

Department of Information Science and Technology

**A Novel SDN-Based Intrusion Detection System for Early
Detection and Mitigation of DDoS Attacks**

Pedro Caetano Manso

Dissertation submitted as partial requirement to obtain the degree of
Master in Telecommunications and Computer Engineering

Supervisor:
Doctor José André Moura, Professor of Digital Networks II and III
ISCTE-IUL

Co-supervisor:
Doctor Carlos Serrão, Assistant Professor of Networks and Information Systems
Security
ISCTE-IUL

October, 2018

Agradecimentos

À minha família e amigos pelo apoio incondicional.

Aos professores José Moura e Carlos Serrão por toda a disponibilidade, motivação e conhecimento partilhado.

A todos um sincero obrigado.

Resumo

A Internet teve o seu “boom” no meio dos anos 90. A sociedade mergulhou numa realidade “dot.com” com muitos serviços e plataformas disruptivas por todo o mundo. Atualmente, depois de quase 25 anos, a Internet provou que é uma tecnologia indispensável. Com este uso mundial da Internet, o esforço colocado no processo de gestão de redes aumentou. Para facilitar este processo, uma nova ideia surgiu, chamada de *Software Defined Networks* (SDN). SDN é um paradigma emergente que começou a ser amplamente usado em várias áreas, como por exemplo a *Cloud Computing*. Para além disso, com o crescimento da Internet, a Segurança da Informação está a começar a ser uma área alvo de interesse para investigadores em áreas como Direito e Engenharia Informática. Um tópico de Segurança muito relevante é aquele que incorpora os ataques do tipo *Distributed Denial-of-Service* (DDoS).

Nesta tese, desenvolvemos e testamos uma solução completa que deteta e mitiga ataques DDoS juntando o conceito SDN com uma ferramenta *Intrusion Detection System* (IDS). A nossa arquitetura inclui um OpenFlow *switch*, um IDS e um controlador SDN. O switch copia todo o tráfego da rede para o IDS, via *port mirroring*. O IDS deteta se um ataque está a acontecer. O processo de deteção é um mecanismo baseado em regras. Uma vez que o ataque é detetado pelo IDS, este ataque é comunicado para o controlador SDN, com a finalidade de se dar início ao nosso processo de mitigação desse ataque. Este processo de mitigação consiste em atualizar a tabela existente no *switch* com regras de *forwarding* de tráfego que bloqueiam somente o tráfego malévolo. Desta forma, obtivemos um *testbed* funcional que combina ambas as valências do SDN e do IDS, e evidenciamos que este *testbed* consegue mitigar com sucesso vários cenários de ataque DDoS, minimizando a perturbação sofrida pelo tráfego normal.

Palavras-Chave: SDN; *Ryu controller*; DDoS attacks; Mininet; IDS; Snort; Botnet; OpenFlow.

Abstract

Internet has its boom in the mid of the 90s. Society has dived in a “dot com” reality with many world-wide disruptive services and platforms. Back to these days, after around 25 years, the Internet has proven that is an indispensable technology. With this world-wide use of the Internet, the effort placed in network manage process has increased. In order to facilitate that process, a new idea as come around called *Software Defined Networks* (SDN). SDN is an emergent paradigm that has started to be widely used in many areas such as Cloud computing. Also, with the growth of the Internet, Information Security has become a field of interest for researchers in many areas such as Law and Computer Science. One particular topic of Security is the *Distributed Denial-of-Service* (DDoS) attacks.

In this thesis, we developed and tested a fully solution that detects and mitigates DDoS attacks compounding SDN with an *Intrusion Detection System* (IDS). Our architecture includes an OpenFlow switch, an IDS and the SDN controller. The switch mirrors the traffic to the IDS, through port mirroring configurations. The IDS detect if an attack is happening. The detection process is a rule-based mechanism. Once an attack is detected, it communicates to the SDN controller, in order to start the mitigation procedure. The mitigation procedure consists on updating the switch flow rule table, with a blocking rule. This way, we got a functional testbed where of DDoS attacks scenarios were successfully mitigated.

Keywords: SDN; Ryu controller; DDoS attacks; Mininet; IDS; Snort; Botnet; OpenFlow.

Index

Agradecimientos	i
Resumo.....	ii
Abstract.....	iii
Index	iv
Tables Index.....	vi
Figures Index	vii
List of Abbreviations and Acronyms	ix
Chapter 1 – Introduction.....	1
1.1. Contextualization	1
1.2. Questions and Investigation Goals.....	3
1.3. Methodology Approach.....	4
1.4. Dissertation Structure and Organization	5
Chapter 2 – Literature Review	6
2.1. Software Defined Networks	6
2.1.1. SDN Controllers.....	8
2.1.2. SDN/Southbound APIs.....	10
2.1.3. SDN/Northbound APIs.....	14
2.2. Distributed Denial-of-Service Mitigated with SDN	18
2.2.1. Signature-Based	22
2.2.2. Anomaly-Based.....	25
2.2.3. Other Related Techniques.....	27
Chapter 3 – System Design and Deployment.....	28
3.1. System Design	28
3.2. System Deployment	29
3.3. Deployed System Components.....	30
3.3.1. Virtual Machines.....	30
3.3.2. SDN Controller	34
3.3.2.1. Controller Performance - Enhanced Version 1	35
3.3.2.2. Controller Performance - Enhanced Version 2	36
3.3.3. Snort	36
3.3.4. Mininet	37
3.3.4.1. NAT	39
3.3.4.2. OpenFlow Switch	40
3.4. Our Proposal Workflow	41

Chapter 4 – Evaluation Scenario, Performed Tests and Results Discussion.....	43
4.1. Tests description	45
4.1.1. Description of Scenario I Test	45
4.1.2. Description of Scenario II Test.....	46
4.1.3. Description of Scenario III Test.....	47
4.2. Tests Results Presentation	49
4.2.1. Scenario I - Results Presentation.....	52
a. System Performance - Enhanced Version 1	59
b. System Performance - Enhanced Version 2	61
4.2.2. Scenario II - Results Presentation	62
4.2.3. Scenario III - Results Presentation	67
4.2.4. Scenario I - Discussion of Results.....	72
4.2.5. Scenario II - Discussion of Results	73
4.2.6. Scenario III - Discussion of Results	74
Chapter 5 – Conclusions and Future Work.....	75
5.1. Main Conclusions	75
5.2. Future Work.....	76
Bibliography	77
Appendixes.....	83
Appendix A – <i>packet_in_handler</i> Code	83
Appendix B – <i>add_flow</i> Code	83
Appendix C – <i>process_snort_alert</i> Code.....	84
Appendix D – <i>send_blocking_flowrule</i> Code	84

