enforced on the switches by the OpFlex protocol. This protocol allows not only the communication from the controller to the switches but also the communication in the opposite direction, enabling the exchange of policies, networking events and statistical monitoring information, which may be used to make networking adjustments since it is real-time information regarding the network. The switches have an OpFlex agent that supports the Cisco OpFlex protocol, and this relies on traditional and distributed network control protocols so it can push commands to the embedded agents in the former. This protocol has yet to be standardized, the reason why Cisco submitted it to the Internet Engineering Task Force (IETF) standardization process, and why several vendors are working also towards this and to increase the adoption of the protocol [1].

XMPP was originally designed as a general communications protocol offering messaging and presence information exchange among clients through centralized servers [13]. Considering the

Simple Mail Transfer Protocol (SMTP), XMPP is quite similar, having two differences from the former: its schema is extensible through Extensible Markup Language (XML) encoding for user customization, and it provides near real-time communication. The XMPP works in a client-server model, where each XMPP client is identified by an Identifier/Identification (ID), which could be something simple, like an email address. The client machines set up connections with a central server to notify their presence, so the latter can maintain contact addresses and may let other contacts know that a particular client is online. Regarding the communication between clients, it can be done through pushed chat messages, as opposed to polling, which is used in SMTP/Post Office Protocol (POP) emails. This protocol found new applications in hybrid SDN, Internet of Things (IoT), and data centers, and is used for managing individual network devices, by having these run XMPP clients which respond to XMPP messages containing Command Line Interface (CLI) management requests. In the case of data centers, every object (virtual machine, switch, hypervisor) can have an XMPP client module, which awaits instructions from the XMPP server for authentication and traffic forwarding, that, once received, the client can update their configurations as requested by the server. The XMPP protocol is standardized by the IETF (RFC 6121 [1]) and follows an open systems development and application approach allowing interoperability among multiple infrastructures. However, it still has a few weaknesses, namely does not guarantee QoS of message exchanges between the XMPP client and the server (which, if needed, has to be built on top of the protocol) and does not allow end-to-end encryption (a fundamental requirement in modern and multi-tenanted network architectures). Hereupon, the protocol is still being developed, at least to deal with the message delivery problem [1].

OpenFlow

As said before, OpenFlow protocol [14] is the southbound communication protocol chosen and supported by several vendors and the ONF to use for communication between the control and data planes, being maintained and updated by ONF itself. Besides being the first southbound communication interface (having been developed in the early stages of SDN paradigm), OpenFlow is also the most prominent one and is meant to communicate control messages between the SDN controller and networking components in the data plane. A typical OpenFlow compatible switch contains one or more flow tables (where a path through them defines how packets should be handled), and each entry of the tables has three parts: a matching rule, actions to be executed on matching packets, and counters that keep statistics of matching packets. The tables are used to match incoming flows (and packets) with policy actions, such as prioritization, queueing and packet dropping. When a packet arrives at a switch, a lookup process starts in the first table, by trying to match some rule in it, based on the priorities of the flow entries. If a match is found, the packet is sent to its destination (by the outgoing port), otherwise is sent to another flow table as dictated by network control logic instructed by the controller (or, in other words, the corresponding action of the matched

[1] `https://tools.ietf.org/html/rfc6121`

flow entry is executed). If a match for the packet is not found in any of the tables (what is called a "table miss"), the packet is dropped or a request for processing instructions is sent to the controller. To avoid "table misses", it is common to install a default rule which tells the switch to send the packet to the controller. The packets traverse flow tables (which are assigned numbers in sequence) in the form of metadata communicated between different tables, and generally, they can only be sent from a table of lower sequence to one of higher, to assure that packets move in a forward direction instead of backward in the switch. The flow entries can also point the packet to particular group actions, which allow a further set of complex policies to be executed on packets (when compared to flow tables), such as route aggregation and multicasting. Recapping, packets arriving at the ingress port of a switch are generically processed in the following sequence: first, the highest priority matching flow entry (entries from top to bottom) is found in the first flow table, based on ingress port, metadata and packet headers; then, the relevant instructions are applied; finally, the match data and respective action set are sent to the next table for further processing. The instructions referred can be one of the three: modify the packet as instructed in the actions list and/or transmit through an output port; update the action set by adding/deleting actions in the action list; update metadata. The fundamental difference between an action list and action set is the time of execution, given that an action list is executed as soon as packet's data leaves the flow table, to make necessary changes to this data; whereas an action set keeps accumulating and is executed once it traverses all relevant flow tables. The flow tables are manipulated by the SDN controller, which can do it in two manners: in real-time, reactively (for example, if a packet's forwarding path is unknown and a switch sends a message to the controller asking for forwarding information); or proactively, by sending complete flow entries based on the requirements dictated by higher applications residing in the application plane. An OpenFlow compliant switch maintains a Transport Layer Security (TLS) control channel with the SDN controller, and periodically sends keep-alive "hello" messages to communicate state information. Also, to ensure reliability in message delivery between the controller and the switch, the Transmission Control Protocol (TCP) protocol is used, where the well-known ports for OpenFlow traffic are 6633 and 6653 (official Internet Assigned Numbers Authority (IANA) ports since 2013-07-18). OpenFlow versions have evolved over the years, with some version introducing new match fields, and offering bug fixes and enhancements, being v1.5 the latest version available [1], [10].

SDN is built (conceptually) on top of open and standard interfaces (e.g. OpenFlow), which is crucial for ensuring configuration and communication compatibility and interoperability among different data and control plane devices. This is possible because open interfaces enable controller entities to dynamically program heterogeneous forwarding devices, something that is very difficult in traditional networks due to the existence of a large variety of proprietary and closed interfaces, and also due to the distributed nature of the control plane [10]. Concerning an SDN network that uses OpenFlow protocol (which, as stated, are the most of them), the switches can be of two types: pure OpenFlow switches, which have no legacy features or on-board control, completely relying on a controller for forwarding decisions; or hybrid switches,

Switch	Implementation	Category
Open vSwitch [15]	C/Python	Software
Indigo [16]	C	Software
OpenFlowJ [17]	Java	Software
OpenFaucet [18]	Python	Software
ofsoftswitch13 [19]	C/C++	Software
Pantou [20]	C	Software
Oflib-node [21]	JavaScript	Software
OpenFlow Reference	C	Software
Pica8 [22]	C	Physical and Software
A10 Networks - AX Series [23]	Proprietary	Physical and Software
Big Switch Networks - Big Virtual Switch [24]	Proprietary	Physical and Software
Brocade ADX Series [25]	Proprietary	Physical and Software
NEC ProgrammableFlow Switch Series [26]	Proprietary	Physical and Software
ADVA Optical - FSP 150 & 3000 [27]	Proprietary	Physical
IBM RackSwitch G8264 [28]	Proprietary	Physical
HP 2920, 3500, 3800, 5400 series [29]	Proprietary	Physical
Juniper Junos MX, EX, QFX Series [30]	Proprietary	Physical

Table 2.1: Common OpenFlow Compliant Switches and Standalone Stacks. Source: [1]

which support OpenFlow in addition to traditional operation and protocols [11]. Besides this classification, there are others, namely regarding hardware implementation (on general PC hardware, on Open Network Hardware or on vendor's switch) [4], ownership (proprietary or open-source) [8], or type of implementation (software or physical) [1]. Regardless of the classification used for switching devices, Table 2.1 presents some of the most common OpenFlow compliant switches, and a brief overview of the most notable software switches is provided.

One of the most widely deployed software switches is Open vSwitch (OVS) [15]. This switch is part of the Linux Kernel since version 3.3, and its stack can both be used as a virtual switch in virtualized network topologies and has also been ported to multiple hardware/commodity switch platforms [1]. It supports multiple virtualization technologies, namely Xen/Xen Server, Kernel-based Virtual Machine (KVM), and VirtualBox [8].
The switch ofsoftswitch13 [19] runs in user space and provides support for multiple OpenFlow versions. It supports Data Path Control (Dpctl), a management utility to directly control the OpenFlow switch, that allows the addition and deletion of flows, query switch statistics and to modify flow table configurations. Nonetheless, this switch encountered some compatibility issues with the latest versions of Linux (from Ubuntu 14.04 and beyond) and its developer support has stagnated [1].
Indigo [16] is an open-source implementation of OpenFlow that can be run on a range of physical switches. Its implementation is based on the original OpenFlow reference, and it utilizes the hardware features of existing Ethernet switch Application-Specific Integrated Circuits (ASICs) to run OpenFlow pipeline at line rates [1].
PicOS by Pica8 [22] is a network operating system that allows network administrators to build

flexible and programmable networks using white-box switches with OpenFlow. It is proprietary software and allows integration of OpenFlow rules in legacy layer 2/layer 3 networks, without disrupting the existing network and creating a new one from scratch [1].
Pantou [20] modifies a commercial wireless router and access point to an OpenFlow-enabled switch. The OpenFlow protocol is implemented as an application on top of OpenWRT platform, where the platform is based on the BackFire release (Linux v2.6.32) and the OpenFlow module is based on the Stanford reference implementation in userspace. Despite being reference in a number of articles, the online page of the project is not accessible [1].
One has also to consider the existence of white box switch implementations, where the software and hardware are sold separately and the end-user is free to load an operating system of its choice [10]. This type of switches allows network operators to use off-the-shelf switching hardware in the SDN data plane. With this ability to use generic switching hardware, organizations leverage the benefits of individual components suited for specific SDN applications as opposed to investing in relatively costly all-in-one vendor solutions. The white box can have a preinstalled operating system or be a bare metal device, where the operator selects and loads software which would aid in integrating the product into a larger SDN ecosystem to facilitate networking features. The use of SDN tools in combination with off-the-shelf hardware lowers Capital Expenditure (CAPEX) and also allows a reduction in time for provisioning new services. However, to accurately configure devices for subsequent use, white-box solutions require a significant deal of expertise from administrators, having to do it sometimes without a sophisticated manufacturer after-sales support [1].

2.1.3 Control Plane: Controllers

When talking about the control plane, two terms are frequently used to name the control logic: NOS and controller. Throughout the years, both terms have been used to describe the control plane, sometimes being interchangeable, other times not, and other times not being totally clear [1], [4], [7], [10], [11], [31], [32]. To simplify, here both terms will be used interchangeably.
SDN is promised to facilitate network management and ease the burden of solving networking problems by means of the logically-centralized control offered by a NOS [33], implemented in the control plane. The control platform (NOS or controller) abstracts the lower-level details of connecting and interacting with forwarding devices, alleviating the work of developers, who do not need to care about them to define network policies [10]. Therefore, NOS/controller is the brain of the SDN architecture, which performs the control decision tasks while routing the packets, and this centralized decision capability for routing enhances the network performance. Similar to a basic Operating System (OS), the NOS should provide basic functionalities, such as execution of a program, management of input/output operation, security and protection mechanisms, and other functionalities [32]. It should also provide networking functionalities and services, such as topology related functions (network state and network topology information), shortest path forwarding, device discovery, distribution of network configuration [10], [32], to give a global view of the SDN status to the upper layer (application

plane) [31]. To summarize, the control plane is the intermediary layer between application and data planes, where a controller communicates with the applications via the northbound interface and with the switches via the southbound interface [8]. SDN controllers centralize the network intelligence, which is responsible for maintaining and applying network policies required by higher applications and services, by translating and configuring the desired policies in individual network devices. As mentioned before, once a packet arrives at a switch, it will try to find a match in its flow tables. In case of a table miss (absence of flow entry), the switch may forward the packet to the controller, which will determine the next course of action for the respective traffic flow. To carry out the computed actions (which can be the addition, deletion or modification of flow entries) in the switches, the controller uses a southbound interface (e.g. OpenFlow). To establish the control channel (which is independent of the traffic forwarding framework) between the controller and each individual switch, the former is assigned an Internet Protocol (IP) address and the latter communicate with it using a predetermined port number, through a standard TLS or TCP connection [1].

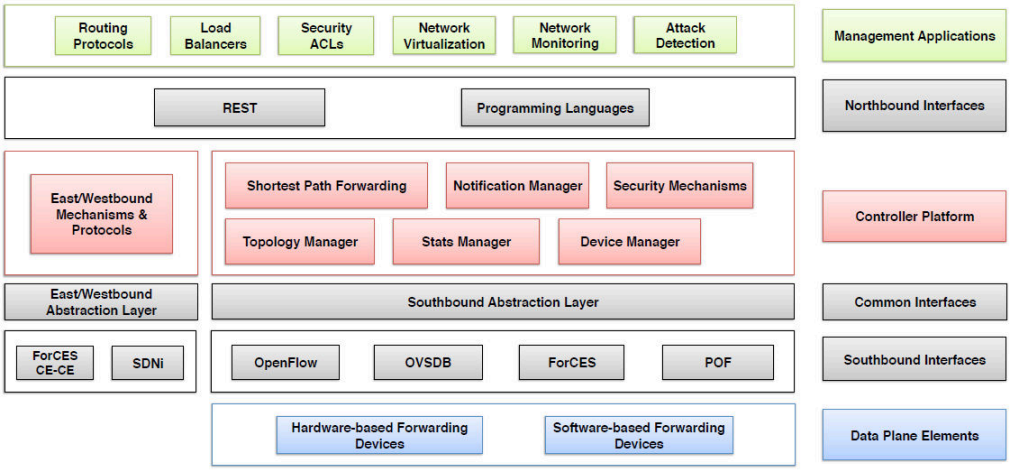


Figure 2.3: SDN control platforms: elements, services and interfaces. Source: [10]

Figure 2.3 is the result of an analysis of different SDN controllers, and provides the first attempt to clearly and systematically dissect an SDN control platform [10]. It thoroughly shows the elements, services, and interfaces that compose or that collaborate with the controller platform. Some of the elements displayed in the figure are referenced and explained in this section, while others that are a little out of scope were or will be explained in other sections (sections 2.1.2 and 2.1.4).

A controller has four limits: northbound (top limit), southbound (bottom limit), eastbound (right limit) and westbound (left limit), and each one comprises interfaces for the controller to communicate with other elements. Through the southbound, the controller establishes communication channels with the switches (in the data plane); whereas through the northbound it communicates with SDN applications (in the application plane). The east and west bounds are used in the context of distributed controllers, for several controllers to communicate with each other, which will be further explained.

In small networks, a single controller may be enough to manage it, but it represents a single point of failure [10]. Besides, regarding scalability, when the number of switches in the network starts to increase, the traffic towards the centralized controller also increases, possibly becoming a bottleneck. One solution for these problems is the implementation of multiple controller replicas [32]. To reduce the impact of a single controller failure, independent controllers can be spread across the network, each of them managing a network segment. At the same time, if the control plane availability is critical (because of a great number of requests towards the controller), a cluster of controllers can be used to achieve a higher degree of availability and/or for supporting more devices. Ultimately, a distributed controller can improve the control plane resilience, scalability and reduce the impact of problems caused by a network partition [10].

Hence, based on an architectural point of view, controllers can be categorized into two types: centralized or distributed. The first type corresponds to a single entity (controller) that manages all forwarding devices (switches) of the network [10], while the second type corresponds to several entities performing that management. Both types of architectures have to perform the same tasks (which were already described), but the distributed one is more complex since it involves more entities. As said, in comparison with the centralized architecture, the distributed one brings two advantages: scalability (a NOS can be scaled up to meet the requirements of potentially any environment, from small to large-scale networks [10]) and high performance during increase demand of requests [32], but some questions arise regarding the interaction between the several entities. There are two popular schemes for the controllers' implementation: a vertical approach (multiple controllers are in effect managed by a controller(s) at a higher layer) or a horizontal approach (controllers establish a peer-to-peer communication relationship). Regarding the switches (data plane), they may communicate with either a single or several controllers, which will depend on redundancy requirements, and, in case of failure of one controller or control channel, a switch can obtain flow forwarding instructions from another controller instance. With regard to inter controller communication to exchange routing information, it is usually served by an external legacy protocol, either Border Gateway Protocol (BGP) or Session Initiation Protocol (SIP) over TCP channels [1].

It is in the context of the distributed control plane that the east and westbound appear since they are a special case of interfaces required by distributed controllers for communication between different controllers. Each controller implements its own east/westbound API, which has: functions to import/export data between controllers, algorithms for data consistency models, and monitoring/notification capabilities (e.g. check if a controller is up or notify a take over on a set of forwarding devices). Nevertheless, it is necessary to have standard east/westbound interfaces, to identify and provide common compatibility and interoperability between different controllers. One example is SDNi [34], which defines common requirements to coordinate flow setup and exchange reachability information across multiple domains. Such protocols can be used in an orchestrated and interoperable way to create more scalable and dependable distributed control platforms. And the interoperability can be leveraged to increase the diversity of the control platform elements, which in turn increases

Name	Architecture	Language	License	OpenFlow Version
Beacon	Centralized Multi-threaded	Java	GPLv2	v1.0
Fleet	Distributed	—	—	v1.0
Floodlight	Centralized Multi-threaded	Java	Apache	v1.1
HyperFlow	Distributed	C++	—	v1.0
Maestro	Centralized Multi-threaded	Java	LGPLv2.1	v1.0
Meridian	Centralized Multi-threaded	Java	—	v1.0
MUL	Centralized Multi-threaded	C	GPLv2	v1.0, v1.3
NOX	Centralized	C++/Python	GPLv3	v1.0
NOX-MT	Centralized Multi-threaded	C++	GPLv3	v1.0
Onix	Distributed	Python; C	Commercial	v1.0
ONOS	Distributed	Java	(Open Source)	v1.0
OpenDaylight	Distributed	Java	EPLv1.0	v1.0, v1.3
PANE	Distributed	—	—	—
POX	Centralized	Python	GPLv3	v1.0
Rosemary	Centralized	—	—	v1.0
Ryu	Centralized Multi-threaded	Python	Apache 2.0	v1.0, v1.3
SMaRtLight	Distributed	Java	Apache	v1.0
SNAC	Centralized	C++	GPL	v1.0

Table 2.2: OpenFlow compliant SDN controllers. Source: [1], [10], [32]

the system robustness by reducing the probability of common faults, such as software ones [10].

Table 2.2 compiles information about some SDN OpenFlow compliant controllers (namely their architecture, the programming language used to designed them and OpenFlow versions that they support), which will be briefly described.
Beacon [35] is one of the most popular open-source centralized controllers in SDN, due to its importance in the research community. This controller is designed in Java and it provides effective memory management and proper segmentation fault and memory leaks handling. Its development had three main objectives: provide developer side productivity (which resulted in a rich set of libraries for application development), achieve high performance, and increase runtime capability to stop and start applications. Beacon has three major interfaces, each one with a different purpose. For routing purposes it has the IRoutingEngine interface, which helps design different routing modules, being an example the shortest path routing [36]. For network topology, Beacon has the ITopology interface, which contains a set of operations to retrieve information related to link discovery and link registration/deregistration. Lastly, it has the IBeaconProvider interface, which uses OpenFlowJ [17] API, to interact with the

OpenFlow switches. To deal with the runtime modularity feature, the controller uses Open Services Gateway Initiative (OSGi) [37], which provides easiness in adjusting an instance of an application at runtime, by allowing to start and stop existing applications at runtime or even create completely new ones. OSGi also provides a service registry that allows new services to register themselves so that a user can pick any service from the pool base don their requirements [32].

Fleet [38] is one of the first controllers to address the malicious administrator problem. The idea behind its development was to prevent the controller from malfunctioning due to a malicious administrator's configuration. Since 50% to 80% of network outages are caused by human errors [39], concerns arise regarding network management by administrators, given that a misconfiguration of a controller can damage the routing and forwarding network abilities of it, resulting in a degradation of the performance of the system. From the design of Fleet, two versions appeared: a single configuration approach (all administrators agree upon a single threshold value for making a high-level routing decision which is installed in corresponding switches) and a multi-configuration one (which allows a set of n different routing configuration decisions from different administrators and then selects anyone for particular switch flow based on metrics) [32].

Floodlight [40] is a controller written in Java, that can handle mixed OpenFlow and non-OpenFlow networks. It supports a broad range of virtual and physical OpenFlow switches, as well as various network services, including path discovery and link-level information discovery [8], [32].

HyperFlow [41] is the first distributed control plane designed for OpenFlow, whose original design was inspired by NOX. Its designed is distributed due to the physical availability of different controllers, but they still form a logically centralized environment. Fleet's functioning is based on a public/subscribe message system, used to send event messages towards other controllers. Each controller sends periodic messages to show its presence in the network, and if anyone of them fails to send one message within three advertisements intervals it is assumed that the respective controller failed. In this case, the switches associated with it need to migrate to some other controller to continue operating. Each controller can only program the switches which are directly controlled by it but can also control other ones. If this type of control has to happen, the controller that has to execute the operation publishes a message that contains the source controller identifier, the target switch identifier, and the local command identifier, so the controller corresponding to the switch can execute the operation [32].

Maestro [42] is a Java-based multi-threaded controller, that explores an additional throughput optimization technique to achieve maximum performance with the exploitation of parallelism. It introduces the concept of batching, where multiple requests are grouped in a single batch from users and once a thread is free it can pick any available pending request from the batch and start executing. This allows the execution of multiple flow requests by different worker threads, thus achieving parallelism. The controller provides task priority (to resolve some problems resulting from the parallelism tasks) and a rich set of interfaces and libraries [32].

Meridian [43] is a controller originally designed for the applicability of the SDN architecture in a cloud environment, since it fits perfectly, either in Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS). Its cloud network architecture (which is SDN architecture for cloud computing) is composed of three layers: abstracted API layer, network orchestration layer, and network driver layer. The Abstracted API layer is responsible for exposing the required abstract details for a network model. The network orchestration layer collaborates with the previous layer to convert logical commands into their corresponding actions, having the additional responsibility of generating network services, namely routing action and shortest path computation. The network driver layer serves as an interface between the controller and the various networking tools, thus consisting of plugins or drivers which allow devices to work accordingly to the command issued [32].

MUL [44] is a controller written in language C which supports a multi-threaded infrastructure and a multi-level northbound interface [8].

NOX [33], [45] was the first OpenFlow controller, developed to study performance characteristics of the SDN architecture, being introduced as open-source. It is written in C++ and Python and is single-threaded, not being optimized for performance. NOX has a very low flow setup throughput and a large flow setup latency [8], [31], [32].