Tables Index

Table 1 – Existing Controllers Part 1 of 2	8
Table 2 – Existing Controllers Part 2 of 2	9
Table 3 – SDN/Southbound APIs	10
Table 4 – Northbound APIs Part 1 of 2	14
Table 5 – Northbound APIs Part 2 of 2	15
Table 6 – Literature SDN DDoS Mitigation Solutions Part 1 of 4	18
Table 7 – Literature SDN DDoS Mitigation Solutions Part 2 of 4	19
Table 8 – Literature SDN DDoS Mitigation Solutions Part 3 of 4	20
Table 9 – Literature SDN DDoS Mitigation Solutions Part 4 of 4	21
Table 10 – Hping3 Tool Arguments Used in our Test Scenarios	44
Table 11 – Summary of Tests Descriptions Part 1 of 2	48
Table 12 – Summary of Tests Descriptions Part 2 of 2	49
Table 13 – Normal Usage Latency	51
Table 14 – Scenario I Results	59
Table 15 – Scenario I Performance Enhanced Version 1 Results	60
Table 16 – Scenario I Performance Enhanced Version 2 Results	62
Table 17 – Scenario II Performance Results Part 1 of 2	66
Table 18 – Scenario II Performance Results Part 2 of 2	67
Table 19 – Scenario III Results	72
Table 20 – Scenario I Summary Results	73
Table 21 – Scenario II Summary Results	73
Table 22 – Scenario III Summary Results	74
Table 23 – Scenarios Results Comparation Summary	76

Figures Index

Figure 1 – Design Research Adopted (Peffer et al., 2007)	5
Figure 2 – System's Basic Architecture	28
Figure 3 – System Deployment.....	30
Figure 4 – VM A Network Adapter 1	31
Figure 5 – VM A Network Adapter 2	31
Figure 6 – VM A Network Interfaces.....	32
Figure 7 – VM B Network Adapter 1.....	32
Figure 8 – VM B Network Adapter 2.....	33
Figure 9 – VM B Network Interfaces.....	33
Figure 10 – VM C Network Adapter 1.....	34
Figure 11 – VM C Network Interfaces.....	34
Figure 12 – Launching Ryu Application	35
Figure 13 – Snort Alert Handler Function Performance Enhanced Version 1	36
Figure 14 – Snort Alert Handler Function Performance Enhanced Version 2	36
Figure 15 – Snort Launch Command	37
Figure 16 – Python Script for Network Creation	38
Figure 17 – Dump Command Output.....	38
Figure 18 – NAT General Overview	39
Figure 19 – NAT Routing Table	39
Figure 20 – Port Mirroring Concept.....	40
Figure 21 – System Workflow for Each Packet.....	41
Figure 22 – System Workflow Exchanged Messages Diagram.....	42
Figure 23 – Home Network Topology Simulated.....	44
Figure 24 – Scenario I Test Diagram	46
Figure 25 – Scenario II Test Diagram	47
Figure 26 – Scenario III Test Diagram.....	48
Figure 27 – OpenFlow Switch Dump Flows Before Any Packets Exchange	49
Figure 28 – OpenFlow Switch Dump Flows After Pingall Mininet's Command Execution	50
Figure 29 – Wireshark Usual System Usage	51
Figure 30 – Online Service Under DDoS Attack Top Command Execution	52
Figure 31 – Wireshark I/O Graph of DDoS Attack Without Our Defense System	52
Figure 32 – Host 1 DDoS Attack Engagement	53
Figure 33 – Host 3 DDoS Attack Engagement	53
Figure 34 – Host 2 Ping Command Output While Server is Under DDoS Attack	54
Figure 35 – Host 2 Wireshark I/O graph example While Server is Under DDoS Attack	54
Figure 36 – DDoS attack Host 1 Detection Controller's Output	55
Figure 37 – DDoS Attack Host 2 Detection Controller's Output	55
Figure 38 – OpenFlow Switch's Flow Table with Mitigation Rules	56
Figure 39 – Host 2 Successful Ping After DDoS Attack.....	57
Figure 40 – Host 1 Unsuccessful Ping After DDoS Attack Mitigation	57
Figure 41 – Host 3 Unsuccessful Ping After DDoS Attack Mitigation	57
Figure 42 – DDoS Attack and Mitigation.....	58
Figure 43 – Server top Command after DDoS attack Mitigation	58
Figure 44 – Graphic of DDoS Attack Mitigation with System Version 1 Performance Enhanced.....	60
Figure 45 – Snort Extra Configuration	61

Figure 46 – DDoS Attack and Mitigation Performance Enhanced Version 2	61
Figure 47 – Host 1 DDoS Engagement	63
Figure 48 – Host 2 Ping Output while DDoS Attack	63
Figure 49 – Host 3 Ping Output while DDoS Attack	64
Figure 50 – Host 2 and 3 Wireshark I/O Graph while DDoS Attack	64
Figure 51 – DDoS Attack Host 1 Detection Controller's Output	65
Figure 52 – OpenFlow Switch's Flow Table with Mitigation Rules	65
Figure 53 – Wireshark I/O Graph of DDoS Mitigation	66
Figure 54 – Host 1 DDoS Engagement	67
Figure 55 – Host 2 Ping Command Output while the Server is under DDoS Attack	68
Figure 56 – Host 3 Ping Output while Server is under the DDoS Attack	68
Figure 57 – Hosts 2 and 3 Wireshark I/O Graph while Server is under the DDoS Attack	69
Figure 58 – Controller's Output	69
Figure 59 – OpenFlow Switch's Flow Table with Mitigation Rules	70
Figure 60 – Wireshark I/O Graph of DDoS Attack Mitigation	71
Figure 61 – Server tcpdump Command Output	71