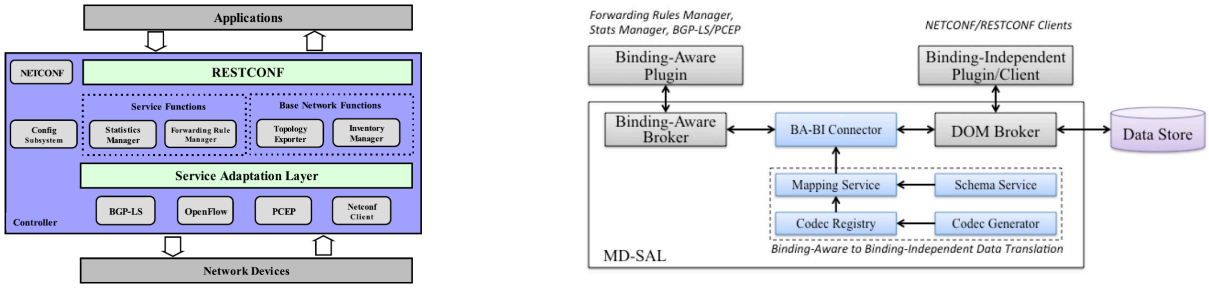
NOX-MT [45] is the successor of NOX, being modified to be multi-threaded. Being the first effort in enhancing controller performance, NOX-MT improved NOX's throughput and response time by using Input/Output (I/O) batching, improving performance by a factor of 33 in comparison to it. However, it still did not address some of Nox's problems, namely the heavy use of dynamic memory allocation and the redundancy in multiple copies for each request [31], [32].

Although Onix [46] is a controller written in C++, it allows its instance to be written in multiple languages (supporting C++, Python, and Java), which subsequently run in different processes. So, it offers a common platform, allowing the implementation of various control functions like routing, access control, and traffic engineering [32].

Open Network Operating System (ONOS) [47] is an open-source controller, designed for the SDN environment, that supports a logically centralized but physically distributed architecture. It provides an operating system resilient enough for the carrier-grade deployment of Software Defined Networks since it is mainly designed to address scalability, availability and performance issues. ONOS has two prototype specifications. Prototype 1 focuses on building a network architecture that provides a global network view with fault tolerance and scalability features, while Prototype 2 focuses on improving the performance of the overall system. This controller has a Graphical User Interface (GUI) which provides a multilayer view of the underlying network to allow operators to study network devices, links, and errors, so then they can implement policies based on that [1], [32].

ODL [48] is a controller platform founded and led by several industry giants, which offers Java-based development and deployment of carrier-grade SDN solutions [1]. This controller makes use of OSGi, same as Beacon, given that the former's architecture was inspired by the latter. ODL started with the concept of Model-Driven Software Engineering (MDSE),

a framework that defines models and their relationships with each other. The models are platform-independent to support different business policy needs, and they communicate with each other by a data modeling language. ODL uses a model-driven management protocol, being Network Configuration Protocol (NETCONF) and Representational State Transfer Configuration Protocol (RESTCONF) used as such. NETCONF supports Remote Procedure Call (RPC) operation as well as the basic operations of Create, Retrieve, Update and Delete. RESTCONF is, in some aspects, similar to the typical REST-like protocol, and in the context of ODL is responsible for providing programmatic interface over the Hypertext Transfer Protocol (HTTP). As stated, models communicate through a data modeling language, being used Yet Another Next Generation (YANG) [2] in ODL. This language is used to describe other network constructs, namely services, policies, and protocols, and it has a tree-like data structure. [3] Figure 2.4a shows the architecture of ODL controller. ODL has a set of northbound and southbound plugins, which are separated by Service Abstraction Layer (SAL). [4] The northbound plugins can include a topology exporter, a forwarding rule manager and/or a statistics manager, whereas examples of southbound plugins are OpenFlow, NETCONF client, and Path Computation Element Configuration Protocol (PCEP). By using the MDSE concept, SAL was modified to meet the ODL objectives, resulting in Model-Driven Service Abstraction Layer (MD-SAL). Figure 2.4b presents the MD-SAL design, where two brokers can be seen, being their function data handling. The Document Object Model (DOM) broker deals with architecture's runtime activity, while the Binding-Aware (BA) broker deals with Java APIs for plugins, having the BA-BI connector has a mediator between them. Regarding the MD-SAL, there are some relevant terms that have to be mentioned and briefly explained. A RPC is a call from a consumer to the provider which is processed either locally or remotely, and its connection is of the one to one type; a notification is a reply expected by the consumer from the provider side; a data store is a tree-like structure described by the YANG schemas; and a path is the location of the specific leaf in the tree (data store) [32].



(a) ODL Architecture (network view). Source: [32]

(b) MD-SAL Design. Source: [48]

Figure 2.4: ODL-related Designs

PANE [49] is an SDN controller that contains fault tolerance and resilience procedure,

[2] `https://tools.ietf.org/html/rfc8328`

[3] `https://tools.ietf.org/html/rfc6020`

[4] Some literature and ODL documentation uses the term "Adaptation" for SAL, while others use the term "Abstraction". Despite the first term appearing in Figure 2.4a, in this thesis, the second term is used.

supporting two types of failures: failure of networking elements (i.e. links, ports, switches), and failure of the controller itself. It was developed based on the idea that there should be a configuration API between the user and the control plane. PANE deals with two problems: decomposition of control and visibility in the network (which can be resolved with the use of privileges), and conflict resolution among users and their requests (which can be resolved by making use of conflict resolution operator and hierarchical flow table) [32].
POX [50] is a controller written in Python that offers OpenFlow support and a visual topology manager [1].
Rosemary [51] has a feature of controller resiliency and is designed to provide security to OpenFlow applications. This security is achieved by making a sandbox-like structure around each application, thus separating the network application's context from the controller context. This controller has a micro-NOS architecture, which has three design pillars. First, it schedules each network application separately in a different address space other than the controller. Second, it implements a resource monitoring system, one that can track resource consumption patterns of each application to find out its behavior. Finally, it introduces a permission structure for each micro-NOS instance, allowing constraints for each instance regarding libraries, resources and other parameters [32].
Ryu is a component-based SDN framework that found increased applicability in several research studies, being written in Python [1]. It provides software components with well-defined API that help developers to create new network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow (supporting versions 1.0, 1.2, 1.3, 1.4, 1.5), NETCONF and OF-config [52].
SMaRtLight [53] is designed to address fault tolerance issues in networks, such as switch or link failures in the data plane, switch-controller connection failure in the control plane, and failure of the controller. It achieves fault tolerance by having several controllers that work when needed. The overall network management is done by a single controller (called "primary controller"), and there are other controllers in the network that serve as backups. In the event of failure of the primary controller, a smooth transition is initiated, where a new primary controller is chosen from the backup ones. SMaRtLight is designed with a cache for fast access to state information [32].
SNAC [54] is an OpenFlow controller based on NOX and written in C++. It supports a graphical user interface and policy definition language [8].

2.1.4 Application Plane: Applications

As the name implies, the application plane comprises the SDN applications. They can exist at any level of abstraction and they vary in scope, with some providing comprehensive network monitoring and control solutions while others solely target a particular aspect of load balancing, security, and traffic optimization through SDN controllers [1].
However, there is not a precise distinction between applications and controller [55], [56], even being mentioned in the ONF SDN framework [9] that applications might act as an SDN controller in their own right or collaborate with one or more SDN controllers to gain exclusive

control of resources exposed by them. Consequently, controller-application interface may mean different things to different vendors, and the architectures and APIs (northbound) of SDN applications also vary between them, some having incorporated SDN controllers inside applications and others choosing to define custom northbound APIs for policy translation between controllers and their own SDN services of the application layer [1].

These problems made come to light the need for a common northbound interface, which is still an open issue. To promote application portability and interoperability among the different control platforms the northbound API has to represent an abstraction that guarantees programming language and controller independence, and, for this to be achieved, is essential the development of open and standard northbound interfaces [10]. The benefits that arise from the existence of these interfaces are significant since they allow developers from different areas of industry and research to develop a network application (as opposed to only equipment vendors), and also give network operators the ability to quickly modify or customize their network control [1]. There are two examples of efforts made to create said interfaces: NOSIX and SFNet. NOSIX [57] is one of the first examples of an attempt at standardizing the northbound API. It tries to define portable low-level (e.g. flow model) application interfaces, so the southbound APIs (such as OpenFlow) look like "device drivers". That is why NOSIX is not exactly a general-purpose northbound interface, but more like a higher-level abstraction for southbound interfaces. Another example of a northbound interface is SFNet [58], which is a high-level API that translates application requirements into lower-level service requests. Despite that, it has a limited scope, since it focuses on the use of queries to do its job, suing them to request the congestion state of the network and services (such as bandwidth reservation and multicast) [10].

As said before, some controllers can define their own northbound API (namely, Floodlight [40], Trema [59], NOX [60], Onix [46], and OpenDaylight [61]), yet each of them has its own specific definitions [10], hence the necessity for a standard northbound interface. While this work is still in progress, there are some northbound APIs that are currently used by a variety of controllers, following a brief description of two of the most popular: RESTful and Java-based OSGi [1].

Representational State Transfer [62], along with Java APIs, is the northbound interface most commonly used in controller architectures, being included in almost all major SDN controllers (namely Ryu and ODL), as well as several vendor-proprietary platforms. Its main goals are to offer scalability, generality, and independence, allowing the inclusion of intermediate components between clients and servers to achieve that, and both (clients and servers) can be developed independently or alongside). Figure 2.5 illustrates RESTful calls and the integration of the API in the SDN architecture. To allow scalability, the server component is stateless, and the clients keep track of their individual states. Every entity or global resource can be identified with a global identifier (for example, Uniform Resource Identifier (URI)), and all of them are able to respond to Create, Read, Update, and Delete (CRUD) operations. Regarding the resources, each one has a uniform interface to perform the referred operations, where the terms used are GET (read), POST (insert), PUT (write), and DELETE (remove). Despite

being one of the most used APIs for the northbound interface, REST has one major drawback: the lack of public subscriptions or live feed informing the application/service of network changes, since, like HTTP, it does not tell when a page has changed, requiring a frequent refresh. One of the solutions used by application developers is to periodically use loop calls to retrieve and subsequently post updates to individual switches based on predefined policies [1].

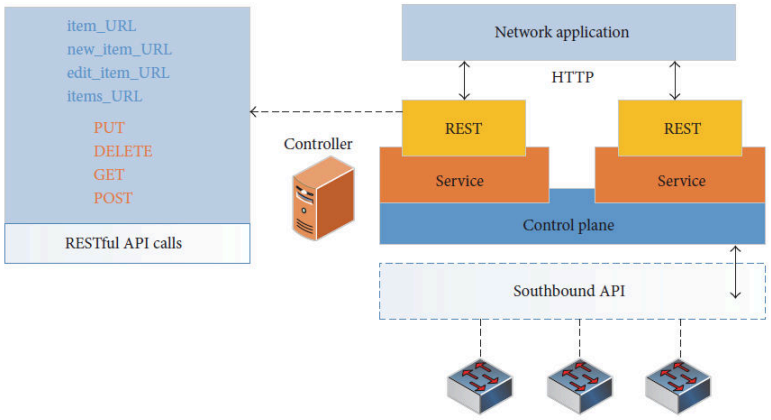


Figure 2.5: REST Application Programming Interface. Source: [1]

OSGi [63] is a set of specifications using reusable Java components, called bundles, to dynamically create applications. Through the OSGi services registry, bundles can publish their services and also find/use services of other bundles, actions represented in Figure 2.6. By using a lifecycle API, bundles can be installed, started, stopped, updated and uninstalled, and modules define how they can import and export code. The security layer, as it implies, handles security, whereas the execution environment is the one that defines what methods and classes are available in a specific platform. The services offered by the bundles have two characteristics: they have properties and they are dynamic. The first characteristic is needed to allow for a better selection of services since multiple bundles can offer the same service. The second characteristic comes from the fact that bundles can decide to withdraw their services (also because they can be installed and uninstalled on the fly), and, consequently, bundles using withdrawn ones have to stop doing so. One major example of an SDN controller platform that uses OSGi is ODL, has it is built using the framework, and other examples of controllers that support OSGi are Beacon [64], Floodlight [40], and ONOS [65]. As explained, OSGi allows controllers to start, stop, load and unload Java-based network functionalities, which is a significant improvement comparing to platforms that do not support the framework. In their case, the controller has to be stopped and restarted every time a new module has to be inserted, modified or removed, or, as an alternative to avoid the restarts, a custom REST method has to be built with all the required functionalities [1].

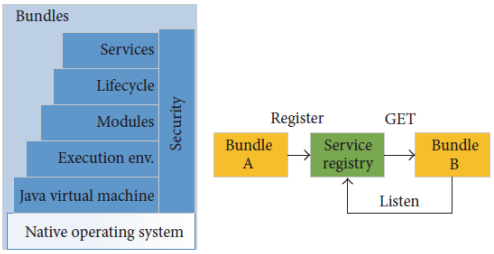


Figure 2.6: Open Services Gateway Initiative. Source: [1]

2.2 Intrusion Detection System

A network, particularly a business one, is under the constant threat of attacks since there is always someone who wants to access information that it conceals. Therefore, the management and security of a network are of utmost importance, which translates into constant monitorization and reaction to anomalous situations. Notwithstanding, it is impossible to have a network administrator do this job. Hence the introduction of IDSs.

The main objective of an IDS is to detect and restrain intrusions. An intrusion is any set of actions that aim to compromise the integrity, confidentiality or availability of a resource. The IDS can be a set of hardware or software components whose role is to detect, identify and react to non-authorized or abnormal activities in a target system [66].

2.2.1 Common Attacks

When talking about IDS, whichever type or features of it, it is important to consider some common attacks that might occur. In this case, the focus will be on network-related attacks. To learn and study common attacks, the Common Attack Pattern Enumeration and Classification (CAPEC) resource was used. [5]

The Cache Poisoning attack consists in exploiting the functionality of cache technologies to induce specific data to be cached that aids the attackers' objectives. To mitigate this attack one can disable client-side caching (when configuring the system) or, regarding the system's implementation, one can listen for query replies on a network, and notification via email is sent when an entry changes [67].

A Footprinting type of attack occurs when an attacker probes and explores an application, a system or a network, trying to learn everything possible about the composition, configuration, and security mechanisms of it. Normally, this type of attack is the opening for more serious attacks, because, using the information retrieved from the attack, one can circumvent or disable certain security measures in order to attack a more high profile victim (for example, a server). Some of the actions to reduce the risk of this attack occurring include shutting down unnecessary services/ports, changing default passwords (by choosing strong ones), and encrypting and password-protecting sensitive data [68].

Man-in-the-Middle (MITM) type of attacks targets the communication between two components (typically client and server) and aims to extract sensitive information without the

[5] `https://capec.mitre.org/`

components knowing it. The attacker places himself in the communication channel between the two components, and whenever one of them attempts to communicate with the other, he intercepts the message. This means that all of the traffic between the two components goes through the attacker. He has the power to cut off some communication, to alter it or simply just to copy it, having in mind the extraction of sensitive information. But there are ways to decrease the risk of an attack, namely signing the public key with a Certificate Authority, encrypting the communication using cryptography, using strong mutual authentication to always fully authenticate both ends of any communications channel, and exchanging public keys using a secure channel [69].

Denial of Service (DoS) consists of engaging with a resource in order to not allowing legitimate users to access it. According to CAPEC, it can be accomplished by four types of attacks: flooding, excessive allocation, resource leak exposure, and sustained client engagement. [6]

A flooding attack consumes the resources of a target by rapidly engaging in a large number of interactions with it, which generally exposes a weakness in rate-limiting or flow. If this attack is successful, it prevents legitimate users from accessing the service and can cause the target to crash. To mitigate the chances of this type of attack occurring, the administrator has to ensure that protocols have specific limits of scale configured, has to specify expectations for capabilities and dictate which behaviors are acceptable when resource allocation reaches limits and has to uniformly throttle all requests in order to make it more difficult to consume resources more quickly than they can again be freed [70].

When performing an excessive allocation attack, an attacker will cause the target to allocate excessive resources to servicing his requests, thereby reducing the resources available for legitimate services and degrading or denying services. Unlike the flooding attack, this attack does not attempt to force the allocation through a large number of requests but instead uses one or a small number of requests that are carefully formatted to force the target to service this request(s), allocating excessive resources. To reduce the chances of an attack, it is necessary to limit the amount of resources that are accessible to unprivileged users, to assume all input is malicious (do a more careful and thorough validation), to consider uniformly throttling all requests (to make it more difficult to consume resources more quickly than they can be freed), and, if possible, to use resource-limiting settings [71].

The resource leak exposure attack consists of utilizing a resource leak on the target to exhaust the quantity of the resource available to the service legitimate requests. The attacker determines what activity results in leaked resources and then performs it on the target. But, since the leaks may be small, it may require the attacker to perform a large number of requests to try to find it. This attack concerns the memory, so, to decrease the chance of it happening, some actions have to be performed regarding the memory. If possible, during the implementation of the service, the administrator should leverage coding languages that do not allow this weakness to occur (for example, Java, Ruby or Python), should implement best practices with respect to memory management, and should always allocate/free memory

[6] `https://capec.mitre.org/data/definitions/343.html`

using matching functions [72].
Another attack that can accomplish DoS is sustained client engagement, where the attacker attempts to deny legitimate users access to a resource, by continually engaging a specific resource in an attempt to keep the resource tied up as long as possible. However, this requires great skill by the attacker, since he does not want to crash or flood the target, as it would alert the administrator. Some of the actions to mitigate this attack include requiring a unique login for each resource request, constraining local unprivileged access by disallowing simultaneous engagements of the resource, or limiting access to the resource to one access per IP address [73].

Some of these attacks are more serious than others (e.g., DoS), but all of them have ways to mitigate their risk of occurrence. By using tools, such as CAPEC that tries to classify attacks, IDSs can more easily identify attacks. In the case of knowledge-based IDSs, it is particularly important this classification, to allow the creation of signatures that more accurately identify attacks.

2.2.2 IDS Classification

IDS have a variety of operational features, which can be used to classify them to understand what is the best one to use in a specific setting. However, there is not a unique classification scheme, existing several ones [74], [75]. To learn about the different types of IDSs and their properties, one was chosen (the one presented in [66]), which is summarized in Table 2.3.

The first operational characteristic shown is the detection method, which can be knowledge or behavior-based. The knowledge-based detection method, also known as signature-based or misuse detection, analyzes the system's activity while searching for known attack or intrusion patterns (known as signatures). For example, the signature of an SYN flooding attack is the absence of TCP ACK segments. This method is usually very efficient and allows for the IDS to have a reduced percentage of false positives. However, its main drawback is that it only detects known problems, the ones he has signatures for. This disadvantage can increase the percentage of false negatives, and, regarding newly discovered attacks, the system is useless in

Operational Characteristic	**Classification**
Detection Method	Knowledge based
	Behavior based
Events Source	Host based
	Network based
	Hybrid
Moment of Detection	Real time
	Virtual real time
	A posteriori
Reactivity	Active
	Passive
Analysis type	Single
	Collaborative

Table 2.3: Example of IDS classification. Source: [66]

the time between the attack is discovered until the moment when its signature is installed in the IDS. The other detection method is behavior-based, or also known as anomaly detection. This method detects deviations in the target system's normal behavior. To achieve this, the IDS first has to be trained, that is, it has to be put in the system and allowed to learn its normal functioning for it to work. Then, when its "training is finished", he can work as a normal IDS, analyzing the system and detecting changes to its normal behavior. For it to know when something is normal or abnormal, it makes use of statistic analysis and artificial intelligence techniques. This method's main advantage is that, in theory, it can detect new forms of attack and penetration, assuming that they represent an abnormal activity or they can provoke it. Nevertheless, the method has some operational issues, namely the definition of the target system's normal behavior (it can change over time and it is difficult to assert), it can be fooled (with artificial abnormal but non-harmful behaviors), its sensibility degree (how he asserts that some event is normal or abnormal, which is something critical to the IDS but very complex to manage), and the useful information it supplies to reaction components (an anomaly detection does not clearly allow to identify its cause) [66].
An IDS can also be classified based on the source of events, which can be from a host, a network, or can be a hybrid source, but the last one is not very common. If an IDS collects events from a host it is called a Host-based Intrusion Detection System (HIDS), and it monitors the state of the several components of a machine, namely hardware, OS kernel, file system, servers, and others. Its primary advantage is that this type of IDS is closest to the location where an intrusion occurs, which, presumably, makes it easier to collect trustworthy evidence of it. Another advantage is that, in theory, the IDSs can observe ciphered communication made with the machine where he is located. Despite that, Network-based Intrusion Detection System (NIDS) have a major disadvantage: they can observe what is happening in the whole system (because they are "closed" inside the machine. They can not detect reconnaissance that precedes attacks (such as network scans), nor can they monitor attacks to other machines or other HIDS inside the same organization, and they reduce the performance in the machine where they reside. The other source of events is the network, and the IDSs that collect this type of information are named NIDS. This type of IDS monitor machine interactions by having sensors in the network that capture all of the datagrams that circulate through it. The sensors can be normal machines, interconnection equipment (like gateways or routers) or simply dedicated equipment that captures datagrams. NIDSs main advantages are the simplicity (a small set of sensors can monitor a vast heterogeneous machine and OS network), the possibility of executing sensors in dedicated machines (allowing a small or null impact in the network performance, as well as not overloading machines with other goals), and the possibility hide dedicated machines from the attackers (not allowing attackers to deactivate sensors through attacks to the machines where they are). But some drawbacks arise from the fact that NIDS collect events from a low level, some of them are difficulty in analyzing all of the datagrams in high-throughput networks, inability to analyze ciphered datagrams, and inability to clearly identify if an intrusion occurred or not. Lastly, if an IDS acts simultaneously as a HIDS and a NIDS by collecting events from both hosts and networks, they are classified as

hybrids [66].
Regarding the moment of detection, IDSs can have three types: real-time, virtual real-time and *a posteriori*. Real-time detection allows the detection of attacks at the exact moment that they occur, and, this way, they can be terminated and prevented from being successful. But this ideal moment is very complex to attain since an attack is composed of several steps and some are just to illude the system. Nevertheless, there are some activities that can easily be reported in real-time that alert for future problems, such as the detection of reconnaissance probes. A more realistic time of detection is the virtual real-time, which allows detecting attacks shortly after they are successful and prevent them from being successful a second time. It is presumed that, once an intrusion is detected (but not the attacks that allowed it), it is possible to quickly react to eliminate the vulnerabilities that made it possible, or at least isolate the compromised system until it is known what happened and avoid it to happen again. *a posteriori* detection allows detecting one or more past attacks, usually by analyzing operation log files and detecting anomalies. This method is used by audit techniques and has the risk of not acting fast enough after the occurrence of intrusions [66].
An IDS can react passively or actively to attacks or anomalies. Passive IDSs only react with alerts, giving reports with the most useful information possible, so the human operators can understand what is happening and be able to react quickly and correctly. Active IDSs can react automatically to attacks and intrusions, where they react through defense or counterattack, but always considering the effects of the reaction to be applied [66].
IDSs can have to types of analysis: single or collaborative. The first type is simpler but does not allow a whole view. The second type is more complex and may entail the need for standards, if it is used IDSs from different manufacturers. The advantage of this type is that it allows a more embracing analysis of distributed systems, which can be composed of heterogeneous machines and networks [66].