List of Abbreviations and Acronyms

API – Application Programming Interface

CLI – Command Line Interface

CONA – Content-Oriented Networking Architecture

DDoS – Distributed Denial-of-Service

FL-GUARD – Floodlight-based Guard

FML – Flow-based Management Language

ForCES – Forward and Control Elements Separation

FRP – Functional Reactive Programming

HTTP – HyperText Transfer Protocol

ICMP – Internet Control Message Protocol

IDS – Intrusion Detection System

IP – Internet Protocol

IoT – Internet of Things

I2RS – Interface Routing System

JSON – JavaScript Object Notation

LAN – Local Area Network

LSP – Label Switch Path

MPLS-TP – Multiprotocol Label Switching – Transport Profile

NMS – Network Management System

ONOS – Open Source Network Operating System

OSGi – Open Services Gateway Initiative

OVSDB – Open vSwitch Database

PCC – Path Computation Client

PCE – Path Computation Element

PCEP – Path Computation Element Protocol

PCNtf – Patch Computation Notification

PCReq – Path Computation Request

PCRep – Path Computation Reply

REST – Representation State Transfer

RPC – Remote Call Procedure

RTT – Round-Trip Time

SASL – Simple and Security Layer

SDN – Software-Defined Networks

TCP – Transmission Communication Protocol

TLS – Transport Layer Security

UDP – User Datagram Protocol

URI – Uniform Resource Identifier

VM – Virtual Machine

XML – Extensible Markup Language

XMPP – Extensible Messaging and Presence Protocol

Chapter 1 – Introduction

1.1. Contextualization

According to (Internet World Stats, 2017) in the beginning of 2018, more than 50% of the world population uses the Internet. In particular, 85% of the Europeans and 95% of North Americans use the Internet. The number of devices connected to the Internet has skyrocketed over the expectations during the last decade (Nordrum, 2016). (Cisco, 2016) forecasts that, in 2021, there will be 27.1 billion network devices and 3.5 networked devices per capita. This exponential growth represents not only the progress of technology, but also the opportunity for the eventual compromising of a larger amount of network devices.

In 2016, the world witnessed one of the world's largest DDoS attack ever seen (Newman, 2016). This attack consisted on a malware infection that, once in use, gives access and control of network devices running Linux. Therefore, as this infection scales over a considerable number of devices, orchestrated by an attacker, we have a *botnet*. Then, the compromised devices generate high volume of traffic, through the botnet, in direction to an online service. If the attack has success, the online service will find its resources exhausted. Concluding, the online service might come down with its service denied for its normal clients. This scenario is called as a *DDoS* attack. The landscape of *Distributed Denial-of-Service* (DDoS) attacks points also to more complex and frequent attacks. Putting also the increase of DDoS attacks into numbers, a report on (NETSCOUT, 2018) shows that:

- “Fifty-seven percent of enterprises and forty-five percent of data centers operators saw their Internet backhaul links saturated due to DDoS attacks”;
- “There were 7.5 million DDoS attacks in 2017”;
- “Fifty-nine percent of service providers and forty-eight percent of enterprises experienced multi-attacks: attacks that combine high volume floods, application-layer attacks and TCP-state exhaustion attacks”.

The *Mirai botnet* had made its fame when it successfully bring down a key part of the Internet's backbone, Dyn (Newman, 2016). This is an Internet infrastructure company

that provides *Domain Name System* (DNS) services. In fact, the Internet DNS service was severely affected after hundred thousand compromised devices sent DNS query messages at very high rates to specific DNS servers. These servers were down during a significant amount of time. As if it was not enough, on January 3rd of 2018, the biggest DDoS attack ever recorded happened. It reached to 1.35 Tbs of traffic against the GitHub platform (Newman, 2018). Hence, the security community is concerned about the next big dimension of these attacks and their consequences. Researchers believe that the next attack traffic volume can reach up to dozens of terabits-per-second. As (Arghire, 2018) presents, *Wicked* a variant of Mirai botnet already emerged. We can expect more variants of Mirai to appear, in the future, that will be more sophisticated and pose increasing security risks to the Internet. Therefore, the fight against DDoS attacks existed, exists and will be existing in a far future from now.

Nowadays, according to (White & Zandi, 2016), networks are designed with the following concerns in mind: isolating up faulty domains, managing complex policies, choosing topologies based on fast convergence characteristics and the interaction of multiple distributed control planes running on a single network. Hence, a new emergent network technology appeared, called *Software Defined Networks* (SDN). SDN is a network technology that allow us to: program the network's operation, logically manage the network in a centralized way and separate the control plane from the data plane. As (Ieee et al., 2015) mentions, companies as: Google, Facebook, Yahoo, Microsoft, Verizon and Deutsche Telekom had founded the *Open Network Foundation* (ONF) in order to adopt and promote SDN will be existing in a far future from now.

As (Lim, Ha, Kim, Kim, & Yang, 2014) does reference, owing to the trouble in facing a botnet based attack inside the network, DDoS mitigation techniques tend to be implemented in the server side. In this thesis, in order to mitigate DDoS attacks it is discussed, implemented and tested a novel solution empowered by the SDN paradigm aggregated to an IDS, which tries to mitigate the DDoS attacks at their origin, i.e. at the local network of the potential compromised clients. Thus, our proposal consists on a DDoS attack mitigation strategy and system, to be implemented at the client side, that seeks for: scalability, differentiation between malign from benign traffic, reduced overhead over the network and mitigation time.

1.2. Questions and Investigation Goals

The main research question of the current work is the following:

“Given the network security vulnerabilities introduced by network devices and the urgent need for mitigating the negative effects of DDoS attacks which are exploring, mostly, the IoT security weaknesses, can a security system based on the SDN paradigm be designed and implemented, at the client side, to ensure in a centralized way the “normal operation” of the home network infrastructure potentially affected by the aforementioned threats?”

To answer the previous research question, we would like to study the emerging paradigm of Software Defined Network (SDN), in order to supervise all the traffic exchanged within a LAN environment. This supervision, empowered by the additional collaboration of an IDS, is required to detect and block malicious traffic. As the attack sources could be either external or internal nodes, we envision as a more suitable design for the necessary security solution a centralized logical control. This design is very well aligned with a SDN controller operating within a LAN infrastructure. In this way, the probability of some internal LAN nodes become compromised, through the local device installation of malicious code, is considerably diminished, mitigating future DDoS attacks, against not only that LAN but also other external networks.

The following goals will be achieved during the timeframe of this work:

- Incorporate an IDS on the SDN architecture;
- Design and implement a SDN-based framework that detects and mitigates DDoS attacks, at their origins;
- Set up a testbed;
- Evaluate the overhead that our proposal adds to the network, through the comparison of the network metrics average RTT and percentage of packet loss, considering our solution enabled or disabled;
- Evaluate the effectiveness of our system, through the time that our proposal spends to detect and mitigate a DDoS attack.