Nonetheless, IDSs have some operational limitations, such as adaptation to various work environments (always needing to be fine-tuned to the setting and situation), scalability, and lack of a universal taxonomy to describe attacks and intrusions (which is important to provide useful information to operators and system administrators, and to allow interoperability with other IDSs or other security means) [66].

2.2.3 Open Source NIDS

Currently, there are several open source IDSs, some more known and developed than others, and some of which will be further described.

Snort [76] was for many years the de facto open-source IDS/Intrusion Prevention System (IPS) solution. It was initially released in 1998 by Martin Roesch as a lightweight open-source IDS, and it has been the most popular one since then. Snort does not have a native packet capture facility and requires an external packet-sniffing library, namely LibPcap [77]. This library is widely supported across various OSs, granting Snort with OS flexibility. Although LibPcap is versatile and widely adopted by industry, it is not a very efficient packet capture engine for high throughput systems, since it can only process individual packets (one at a

time), causing a bottleneck in high-bandwidth monitoring environments. Snort is an extremely well-tuned, single-threaded product, being a great fit for commercial organizations, where enterprise support options, along with a broad user base, are advantageous. It can be deployed at the organization perimeter or between enclaves, where bandwidth is below 1 Gbps [78].

Suricata [79] is also an open-source IDS that introduced features previously only offered by commercial IDS/IPS products. It was released in 2009 by the Open Information Security Foundation (OISF). Suricata's data acquisition is similar to Snort, supporting multiple-packet capture engines, emerging signatures and Snort's native signatures. Its features include alert and filtering (offers rate limits and thresholds per host or subnet), IP reputation (blocks know bad IPs based on reputation reporting), and Computer Unified Device Architecture (CUDA) Graphics Processing Unit (GPU) acceleration for pattern matching (distributing the workload away from valuable Central Processing Unit (CPU) resources, and boosting the systems' processing speed). Besides these, Suricata also adds more features to the ones from Snort, including multi-threading, file extraction, and hash white and blacklisting. It is advantageous for Internet Service Providers (ISPs) and hosting providers using 10-gigabit links, due to its multi-threading features [78].

Zeek (formerly Bro) [80] distinguishes itself from Suricata and Snort by being an event-driven IDS, whereas the other two are signature-driven. Bro started in 1995 by Vern Paxson in the Lawrence Berkeley National Laboratory (LBL) (altough it was published until 1998), and it is being under development ever since. In 2018, due to some questions that rose about its name, it was changed to Zeek [7]. It was initially developed as a research tool, meaning that its initial focus was not on GUIs, usability or ease of installation, and, due to its complexity, it resulted in the steepest learning curve when compared to Snort and Suricata [78]. Its main goals are separating policy from mechanisms (it gains in simplicity and flexibility), efficient operation suitable for high-speed, large-volume monitoring (since every dropped packet may potentially lead to a missed attack), and withstand attacks against itself (since an attacker may first disable it before targeting the real victim it protects).

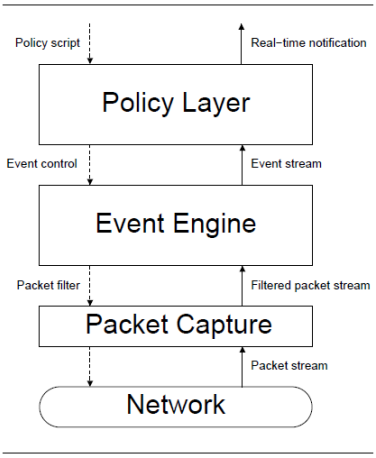


Figure 2.7: Overview of the Zeek design. Source: [81]

[7] `https://blog.zeek.org//2018/10/renaming-bro-project_11.html`

As shown in Figure 2.7, Zeek's architecture is layered, each with a component: packet capture, event engine, and policy layer. The lowest layer, the packet capture, is the one that gets the packets of the wire, and then sends them to the core of Zeek (the event engine). This component uses the LibPcap library [77], which provides a platform-independent interface to different network link technologies and implements small but powerful filter expressions, which can be used to reduce the number of analyzed packets (by selecting only those which are indeed needed by the current policies). This library interacts directly with kernel-level packet filters, providing efficient access to the needed subset of network packets. The second layer, the event engine, as mentioned, is the core of the IDS. It performs low-level processing like state management and protocol analysis, generates a stream of events, and then passes it to the policy layer. Its workflow includes getting raw IP packets (from the packet capture), sorting them by connection, reassembling TCP data streams and decoding application layer protocols. The event engine consists of several analyzers, each one responsible for a well-defined task (decoding specific protocol, performing signature-matching or identifying backdoors), and they may raise a set of analyzer-specific events. This component has to be efficient (to cope with large amounts of traffic) and as robust as possible (to withstand attacks against the IDS itself). Lastly, the highest layer, the policy layer, is responsible for evaluating the events according to user-supplied scripts. The user can define event handlers which specify the actions to be performed regarding it [81]. Being Zeek script-driven, it provides several features that are not at all possible in the other two IDSs. To attempt to scale the system to a high-throughput network, Zeek supports clustering, and, to achieve that, it provides primitive communication between sensors, so a high volume of traffic can be spread across multiple sensors. It offers scripting features that cannot be utilized with Snort and Suricata, bringing additional features through its script-based analysis engine and its capability to extend the response via scripts. It is best suited for high-throughput research environments [78].

Several works were published that presented comparisons between the mentioned IDSs [82]–[84], mainly comparing Snort and Suricata. However, there is one more recent ([78]) that compares all three of them . This paper primarily focuses on comparing their features and performance. Regarding IDS/IPS systems, they strongly depend on computer processing power to perform deep packet inspection and pattern matching, the reason why they start dropping packets when their processing limits are reached. A single thread or single processor is unable to sustain 10/40 gigabit throughput, and a saturated IDS/IPS starts losing packets, which increases the potential of false negatives occurring. If the percentage of unanalyzed traffic increases, then it is more likely to miss events of interest, like attacks. This problem arises because the network throughput is growing at an exponential rate, and, while commercial IDS/IPS products increase their processing power through custom ASICs and proprietary software, the open-source community has to seek scalable solutions that utilize commodity hardware to try to keep up with that. In conclusion, each IDS has its advantages, which should be considered when trying to select the best option to implement in a specific setting or else opting to implement several of them, leveraging the advantages of each one. Snort is an extremely well-tuned, single-threaded product. Suricata leverages the Snort ruleset and other

supporting products, as well as having features of multi-threading, file extraction, and hash white and blacklisting. Zeek introduces additional features through a script-based analysis engine and has the capability to extend responses to events via scripts.

2.2.4 Network-based IDS and SDN

With SDN advantages such as flexibility and scalability, the networks can become very vast and complex, arising the need for more security measures, one of which can be NIDS. There are in literature several works that use SDN in conjunction with NIDS, leveraging its features to develop a variety of solutions.

In [85] is presented NICE (Network Intrusion detection and Countermeasure sElection in virtual network systems), which proposes to detect and mitigate collaborative attacks in the cloud virtual networking environment. It utilizes the attack graph model to conduct attack detection and prediction. The proposed solution investigates how to use the programmability of software switches-based solutions to improve the detection accuracy and defeat victim exploitation phases of collaborative attacks. NICE only investigates the NIDS approach to counter zombie explorative attacks. In order to improve the detection accuracy, HIDS solutions are needed to be incorporated and to cover the whole spectrum of IDS in the cloud system, which is noted as future work. The NIDS used in this work is Snort.

SDNIPS (SDN-based Intrusion Prevention System) is presented in [86] for cloud virtual networking environments. This solution is designed to establish a comprehensive IPS solution that is capable of reconfiguring the cloud networking environment on-the-fly to counter malicious attacks. It has detection and prevention capabilities. It inherits the intrusion detection capability from Snort and flexible network reconfiguration from SDN. The solution uses the controller POX, which was extended to contain the SDNIPS daemon, an alert interpreter and a rules generator.

In [87] is implemented an Advanced Metering Infrastructure (AMI) threat detection mechanism based on SDN, which efficiently builds a threat detecting mechanism in AMI systems with the help of SDN. The solution is designed to enhance the security of AMI systems, while preserving the traffic quality of this structure. It integrates Snort with SDN, by mirroring all traffic that goes through the switch to Snort so it can analyze it and communicate the results to the controller.

In [88] a new method to efficiently prevent Distributed Denial of Service (DDoS) attacks is proposed, which is based on a SDN/Network Function Virtualization (NFV) framework. It utilizes SDN to improve various inefficiencies (such as network inefficiency and high cost) of existing defense measures against DDoS attacks. To resolve the problem that normal packets are blocked due to the inspection of suspicious packets, they developed a threshold-based method that provides a client with an efficient, fast DDoS attack mitigation. It confirms that network and web administrators can control the network packet capacity through the SDN controller. Suricata is used in the implementation as a security function.

2.3 Honeypots and Honeynet

Concerning a network's security, there are already plenty of off-the-shelf tools capable of identifying and terminate attacks executed onto a business network or onto equipment in the same, namely IDSs [section 2.2], IPSs, and Intrusion Detection and Prevention Systems (IDPSs). However, these tools are based on the pre-defined behavior of attacks, which means that they identify attacks by analyzing the traffic behavior against pre-established behaviors of known attacks. This makes the identification of unknown attacks very difficult; and even in the case that such attacks are identified, they are simply blocked, which means that if the same attack occurs again it may not be stopped.

Thus, new ways to analyze attacks and learn more from them started to be investigated. Some of these are the honeypots and honeynets. Honeypots are systems intentionally vulnerable (ideally being replicas of the real systems being protected) that are used to attract attackers, aiming to analyze how they act, as a means to identify possible vulnerabilities in the system [8]. Honeynets are like the honeypots, intentionally vulnerable and with the same purpose, only they are not just a system but a whole network that is recreated [9]. The honeynets have, therefore, the advantage of not isolating an attacker solely in a system, but allowing him to navigate all-over the network and interact with different systems.

Regarding the honeypots, they can be classified into three types: low, medium and high interaction. A Low Interaction Honeypot (LIH) is normally just a program that emulates the protocols of an OS, having a limited subset of the full functionality (hence the low interaction). Comparing to a LIH, a Medium Interaction Honeypot (MIH) provides much more interaction capability, often being able to emulate well-known vulnerabilities and capture malicious traffic accessing them. A High Interaction Honeypot (HIH) is a genuine computer system running as a honeypot, proving a fully functional OS for attacking and being able to capture not only network activity but also system activity. Despite that, it has a major limitation when compared to the other to types of honeypots: its resource consumption for large-scale deployment, because it has minimum hardware requirements to be fully operational [89].

Concerning the honeynet, there are many solutions, but to evaluate them is essential the existence of a taxonomy, being one proposed in [90]. This classification scheme contains five main criteria: adaptability level, IP network scope, physical placement, logical deployment, and resource level. But first, it has to be considered the three essential capabilities of a honeynet: data control, data capture and data collection. In order for outbound attacks to be controlled and to protect the non-honeynet systems, data control is required, which means having a set of measures to mitigate the risk of an attacker using a compromised honeypot to attack other non-honeynet systems. Also, to later investigate the attacks made on a honeypot, it is necessary to capture the data, in other words, log all attacker's activity. This data capture has three critical layers: firewall logs (inbound and outbound connections), network traffic (every packet and its payload) and system activity (attacker's keystroke, system calls, modified files, etc). Lastly, all of the captured data needs to be securely forwarded from

[8] `https://searchsecurity.techtarget.com/definition/honey-pot`
[9] `https://searchsecurity.techtarget.com/definition/honeynet`

distributed systems to a centralized secure data collection point, and the means to do it form the data collection capability. Given that these capabilities are the base of honeynets, they are considered and referenced in the scheme being presented.

The capability of dynamically modifying the configuration and topology of a honeynet is referred to as the adaptability level and can be of two types: static or dynamic. If a honeynet is static, it means that it does not have the capability of being modified or reconfigured, which is the main drawback, since the whole honeynet has to be redeployed if one honeypot needs to be reconfigured. For this reason, this type of honeynet is not adequate for systems requiring honeynet reconfiguration as a real-time response to network events. On the contrary, a dynamic honeynet is able to change the configuration of their honeypots, their topology, or adding or deleting honeypots dynamically in real-time, which can be a result of a management request or a response to a networking event. This type of honeynet is very useful considering that they can react, for example, to events triggered by an IDS, being able to reconfigure themselves according to the characteristics of the attacker's behavior. In addition, these honeynets overcome some of the static honeynets limitations, namely, by allowing a partial reconfiguration without the need for restarting and redeploying the whole honeynet [90].

One important feature of the honeynet is its IP network scope, in other words, how IP addresses are assigned to honeypots since it affects the mechanisms used for isolating them from the rest of the network. The IP network scope of the honeynets can be stand-alone or distributed. In the first case, all honeypots use IP addresses from a common IP network prefix, which can ease the data capture given that they can share one security toolkit that can capture all of the data from the whole honeynet. When an organization has multiple honeynets in distributed environments, it is best to have a distributed network scope, which covers multiple networks concurrently in order to provide a greater ability to capture suspicious network events [90].

The physical placement (physical location) of a honeynet is also an essential aspect, especially when considering large-scale networks. A network of honeypots located within a physically limited area has a local physical placement, for example, areas like a computer laboratory or an office building. On the other hand, if a honeynet is remote it means that it is not forced by any physical placement limitation, meaning that the honeynet can be located in any place of the world. This allows a group of physical remote honeypots to be integrated into one production network by tunnel technology (i.e. Generic Routing Encapsulation (GRE) tunnel) [90].

The logical deployment of a honeynet is the logical relationship between it and the production network, which can be of two types: minefield (honeynets passively capture data upon interaction) or shield (honeynet always acts as a mirror of the production network). In the case of a minefield logical deployment, the honeypots are often logically deployed among production systems, possibly cloning some of their real data. The honeynet is used to handle any kind of traffic (regular and intrusion), and the IP addresses assigned to the honeypots are chosen from the unused ones of the production network. In the shield logical deployment, the honeynet shield and the production network are coupled (tightly or loosely), the honeynet can

reside in the same address space of the production network or on another subnet alongside it, and some approaches (for example, Network Address Translation (NAT) or GRE tunnel) must exist to redirect the interesting traffic to the honeynet [90].
All of these guidelines can be used to classify and to study honeynet solutions, to better understand them. To build a honeynet is vital to have resources, which can be physical, virtual or mixed. When several honeypot systems are run directly on physical machines following certain network topology, the honeynet is considered physical at the resource level. This type of honeypots can be high-interaction, since they can get a high-level fidelity of data capture, implying, however, a higher resource cost. In contrast, a virtual honeynet is several honeypot systems, following certain network topology, deployed using virtualization software, meaning they can be hosted by one or more physical machines. Finally, a honeynet can be of mixed type at the resource level, which means that honeypots are deployed physically and virtually, following certain network topology. This type of resource level can get a good balance between resource efficiency and service fidelity, combining advantages from both previously mentioned types of resources [90].

2.3.1 Virtual Honeynets

A virtual honeynet combines all the physical elements of a honeynet inside only one computer by using virtualization software and can be of two types: self-contained or hybrid.

A self-contained virtual honeynet is an entire honeynet system deployed onto a single physical system, where the data control and data capture functions are run by the host OS [90]. Its main advantage is portability, given that, if it one wishes to deploy the honeynet in several locations or wants to relocate one already in place, it simply has to copy or move the physical machine where it resides, in other words, its setup is of the type plug and catch. Some other advantages of this type of virtual honeynet include a low cost, low maintenance time, low physical space and centralized management. However, it has some drawbacks, namely, it has limited flexibility, its security is software dependent, and the data obtained is limited [91].

Regarding the hybrid virtual honeynet, it comprises at least two physical machines, one implementing the data control and data capture functions, and another running the virtual honeypots [90]. Its main advantage when comparing to the self-contained honeynet is that it is safer, considering that if an attacker compromises the physical machine, the honeynet preserves control of the situation through the machine with the data control and capture functions. It also has more advantages, such as it can obtain a higher quantity of data and it has moderate flexibility, comparing to a self-contained honeynet. Despite that, it has a major disadvantage: it is not portable, which increases its cost and space. In addition, it has some other drawbacks, namely, medium maintenance time and distributed management [91].

Regardless of its type, virtual honeynets have some advantages and disadvantages that are worth mention. A virtual honeynet, when compared to a physical one, introduces a great reduction in cost, is easier to do the maintenance of the entire infrastructure, and is feasible to design and put into operation. Nevertheless, it also has disadvantages, namely inherent

limitations (limited types of OSs, due to subjacent hardware and virtualization software), an increased risk (attacker could compromise the physical machine and gain full control of the honeynet, and also corrupt all of the control and information capture mechanisms), and fingerprinting risk. Fingerprinting can be seen as the ability to identify honeynets remotely or locally, based on certain elements that result from the virtualization process [91]. As explained before, virtual honeynets make use of virtualization software to build their honeypots, and some of the software used can leave traces ("fingerprints") that clearly identify the honeypots' OS or its services, which can be used by attackers to identify the system as a honeypot. One example of a honeypot that has an identifiable fingerprint is Kippo [92], which is a medium interaction Secure Shell (SSH) honeypot whose objective is to log brute force attacks and the entire shell interaction performed by the attacker. Its known fingerprint is the print of the system's information. When the honeypot emulates a fake SSH service, it always returns the same string when an attacker accesses its system information, because it is hard-coded. So, if an attacker, when entering a machine, always performs checks for known fingerprints, or if he just suspects he is in a honeypot, one of the verifications can be to print the system's information, and if the response is ***Linux*** or ***Linux <hostname> 2.6.26-2-686 #1 SMP Wed Nov 4 20:45:37 UTC 2009 i686 GNU/Linux*** (from Kippo's source code) the attacker knows he is inside a honeypot. Another example of an SSH honeypot with a known "fingerprint" is Kojoney2 [93], which also a medium interaction honeypot. Its "fingerprint" is also regarding the system information but is not something hard-coded, although it is still easily detected. When this honeypot is installed, it generates the system information using the timestamp of the real-time, meaning its system information will have the timestamp of the moment the honeypot was created. If an attacker knows the time he started launching the attack and compares it to the timestamp of the system information, it can easily verify that is inside a honeypot. In both examples, the second the attacker realizes he was deceived by a honeypot, he will stop the attack and try to launch it by other means, and probably be successful, since he already knows some of the security measures existing [94].

2.3.2 Honeypots and Honeynet in SDN

With the rise in popularity of honeynets, honeypots, and SDN, it should not be a surprise that solutions are being developed that take advantage of their features and functionalities and combine them all.

The advantages of the SDN technology include ease of management, efficient flow control, and extendable integration [89], all of which can leverage the deployment of honeynets and honeypots. Honeynets can be very complex, not only because they have to capture data inside it, but also because they have to control the data flow since they are by definition compromised. With SDN, their management can be eased by transferring some of the data control features to the controller. With the flow control provided by the SDN controller, data from and to a honeypot and/or honeynet can be easily selected and more carefully handled. The controller can also be extended (by applications or by extending its capabilities)

to accommodate data capture functions, further alleviating the honeynet tasks. All of this can also be done in an automatic manner, which decreases the chance of a human error and increasing the security of the honeynet, not allowing an attacker inside it to compromise the rest of the network.

As mentioned before, honeynets and honeypots can be implemented in physical machines or virtually. When considering traditional networks, most probably it will be the first case. However, SDN brings some advantages when compared to them, namely lower cost (needs fewer infrastructures) and faster deployment (by using the controller to deploy several devices at the same time). It also allows a more dynamic and exhaustive data collection by leveraging the programmability of the controller. SDN also provides a certain level of abstraction, since it allows the definition of a logical network topology through software, and to structure it through virtualization technology. With these features, there is no need to consider the physical topology of the underlying network, which provides means for more dynamic deployment of honeypots [95].

Some solutions of honeynets and honeypots that use SDN technology are further explained in section 2.4.

2.4 Similar Solutions

Regarding the solution that will be proposed in this thesis, some solutions already exist that are within the same scope of it and that address some of the same problems, following a brief explanation of some of them.

HoneyMix[94] is an intelligent SDN-based honeynet. This system is composed of pre-instantiated honeypots that used to establish connections to possible attackers accordingly to their intentions. Summarizing, the honeypots have several and distinct services, being these repeated in different machines, in order to create redundancy of services, and ensuring that there are not two machines with exactly the same services. When there is a connection attempt to certain service, this is first identified, and then is established a connection with a honeypot that has it and that is perceived as more desirable to the particular attacker, in order to keep him connected to the honeypot for as long as possible without identifying it as such. Nevertheless, as previously mentioned, the honeypots are pre-instantiated and there is only the redirection of traffic to the selected one, which limits the actions of the attacker to the honeypot and can contribute to faster identification of it, in addition to the system not being tested in real-world deployments. HoneyMix architecture centrally computes the service distribution in the network, to enable fine-grained data control, and deploys the corresponding rules via SDN switches. It also conducts security incident response, which includes quarantine and recovery using SDN and NFV. This architecture main goal is to prevent honeypot fingerprinting by using a connection selection engine that transfers from one connection to another, in other words, it has a dynamic selection of the most desirable connection to an attacker. This selection engine helps increase the anonymity of existing honeypots by changing active connections, and also sanitizes specific payloads that reveal information of honeypots, which are major advantages when comparing to other more simpler

architectures. HoneyMix uses some ideas of Moving Target Defense (MTD) [96], [97], and some of its techniques can also be added to the architecture, as means to minimize the possibility of exposing network infrastructure (like OS, service, and host). One example of a technique that can be used is the random host mutation that can be considered to hide honeypots. Another technique is to generate virtual IP and Media Access Control (MAC) addresses to dynamically create corresponding Domain Name System (DNS) information to hide network configurations. Summarizing, HoneyMix decreases the risk of honeypot fingerprinting, when comparing to other honeynet solutions, while also providing an array of services, increasing the possibility of an attacker not recognizing that is inside a honeypot and, therefore, making the attacker spend more time in it and collecting more data. Additionally, it has the capability to support MTD techniques, which further increases the likelihood of an attacker never realizing he is being deceived.

Also in the topic of intelligent honeynet, Lian et al.[95] proposed an SDN Virtual Honeynet for Network Attack Information Acquisition, which acquires information during the occurrence of attacks and acts upon this adjusting the network's structure, diminishing the threat of network attacks. This system acts as a proactive defense mechanism since it monitors the behaviors of attackers and lures them when they enter the pre-designed honeypot network, thus protecting the real network, obtaining more information about the situation, analyzing and evaluating it, and do predictions based on this. For satisfying different demands of network resources, the system defines a logical network topology through software and uses virtualization technology to structure it, thereby not having the need to consider the physical topology of the underlying network and being able to realize dynamic deployment. This means that the network structure can be changed dynamically to decrease the level of threat and raising the security level of the network. To test the system, Mininet [98] and ODL [61] were used, the former to build the virtual honeynet and the latter to control the flow of the honeynet. The experiment proved that a dynamic virtual honeynet based on SDN can solve difficult problems in flow control on traditional honeynets, and solve the inconvenience and the complexity of dynamic deployment of physical machine deployments. Regarding the experiment, the services made available were limited, because the test platform used was Mininet, lightweight software for network definition. This architecture brought some advantages to previous solutions, namely facilitating the honeynet deployment, being able to add or remove all kinds of services, saving in cost, making the deployment more convenient, and making the system easier to restore. By analyzing the network situation on-demand and dynamically deploying honeypot services, the system can get more information about the situation and provide more support for decision-making, which can improve the network's security system. To sum up, an SDN Virtual Honeynet for Network Attack Information Acquisition validates the significance of SDN dynamic virtual honeynets, proving that it has several benefits when comparing to traditional honeynet systems, namely a lower cost and a faster deployment, while also allowing a more dynamic and exhaustive data collection.

Very recently, Fan et al. presented HoneyDOC[89], which can be considered almost within the same scope as the proposed solution of this thesis. HoneyDOC is a honeypot architecture

with the capability of traffic analysis and redirection according to the type of attack. Figure 2.8 shows the components of the architecture, of which three essential modules stand out: Decoy, Captor and Orchestrator, which are independent and collaborative.

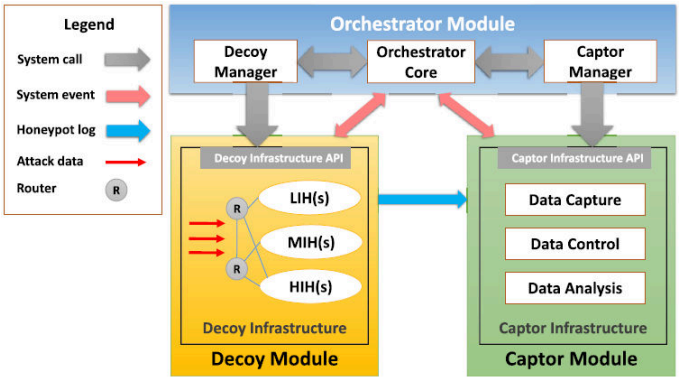


Figure 2.8: Overview of the HoneyDOC architecture. Source: [89]

The Decoy module is the one responsible for provisioning and deploying the honeypots/honeynet over the decoy infrastructure, and it has to expose an API that the orchestrator module can call. Dedicated honeypots are effective for meeting single criterion (because they are specialized), but have difficulty in satisfying multiple criteria, due to being expensive in scalability (in the case of HIH) or having problems in collecting detailed attacking data (such as the MIH and the LIH). This is why the decoy infrastructure aims to host the three types of decoys, and it should allow the deployment of arbitrary honeynet since multiple decoys can be deployed as a honeynet by following certain network topology. The attack data goes through the decoy infrastructure, where t should be captured to provide a honeypot log, which then will be transferred to the captor module, and this attack data will cause a system event that will be sent to the orchestrator module.

The Captor module is responsible for providing functionalities that can be applied to the attack data, and it includes three basic submodules: data capture, data control, and data analysis. The data capture submodule has three critical layers: firewall logs, network traffic, and system activity. The data control submodule leads the attack flow according to the honeypot's intention instead of the attacker's, so the latter can not gain control over the former. The control is done over the attack flow, being considered two directions: inbound and outbound. Concerning the inbound attack flow, the traffic will be classified and actions will be performed, specifically discard uninteresting data, forward interesting data and redirect the most interesting data to dedicated decoy resource. Regarding the outbound attack flow, it is used to mitigate the risk that the adversary uses the compromised honeypot to attack other non-honeypot systems. Therefore, it is necessary to minimize the chance of the attacker or malicious code detect this functionality, being one example of a solution to transparently redirect outbound connections to another honeypot that emulates the targeted system. About the data analysis submodule, it does exactly what the name states, being able to employ and integrate third-party analysis services.

The Orchestrator module has the responsibility of coordinating the other two modules so

they work efficiently together, having three main components: the Decoy Manager (DM), the Captor Manager (CM) and the Orchestrator Core (OC). The DM can actively call the decoy infrastructure API in order to provision and deploy the needed decoy or a network of decoys (honeynet). The CM is used to integrate the captor infrastructure by calling its API, and can also call the captor module to process the system event as well as the honeypot log. The kernel of the orchestrator module is the OC, handling all system events and managing the interaction between the DM and the CM by scheduling calls to their APIs in order to perform needed operations.

The design of SDN-enabled HoneyDOC takes into consideration three key features: sensibility, countermeasure, and stealth. Sensibility can greatly increase the efficiency of data capture, since it consists in the ability to classify and processing attack data in fine-grained ways, representing advanced detection capability (instead of merely alerting malicious activity). The capability of a honeypot system providing a response to the attack with the purpose of capturing high-quality attack data (instead of being a victim) is expressed through the countermeasure feature. In order to prevent the honeypot system's functions and operations from being detected by the attacker, the system has to have stealth. This feature can guarantee the effectiveness of the honeypot, since it aims to deceive the adversary to behave as in its self-righteous network environment, so that it can show off its attacking ability, to be fully observed by the honeypot system. One essential aspect to have in mind when trying to apply this feature is traffic redirection. In it, the header information of three protocols (Ethernet, IP, and TCP) need to be handled, so the attacker can not recognize that its destination changes. Ethernet and IP protocols are stateless, so the only thing that needs modification is the destination fields, and the problem is solved. Regarding the transport layer protocols, User Datagram Protocol (UDP) is also stateless, so the solution previously presented to the other protocols also applies to this. However, TCP (the other transport layer protocol) is stateful, making some problems appear, since multiple fields need to be updated for migrating the connection. The main issue of a transparent TCP connection handover is to synchronize the Seq and Ack values related to the sequence numbers chosen by the different endpoints. Based on this conclusion, two technical challenges need to be addressed in order to facilitate a stealthy traffic redirection: identical fingerprints of different decoys and the sequence number synchronization for connection transfer. A solution for the first problem is to use distinct outports of the SDN switch to identify end-points with identical IP Version 4 (IPv4) and MAC addresses. To solve the second problem, a TCP replay approach [99] is used to transfer connection, which updates the Seq number on-the-fly by the SDN controller.

When applying the HoneyDOC architecture in the SDN framework, it correlates a plane with a module: Application plane to the Captor module, Control plane to the Orchestrator module, and Data plane to the Decoy module. The Data plane (Decoy module) is responsible for directing traffic to heterogeneous decoys with user-defined network topologies deployed by the system. The Control plane (Orchestrator module) can use a specific SDN controller software to provide network services for supervising and managing data communication over the data plane. The Application plane (Captor module) can include various captors upon the SDN

controller's framework.

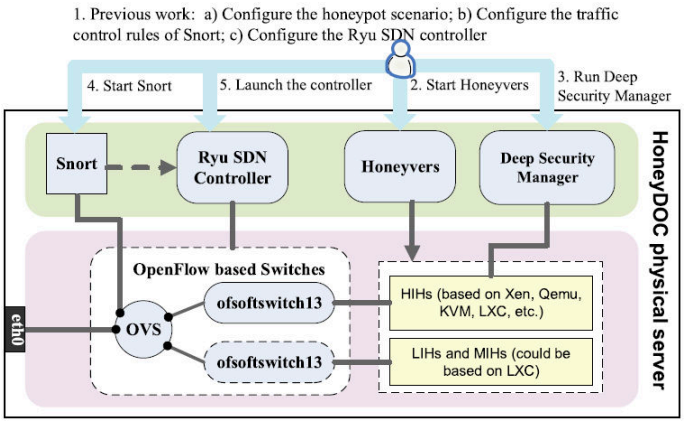


Figure 2.9: Implementation of HoneyDOC proof-of-concept. Source: [89]

In Figure 2.9 is demonstrated the proof-of-concept implementation of the HoneyDOC in a single physical machine. The technologies used can be seen, which include Snort IDS [76] (run in NIDS mode), Ryu SDN controller [52], Honeyvers [100] (a generic virtual decoy deployer), Deep Security Manager [10] (which needs to be installed into the HIH), virtualization software to deploy the honeypots (Xen [11], Qemu [12], KVM [13], and LXC [14]), and OpenFlow-based switches (OVS [15] and ofsoftswitch13 [19]). As shown, the system's deployment has several steps. First, some work has to be done previously to starting and running the system: the honeypot scenario has to be configured, as well as the traffic control rules of Snort and the Ryu SDN controller. Next, Honeyvers has to be started, followed by running the Deep Security Manager. Lastly, Snort has to be started and, next, Ryu has to be launched.
In conclusion, HoneyDOC architecture enables the implementation of honeypots with different interaction degrees, that is, depending on the severity of the attack, the traffic is redirected to a honeypot with the appropriate interaction degree, ranging between low, medium and high. The honeypots are instantiated using virtualization software, in order to not overload the network's resources. This solution as also an orchestrator, responsible for the traffic redirection to the correct component. To classify the traffic for attack identification or for traffic selection using other methods, the Snort IDS is used. The architecture introduces several improvements in terms of sensibility, countermeasure and stealth features, although it seems to be more focused on the last one. It uses a range of different tools, demonstrating that a system can be built by making distinct components work together, as long that exists one of them capable of coordinating the interactions between the rest of them. Also, HoneyDOC demonstrates that SDN can be beneficial in the deployment of more evolved virtual honeynet solutions.

[10] `https://help.deepsecurity.trendmicro.com/software.html`
[11] `https://xenproject.org/`
[12] `https://www.qemu.org/`
[13] `https://www.linux-kvm.org/page/Main_Page`
[14] `https://linuxcontainers.org/`

CHAPTER 3

Solution Architecture

The business world is very broad. However, there is a constant overlap of companies with the same goals and that provide the same services, which leads to the existence of strong competitiveness between them. For a business to thrive in the current economy, it has to be several steps ahead of the others. This drives the companies to desperately trying to know what its competitors are developing.

An enterprise network contains many varied elements. Inside a company, its employees have a computer and a smartphone, so they can communicate with other employees that are associated with the same projects. Besides this, the company also needs landline phones (for its clients to contact it and for different departments to communicate), printers and scanners (to reproduce work), surveillance cameras (for the people's and company's physical space security), and possibly more types of hardware. All of these pieces of equipment will be connected through the business' network, which means that all of the data, be it about projects or related to interpersonal relationships, will circulate through there.

Therefore, network security is one of the essential aspects to look for in a network's management, since the network contains all of the company's secrets, that is, all of the projects and work that it is developing. Without good network security, the company can easily be hacked and its secrets discovered by a rival company. Notwithstanding, this is not an easy task to be done, especially in a business context. Despite all types of equipment being connected to the same network, one can not protect all of them in the same way. Which leads to the categorization of the business' pieces of equipment in two types: equipment with the ability to implement security measures within itself (capable of self-protection); and equipment without the ability to implement anything besides its proprietary software (without the ability of self-protection). Computers and smartphones are included in the first category, whereas printers, surveillance cameras, scanners and other types of equipment belong in the second category. The first group of hardware as the ability of self-protection since they can have firewalls, anti-virus, and other security measures so that, in case someone tries to access them or some malicious software is installed or downloaded into them, these are identified

and blocked or eliminated. The second group of hardware doesn't have this capability since they only execute proprietary software to perform their functions and nothing else. This type of equipment will have to be better protected by the network where they will be inserted because if something breaks through the network's security this group of equipment will be completely vulnerable to attacks and intrusions, while the other group (the self-protected ones) will have a few more layers of protection.

The identification of different equipment creates the need to implement additional security measures in the business' network, such as more layers of protection, and also the distinction between the security measures implemented for the self-protected hardware and the other group of hardware. To do this, the network's administrator needs to implement separation mechanisms to divide the two groups, such as the creation of private networks or Virtual Local Area Networks (VLANs) for each of the groups, in addition to the already existing networks and VLANs to differentiate the several departments of the company.

A business' network security is never easy to maintain and manage, not only because there is always someone capable to overcome the barriers imposed on the network or system, but also because it is necessary constant supervision of the implemented mechanisms. The network's administrator has to have the ability to, before the rise of anomalous situations or even attacks in the network, quickly and discretely (that is, without disrupting the normal operation of the business) solve the situation. Nevertheless, the administrator can't be monitoring and manually applying security measures 24/7 (twenty-four hours a day, seven days a week). This implies the existence of automation mechanisms capable of monitoring and dynamically implementing defense mechanisms, without the required and constant intervention by the network's administrator. If such mechanisms are implemented, the network's and security mechanisms' performance will be much better, and they will facilitate the network's administrator job.

All of this raises a very relevant question: how to manage the security of a business network when having a minimal intervention by its administrator?

3.1 Requirements

To tackle the problem at hand, it's necessary to identify the necessary involved parameters.

SDN networks are a recent and evolving technology, that provides several advantages to the management and security of a network, such as flexible network configuration, centralized point of control and on-demand traffic control. Based on the investigation done in this area, it can be concluded that is advantageous to use SDN to build a business network. By using it one can implement new tactics and mechanisms to secure and manage the network as a means to reduce the work of the network's administrator and to automatize some of its tasks. From this arises one requirement: to manipulate traffic automatically and more dynamically, which can be done by using the SDN network, specifically the controller, to handle flows as a means to automate the network's management and security.

Regarding network security, this is one of the most important and sensitive aspects of a system, since it has to have several components and functions so it can provide maximum

safety possible. Other requirements derive from the combination of this awareness and the requirements previously referenced, leading to approaches such as automatic traffic classification. If using SDN automates traffic manipulation, then the traffic has to be classified to ensure that each user only has access to what concerns him and has clearance to, implying a blockage or redirection of the traffic accordingly to its provenance and destination. To achieve this, the network should include a component responsible for traffic classification, which can include one or several functions that can be implemented using several tools whose goal already is to automate this process. Also, this component should be able to communicate and interact with the controller, since this is a fundamental piece of an SDN network and its central point of control.

In the day-to-day of a company, its network handles the traffic exchanged between the employees as well as traffic originated on the outside (from the Internet). So, the problems that may appear in the network can be inoffensive (such as disconnected cables, unaccessible servers, and others) or can be harmful, namely attacks with the intention of stealing the company's information. Therefore, in order to automate almost all the administrator's tasks, the system should have not only means to identify and handle traffic, but also a way to react to problems that may arise from the network's operation, which requires the existence of one or more components capable of implementing reaction measures to anomalous situations. One of these components may be the controller, who is capable of flow manipulation, but can also be other components (controller's independent or not) that allow the implementation of another type of defenses mechanisms. One example of these can be the existence of a private network that isolates certain traffic from the rest of the company's network, to which possible attacks are redirected. Another example could be the creation of a honeynet that allows attack inspection (whose data can be used to improve the already existent network's security measures, making them more advanced and/or specific) and attack illusion mechanisms, while also doing the same as a private network.

Sometimes there are events happening in a network that are not perceived as urgent problems but are worth checking, such as a server becoming increasingly full (without space), IP address exhaustion (Dynamic Host Configuration Protocol (DHCP) address pool becoming almost used), and others of the type, which, usually, the administrator identifies and acts upon, normally adding or redistributing resources, in the case of the examples given. The log files are one of the most import registries to check when looking for these types of events, be them network, server or machine logs. Thus, a system that automates the administrator's job requires the existence of a central log server, which should be capable of processing the logs of the network's components, communicate with them and compile the most important information to display to the administrator.

In a system, there always are certain parameters that have to be met so its operation does not become compromised or stops altogether. In a network, be this a small or an enterprise-size network, some of the parameters considered are bandwidth, throughput, latency, error rate and response time, but a system can have others based on its components and their function. Regarding a network where the administrator has components responsible for task automation,

Requirements	Overview
Traffic manipulation	Automatic and dynamic manipulation of traffic by the controller to automatize the network management and security
Automatic traffic classification	Existence of component responsible for it, with one or more functions, and able to communicate with the controller; To apply QoS and security measures
Reaction to anomalous situations	Component capable of implementing reactive measures, namely defense mechanisms
Central Log Server	Component capable of gathering logs and analyzing them
System performance	Introduction of new components in the network can not negatively impact its performance
Percentage of false positives	Traffic classification component can not have a significantly high percentage of false positives
System's response time	The time needed by the system to identify traffic and implement security mechanisms can not be noticeable by an attacker

Table 3.1: Brief overview of the solution's requirements

it is very important that these components do not impair the performance of the remaining network, negatively affecting the normal work of the company, or else their installation in the network does more harm than good. To test this impact, the previously mention parameters have to have similar values on the network with and without the components for these to bring an improvement to the system. About the component responsible for traffic classification, besides its impact on the network, it also has other parameters that influence the choice of its functions and tools, namely the percentage of false positives it has. The percentage of false positives of a traffic classification tool is the portion of results that are misinterpreted as positives, for example when the tool identifies a traffic flow as being part of an attack (positive result) when in fact it is just a user (negative result). If a specific tool or function does not have a low percentage of false positive, it is considered too damaging to implement, because it will compromise the system and lead to the blockage of several legitimate users' traffic. If this happens, it will mean the disruption of the company's work by the network's defense mechanisms, something that should be avoided at all costs. Besides the traffic classification component, one has also to consider the introduction of the network's defense automation mechanisms. If traffic redirection and the creation of illusion mechanisms (namely honeypots) is possible, the response time when these mechanisms are used has to be examined, since they introduce layers of protection, taking longer for the traffic to arrive at its destination and get a response. The response time measured when these mechanisms are used against specific traffic cannot be much greater than the normal values, in other words, so great that it makes the attacker get suspicious, distrust the network and try to circumvent these measures.

Table 3.1 presents a summary of the system's requirements, to better identify them and understand their significance.

3.2 Use Cases

To design a system it is necessary to determine what it needs to do, that is, which actions it has to be able to perform. One of the methods used to achieve that is identifying use cases, which are descriptions of interactions between a user and the system where the former achieves a particular goal. Since the problem described requires several components, is of large-scale and too complex, the identification of use cases is of great importance to better understand the aim of the system and to design its components to have the necessary functions. To sum up, use cases are used to simplify and better understand the problem.

3.2.1 Use Case 1: Manage network

If the aim of the system is to automate the administrator's tasks, then he has to be able to manage the network, in this case with minimal interaction. So, the first use case will be "Manage network", which is described as the network's administrator starting the system and receiving information about what is happening, where the actor of the use case is the administrator. Some preconditions have to be met in order to start the use case's flow, such as the actor has to be authenticated (to ensure that nobody besides the proper technician controls the network), the platform where the network will be built have to support SDN (the most important requirement of the solution) and has to be configured and running, and it should have enough resources to install and run the system. The flow starts with the administrator running the initial script (responsible for configuring and starting the system), which will ask for information on some parameters, so the system will be configured as the actor desires. Next, the administrator has to input the answers, which will be used to configure, install and run the system. An alternative to this flow is, instead of the administrator inputting the answers, he leaves blanks, so the system will be configured with the default information for each parameter. After the use case, its post-conditions have to be met, that is, the system should be up and running and the administrator has to be receiving information regarding the system's operation. This means that the system should generate alerts of, at least, anomalous situations and should display them to the network's administrator, so he can act upon them, if necessary because the goal of the system is also to provide an automatic response to alerts. Additionally, one can verify that, even if the administrator does not input information to configure the system, it is built regardless, and it already includes some measures to administrate it, namely security measures.

Figure 3.1 shows the flowchart of the use case "Manage network", which was previously described.

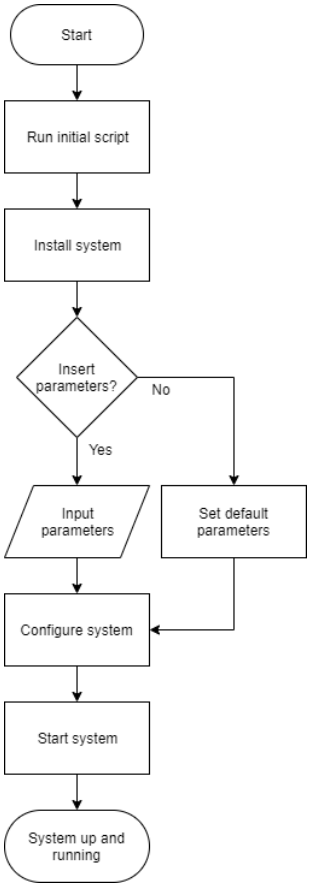


Figure 3.1: Flowchart of Use Case 1

3.2.2 Use Case 2: Handle security measures

An administrator's tasks include not only the identification of anomalous situations but also the implementation of measures to correct them, specifically the application of security measures dynamically. This reveals another use case: "Handle security measures", where the actor is the same (the network's administrator), and he has to implement new security measures or modify the existing ones on the fly (the brief description of his goal). Before proceeding to the steps necessary to achieve the goal, first the administrator has to be authenticated and the system has to be up and running, or else the actor can not continue his actions. When these conditions are met, the administrator starts his actions by accessing the machine where the component responsible for applying the security measures resides, possibly the controller's machine, since this is the central point of the network. There, he executes the script to implement new security measures, which will ask information about the necessary parameters to generate said measures. After the administrator inputs the information, the script will compute it and will add the new security measures to the system. If the administrator does not want to add a new measure, but instead to modify one, he has to follow a slightly different flow of actions. In this case, the administrator starts the same way, by accessing the specified machine, but instead executes a different script, the one responsible for modifying existing security measures. When this script is executed, it displays the measures available and asks what is the one to modify,

and then the actor specifies which is the one he wants. After that, the script will inquire about which parameters to change, the administrator has to answer, and the script will compute the information and modify the measures desired. When the administrator and the system are done, the latter has new security measures and the former starts receiving information based on these. This use case reflects the dynamic implementation of security measures, meaning that at any moment in time the administrator can modify the system without jeopardizing the operation of the entire network. Also, since the measures are defined by parameters, the system can display information regarding these to the administrator, so he can act upon the generated alerts by knowing exactly what to change or add to the network.