The following tasks were conducted in order to address and attain the goals above:

- Design and emulate an end-user network with Mininet;
- Make the configurations needed to incorporate an IDS in the SDN architecture;
- Adopt DDoS rules to define suspicious traffic;

- Process alerts with the controller, launched by an IDS;
- Define flow rule updates in order to mitigate the attack;
- Measurement of the next metrics: average RTT, percentage of packet loss and, the time to detect and mitigate a DDoS attack;
- Perform functional and performance tests.

1.3. Methodology Approach

The Design Research Methodology process model used to conduct the research activities in this work was adopt from (Peffer, Tuunanen, Rothenberger, & Chatterjee, 2007). It is composed by six steps as shown in Figure 1:

1. Problem Identification
2. Define Objectives for a solution
3. Design and Development
4. Demonstration
5. Evaluation
6. Communication

In this thesis, on the first approach, from the step 1 to step 4 was practically sequential. The Problem Identification and the definition of objectives for a solution are defined in the section 1.2. Step 3, the system design, development and evaluation are in the subsection 4.2.1. Seeking for improvements the iteration process started. On total, in our proposal, we did two iterations. After getting the scenario I results and evaluate them (subsection 4.2.1), the first iteration started by taking a step back to the research entry point Design & Development. We changed the SDN controller code, in order to enhance the performance of our proposal (subsection 3.3.2.1) and made its evaluation. Then, in the iteration number two, we perform code changes (subsection 3.3.2.2) and add extra configurations to our IDS (Figure 45). Finally, we evaluated this last version and proceed to the communication phase that consisted on writing this proposal.

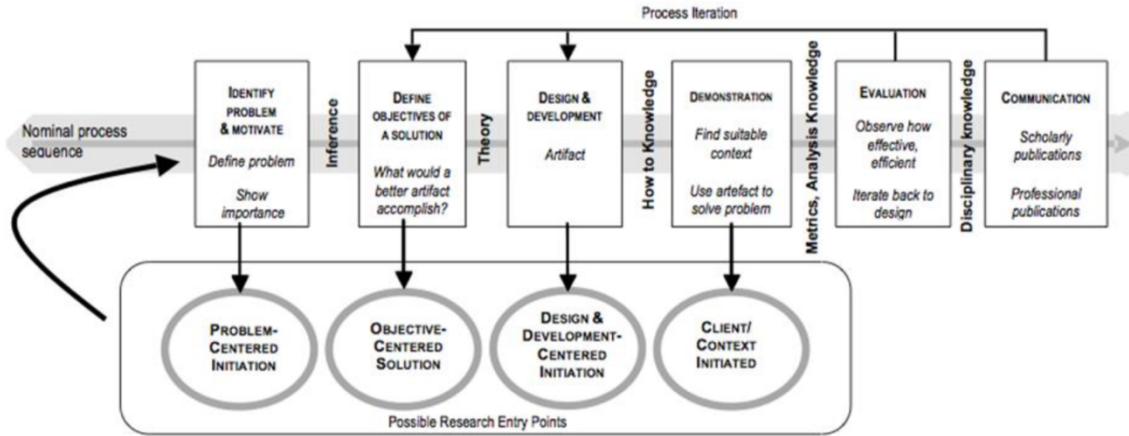


Figure 1 – Design Research Adopted (Peppers et al., 2007)

1.4. Dissertation Structure and Organization

This thesis is organized in five chapters. The first chapter introduces the motivation and the research theme of this thesis, lays out the different goals and presents the structure of this work.

The second chapter reflects the literature review, providing an overview of the most recent and relevant topics in the context of this thesis.

The following chapter is devoted to the design, specification and deployment of the proposal that will address the research question that was previously identified.

The fourth chapter describes how the proposal has been evaluated. It also discusses in a comprehensive way the diverse evaluation results.

Finally, the last chapter of this work presents the conclusions and provides some directions for future related research work.

Chapter 2 – Literature Review

In this chapter, the solutions presented in the literature are discussed. The section 2.1 is dedicated to the SDN technology. In subsection 2.1.1 is presented a list with the existing SDN controllers. Subsection 2.1.2 presents a list with the SDN/Southbound APIs and its discussion. Subsection 2.1.3 presents the Northbound APIs existent in literature and its discussion. The subsection 2.2 is dedicated to DDoS detection and mitigation techniques compounded with SDN.

2.1. Software Defined Networks

Software Defined Networks (SDN) is a new network model that has been emerging in the last years. Although, the idea beneath SDN appeared initially in the 1990s as presented in (Sezer et al., 2013) and (Scott-Hayward, S.a , Natarajan, S.b , Sezer, 2016). The main features of this emerging technology are as aforementioned: the separation of the control plane from the data plane, a centralized logical control, an abstracted view of the network and the programmability of the network by external applications.

The separation of the control plane from the data plane enables the network configuration to be elsewhere in the network rather than within the usual network elements forming the data path, i.e. switches and routers. Those are just responsible for forwarding functions. The control plane is responsible for the network configuration and the management of its main functionalities, e.g. drop traffic from a specify source. The data plane is where the traffic exchanged is “consumable” and processed by network elements. It works with the control plane in order to know how to forward packets across the network topology.

The centralized logical control and abstracted view of the network resides in one place, i.e. the controller. This centralization means that the controller supervises all the network characteristics. For instance, the controller discovers the topology of the entire network. The controller is the responsible for the aforementioned control plane functions. In design terms, it does the bridge between the data plane and applications to the network. Such architecture provides the freedom to develop these applications, with the abstraction of what is in the data plane, in one single network element, i.e. the controller. In contrast, in the legacy ways of networking, in order to make changes to the network operation we had to change the configuration of every single network element. For instance, supposing

we wanted to mirror the traffic in all routers. In that case, we had to go configure every single router for its operation. With SDN, we are able to do it in one only network element, the controller. This feature is also emphasized in (Alsmadi & Xu, 2015). Another peculiarity with this architecture is that there can be four programming interfaces: The Southbound API, the Northbound API, the West API and the East API. Nevertheless, in this thesis, we will be only focused on the Southbound API and Northbound API:

1. *Southbound API*: this programming interface is located between the data plane and the control plane. Its main goal is to establish the connection between the controller and the data plane, i.e. the switches. The most common protocol used here is *OpenFlow*.
2. *Northbound API*: this programming interface is located between the control plane and the applications above the control plane. Its main purpose is to make the communication between the business logic and the controller. The most common protocol used here is *REST*.