Figure 3.2 shows the flowchart of the use case "Handle security measures", which was previously described.

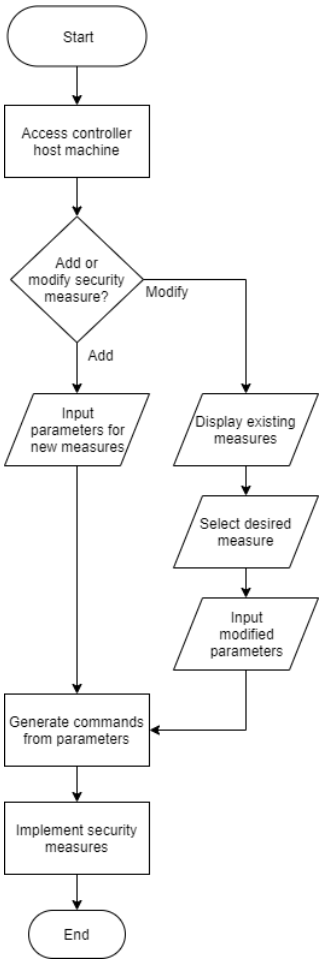


Figure 3.2: Flowchart of Use Case 2

3.3 Architecture

When a system is very complex and requires many components, its design has to be done in several steps, in other words, the problem has to be fragmented into small and understandable parts, so then they can be connected to build the final solution. Since the requirements

and some use cases were already defined, now it is time to construct the architecture, which represents the structure of the system, with its several components, necessary functions and the elements it will interact with. Seeing that the system described is in the enterprise scope, the dimension and elements of a company have to be considered in the architecture.

Given its magnitude, when dealing with a business network it is good practice to divide it, to consider a portion of it one at a time. First, two big areas can be defined: external (everything outside the boundaries of the company, including the Internet) and internal (everything directly connected to the network inside the company). In each of these areas, there are several types of equipment (each with different functions and aims) and users, whose intentions are not the same. All of these sections have to be considered in the architecture because the traffic originated from each one of them has to be handled using different methods.

The external area of a company's network comprises all kinds of devices and users that try to access something located inside it, be a computer or a file in the company's server. Concerning the users, different people have different intentions, that is to say, one person may simply try to download the file she wants and another may try to overflood the company's network so it stops working. Therefore, all traffic originated from this area should be treated with maximum care possible, even making it undergo extra security measures if necessary. Consequently, these security measures should be the most effective and reliable in the treatment of this type of traffic, so the operation of the business' network may not be put in jeopardy. All of the devices and users internally connected to the company's network make up for the internal area. Here, despite having the same problem of the user's intentions (although not so serious as the users from the external area), the traffic can be a little bit more reliable, but not much more, since the employees' devices can be transported to the outside. This means that some devices, such as smartphones and computers, can be connected to untrustworthy networks, be infected by a virus and then pass it on to the company's network once they reconnect with it. Therefore, the internal traffic must be submitted to additional security measures, but not the same as the external traffic, since they are different situations with distinct types of attacks and underlying problems.

The architecture should include the external and internal areas and two types of devices (those with the capability of self-protection and those without it), seeing that these are the elements that interact with the system and influence its behavior. Furthermore, the architecture has to include the essential part, that is the system's components and their connections. First, because it will be an SDN network, its elements have to be present. This type of network allows a dynamic topology, in other words, an easy and quick instantiation of traffic rules and components, namely switches and, at least, a controller. So, the architecture has to have a controller and an area to represent the switches that will be instantiated to compose the network. The traffic circulates through the switches, elements responsible to forward it to the right destination. For that reason, at least one switch has to be connected to the external area (the Internet), which will be the entry point to the company for the external traffic. Regarding the controller, the core of an SDN network, it has to be connected with all of the switches, so it can compute their behavior and do the processing part of the traffic

forwarding. Besides the network, the system has to have components to meet the requirements defined: a component to classify traffic and another to implement defense mechanisms. As said before, the traffic classification component can have only one or multiple functions, knowing that some of them can be implemented using existing tools. Also, it has to be connected to the entry point of the network, one of the other switches and the controller, so it can receive the traffic from the outside, the traffic circulating inside and to send alerts and share information, respectively. The component responsible for the defense mechanisms' implementation can have one or several functions, implemented using tools or not, but, unlike the traffic classifier, it has to be the most isolated possible from the network, so it does not interfere with it. One of the suggested defense mechanisms was the use of honeynet, which consists of a network of machines (or containers) where the attackers can mess with the system all they want and information of their behavior is gathered, for learning purposes. This type of mechanism needs to be isolated from the rest of the network, or else the attacker can pass the defenses and damage the network. Hence, it only needs to be connected with one switch of the network and the controller, so the desired traffic can be redirected through it, and it receives orders of what to do, respectively.

The combination of the components and elements referenced before results in the architecture of the system, which can implement different kinds of defenses. First, since the traffic classifier is linked to the entry point of the network, the system is capable of performing a perimeter defense, that is, to defend the company's network from threats originating in the outside. Second, because the component of traffic classification is also linked to the switches inside the company's network, the system is able to deal with threats from the inside of it. Third, seeing that exists a component capable of isolating attackers, the system has the ability to inspect and gather information about specific attacks, so the system's administrator can modify or implement new security measures based on this.

All of the features and functionalities previously described can be compiled in a network architecture, which can be seen in Figure 3.3. It shows how a company's network can be divided into segments, some with components responsible for the operation of the network, while others just grouping equipment based on their capabilities to facilitate the job of the network administrator.

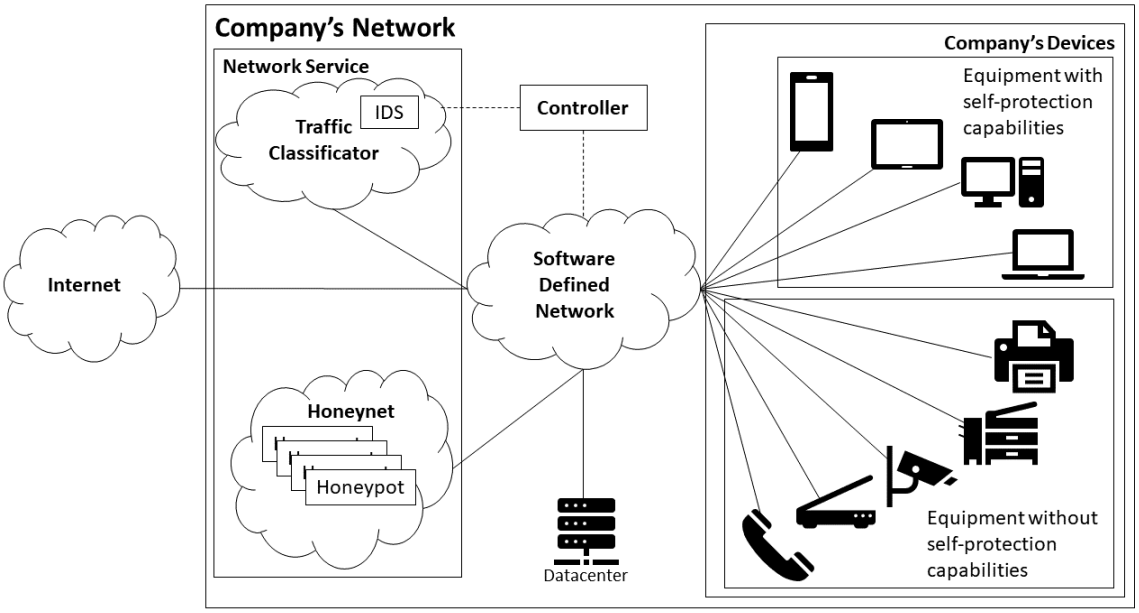


Figure 3.3: Business network's architecture with attackers deception functionalities

3.4 Modules

In a system, there are several modules responsible for its operation, each with one or more functions, which can be seen in the architecture design (Figure 3.3). For the system to work properly, each module has to have functions that can be linked to the others, so they can operate as a whole.

3.4.1 SDN Network

One of the system's requirements is the use of an SDN network. This type of network is distinguished by the separation of the control plane from the data plane, where the former computes and controls the functions of the latter. The control plane is normally composed of one controller, but it can have more, working in a distributed matter. One or more switches with OpenFlow support form the data plane, responsible to circulate and redirect the traffic to the correct destination.

The controller, since it controls the whole network and interacts with other modules, can be considered one itself. So, it will be described in another subsection (3.4.2). Still, for the description of this module, it is important to note that the controller is capable of implementing rules that redirect traffic, i.e., is capable of changing the normal course of specified traffic.

As said, the switches constitute the data plane of the network, since they are responsible for the traffic redirection (packet forwarding). When the switches start working and the controller does not implement any rule yet, they operate as normal switches, that is, forwarding packets to the right destination depending on their headers. However, if the traffic needs to be changed, the controller will implement rules in the switches to do that, and they will

apply them. The rules support several actions, but they all sum up to three types: forward, drop or modify-field, which means that when the switches receive traffic, they will verify the rules they have implemented and will try to match the traffic to one of them. If the traffic matches a rule, then it will be redirected to certain ports/destinations, it will be dropped or some of its fields will be modified, depending on what the rule stipulates. If the traffic does not match any rule available, then the switch will forward the traffic as normal. The fact that the controller is capable of implementing rules to control the traffic allows for a quick configuration of the data plane, thus helping a quick reaction to any type of anomalous situations.

All of these features make SDN a technology to take into consideration when developing more complex network solutions. Due to its flexibility, programmability, and scalability, SDN can ease the job of a network administrator. By using flow rules, traffic can easily be separated and redirected, and by using the controller, a network administrator can make changes to all of the devices with just one interaction. Also, the administrator can more rapidly check resource consumption and assignment just by using an SDN application that extracts network information. Combining all of the tasks, a system can be developed that automates the management of the network and makes it more dynamic.

3.4.2 Controller

The controller, one of the SDN network's components, is considered a module in this solution because, in addition to its SDN capabilities, it also has more functions and interacts with the other system's modules.

The control plane of an SDN network consists of one or more controllers, which are responsible for the decision-making process of the network. It has four semantic limits: northbound, eastbound, southbound and westbound. The east and westbound are only considered when the network has multiple controllers since these bounds are used for communication between them. The northbound is used to apply applications that work on the network, be to extract information about it or to manage it. After the controller computes information received by the northbound, it will communicate with the data plane (the switches), through OpenFlow protocol, using the southbound, so it can implement rules or extract information to respond to the applications. However, the southbound is bidirectional, so the switches can also communicate with the controller. One example of this is the PacketIn message. The traffic does not go through the controller (or else it would create a bottleneck), but in some situations, it is necessary for the controller to have access to a packet so it can compute what to do with it. Therefore, a PacketIn message is sent to the controller containing a certain packet when a switch cannot match it to any rule, or when there is an explicit action in a rule to do so, and the controller will analyze it and compute a response (normally, a new rule to implement).

Since the controller is the central point of the network, it is responsible for implementing security measures and to communicate with other components responsible for them, such as the traffic classifier (3.4.3). When one of them has information about anomalous situations

(for example, ongoing attacks) whose solution implies a traffic modification, they have to send it to the controller so he can compute adjustments to the existing rules. Despite using the OpenFlow protocol to communicate with the data plane (using the southbound), the northbound of SDN controllers is not regulated, so it does not have a standard protocol to use between the applications and the controller. Therefore, one can use the preferred protocol depending on the situation.

A variety of applications, with distinct functionalities, can be developed to communicate with the controller's northbound and to exchange all kinds of information. Applications can be used by a network administrator to check the network and its devices' status, as well as to implement some control logic that further enhances the management of traffic. Additionally, the controller's functionalities can be extended by modifying its application to accommodate more control logic, eliminating the need for external applications. Both of these features can be used to develop systems that automate network administration tasks. By adding functionalities to the controller, steps regarding network management can be eliminated, saving time in those tasks and decreasing the risk for human error.

3.4.3 Traffic Classification

The traffic classification components can have multiple functions, to better check the traffic that circulates through the network, especially the one that comes from the Internet. However, the aim of this thesis is not to deepen the knowledge about classifying traffic, but to use this as a component of a greater security system. That is the reason why the traffic classifier will be considered as having only one function, and this will be implemented using an existing tool, in this particular case, an IDS (and more specifically, a NIDS).

An NIDS is capable of monitoring a network and inspecting packets (which are important functions in a traffic classifier), but can also generate alerts, which facilitates its interaction with other components in the network, namely the controller, in order to improve the security measures and the system itself. Most common NIDSs, for example, Snort [76], Suricata [79], and Zeek (former Bro) [80] use a signature-based detection method, meaning that they have a set of signatures to which they compare the inspected packets. This method allows not only to detect known attacks (whose signatures have already corresponding rules in the IDS) but also to detect specified traffic since IDSs allow the user to write his own rules, meaning the traffic classifier can identify attacks and also other important traffic.

All of the network's traffic has to pass through the traffic classifier, so the IDS can do its job. When the traffic is sent to the IDS, he will compare it against the set of rules in its configuration and try to match the traffic. If the traffic does not match any rules, it will be allowed to be forwarded to its destination. If the traffic matches some rule, the IDS will generate an alert and send it to the controller, where he will process the information and apply actions accordingly to its configuration: block or redirect traffic. If the traffic is to be blocked, the controller applies the corresponding rule to the switches and it will be dropped. If the traffic is to be redirected (possibly to the honeynet), the controller will apply rules to the switches that will redirect it to the new destination. This last option is not applied to all

traffic identified as an attack, but only to the one corresponding to specified attacks worthy of further inspection.

3.4.4 Honeynet

There are many attacks whose signatures are known, and which are easily identified, especially when using an IDS. However, many times the attackers simply use these attacks as a means to start a more complex and intrusive one, which is unrecognizable to the system. One way to take advantage of this is to use honeypots or even a honeynet.

A honeynet can be seen as a deception network which lures attackers to gain information on threats. It is composed of several honeypots, which provide real systems with vulnerabilities (for example, by exposing sensitive information), to appeal to attackers and attract them. Once inside a honeypot, it will collect information concerning the behavior of the attacker to learn more about his attack, and afterward, the system can be improved using this information.

Since this component is by definition compromised, it has to be isolated from the rest of the network so the attackers that will interact with it don't damage the company's operation. In this isolated network, honeypots will be implemented that mimic one or more services of the company, which means that they have to be as equal as possible to the real attackers' target. Seeing that the services provided by the company are regularly updated, a better way to replicate them as honeypots is to do it dynamically, that is, at the moment they are needed, therefore reducing the possibility of the attacker instantly realizing that he is being tricked and terminating its actions. One of the methods used to replicate services is containers, software that runs on top of the operating system and that mimics the function of a Virtual Machine (VM). The containers can be used instead of full VMs to save resources and to ease the implementation of services. Full VMs normally virtualize hardware, which increases resource consumption, making them a great solution when there is the need for a fully emulated physical machine. However, when is necessary to implement many VMs and the lack of full implementation of a machine is not an issue, containers can be a great alternative. They allow for a greater number of VM deployments in only one virtual machine, while fully emulating an OS. Despite having the drawback of being limited to the same OS of the host machine, they can be used in a variety of situations, having the major advantages of reducing infrastructure cost and resource consumption.

The function used in the traffic classifier is an IDS, and it is not capable of recognizing zero-day attacks, that is, attacks that are not previously known. Because of this, the honeynet will be used to trap attackers that perform attacks known to be the opening for others. So, when the traffic classifier matches traffic with one of these types of attacks, it will generate an alert and send it to the controller. Then, the controller will evaluate the information and compute the necessary actions, in this case, the replication of the service being attacked and the redirection of the attack's traffic. The honeynet (which, initially, will be only one VM) will be notified of this, will create a copy of the required service and will create a container to implement said copy. Depending on the target of the attack, pre-configured containers or images can exist that have a copy of some service. For example, surveillance cameras or

printers only require updates to its proprietary software (since they don't support any software besides this), so containers that implement a copy of this software can already be running and the honeynet only has to redirect the attack's traffic to it, saving time and reducing the risk of the attacker sensing something and give up on the attack.

3.5 Functionalities

The aim of the proposed solution is to ease and complement the network's administrator job, that is, to automate some of its tasks and to add more functionalities to help its maintenance. Since the solution is composed of independent components, one may use only some of them or may implement them with others or in other systems, meaning it can be adapted to several network scenarios and configurations and can solve different problems.

Concerning the problem's description, it was mentioned the existence of two groups of equipment, those who can self-protect themselves and those who don't have that capability. Since the proposed solution is based on SDN, it has the ability of more dynamically and efficiently implement segregation mechanisms, through the use of flow rules. These mechanisms can be, for example, the configuration of a switch for each group of equipment, staying each on different private networks; or else the configuration of VLANs (one for each group), not being necessary the configuration of additional switches. Besides these segregation mechanisms, SDN also allows a more fluent traffic redirection, which facilitates the passage of traffic through specific components that are responsible for the security measures and its consequent redirection to the desired destination. The segregation mechanisms can also be used for other functions related to the division of the company's traffic, namely the separation of traffic containing restrict information (containing it inside the company's network) from the traffic related to employees' communications with the outside of the company.

The solution includes a traffic classification component, which in this case is just an IDS that identifies attacks. Despite the IDS being able to identify not only attacks but also certain traffic, the classifier can include more functions to expand its features. If the component has different classification chains, that is, distinct classification flows for the traffic to go through, the component itself can separate traffic accordingly to several parameters and the only difference is the traffic being injected in the initial point of the desired classification chain instead of having only one entry point.

Another component of the solution is the honeynet. This network, being isolated, introduces the capability of malicious traffic containment and also the capability of equipment replication. This translates into a function of traffic analysis, which can be used to improve the current security measures. Besides this, since it is an independent component, there is the possibility to replicate this network to be used as a test center, that is, existing a network in the company in which equipment is replicated to test newly developed software without taking the risk of damage the company's networks.

The SDN controller is the central point of the network's control, and it knows the network's topology, which enables easy and agile implementation of components and functions. Because of this, there is the possibility to add, eliminate, replicate or modify them without affecting

the rest of the company's network. Furthermore, it allows the automation of the network's management and supervision. This means that an application in the controller, with access to all the network's information, is capable of reacting to situations and problems that may arise and is able to implement new measures. This eliminates the need for constant supervision by the administrator since he can receive notifications of what is going on in the network and only react (by manually implementing measures) when strictly necessary.

Because of the functionalities previously described, the SDN controller is largely responsible for the fulfillment of the use cases. Due to its logic, management tasks can easily be translated to commands that implement them in the network devices. Using the northbound, information about the network status can be extracted and used to compute responses to network events, without the need for human intervention. With these features, applications can be developed that exchange information with other modules and interact with the controller to implement managing tasks and to handle security measures scattered in the network.

CHAPTER 4

Implementation

In order to validate the solution described [chapter 3], an implementation of it would be needed, to evaluate its components and their interaction with each other. Given the resources available for the elaboration of this thesis, it was not possible to fully implement the solution since that required a real company network. Therefore, it was decided to implement some scenarios on a smaller scale that could demonstrate that the solution could work. To do this, it was necessary to choose the technologies that better suited the solution and further specify how the components would be implemented and linked.

4.1 Technologies

In the solution design, it was determined that SDN would help to meet the requirements, considering its advantages. So, in the implementation, this was the first technology to be considered. However, some of the requirements do not restrict which technologies could be used to implement them, so, to choose these, some deliberation and experimentation were required. Figure 4.1 shows the technologies used in each component of the architecture.

Regarding the SDN network, this was chosen based on its popularity and functionalities. SDN networks are rising in popularity since their appearance, because of their functionalities and advantages. This type of network allows a dynamic topology, with an easy and quick instantiation of traffic rules and components (namely, switches), since the controller is the center of the network and has a view of its entirety. This is the reason why this type of network is chosen over a legacy network because the latter is practically inflexible (its topology is fixed, too many initial configurations are required and network re-configuration is almost impractical). The main components of SDN networks are the controller and the switches, which translates in only two technologies used, having the switch technology to support OpenFlow (the protocol used by the controller to communicate with the switches and vice-versa). There are some switch software that support OpenFlow, and the one chosen for this implementation was the Open vSwitch, which is a software that simulates a virtual switch, providing functionalities in the context of security, monitoring, QoS, and automated control, being one of these

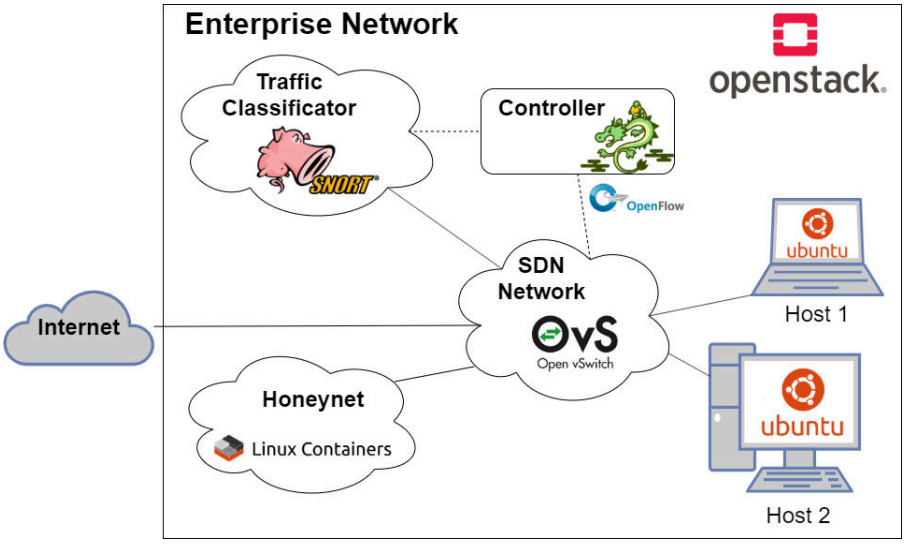


Figure 4.1: Architecture of the system implemented

functionalities the support for use and development of the OpenFlow protocol. [15] As for the controller, there are many options available, but for this implementation Ryu was used, which is a controller that provides software components to facilitate the management and control of an SDN network, having also support to the OpenFlow protocol. [52]

The SDN network's components were chosen, but it was missing a platform to implement them, one capable of supporting not only the network and its components but also the additional components of the system and hosts. Thus, the OpenStack platform was used, seeing that it is an IaaS that provides a cloud operating system capable of controlling computing, storage, and network resources. This allows the minimally faithful replication of a business network on a virtual level, monetizing the physical resources available. [1] OpenStack supports the SDN network but also allows the creation of VMs, which were used to simulate types of hardware that can exist inside a company and hosts. To standardize the initialization of the VMs and to minimize the risk of problems between the components, only one OS was used to implement them: Linux, specifically Ubuntu Server 18.04.