The programmability of the network by external applications is a key feature of SDN as presented in (Lim et al., 2014). Transcribing (White & Zandi, 2016): “If there’s one point virtually every network engineer agrees on, it’s that large-scale networks are difficult to provision, monitor and troubleshoot”. In traditional networks, routers are very painful to modify their behavior. They run high-level algorithms in order to define routing and control rules (e.g. *Border Gateway Protocol*). In addition, they are distributed through the network. With SDN, the opposite happens. Due to the centralization of the control plane, the abstracted view of the network and the possibility to program through the diverse APIs, we can code how the network devices will perform. Hence, it enables us to implement business policies in an agile way (e.g. load balancing, distributed firewall). OpenFlow is the pioneer in the protocol between the data plane and control plane (You, Feng, & Sakurai, 2018). So far, it has been proposed six versions of OpenFlow (from version 1.0 to 1.5). In this thesis, OpenFlow v1.3 has been used.

2.1.1. SDN Controllers

In this section, we present the main existent SDN controllers (Table 1, 2, 3 and 4), with two characteristics in mind: the controller architecture and its implementation language. The controller architecture can be Centralized or Distributed. On one hand, a distributed architecture it is able to support the requirements of potentially any environment dimension, from small to large networks. It can consist on a centralized cluster of nodes of a physically distributed set of elements. On the other hand, a centralized architecture consists on a non-scalable system where it supports only small networks dimensions. Thus, it is usually composed by a single node or a physically single controller. The implementation language represents the language that the controller was developed (e.g. Python, Java, C.).

Table 1 – Existing Controllers Part 1 of 2

Name	Architecture	Implementation
POX (McCauley, 2011)	Centralized	Python
NOX (Gude et al., 2008)	Centralized	Python, C++
Maestro (Cai, Cox, & T. S. Ng, 2011)	Centralized	Java
Trema (Trema, 2015)	Centralized	C, Ruby
Beacon (Erickson, 2013)	Centralized	Java
Jaxon (Ohyama, 2012)	Centralized	Java
Helios (Nec, 2012)	Centralized	C
Floodlight (Floodlight, 2014)	Centralized	Java

Table 2 – Existing Controllers Part 2 of 2

Name	Architecture	Implementation
SNAC (SNAC, 2012)	Centralized	C++
Ryu (RYU project team, 2013)	Centralized	Python
Ovs-Controller (Open vSwitch, 2009)	Centralized	C
DISCO (Phemius, Bouet, & Leguay, 2014)	Distributed	Java
Fleet (Matsumoto, Hitz, & Perrig, 2014)	Distributed	Python
HP VAN SDN (HP Enterprise, 2013)	Distributed	Java
HyperFlow (Tootoonchian & Ganjali, 2010)	Distributed	C++
Onix (Koponen et al., 2010)	Distributed	Python, C
OpenContrail (Gorbunov & Architect, 2013)	Distributed	Python, C++, Java
ONOS (Berde et al., 2014)	Distributed	Java
SMarRLight (Botelho, Bessani, Ramos, & Ferreira, 2014)	Distributed	Java

2.1.2. SDN/Southbound APIs

There are many Southbound APIs currently available. In Table 3 is presented the solutions mentioned in this section.

Table 3 – SDN/Southbound APIs

SDN/Southbound APIs	Description
OpenFlow (McKeown et al., 2008)	Protocol
ForCES (A. Doria et al., 2010)	Protocol
PCEP (Vasseur & Le Roux, 2009)	Protocol
Netconf (Bierman & Bjorklund, 2012)	Protocol
OVSDB (Pfaff & Davie, 2013)	Protocol
XMPP (Saint-Andre, 2011)	Protocol
MPLS-TP (IETF RFC 6373, 2011)	Protocol

OpenFlow (McKeown et al., 2008) is a very popular protocol being used in SDN architecture. This protocol, provides the communication between the controller and the network devices. It explores the fact that the network devices have flow-tables. The network devices which “speak” OpenFlow protocol are called OpenFlow switches. These switches, are normally composed by 3 features: *Flow Tables*, a *Secure Channel* and the client-side of the *OpenFlow Protocol*. The *Flow Tables* store OpenFlow rules associated

to *flows*. A *Flow Rule* is composed by a *match*, an *action* and *statistics*. A *match* is expressed with specific values on fields of the packet headers that identifies a specific flow. An *action*, is a specific control that the switch performs over each packet of the well-identified flow. The *Statistics* collect some useful information from the data plane, e.g. number of packets and bytes for each flow processed by a specific switch. For instance, in a scenario where all packets from a specific source IP are commuted to the output port N of a switch. The source IP is the match. The internal commutation process to the output port N process is the action. The statistics could be the number of packets that source IP had sent in an interval of time though that switch. The *Secure Channel* is responsible for supporting the connection between the switch and the controller. In general, an OpenFlow Switch does four simple actions:

1. Forward – Giving the arrival of a packet to a port of the switch, the switch will check its Flow Table. If there is any match for that packet, it matches and the rule previously defined is applied, e.g. “forward” the received packet from the switch port that has received the packet to the port that it should send the received packet to the next network device.
2. Drop – Giving the arrival of a packet to a port of the switch, the switch will check its Flow Table. If there is any match for that packet with a defined rule to “drop”, it will drop the packet.
3. Forward to Secure Channel – Giving the arrival of a packet to a port of the switch, the switch will check its Flow Table. If there is any action for that packet to forward it to a specific output port, the switch should have a default action to forward the received packet through the Secure Channel to be processed by the SDN controller. After this processing, the SDN controller sends a specific OpenFlow rule to the switch in order to all the packets of the same flow could be locally controlled by the switch.
4. Set - Giving the arrival of a packet to a port of the switch, the switch will update a field oh the packet header. Then, commutes the packet for the specified port. For instance, this can be used for updating the existing TTL hop limit and the IP checksum.

The *OpenFlow Protocol* is the basis of all the behaviors that each switch has. (Heller, 2009) discusses the OpenFlow 1.0 specification. To date, it has been already deployed

the version 1.1, 1.2 and 1.3. The process of discover about the network's topology is based in *packet_in* messages. When a switch does not know how to handle a packet, it will send a *packet_in* towards the SDN controller. Since this process will be repeated to every packet that the switch does not know how to handle it, the controller will have the knowledge of every machine connected to each switch, because initially, the switch never knows how to handle a packet. In other words, the SDN controller will receive a *packet_in* message every time an "unknown" host wants to send a packet through a specific switch.

Forward and Control Elements Separation (ForCES) protocol specification is presented in (A. Doria et al., 2010). They separate two type of network devices: the Forward Elements (i.e. the switches) and the Control Elements (i.e. the controller). A group of Forward Elements is designed *Logical Function Blocks*. They summarize the protocol in 2 phases: pre-association and post-association. Pre-association is the whole boot process of configuration of a Forward Element before starting an association with the controller. This configuration is done by loading a *XML* configuration default file. Post-association is the process of communication between the Forward Element and the controller. It has 3 sub-phases: association setup stage, established stage and association lost stage. The association setup stage refers to the attempting of a Forward Element join to the controller. The established stage corresponds to when the connection between the Forward Element and the controller are active. In this phase, the controller sends query messages periodically and the Forward Element notifies the controller of any event in the network (e.g. node down). The association lost stage comes up when there is a disconnection. (T. Anderson & Khosravi, 2003) specifies the ForCES requirements. (Dantu, Anderson, Gopal, & Yang, 2004) specifies the ForCES framework.

Path Computation Element Protocol (PCEP) is another option to use as the ruler in the communication process in Southbound API as presented by (Vasseur & Le Roux, 2009). This protocol operates over TCP protocol. The controller element is defined as *Path Computation Element* (PCE). The forward elements, i.e. switches, are defined as *Path Computation Client* (PCC). PCEP uses 5 main messages: the *Open message*, the *KeepAlive message*, the *Notification (PCNtf) message*, the *Path Computation Request (PCReq) message* and the *Path Computation Reply (PCRep) message*. The *Open message* its used to stablish a connection. The *KeepAlive message* its used to maintain a connection stablished. The *PCNtf message* its used to notify the controller of specific events in the network (i.e. congestion link, etc). The *PCReq message* is sent by a PCC and processed

by the PCE in order to request a path. The *PCRep* message is sent by the PCE and processed by PCC to reply a previous *PCReq* message. The initialization phase of PCEP is defined by 2 phases: the TCP connection between the PCE and PCC and the establishment PCEP session over TCP. In order to establish the PCEP session, the PCC sends an *Open* message to the PCE and vice versa. In this message, various parameters can be negotiated (e.g. timer to the *KeepAlive* message). Both, PCC and PCE, acknowledge the *Open* message with a *KeepAlive* message. Then, PCC and PCE use *KeepAlive* message to maintain the session available. When an event on the network occurs (e.g. path selection), the PCC sends a *PCReq* to PCE. The PCE responds with a *PCRep*, the path is selected and the packet is forwarded. In case of overloading, both elements can use *PCNtf* to notify and react.

Alternatively, (Bierman & Bjorklund, 2012) presents Netconf. Netconf provides a mechanism to manage (e.g. install configurations) network devices. It uses *Extensible Markup Language* (XML) data encoding for the configuration data as well as the protocol messages and operates with *remote procedure calls* (RPC). An RPC, as its name indicates, is when a program on a computer causes the execution of a procedure, usually, on a different machine. It is realized by exchanging *<rpc>* and *<rpc-reply>* messages. This way, this protocol, is able to provide a set of operations for managing the devices configurations (e.g. get the devices configurations, edit it and delete the configuration).

In (Pfaff & Davie, 2013) is presented *Open vSwitch Database* (OVSDB) management protocol, which allows programmatic access to the *Open vSwitch* (OVS) database. OVS switch is a virtual switch that supports different protocols. It has 2 crucial components: the *ovsdb-server* and the *ovs-vswitchd*. The *ovsdb-server* is a database server that *ovs-vswitchd* queries to retrieve its configuration. The management protocol of OVS instances is made by a Manager. An OVS instance is managed by at least 1 manager. The OVS management interface is used for performing changes on the configuration of an OVS switch (e.g. the deletion of a datapath).

Another Southbound API candidate is the *Extensible Messaging and Presence Protocol* (XMPP) where its purpose is: "... to enable the exchange of relatively small pieces of structured data, called XML stanzas", over a network between any two (or more) entities" (Saint-Andre, 2011). These entities, usually are a client and a server. The client is the entity that establishes the XML stream with the server. The server is responsible for managing and delivering the *XML stanzas*. The next paragraphs summarize this protocol

giving an example, where a network entity pretends to transmit a XML stanza to another entity. First, the entity chooses the IP and port at which to connect. Second, establish a TCP connection. Third, open an XML stream over TCP. Fourth, negotiates the *Transport Layer Security* (TLS) for channel encryption. Fifth, authenticates using a *Simple and Security Layer* (SASL) method. Finally, it exchanges the *XML stanzas*, closes the XML stream and terminates the TCP session.

Lastly, MPLS Transport Profile (MPLS-TP) is presented (IETF RFC 6373, 2011). MPLS-TP supports static provisioning of transport paths via a control plane via a *Network Management System* (NMS) and dynamic provisioning of transport paths via a control plane. However, the use of a control plane is optional. Operators can configure the *Label Switch Path* (LSP) using NMS. LSP is used in data plane in order to fulfill its main functions.

2.1.3. SDN/Northbound APIs

Currently there is not a standard for SDN/NorthBound API, consequently there are a few options to study in this topic. The alternatives found in literature can be assembled in one big group characterized as “declarative configuration languages”. Although, some solutions do not fit in the previous group. Table 4 and 5 summarizes the solutions that we discuss in this section.