In addition to the network and the required components (hosts and company's equipment), the solution introduces others that constitute the system. One of them is the traffic classificator, which in this case is composed by an IDS. The one used was Snort, which, besides detecting intrusions, is also capable of working as an IPS, being capable of analyzing traffic on real-time and logging packets (but this last feature was not used). [76] Snort is based on rules to perceive if certain traffic could be related to an attack or an attempt. Although there exist rules already developed that are capable of identifying malicious traffic (usually, applicable to known and recurrent attacks), Snort can accept rules developed by the user, which means the network's administrator can select traffic that he considers relevant. When the traffic is passed through the IDS, this will try to match it with the rules available, and Snort can be configurated to generate alerts when this happens. In the system, when an alert is raised, the

[1] `https://www.openstack.org/software/`

controller has to be informed, so it can act upon it and apply rules accordingly. Thus, the IDS and the controller have to communicate (through the respective VMs). To achieve this, a network socket was used to create a communication channel between the two components. A network socket is a two-way link, composed of two endpoints, which are two programs that are running in the network, and through which they transmit information. This communication is made using the TCP protocol. [2]

Besides the traffic classifier, the system also has another component added to the network: the honeynet. In this, honeypots are created to redirect attackers' traffic in order to collect information about it. Virtualization software was used to deploy the honeypots, and Linux Container Daemon (LXD) was chosen to do it, which allows the imitation of existing systems and hardware and to create these copies dynamically. LXD is a hypervisor built on top of LinuX Containers (LXC), using this to mimic VMs, but making it easier to scale the system, because it tolerates a greater number of containers than the replication of full VMs. [3]

4.2 Framework Architecture

The system was implemented based on the proposed solution's architecture. Recalling, it has an SDN network (with a controller and switches), a traffic classifier (an IDS) and a honeynet, which were all implemented as components in the system. Figure 4.2 displays the implementation's architecture, which will be further explained.

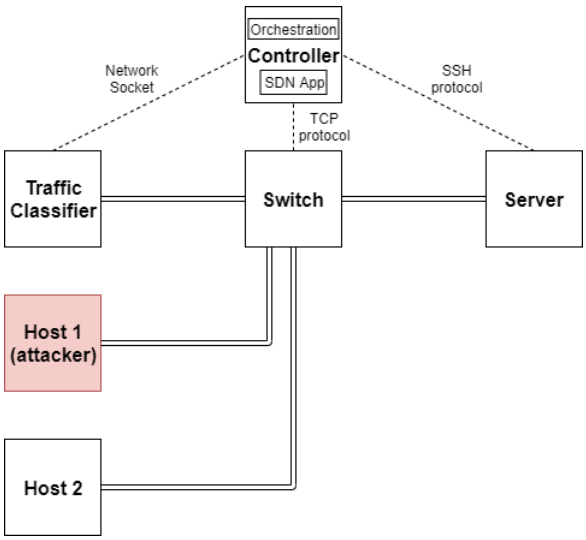


Figure 4.2: Architecture of the system implemented

In the system, the SDN network is its core, since it has control of all of the traffic that circulates through the business' network, being constituted by Ryu (the controller) and an OVS (the switch). Each network element was installed in a VM, to better replicate a real-world scenario, since the switches are normally scattered throughout the business' building. The controller has to be "isolated" from the common traffic of the network, to ensure that only

[2] https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html
[3] https://linuxcontainers.org/lxd/introduction/

authorized people (namely, the network's administrator) have access to it and so it is not flooded with useless information (company's traffic). To try to achieve that in OpenStack (the platform used), Virtual Extensible Local Area Network (VxLAN) tunnels were used to create connections between all of the system's elements, except for the controller. To simplify the configuration of the tunnels and to ensure that they were always up and running, they were set up using "internal" OVS switches, that is, an OVS was installed in each elements' VM and the tunnels were created in there. Therefore, each system's element has an OVS (inside the respective VM) and a VxLAN tunnel connecting it to the central OVS (the VM implementing it), except for the controller, which does not have any tunnel or OVS installed. This way, Ryu communicates with the necessary system's components (OVS, IDS, and honeynet) only through TCP protocol.

Regarding the IDS (traffic classifier), besides its connection (VxLAN tunnel) to the switch, it has a connection to the controller, since it has to transmit to it information concerning the traffic it analyzes. To send this information to Ryu, a network socket is used, one that already existed. Ryu already had a version of the controller that was capable of communicating with Snort, using "snortlib", included in the Ryu libraries folder. This controller's version, if implemented without modifications, is only capable of processing Snort alerts if the IDS is installed on the same machine (through the use of a Unix Domain Socket). However, modifying a parameter ("unixsock") on the controller and running a script ("pigrelay.py") on the Snort VM, Ryu can receive and process alerts originated from other VM. "Pigrelay" is a script that sends the alerts generated by Snort to the IP address and port specified, in this case, the controller's IP address. This was the method used for the IDS to communicate with the controller.

The honeynet was implemented using virtualization software, in this case, LXD. This software was installed in a VM (the "server") so the containers could be created. The controller communicates with this machine via SSH when there is the need to create new honeypots, that is, new containers.

To simulate hosts, VMs with Ubuntu Server 18.04 were used. Two hosts were created, one to emulate a normal user and another to emulate an attacker.

Recapitulating, the system consists of six VMs: a controller, an OVS, a honeynet ("server"), an IDS and two hosts. Between the OVS and the other machines (except the controller), exist VxLAN tunnels, and the controller communicates with the other machines through TCP protocol (or SSH).

4.3 Components

Since the proposed solution is quite complex, it was necessary to divide its implementation into several parts, which match the modules described in section 3.4.

In order to make the solution more versatile and flexible, the modules were implemented independently and only after they were interconnected. This translates into a scalable (more modules can be added or removed, as needed), malleable (one can modify modules without

the need to modify and make the whole system temporarily unavailable) and contained (the unavailability or inoperability of a module does not affect the system as a whole) system.

4.3.1 SDN Network

Initially, the platform that was being used to replicate the SDN network was Mininet [98], which creates an instant virtual network. This software allows the creation of a network, comprising controllers, switches, and hosts, granting the flexibility to build and use different topologies, permitting the use of any number of hosts desired and allowing the use of independent controllers (like Ryu, ODL, or any other) and different types of switches (like, OVS). However, the use of Mininet was dismissed due to some aspects, namely, although it is flexible and scalable, it limits the resources and services that can be deployed, therefore, not being able to replicate a minimally realistic business network. In addition to the limitation of Mininet in the context of the proposed solution and it's consequent dismissal as a platform to replicate the SDN network, it came to the knowledge that an OpenStack platform was already available as a resource to the members of the project of which this thesis is a part of. After some research on the platform, it was deliberated that it would be the one on which the solution was to be implemented.

Therefore, OpenStack is the platform where the SDN network is reproduced, having a VM whose sole purpose is to function as a switch (the OVS switch) and another containing the controller (Ryu). In a real network, the equipment and components of the network will be connected through physical cables and policies will exist to ensure that each network element only interacts with what concerns it. Meaning, for example, that a smartphone of a visitor that connects to the Wi-Fi only has access to the Internet (the outside of the company); and that a computer of the Human Resources cannot access the information concerning the Investigation Department. In order to reproduce this scenario, VxLAN tunnels were used to connect the VMs of the different components, since all of them share the same private network in the OpenStack platform. The tunnels ensure that the traffic circulates in the predefined paths and guarantees that the experiment is not jeopardized.

The network switch (and the switches inside each VM) were implemented using the version 2.9.2 of OVS. The network switch was configured to allow OpenFlow version 1.0 and 1.3.

4.3.2 Controller

Since the solution was meant for a business-like network, the first choice for the controller was the OpenDaylight, because it is supported by companies and users worldwide and can be integrated with several vendor solutions and apps [4]. ODL is a widely known SDN controller and a well developed one, offering an extensive range of functionalities and applications. Despite having very good and extended documentation, this controller also has a long learning curve. Since the beginning of the solution's implementation and for a very long time, the ODL controller was tried to be used. First, the development of a northbound application to implement the desired functions into the controller was tried, but after many tries and

[4] `https://www.opendaylight.org/what-we-do/odl-platform-overview`

experimentations, this was not achieved. So, the next step was to make use of the REST API of the controller, designated *Restconf* [5], through which flows would be added, modified and deleted. This approach worked up to a certain extent, but in spite of that, it was not possible to make it fully functional. So, a point was reached where it was no longer possible to work with the controller and to implement the intended functionalities.

As a result of this, other controllers were researched to find a substitute. The chosen alternative was Ryu. This controller, despite being more simple and not offering as many functionalities as the ODL, was considered sufficient for the demonstration of a working solution. Furthermore, much time had already passed since the beginning of the project, and since the learning curve of Ryu is relatively short, it was decided to use it so as not to waste much more time. That said, Ryu is utilized as the controller for the SDN network, being the central point of the network, responsible for controlling the traffic. To do that, it controls the OVS switch, by implementing the necessary flows on it, for them to route the traffic to where it needs to be sent. The flows concerning the security and QoS of the company's traffic are implemented as soon as the controller starts working, to guarantee the normal activity of the business. Besides these flows, others are implemented dynamically, based on the alerts sent by the traffic classification component. When the controller receives an alert sent by the IDS, it acts upon it, first retrieving information about the traffic (namely, IPv4 source and destination addresses, source, and destination MAC addresses and protocol), then generating the request and finally implementing the flow to modify the normal route of this traffic. The REST API of Ryu (*ofctl_rest*) was tried to add flows to the switch, however, after many tries, experimentation and not having been discovered the problem, this method had to be left. Instead, the flows are implemented on the OVS through *ovs-ofctl* (OVS command-line tool to administer OpenFlow switches [6]) and SSH commands.

As described, the Ryu controller interacts with the OVS switch and with the IDS through its south and northbound, respectively. To communicate with the switch, theoretically, the controller uses the OpenFlow protocol, to implement the flows on it (but, in this case, SSH protocol is used). Regarding the IDS, the communication between the two components is done by a network socket, using the TCP protocol.

Normally, it is through an SDN application, linked to the northbound, that the network administrator interacts with the controller, to implement some control logic or to extract information regarding the network devices. However, in this implementation the controller application was extended to accommodate inside it the SDN application, meaning that the latter is integrated into the former. This application reacts to network events and also acts as an orchestration module. When the IDS sends an alert to the controller, this will receive it and the application will compute a response. First, the message camp in the alert will be extracted and processed to assert which action to take: block or redirect traffic. Then, according to the action, the application will generate the respective OpenFlow commands, and the controller will apply them in the switch. When dealing with traffic redirection, it is

[5] `https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Restconf`
[6] `http://www.openvswitch.org/support/dist-docs/ovs-ofctl.8.txt`

also necessary to send orders to the honeynet, being the application responsible for sending the command to execute the honeynet script, thus having also an orchestrator functionality.

4.3.3 Orchestration

A business network, being wide and complex, requires the need for an orchestrator, to make its management and monitoring easier. However, regarding the scope of the thesis, and taking into consideration the time frame for its elaboration, it was decided that there was no need for implementing a fully functional orchestrator, being enough the existence of an element capable of carrying out the basic functions required by the scenario being implemented.

Since the central point of the network is the controller, its VM is the obvious and consequential choice to hold the component responsible for the orchestration, being its function to operate the underlying network and the elements that constitute it. Bash was the language used to perform the tasks of an orchestrator, mainly because it can be executed almost across all OSs. Several scripts were developed, each one for a different set of tasks.

The first task to do is install and configure the system. To that end, scripts were created, one for each component (VM). In these scripts, there are some commands that are the same, namely the first to be executed, responsible for updating the OS and installing some useful software. In the controller's script, there are only two tasks: installing Ryu and copying the developed controller's script to the respective folder. In the scripts to configure the hosts, there are only commands for the installation of OVS, the VxLAN tunnel creation and the configuration of the bridge's interface. In the switch configuration script, in addition to the same tasks as in the hosts' scripts, there are also commands to set the controller's IP address (and port) and to create the traffic mirror to the IDS port. These scripts are relatively simple, unlike the server and IDS scripts. These two components are more complex, so it is understandable that the respective scripts would be too. Both components' scripts have the same tasks as the hosts' script (install OVS, create a tunnel and configure bridge), but they have additional tasks. In the case of the IDS script, it includes the configuration of a flow to accept the mirrored traffic (otherwise, it would be dropped), the installation and configuration of Snort (which is very complex and has many commands) and the installation and configuration of *pigrelay* (responsible for communicating the alerts to the controller). Concerning the server script, it also has the installation and configuration of LXD (the virtualization software), routing configuration (to allow the redirection of traffic between bridges), the creation of the server container and the initialization of its services. All of the scripts described are responsible for the initial configuration of the several system components.

Besides the scripts to configure the initial state of the system, there is one more script, which is responsible for the attacker's deception (meaning it is used while the system is up and running). This script is in charge of the tasks regarding the honeynet system. It starts by accepting the information about specific traffic, and then, replicates the container that is being targeted and creates the necessary iptable rules, so the specified traffic will be redirected to the newly created container. The controller is responsible for starting this scripts' execution whenever the creation of a honeypot is needed.

4.3.4 Traffic Classification

The traffic classification component comprises only the IDS, which is Snort. This was the IDS chosen from the set of Snort, Suricata, and Zeek (former Bro). Of the three, Zeek was the first one to be dismissed, since it has a long learning curve and the time necessary to learning could be used in the development of the remainder of the project. Between Snort and Suricata, both have similar features (namely the rules writing format and some configuration options) and good documentation, but eventually, the choice fell on Snort, for a few reasons. First, Snort is more widely used in the articles of the same scope and as an IDS, so there are more learning material available and more examples to understand its behavior, including the ones present in articles. Second, Snort has a configuration that is easier to understand and modify, which is concentrated in a single file, whereas Suricata has different files for configuration and requires more knowledge of the software to understand where to configure something. Third, Ryu (the chosen controller) already had a library to work with Snort, as well as a script to allow the communication between the two elements when they are installed on different machines (*pigrelay*). Therefore, despite Suricata having the same features as Snort, and additional ones (such as multithreading and leveraging the Snort ruleset [78]), the last was chosen to use in this project.

Snort analyzes all the traffic that circulates in the network, being this possible by mirroring all traffic that goes through the switch. This means that, whenever some packet arrives at the switch, besides being routed to its intended destination, a copy of it is made and sent to Snort, via its port in the switch. Snort goes through every packet in the network and runs them against the rules predefined in its configuration. If a packet does not match any rule, nothing is done. However, when a packet matches a rule, an alert is raised and it, along with the packet, is sent, via a network socket, to the controller, so it can act upon this information. The predefined rules used on the Snort's configuration have messages that can describe them. In this case, this functionality is used to enclose important information that needs to be passed to the controller, such as, identify if traffic matched needs to be blocked or needs to be sent to the honeynet.

The version of Snort used in the implementation was the latest available, version 2.9.15.

4.3.5 Honeynet

A honeynet is a type of honeypot, specifically a high-interaction one, that is mainly used to gather information on threats in a system or network since it is designed to capture extensive information. The difference between low and high-interaction honeypots is that the latter provides real systems, applications, and services, usually vulnerable ones, so attackers can recognize and interact with them. Honeynets can be seen as a network that contains one or more honeypots, and, considering that honeypots are not production systems, the honeynet itself will not have sensitive information or services regarding the environment where it is deployed. Hence, it is assumed that all activity within it is malicious or unauthorized, and all of the information regarding this can be used to analyze the threats, having no need to filter

it. [7]

Considering that there were not enough resources to implement a real simulation of a business-like network, this created some restrictions in the choices made about the honeynet implementation. The OpenStack platform has the ability to create VMs, so this was the first choice to build honeypots to compose the honeynet. However, after some experimentation, it was determined that the instantiation of an OpenStack VM would take up much time and would not be viable to implement the honeypots. Hence, it was decided that virtualization software would be used to this end. Nevertheless, the software still had to be run in a machine on the OpenStack platform. Thus, despite requiring much time to deploy a VM, this technology was made responsible for the replication of the machine where the honeynet resides. To mitigate this disadvantage, when the original VM that constitute the honeynet would be almost at its maximum capacity, another would be instantiated, reducing the time of waiting by the users, since the original VM would continue to fill its resources and when it was full the second VM would assume its place, and by then this would be fully instantiated.

Based on the dismissal of OpenStack as the virtualization software, other software was considered, namely containers, since they would be faster to deploy and easier to manage because they only virtualize the OS and not the hardware (in the case of VMs)[8]. One of the most used software to deploy containers is Docker. However, it is a bit restrictive and as a long learning curve, so it was dismissed and LXD was considered. This software, despite applying the concept of a container, can almost emulate a VM, overcoming some of the problems brought by containers, namely the limited access to the network from inside a container[9]. So, LXD was chosen because it could emulate a VM without the need for more resources and without taking up so much time.

In the system, the honeynet is implemented in a VM (the server), thereby being isolated in it and making it easier to gather information about the attacks. This VM is where the honeypots will be created to capture malicious traffic (attacks or attempts), considering that the honeynet's goal is to inspect the actions of possible attackers to prevent future attacks and improve the security measures already in place. The VM has, initially, just one container, the one responsible for providing the services available in the company's server, and it communicates with any user that needs its services. When the controller receives information about the traffic that is suspicious and redirection is required, it sends an order to VM to replicate the server container and redirect the suspicious traffic to it. As a result, a possible attacker interacts with the replicated container and performs actions that latter will be scrutinized to ensure that the system is protected against it, never interfering with the traffic from legitimate users that try to access the company's server.

To deploy the honeypots, the version 3.0.3 of LXD was used.

[7] `http://old.honeynet.org/papers/honeynet/`

[8] `https://www.electronicdesign.com/dev-tools/what-s-difference-between-containers-and-virtual-machines`

[9] `https://linuxhint.com/lxd-vs-docker/`

4.3.6 Hosts

To emulate a real company's network, some hosts had to be simulated, to demonstrate the interactions of users with the system. As explained before, the platform used to implement the system was OpenStack, already installed and configured by the department, therefore, it had some images of different OSs. The available images included Debian 8 and 9, ArchLinux, CentOS 7, Ubuntu 14.04, 16.06 and 18.04. From these images, the one chosen to use as the hosts' OS was Ubuntu Server 18.04, because it is a Linux distribution with some of the packages needed already available, it has good support and, regarding the release, it was the newest available.

CHAPTER 5

Results and Evaluation

After implementing the system and verifying its operation, it could be tested and evaluated. To that end and based on the system's requirements, some tests were performed to access its validity.

Regarding the system described, after its implementation, the requirements compiled in section 3.1 were revisited to assess the system and understand what needed to be tested in other to confirm if the requirements were met. One of the requirements was the use of SDN, which can be verified in the implementation of the system, since it uses an SDN controller together with a switch, making use of the OpenFlow standard protocol to communicate between the two components, and being the central point of the entire network. Concerning the evaluation, there are two requirements that are related: the existence of automated traffic classification and the dynamic/automatic manipulation of traffic. Since the latter is dependent on the former because there can not be a traffic manipulation if the traffic is not classified, the two must be evaluated together, making this one of the aspects to be evaluated through tests. Another requirement was a component capable of implementing measures as a reaction to anomalous situations, which also has to be evaluated through tests. At this point, two aspects were referred that need to be assessed. However, they are qualitative ones, meaning that its validation is done through a yes or no question: is the requirement done or not? As opposed to this, some requirements require a quantitative assessment. Although one of the requirements can be perceived as a qualitative one, the response time may not be noticeable by the attacker, it also can be a quantitative one, being easier to compare exact values from different scenarios as opposed to subjective values.

The system's operation can be divided into the following steps:

1. All the traffic that comes through the central switch is passed through the IDS to be classified;
2. When the IDS matches the traffic with a rule, it raises an alert to the controller and notifies him about what actions to take regarding it;
3. The controller computes a response;
4. Response can only be one of two actions:

a) Block the traffic;

b) Redirect traffic to a honeypot (in the honeynet).

Based on these steps and the aspects previously mentioned, two scenarios were created: one where the traffic is blocked and another where it is redirected. For each scenario (and for each case), fifty tests were performed, all with the same system configurations. However, due to the use of a platform shared by the entire department, some results were much greater than expected. As to not interfere with the rest of the results, they were discarded based on a confidence interval of 95%.

Regarding the qualitative aspects to be evaluated, specifically, the first one mentioned, both scenarios can demonstrate it since the aspect has to be met so the system can properly function. Both scenarios were measured in time, so the quantitative aspects can also be evaluated by the two. Concerning the other qualitative aspect, the existence of a component capable of implementing measures as a reaction to anomalous situations, it is the target of the second scenario.

5.1 Scenario 1

The first scenario evaluated the response of the system when it was necessary to block traffic. When this is the action needed, the controller simply has to implement rules in the switch to do it. In this scenario, a host tries to perform an attack on the server but is a type of attack well known and not complex, which is not worthy to inspect further. So, the traffic will be blocked and the attack will not be concluded.