Table 4 – Northbound APIs Part 1 of 2

Northbound API	Description
FML (Hinrichs, Gude, Casado, Mitchell, & Shenker, 2009)	Declarative configuration language
Procera (Voellmy, Kim, & Feamster, 2012)	Declarative configuration language
Frenetic (Foster et al., 2011)	Declarative configuration language

Table 5 – Northbound APIs Part 2 of 2

Northbound API	Description
Pyretic (Poonam, 2016)	Declarative configuration language
NetCore (Monsanto, Foster, Harrison, & Walker, 2012)	Declarative configuration language
OSGi (Pham & Hoang, 2016)	Software platform
RESTful (Zhou, Li, Luo, & Chou, 2014)	Network protocol
SFNet (Yap, Huang, Dodson, Lam, & McKeown, 2010)	High-level API
I2RS (Atlas, Halpern, Hares, Ward, & Nadeau, 2016)	Network protocol

Flow-based Management Language (FML) is a language which goal is to configure network policies about flows (Hinrichs et al., 2009). A FML policy is a group of statements. The statements specify what action should be implemented to the matched flows. It uses 5 properties in order to characterize a flow: the sending and receiving host name, the sending and receiving user, the sending and receiving access points, the protocol and if the flow is or not a request. They can configure the policies of the system through 4 words: *allow*, *deny*, *waypoint* and *avoid*. Intuitively, *deny* blocks a flow and *allow* validates a flow. *Waypoint* requires a flow to pass through a particular node in the network. *Avoid* bans a flow from passing through a particular node in the network.

Another solution of nature type proposed by (Voellmy et al., 2012) is *Procera*. *Procera* is a declarative policies language based that allows network administrators to determine how a network controller should react to network events that cannot easily be done in

languages for SDN control. It calls on *Functional Reactive Programming* (FRP). FRP allows users to describe time-varying values by describing how their current value depends on event histories and current and prior values of other quantities. It does it by calling functions that easily support policies that involve arithmetic constraints and other functions. *Procera* has 4 main features: a core language based on functional reactive programming, event comprehensions to manipulate event streams, windowing and aggregation signal functions and the use of function values to represent high-level policy.

Another language for programming OpenFlow networks is *Frenetic* (Foster et al., 2011). It is developed in Python and comprises two integrated sublanguages: a high-level and declarative network query language and a general-purpose, functional and reactive network policy management library. The query language is used to get the representation of the network state. The network policy management manages the packet-forwarding rules. It comprises 4 actions: forwarding through a port, flooding through all ports forward to the controller and modifying the header. There are no drop actions. The switches will drop the packets when there is none of the rules aforementioned. *Frenetic* controls the installation of policies by generating *policy events* using *Functional Reactive Programming* (FRP). *Frenetic* also include *Listeners*. Listeners are the event consumers.

The successor of *Frenetic* is proposed by (Monsanto et al., 2012) - the *NetCore*. It is a declarative language for specifying high-level packet-forwarding policies. The *NetCore* run-time system translate these policies to switch-level rules and commands to install the generated rules on switches as presented. These high-level policies can be declared through predicates. Predicates are a simple word that identifies a characteristic in a header of a packet (e.g. "SrcAddr:10.0.0.0/8" identifies all packets that come from that network). Moreover, *NetCore* allows, also, to demand *dynamic policies*. Dynamic policies are those which forwarding behavior depends more than analyzing a simple header field of a packet, for instance, it can require the traffic history, etc. Although, the computation power limitation of switches is still challenge (Poonam, 2016).

Similar to *Frenetic* is *Pyretic* (Poonam, 2016). It is another programming language from the creators of *Frenetic* that tries to make it easier to creating sophisticated SDN applications. It is a Python based platform that allows programmers to program SDN applications and managing networks with a high level of abstraction. *Pyretic* is open source. It allows module programming and dynamic policies rules as *NetCore*. Also, as *NetCore*, *Pyretic* uses predicates to define policies. These policies can be sequential

composition or parallel composition. Sequential composition are policies that, once executed, the given output will be used as input to another policy. Parallel composition are policies that can be executed at the same time and independent way. Another API for network programming is NetKAT. It is based on Kleen algebra with tests (KAT) and It is inspired on NetCore and Pyretic (C. J. Anderson, Foster, Kozen, & Walker, 2014).

A different type was proposed by (Pham & Hoang, 2016) when *Open Services Gateway Initiative* (OSGi) was presented. OSGi's architecture is decomposed into several modules (services, life cycle, modules, execution environment and security) and allows for dynamic addition of new modules that can be seen as user's applications. These modules can be remotely installed, started, stopped, updated, and uninstalled. Each module (application) is dynamically loadable as a collection of code and configuration files that describe their relations with other bundles (if any). This way, we are able to manage a network by starting and stopping these modules. A practical application of OSGi is in one of the most prominent SDN controller platform that is the OpenDaylight project.

The most common Northbound API was introduced by (Zhou et al., 2014) - *Representational State Transfer* (REST). REST is a networking protocol for the Northbound API for SDN. REST APIs use HTTP/HTTPS protocol to communicate with the SDN controller, through the *Uniform Resource Identifier* (URI) string. URIs are strings that identifies each network device (Darien Hirotsu, 2016). Moreover, inside of HTTP protocol (e.g. 200 OK response), they use JSON or XML configuration script to request or update the configuration of a network element. This is doable by addressing the HTTP messages to the controller IP address.

SFNET (Yap et al., 2010) is a high-level API designed to explore the software friendly network designs and used on top of NOX controller. As (Poonam, 2016) does reference, SFNet allows applications to interact with lower level service requests and the network itself. It support network status requests, resource reservation and other high-level primitives. This exchanged data is represent in JSON data format.

For the last, as (Atlas et al., 2016) specifies, *Interface Routing System* (I2RS) is: "... a standard, programmatic interface for state transfer in and out of the Internet routing system.". It allows applications to change and request information. In, SDN paradigm, these applications match the controller/s. The services of this protocol can be available either on controller machine, i.e. locally or across the network. In order to make feasible

these services, two core actors exist. The local I2RS client and the I2RS agent. The local client locates on the control plane and the I2RS agent locates on the data plane. The I2RS agents serve one or more I2RS client. The I2RS client is the entity that access the I2RS agents. The I2RS clients, through the I2RS agents, can pull information about the network and modify it. The I2RS agent is the entity responsible for changing the state locally on forward elements. The I2RS agents, also, allows the subscription of notifications, by I2RS clients. These notifications are produced by the I2RS agents and are related with some events that occurred in the network (e.g. the deletion of an object). I2RS is based on NETCONF protocol while its data models are based on YANG modeling language. Nevertheless, this protocol focuses only on routing operations.

2.2. Distributed Denial-of-Service Mitigated with SDN

The current section discusses some relevant literature solutions based on SDN that mitigate DDoS attacks. Therefore, we have organized those in three different techniques category: the signature-based, the anomaly based and others related (see Table 6, 7, 8 and 9). Signature-based mitigation techniques are those which detection systems inspects the packet headers exchanged in the network. Anomaly-based mitigation techniques are those which tries to perceive what is the normal behavior of the network through inspection of flows. The others related are those which do not belong to any one before mentioned.