As an example of traffic to be blocked, a ping request was used to simulate it, and the response time was used to establish the time for the system to block the referred traffic.

A sequence diagram (Figure 5.1) is presented to show the steps performed during the scenario and the interactions between the several components of the system.

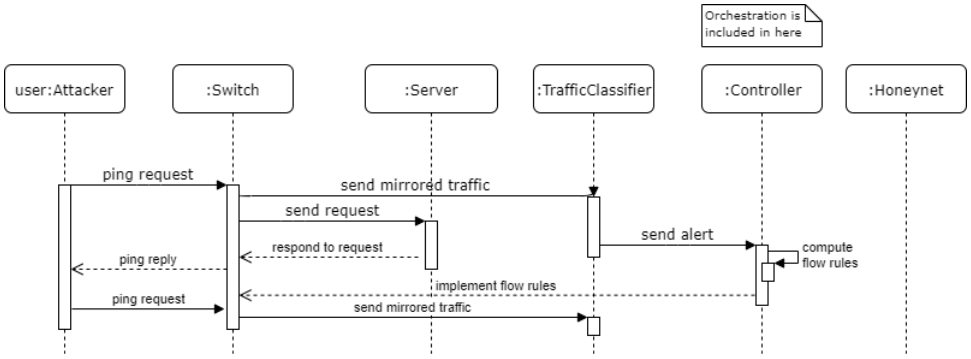


Figure 5.1: Scenario 1 - Sequence Diagram for the attack event

5.1.1 Results

From the tests performed, two values were considered: the response time (time for the traffic to be blocked) and the number of successful responses the attacker receives (number of successful pings).

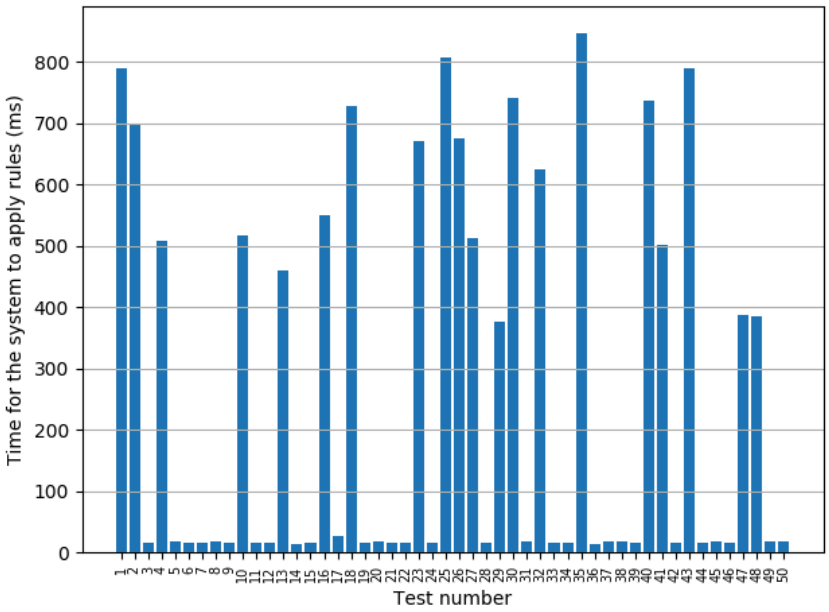


Figure 5.2: Scenario 1 - System blocks traffic

Figure 5.2 demonstrates the response times of each of the fifty tests run. The average of the results is 256.389 milliseconds, whereas the standard deviation is 310.888 milliseconds, probably due to the great difference between the values.

As the graph shows, the response times can be divided into two groups: the ones between 15 and 19 milliseconds and the ones from about 400 milliseconds to higher than 800 milliseconds. The big difference probably occurred due to the platform used to implement the system. As referred, the platform is used by many people and encompasses many resources which generates much latency in the network, introducing delays in the response times and originating values of 400 milliseconds and higher. However, this benefits the tests, since in real networks there are always delays, especially considering that the most likely scenario is for an attacker to be accessing the network from the outside (from the Internet), thus these higher values can correspond to a more realistic setting. But, to draw conclusions from these values is a little difficult, because they have a major range. The lower values probably arise from the fact that the system is setup up in a private network and in a common platform, allowing for more faster exchange of traffic between the several machines, which also can not be used to draw clear conclusions.

Regarding the number of successful connections, the average of the tests was of 2.380 pings, and their standard deviation was 0.490. This means that, in theory, an attacker could only perform two or three successful connections before being cut out of the system, which, considering the type of attacks that are supposed to be blocked, can not result in much damage to the network.

Considering all of these results, the two range of values (inferior to 20 milliseconds and higher than 400) can not be used to draw conclusions by themselves, but the results as a whole can. From the lower results can be concluded that the system in place, specifically the

function of blocking traffic, is of fast action, meaning that it does not introduce almost any delay to the normal flow of traffic. In conjunction with the higher results, it can be concluded that the system can be implemented in a real network, that is, with a vast range of equipment and users, without negatively affecting its normal operation.

Thus, one can conclude that the system can successfully identify and block traffic, doing so in a fast manner.

5.2 Scenario 2

Besides evaluating the same aspects of the first scenario, the second scenario had the goal of evaluating the response to anomalous situations that require deeper inspection. In this scenario, if some of this type of attack is identified, it is necessary the existence of a honeypot so it can be redirected to it. These attacks can be complex or simple attacks that are known to be the opening for more severe attacks. When the controller receives an alert to redirect traffic, he is responsible for implementing the OpenFlow rules, as well as to alert the honeynet (in this case, a VM) to execute actions concerning the honeypots.

In this scenario, a host tries to perform an attack on the server, but the IDS recognizes it and sends an alert to the controller. He will implement rules on the switch to redirect the traffic and, after actions are performed on the honeynet, the attack is redirected to a honeypot. To do these tests, a series of HTTP requests were used to retrieve a web page from the server. And, due to the complexity of the specified aspect to be evaluated, the tests were divided into four slightly different cases.

To better understand the sequence of events and calls performed, a sequence diagram (Figure 5.3) is presented where they can be observed.

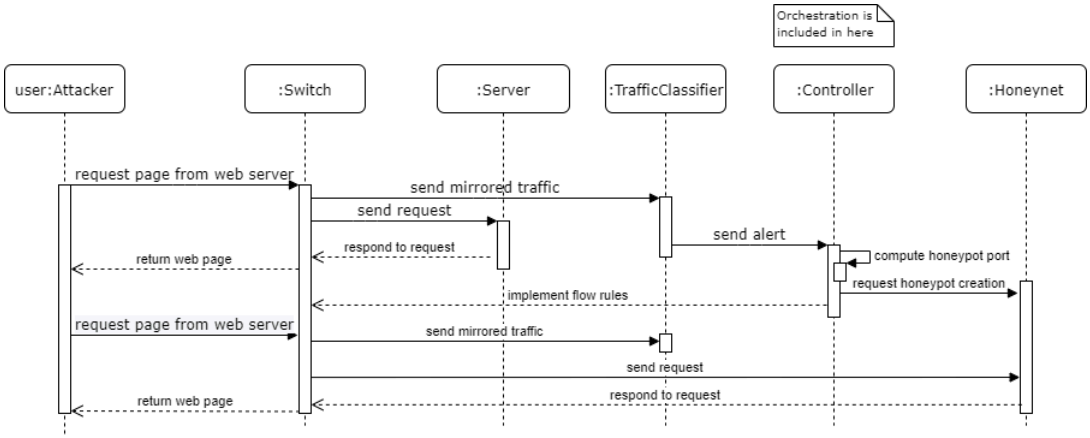


Figure 5.3: Scenario 2 - Sequence Diagram for the attack event

5.2.1 Results

In this scenario, the time between the first request and the first redirected request for the web service was measured to establish the system's response time, as well as the duration of the first redirected request since it is the request that could take up more time and be

noticeable for the attacker. Four cases were tested to assert the best alternative concerning the management of the honeypots and the method for its dynamic instantiation:

Case 1: The container corresponding to the honeypot is instantiated in real-time after the controller receives information to do it;

Case 2: A snapshot of the current honeypot is created and instantiated;

Case 3: There is already a current snapshot of the server, and it is instantiated;

Case 4: A honeypot is already running, being a snapshot of the current state of the server.

Due to the high values of the results obtained regarding the response times, they are presented in seconds, despite having been measured in milliseconds.

Case 1

The first case to be tested was the real-time instantiation of the honeypot. When the honeynet VM receives an order from the controller to create a honeypot, it will create a new container, with the same OS as the server's container, and implement the services available in it. The result is a replica of the server. However, it raises a problem: it can be more easily identified as a honeypot, since, if the attacker can perform at least one successful connection to the server, then it will notice the difference when his traffic is redirected because the services of the honeypot are newly created and configured.

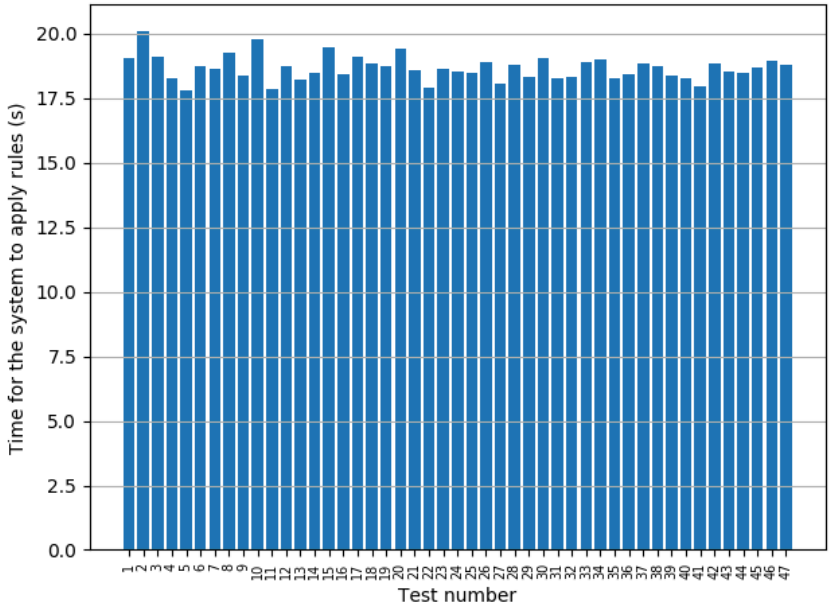


Figure 5.4: Scenario 2, Case 1 Response times - System creates honeypot and redirects traffic to it

The average for the results of this case was 18.797 seconds, with a standard deviation of 0.708 seconds. Concerning the duration of the first redirected requests, their average was of 1.141 seconds, with a standard deviation of 0.276 seconds.

In Figure 5.4 is seen that the response times vary between 17.5 s and 20 s, which can be considered high values, especially taking into account the possibility of an automated attack, which eliminates the delays caused by the human reaction. About the first redirected request,

it takes up to about 1 second, which may be noticeable by an attacker but would be more likely for him to dismiss it, because of the normal delays occurring in networks of this type.

If an attacker performed an automated attack, sending repeated requests (like a DoS attack), he would most probably notice the delay occurred during the first redirected request, making him suspicious and terminating the attack. However, considering that the honeypot is a real-time replica of the state of the server and that real networks will always have delays in their traffic, then perhaps an attacker would be deceived and would proceed with the attack. But, is still important to take notice of the high average of the response times.

Case 2

The second case evaluated the response time of honeypot instantiation by using a snapshot taken in real-time. The honeynet VM, after receiving an order from the controller, will create a snapshot of the server, create a container (honeypot) using it and then start the services inside it. This, as in case 1, results in a server replica where the OS's state is the same as the server, but still having the problem of the services' current state being different from the server's.

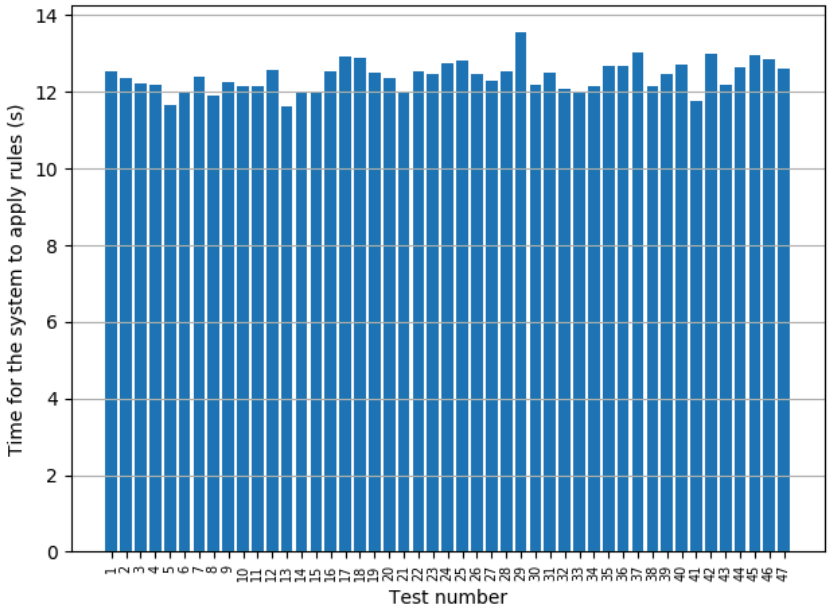


Figure 5.5: Scenario 2, Case 2 Response times - System creates honeypot using created snapshot and redirects traffic to it

The results' average was 12.510 seconds, with a standard deviation of 0.565 seconds, whereas the times for the first redirected request had an average of 1.124 seconds and a standard deviation of 0.254 seconds.

In Figure 5.5 is seen that the response times vary between 11 seconds and 14 seconds, a decrease of 6 seconds compared to the results of case 1. The same conclusions regarding the possibility of an automated attack drawn for case 1 are still applicable in this case. But, a decrease of 6 seconds in the response times is a great improvement towards the deception of

an attacker. Despite that, the first redirected request times, where it is more likely for the attacker to notice the traffic redirection, show an improvement of just 17 milliseconds, which, in human perception, is almost nothing.

Case 3

The third case is almost identical to case 2, only with a step missing. In this case, the honeynet VM, after receiving an order from the controller, still creates a container (honeypot) using a snapshot and then start the services inside it. However, the snapshot used already exists, being previously created using the current state of the server. In this case, it is assumed that every time a change is made to the server, a snapshot is then created to copy its new state. This results in a server replica with the same characteristics and subsequent issues as in case 2.

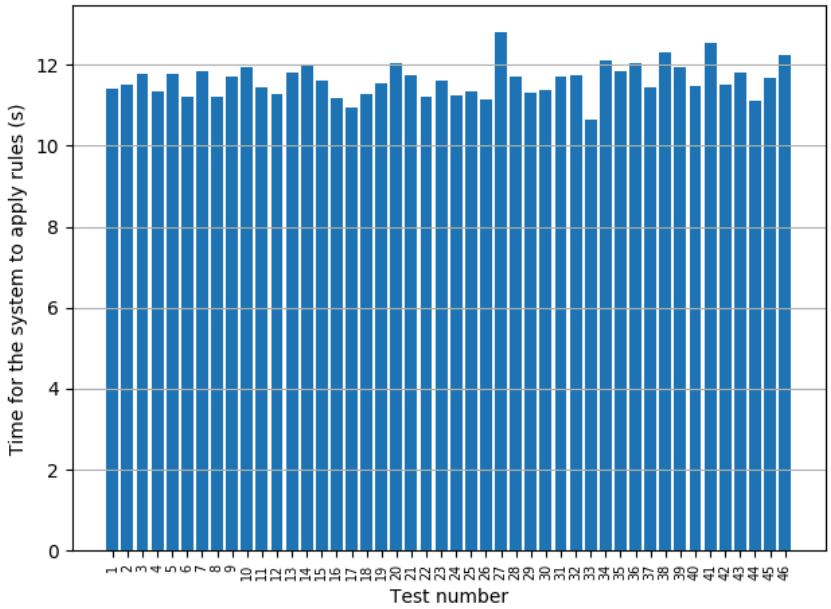


Figure 5.6: Scenario 2, Case 3 Response times - System creates honeypot using existing snapshot and redirects traffic to it

The average of case 3's results was 11.686 seconds with a standard deviation of 0.651 seconds, and the first redirected request times had an average of 1.051 seconds and a standard deviation of 0.124 seconds.

In Figure 5.6 it is seen that the response times vary between 10 seconds and 14 seconds, a decrease of 6 seconds comparing to case 1, thus showing the same improvement of case 2. When comparing this case with case 2, it is observed only an improvement of about a second in the response times, which is not much relevant. But, concerning the first redirected request times, it is shown an improvement of 73 milliseconds, which can be more significant when considering the human perception of time.

Case 4

The fourth case distinguishes itself from the others because only the part referring to the traffic redirection is performed. In this case, the controller does not need to give any order to the honeynet, because it is assumed that a honeypot containing a replica of the current state of the server is already running. Like in case 3, every time the server suffers some change, a snapshot of its new (current) state is created, a container is instantiated using it, and its services are started, creating the honeypot.

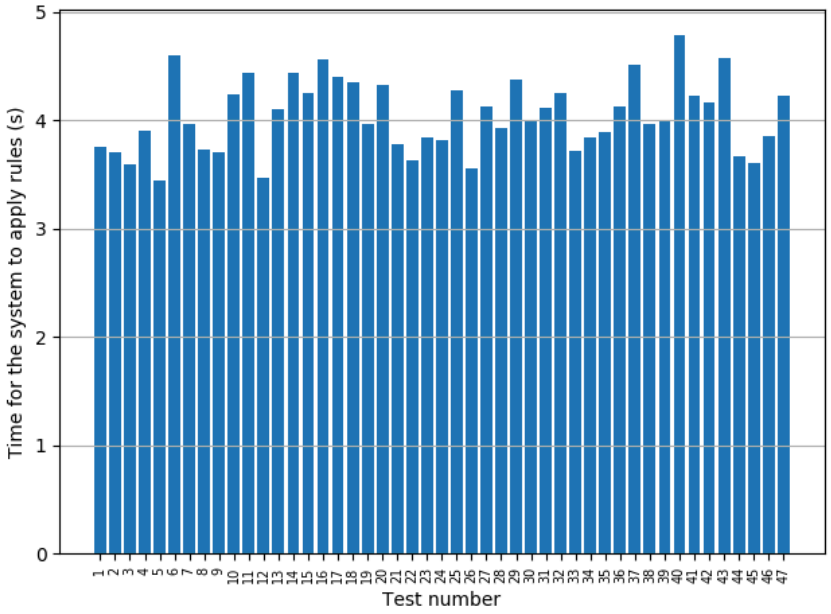


Figure 5.7: Scenario 2, Case 4 Response times - System redirects traffic to existing honeypot

The average of this case's results was 4.114 seconds with a standard deviation of 0.449 seconds, and an average of 1.076 seconds with a standard deviation of 0.186 seconds for the times of the first redirected request.

In Figure 5.7 is seen that the response times vary between 3 seconds and 5 seconds, a huge difference comparing to all of the other cases, especially case 1, existing a difference of 15 seconds between the two. Regarding the first redirected request times, despite having an increase of 25 milliseconds when compared with the case 3 results, it shows an improvement of 65 milliseconds from the case 1 results.

5.3 Evaluation

Based on the results previously shown (sections 5.1 and 5.2), one can conclude that the system implemented can select desired traffic and block or redirect it based on previously defined rules.

5.3.1 Scenario 1

The delays introduced by the platform on which the system was run ultimately contributed to a more realistic setting, demonstrating that in a real scenario the response times vary in a

great range of values, depending on external factors, such as number of connections to the desired resource, number of active users in the system, capacity of the resources, and other ones.

Table 5.1 shows the average and standard deviation for the response times recorded in the tests, as well as for the number of successful pings (successful connections) in each test.

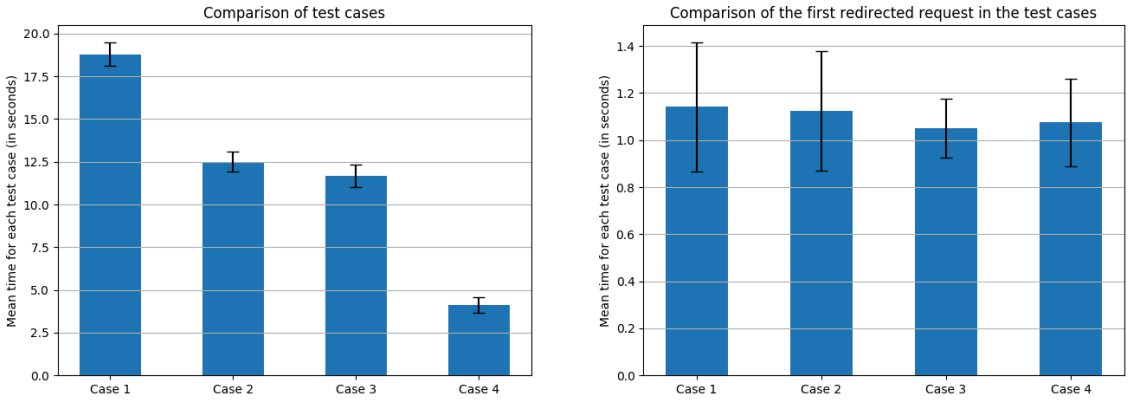
Scenario 1	
Results	256.389 ± 310.888 ms
Number of Successful Pings	2.380 ± 0.490

Table 5.1: Mean and Standard Deviation for parameters in Scenario 1

This scenario had the objective of evaluating the proper operation of the system, having to classify traffic and to dynamically/automatically manipulate traffic, and also evaluating if the traffic manipulation would be noticeable for an attacker. Based on the results obtained, the first objective was met, since the traffic was correctly identified and manipulated, resulting in the blockage of the traffic flow. Regarding the second goal, the results obtained demonstrate that is very probable that an attacker could not notice that his traffic was being blocked, only realizing it when it was too late. As stated, an attacker would not probably see past the higher values registered, since a real network always has great delays and response times. The fact that the number of successful connections was the same for both types of values (higher and lower) also contributes to supporting this claim, demonstrating that only delays were introduced in the system, and not malfunctions or delays resulting from the system itself.

5.3.2 Scenario 2

In scenario 2, since four different cases of tests were performed, one can imply that would necessarily be some case(s) better than the others. To further evaluate and compare the results of each one of them, the average and standard deviation for the response times and for the first redirect request times are shown in Figure 5.8 in the form of graphs, as well as in Table 5.2.



(a) Response Time **(b)** First Redirected Request

Figure 5.8: Scenario 2 - System redirects traffic

Scenario 2		
	Response Time	**First Redirected Request**
Case 1	18.797 ± 0.708 s	1.141 ± 0.276 s
Case 2	12.510 ± 0.565 s	1.124 ± 0.254 s
Case 3	11.686 ± 0.651 s	1.051 ± 0.124 s
Case 4	4.114 ± 0.449 s	1.076 ± 0.186 s

Table 5.2: Mean and Standard Deviation for the cases of Scenario 2

In Figure 5.8a, one can clearly see that case 4 has the best response times. However, this is the case where a new honeypot is not created when an alert is raised, where only occurs the traffic redirection. To guarantee that the copy of the server is exact, each time it would be changed, a new honeypot would have to be created. This raises the question: how would the system respond to several attacks being redirected to that honeypot? This can have two answers. The first is: the first attack would be redirected to the honeypot and the consequent attacks would be disregarded and be blocked instead. Then, when the honeypot would be "empty" again, a new attack would be redirected to it. The second answer could be: all of the attackers would be redirected to the honeypot (or at least some of them). For this case to make sense, one can take into consideration that if a company does not have enough security measures to protect its services, it could occur that several attackers are at the same time trying to perform attacks. This would result in them acknowledging each other and perhaps continuing its "work", which could be simulated with the case of several attackers in the same honeypot. In both cases, there could be several honeypots instantiated and the "rules" of each answer would be applied to all of them.

Regarding the aspects that the scenario was supposed to evaluate, specifically component capable of implementing measures as a reaction to anomalous situations, it can be concluded that all were met. As in scenario 1, the system correctly identified and manipulated the traffic, resulting in the controller giving "orders" to the honeynet, so it could adjust to the situation. Concerning the quantitative results, the traffic manipulation would most probably be noticeable for an attacker, especially in cases 1, 2 and 3, and the results from case 4 are not so decisive. However, as stated, a real network will always have delays and latency, which could result in more attackers being deceived since they could dismiss the higher response times as simply network delays, especially in the case 4, where the values are lower than 5 seconds.

CHAPTER 6

Conclusion and Future Work

In the digital context, there is always someone ready to steal sensitive information or to disrupt the work of others. In the enterprise, this is a more severe problem, since the data handled by it comes from all sources, including people outside the business. This means that when an attack is performed on a company, the consequences will propagate inside and then outside of it, becoming even possible the physical harm of people. Because of this, the security of the company's networks and equipment is a sensitive and very complex matter, which requires constant supervision by an administrator and constant upgrades.

The work developed in this thesis is a contribution to some of these problems. By using SDN, some existing tools and combining them all together, the system developed can be used to implement security measures as needed. Since it was developed based on components, there is the flexibility to implement only what is needed or to extend the functions already existent.

The system is composed of three main components: SDN network, traffic classifier, and honeynet. The traffic enters the company's network by a switch (which supports OpenFlow) and is passed through the traffic classifier. In there, the traffic will be matched against pre-defined rules, and when a match is found, an alert is raised to the controller. He will act upon the information received and will perform actions to block or redirect the traffic, depending on the rules defined in the traffic classifier. If the traffic is to be blocked, it is enough to implement flow rules on the switches of the network. If the traffic is to be redirected, flow rules are implemented also, and an order is sent to the honeynet so it can create honeypots.

Based on the results acquired, one can conclude that, when the traffic is to be blocked, it is done in a fast manner, and a possible attack will be quickly terminated. However, when the traffic is to be redirected, the best case is to already have a honeypot ready with a copy of the current state of the server, and even in this case, the response times are higher than the block traffic scenario. But, the system can still effectively identify traffic, create a honeypot in the honeynet and redirect the identified traffic to it.

6.1 Future Work

One of the improvements to be done should be the conversion of the bash scripts responsible for the initial configuration in a Heat Orchestration Template (HOT) [1], which is used by Heat [2], an orchestration service for the OpenStack platform.

Also, the controller could be changed to the OpenDaylight, since it is the most used in the enterprise context and can probably improve the system's flexibility and scalability.

One other possibility, besides the change of the controller platform, is to add multiple controllers. The environment considered is the entreprise one, which may mean thousands of devices, a wide network (being logically and/or physically), and a great complexity. The use of a distributed control plane, meaning multiple controllers, can improve the scalability of the network, as well as its accuracy, since when there is only one centralized controller and it starts to be flooded with packets, it will create a bottleneck and most probably will start to fail, jeopardizing the entire network. Of course, there are some aspects to consider, some of which were mentioned in the subsection 2.1.3, but is not an idea to be dismissed.

[1] `https://docs.openstack.org/heat/latest/template_guide/index.html`
[2] `https://docs.openstack.org/heat/latest/`

References

[1] T. Bakhshi, "State of the Art and Recent Research Advances in Software Defined Networking", *Wireless Communications and Mobile Computing*, vol. 2017, pp. 1–35, 2017, ISSN: 1530-8669. DOI: 10.1155/2017/7191647.

[2] O.N.F., *SOFTWARE-DEFINED networking: THE new norm for networks*. 2012, vol. 2, pp. 2–6. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[3] K. Greene, "Tr10: Software-defined networking", *MIT Technology Review*, [Online]. Available: http://www2.technologyreview.com/news/412194/tr10-software-defined-networking/.

[4] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A Survey on Software-Defined Networking", *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, 2015, ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2330903.

[5] Open Networking Foundation (ONF), *Open Networking Foundation (ONF)*, 2015. [Online]. Available: https://www.opennetworking.org (visited on 12/05/2019).

[6] Open Networking Foundation, *Software-Defined Networing (SDN) Definition*. [Online]. Available: https://www.opennetworking.org/sdn-definition/ (visited on 12/05/2019).

[7] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks", *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, Jul. 2013, ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6553676.

[8] H. Farhady, H. Lee, and A. Nakao, "Software-Defined Networking: A survey", *Computer Networks*, vol. 81, pp. 79–95, Apr. 2015, ISSN: 13891286. DOI: 10.1016/j.comnet.2015.02.014.

[9] Open Networking Foundation, "SDN Architecture", no. 1, 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR%7B%5C_%7DSDN%7B%5C_%7DARCH%7B%5C_%7D1.0%7B%5C_%7D06062014.pdf.

[10] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey", Jun. 2014. arXiv: 1406.0440. [Online]. Available: http://arxiv.org/abs/1406.0440.

[11] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks", *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014, ISSN: 1553-877X. DOI: 10.1109/SURV.2014.012214.00180.

[12] "Cisco OpFlex", [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html.

[13] P. Saint-Andre, *XMPP The Definite Guide*. O'Reilly, 2009, ISBN: 978-0-596-52126-4.

[14] "OpenFlow Specification (ONF)", [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.pdf.

[15] *Open vSwitch*, 2013. [Online]. Available: https://www.openvswitch.org/ (visited on 12/05/2019).

[16] *Indigo*. [Online]. Available: https://floodlight.atlassian.net/wiki/spaces/indigo1/overview (visited on 12/05/2019).

[17] *Openflowj*. [Online]. Available: https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi (visited on 11/29/2019).

[18] *Openfaucet*. [Online]. Available: https://github.com/rlenglet/openfaucet (visited on 12/05/2019).

[19] *Ofsoftswitch13*. [Online]. Available: https://github.com/CPqD/ofsoftswitch13 (visited on 12/09/2019).

[20] "Pantou: openflow 1.0 for openwrt", [Online]. Available: http://www.openflow.org/%7B%5C%%7D0Awk/index.php/%20(not%20accessible).

[21] *Node.js*. [Online]. Available: https://nodejs.org/ (visited on 12/05/2019).

[22] *Pica8*. [Online]. Available: https://www.pica8.com/product/ (visited on 12/05/2019).

[23] *A10 Networks, Inc.* 2019. [Online]. Available: https://www.a10networks.com/products/ (visited on 12/05/2019).

[24] "Big vSwitch", [Online]. Available: http://www.bigswitch.com/sites/default/files/sdnresources/bvsdatasheet.pdf.

[25] "Brocade ADX Series", [Online]. Available: http://www.brocade.com/en/products-services/software-networking/application-delivery-controllers.html%20(not%20accessible).

[26] *NEC Programmable Switch Series*. [Online]. Available: https://www.necam.com/sdn/ (visited on 12/05/2019).

[27] *Adva optical - fsp 150 & 3000*. [Online]. Available: https://www.adva.com/en/products/open-optical-transport (visited on 12/05/2019).

[28] "Ibm rackswitch g8264", [Online]. Available: http://www.redbooks.ibm.com/abstracts/tips0815.html%20(not%20accessible).

[29] *Hp openflow enabled switches*. [Online]. Available: https://www.hpe.com/uk/en/product-catalog/networking/networking-switches.hits-12.html (visited on 12/05/2019).

[30] *Juniper ju8nos mx, ex, qfx series*. [Online]. Available: https://www.juniper.net/documentation/en%7B%5C_%7DUS/junos/topics/concept/virtual-chassis-ex-qfx-series-mixed-understanding.html (visited on 12/05/2019).

[31] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: from concept to implementation", *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014, ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2326417.

[32] M. Paliwal, D. Shrimankar, and O. Tembhurne, "Controllers in sdn: a review report", *IEEE Access*, vol. 6, pp. 36 256–36 270, 2018, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2846236.

[33] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, p. 105, Jul. 2008, ISSN: 0146-4833. DOI: 10.1145/1384609.1384625.

[34] H. Yin, H. Xie, T. Tsou, P. Aranda, D. Lopez, and R.Sidi, *SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across*, Internet D. Internet Engineering Task Force, 2012, pp. 1–14. [Online]. Available: https://tools.ietf.org/pdf/draft-yin-sdn-sdni-00.pdf.

[35] D. Erickson, "The beacon openflow controller", in *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, New York, New York, USA: ACM Press, 2013, pp. 13–18, ISBN: 9781450320566. DOI: 10.1145/2491185.2491189.

[36] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths", *Journal of the ACM*, vol. 51, no. 6, pp. 968–992, Nov. 2004, ISSN: 00045411. DOI: 10.1145/1039488.1039492.

[37] *Open service gateway initiative (osgi)*. [Online]. Available: https://www.osgi.org/ (visited on 11/29/2019).

[38] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: defending sdns from malicious administrators", in *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*, New York, New York, USA: ACM Press, 2014, pp. 103–108, ISBN: 9781450329897. DOI: 10.1145/2620728.2620750.

[39] J. Networks, *What's Behind Network Downtime?*, May. Sunnyvale, CA, USA, 2008, pp. 1–12.

[40] *Floodlight*. [Online]. Available: http://www.projectfloodlight.org/floodlight/ (visited on 11/29/2019).

[41] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow",

[42] Z. Cai, A. L. Cox, and T. S. E. NG, "Maestro: a system for scalable openflow control", 2010.

[43] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang, "Meridian: an sdn platform for cloud network services", *IEEE Communications Magazine*, vol. 51, no. 2, pp. 120–127, Feb. 2013, ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6461196.

[44] *Mul*. [Online]. Available: https://sourceforge.net/p/mul/wiki/Home/ (visited on 11/29/2019).

[45] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks", in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, San Jose, CA: USENIX Association, Apr. 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/tootoonchian.

[46] T. Koponen, M. Casado, N. Gude, J. Stribling, L. B. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks", in *OSDI*, 2010.

[47] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "Onos: towards an open, distributed sdn os", in *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*, New York, New York, USA: ACM Press, 2014, pp. 1–6, ISBN: 9781450329897. DOI: 10.1145/2620728.2620744.

[48] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: towards a model-driven sdn controller architecture", in *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014, WoWMoM 2014*, IEEE, Jun. 2014, pp. 1–6, ISBN: 9781479947867. DOI: 10.1109/WoWMoM.2014.6918985.

[49] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: an api for application control of sdns", *Computer Communication Review*, vol. 43, no. 4, pp. 327–338, Aug. 2013, ISSN: 01464833. DOI: 10.1145/2534169.2486003.

[50] *Pox*. [Online]. Available: https://github.com/noxrepo/pox (visited on 11/29/2019).

[51] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: a robust, secure, and high-performance network operating system", in *Proceedings of the ACM Conference on Computer and Communications Security*, New York, New York, USA: ACM Press, 2014, pp. 78–89, ISBN: 9781450329576. DOI: 10.1145/2660267.2660353.

[52] *Ryu*. [Online]. Available: https://osrg.github.io/ryu/ (visited on 11/29/2019).

[53] F. Botelho, A. Bessani, F. M. V. Ramos, and P. Ferreira, "Smartlight: a practical fault-tolerant sdn controller", Jul. 2014. arXiv: 1407.6062.

[54] *Snac*. [Online]. Available: https://github.com/bigswitch/snac (visited on 11/29/2019).

[55] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in software defined networks", *Computer Networks*, vol. 71, pp. 1–30, Oct. 2014, ISSN: 13891286. DOI: 10.1016/j.comnet.2014.06.002.

[56] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks", *Computer Communication Review*, vol. 44, no. 2, pp. 87–98, Apr. 2014, ISSN: 19435819. DOI: 10.1145/2602204.2602219.

[57] M. Yu, M. Raju, and A. Wundsam, "Nosix: a lightweight portability layer for the sdn os", *Computer Communication Review*, vol. 44, no. 2, pp. 28–35, Apr. 2014, ISSN: 19435819. DOI: 10.1145/2602204.2602209.

[58] K. K. Yap, T. Y. Huang, B. Dodson, M. S. Lam, and N. McKeown, "Towards software-friendly networks", in *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems, APSys '10, Co-located with SIGCOMM 2010*, New York, New York, USA: ACM Press, 2010, pp. 49–53, ISBN: 9781450301954. DOI: 10.1145/1851276.1851288.

[59] *Trema*. [Online]. Available: https://github.com/trema/trema (visited on 11/29/2019).

[60] *Nox*. [Online]. Available: https://github.com/noxrepo/nox (visited on 11/29/2019).

[61] *Opendaylight*. [Online]. Available: https://www.opendaylight.org/ (visited on 11/29/2019).

[62] R. T. Fielding, "Chapter 5: representational state transfer (rest)", in *Architectural Styles and the Design of Network-Based Software Architectures*, Irvine, California, USA: University of California, 2000.

[63] T. Ward and H. Cummins, *Enterprise OSGi in Action*, 1st editio. Greenwich, CT, USA: Manning Publications Co., 2013, p. 376, ISBN: 9781617290138.

[64] *Beacon*. [Online]. Available: https://openflow.stanford.edu/display/Beacon/Home.html (visited on 11/29/2019).

[65] *Onos*. [Online]. Available: https://onosproject.org/ (visited on 11/29/2019).

[66] A. Zúquete, *Segurança em Redes Informáticas*, 4ª Edição, L. FCA - Editora de Informática, Ed. 2013, p. 432, ISBN: 978-972-722-767-9.

[67] MITRE, *Capec-141: cache poisoning*, 2018. [Online]. Available: https://capec.mitre.org/data/definitions/141.html (visited on 12/11/2019).

[68] *Capec-169: footprinting*. [Online]. Available: ttps://capec.mitre.org/data/definitions/169.html (visited on 12/11/2019).

[69] *Capec-94: man in the middle*. [Online]. Available: https://capec.mitre.org/data/definitions/94.html (visited on 12/11/2019).

[70] MITRE Corporation, *Capec-125: flooding*, 2018. [Online]. Available: https://capec.mitre.org/data/definitions/125.html (visited on 12/11/2019).

[71] *Capec-130: excessive allocation*. [Online]. Available: https://capec.mitre.org/data/definitions/130.html (visited on 12/11/2019).

[72] *Capec-131: resource leak exposure*. [Online]. Available: https://capec.mitre.org/data/definitions/131.html (visited on 12/11/2019).

[73] *Capec-227: sustained client engagement*. [Online]. Available: https://capec.mitre.org/data/definitions/227.html (visited on 12/11/2019).

[74] S. Axelsson, "Intrusion detection systems: a survey and taxonomy", Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, Tech. Rep., Mar. 2000.

[75] H. J. Liao, C. H. Richard Lin, Y. C. Lin, and K. Y. Tung, "Intrusion detection system: a comprehensive review", *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, Jan. 2013, ISSN: 10848045. DOI: 10.1016/j.jnca.2012.09.004.

[76] *Snort*. [Online]. Available: https://www.snort.org/ (visited on 12/01/2019).

[77] *Libpcap*. [Online]. Available: https://www.tcpdump.org/ (visited on 12/10/2019).

[78] G. Khalil, *OPEN Source IDS High Performance Shootout [White Paper]*. 2015.

[79] *Suricata*. [Online]. Available: https://suricata-ids.org/ (visited on 12/01/2019).

[80] *Bro*. [Online]. Available: https://www.zeek.org/ (visited on 12/01/2019).

[81] R. Sommer, "Bro: an open source network intrusion detection system.", *DFN-Arbeitstagung über Kommunikationsnetze*, pp. 273–288, 2003.

[82] D. Day and B. Burns, "A performance analysis of snort and suricata network intrusion detection and prevention engines", in *ICDS 2011, The Fifth International Conference on Digital Society*, c, Gosier, Guadeloupe, France, 2011, pp. 187–192, ISBN: 978-1-61208-116-8. [Online]. Available: http://www.thinkmind.org/index.php?view=article%7B%5C&%7Darticleid=icds%7B%5C_%7D2011%7B%5C_%7D7%7B%5C_%7D40%7B%5C_%7D90007.

[83] E. Albin and N. C. Rowe, "A realistic experimental comparison of the suricata and snort intrusion-detection systems", in *Proceedings - 26th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2012*, IEEE, Mar. 2012, pp. 122–127, ISBN: 9780769546520. DOI: 10.1109/WAINA.2012.29.

[84] A. Alhomoud, R. Munir, J. P. Disso, I. Awan, and A. Al-Dhelaan, "Performance evaluation study of intrusion detection systems", *Procedia Computer Science*, vol. 5, pp. 173–180, 2011, ISSN: 18770509. DOI: 10.1016/j.procs.2011.07.024.

[85] C. J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang, "Nice: network intrusion detection and countermeasure selection in virtual network systems", *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 4, pp. 198–211, Jul. 2013, ISSN: 15455971. DOI: 10.1109/TDSC.2013.8.

[86] T. Xing, Z. Xiong, D. Huang, and D. Medhi, "Sdnips: enabling software-defined networking based intrusion prevention system in clouds", in *Proceedings of the 10th International Conference on Network and Service Management, CNSM 2014*, IEEE, Nov. 2014, pp. 308–311, ISBN: 9783901882661. DOI: 10.1109/CNSM.2014.7014181.

[87] P. W. Chi, C. T. Kuo, H. M. Ruan, S. J. Chen, and C. L. Lei, "An ami threat detection mechanism based on sdn networks", *SECURWARE 2014 - 8th International Conference on Emerging Security Information, Systems and Technologies*, pp. 208–211, 2014.

[88] D. Hyun, J. Kim, D. Hong, and J. Jeong, "Sdn-based network security functions for effective ddos attack mitigation", in *International Conference on Information and Communication Technology Convergence: ICT Convergence Technologies Leading the Fourth Industrial Revolution, ICTC 2017*, vol. 2017-December, IEEE, Oct. 2017, pp. 834–839, ISBN: 9781509040315. DOI: 10.1109/ICTC.2017.8190794.

[89] W. Fan, Z. Du, M. Smith-Creasey, and D. Fernandez, "Honeydoc: An Efficient Honeypot Architecture Enabling All-Round Design", *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 683–697, Mar. 2019, ISSN: 0733-8716. DOI: 10.1109/JSAC.2019.2894307.

[90] W. Fan, Z. Du, and D. Fernandez, "Taxonomy of honeynet solutions", in *IntelliSys 2015 - Proceedings of 2015 SAI Intelligent Systems Conference*, IEEE, Nov. 2015, pp. 1002–1009, ISBN: 9781467376068. DOI: 10.1109/IntelliSys.2015.7361266.

[91] D. V. Silva and G. D. Rodríguez Rafael, "A review of the current state of honeynet architectures and tools", *International Journal of Security and Networks*, vol. 12, no. 4, pp. 255–272, 2017, ISSN: 17478413. DOI: 10.1504/IJSN.2017.088133.

[92] *Kippo - ssh honeypot*, 2009. [Online]. Available: https://github.com/desaster/kippo (visited on 12/08/2019).

[93] *Kojoney2 ssh honeypot*. [Online]. Available: https://github.com/madirish/kojoney2 (visited on 12/08/2019).

[94] W. Han, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeymix", in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization - SDN-NFV Security '16*, New York, New York, USA: ACM Press, 2016, pp. 1–6, ISBN: 9781450340786. DOI: 10.1145/2876019.2876022.

[95] Z. LIAN, X.-c. YIN, R. TAN, and Y.-x. CHEN, "SDN Virtual Honeynet for Network Attack Information Acquisition", *DEStech Transactions on Computer Science and Engineering*, no. smce, Jul. 2017, ISSN: 2475-8841. DOI: 10.12783/dtcse/smce2017/12435.

[96] M. Carvalho and R. Ford, "Moving-target defenses for computer networks", *IEEE Security and Privacy*, vol. 12, no. 2, pp. 73–76, Mar. 2014, ISSN: 15407993. DOI: 10.1109/MSP.2014.30.

[97] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense", in *Proceedings of the ACM Conference on Computer and Communications Security*, vol. 2014-November, New York, New York, USA: ACM Press, 2014, pp. 31–40, ISBN: 9781450331500. DOI: 10.1145/2663474.2663479.

[98] *Mininet*. [Online]. Available: http://mininet.org/ (visited on 12/09/2019).

[99] W. Fan and D. Fernandez, "A novel sdn based stealthy tcp connection handover mechanism for hybrid honeypot systems", in *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: En Route to 5G, NetSoft 2017*, IEEE, Jul. 2017, pp. 1–9, ISBN: 9781509060085. DOI: 10.1109/NETSOFT.2017.8004194.

[100] W. Fan, D. Fernández, and Z. Du, "Versatile virtual honeynet management framework", *IET Information Security*, vol. 11, no. 1, pp. 38–45, Jan. 2017, ISSN: 17518717. DOI: 10.1049/iet-ifs.2015.0256.