Table 6 – Literature SDN DDoS Mitigation Solutions Part 1 of 4

Authors	Signature/Anomaly based	Implemented on the client side	Detection & Mitigation	Differentiates malign/benign packets
(Ujcich & Rausch, 2014)	Signature-based	Not specified	Yes	Yes

Table 7 – Literature SDN DDoS Mitigation Solutions Part 2 of 4

Authors	Signature/Anomaly based	Implemented on the client side	Detection & Mitigation	Differentiates malign/benign packets
(You, Feng, & Sakurai, 2018)	Signature-based	Not specified	No	No
(Xing, Huang, Xu, Chung, & Khatkar, 2013)	Signature-based	Not specified	Yes	Yes
(Ombase, Scholar, Bagade, Kulkarni, & Mhaisgawali, 2017)	Signature-based	Not specified	Yes	Yes
(Khimabhai & Rohokale, 2016)	Signature-based	Not specified	Yes	Yes
(Jakaria, Rashidi, Rahman, Fung, & Yang, 2017)	Signature-based	Not specified	Yes	Yes

Table 8 – Literature SDN DDoS Mitigation Solutions Part 3 of 4

Authors	Signature/Anomaly based	Implemented on the client side	Detection & Mitigation	Differentiates malign/benign packets
(Chowdhary, Pisharody, Alshamrani, & Huang, 2017)	Signature-based	Not specified	Yes	Yes
(Jevtic, Lotfalizadeh, & Kim, 2018)	Signature-based	Not specified	Yes	Yes
(a Wang, Guo, & Hao, 2014)	Signature-based	Not specified	No	No
(Suh et al., 2010)	Signature-based	Not specified	Yes	Yes
(Piedrahita, Rueda, Mattos, & Duarte, 2016)	Signature-based	Not specified	Yes	Yes
(Seungwon Shin, Yegneswaran, Porras, & Gu, 2013)	Signature-based	Not specified	Yes	Yes

Table 9 – Literature SDN DDoS Mitigation Solutions Part 4 of 4

Authors	Signature/Anomaly based	Implemented on the client side	Detection & Mitigation	Differentiates malign/benign packets
(Abubakar & Pranggono, 2017)	Anomaly-based	Not specified	Yes	Yes
(Alshamrani, Chowdhary, Pisharody, Lu, & Huang, 2017)	Anomaly-based	Not specified	Yes	Yes
(Huong & Thanh, 2017)	Anomaly-based	Not specified	Yes	Yes
(J. Liu, Lai, & Zhang, 2017)	Anomaly-based	Not specified	Yes	Yes
(Lu & Wang, 2016)	Anomaly-based	Not specified	Yes	Yes
(R. Wang, Jia, & Ju, 2015)	Anomaly-based	Not specified	Yes	No
(Braga, Mota, & Passito, 2010)	Anomaly-based	Not specified	No	Yes

2.2.1. Signature-Based

Signature-based mitigation techniques are those which detection systems inspect packets exchanged in the network and bases its decisions in predefined signatures stored in a database, like an anti-virus software.

A mitigating DDoS attack *signature-based* proposal that compounds SDN and IDS features is (Ujcich & Rausch, 2014). To inspect the network behavior, they implemented mirror traffic to feed *Snort*. In order to speed up the writing of Snort alerts, they appealed to Barnyard2 application. It converts Snort output to a MySQL database entry. The database will save all the alerts thrown by Snort. Then, they developed a module, called Response Engine. It adds additional information to the alerts of Snort, as alert count. When a predefined threshold is exceeded, it sends a flow rule to the SDN controller. Hence, the SDN controller sends the flow rule, received by the Response Engine, to the network's switches.

A very complete work of this nature proposed by (Chowdhary, Pisharody, Alshamrani, & Huang, 2017). They used Snort as a tool for DDoS detection. Their mitigation procedure consists on limiting the rate limit from a specific source to a specific destination. They have evaluated their proposal for ICMP flood attacks, TCP SYN flood, UDP flood-based attacks with different network topologies.

A study between two different IDS/IPS is proposed in (Ombase et al., 2017). They compared two different IDS/IPS – Snort and Bro. Snort is used in *inline mode* – used as bridge between 2 and 3. If it detects suspicious traffic, it drops packets. Nevertheless, this option increases the delay of delivered packets.

In (You et al., 2018) is presented an DDoS attack detection system based on *Packet In messages*. It is composed by three modules. First, the Data Collection Module, which banks the header fields and counts the number of *Packet In* packets that have been transmitted. It calculates three entropies – one, for the IP destination address, two for the IP destination port and third for the IP source address. The second module is the Trigger Module, which detects an irregular arrival of *Packet In* packets. The third module is the Detection Module that is responsible for evaluate if a DDoS attack is happening and notify the system. For instance, being the entropy the randomness of collected data. Given three entropies: the entropy of IP destination address, the entropy of destination Port and the entropy of IP source. If the entropy of IP destination address destination reduces, the

entropy of destination Port reduces as well and, in the other hand, the entropy of IP source increase, it is probable that the system evaluates it as a DDoS attack.

Another entropy DDoS detection proposal is presented by (S. M Mousavi, 2014). They have simulated a network based on SDN architecture with many subnetworks. Nevertheless, both these proposals do not contain a mitigation process.

Other proposal of this nature is presented by (Kia, 2015). It can be broken in 7 main phases. First, collection of data. Second, calculation of entropy and comparison. Third, flow initiation rate computation and comparison. Fourth, finding the possible attack paths if an attack is suspected in phase two or three. If not return to the first phase again. Fifth, polling the switches in the attack path for flow statistics and studying the returned results from the switches to confirm or cancel an attack state. Sixth, update the thresholds according to the detection result in step 5. Seventh, mitigate the attack. This mitigation is made by setting *idle_timer* shorter.

A lightweight and effective solution for detecting DDoS attack in SDN is presented by (Seyed Mohammad Mousavi & St-Hilaire, 2015). Their system is also entropy based. Although they do not present a mitigation procedure and their detection is "... against SDN controllers".

An architecture that integrates the Snort IDS with OpenFlow is presented in (Xing et al., 2013). In addition, they developed the SnortFlow Server. Its goal is to classify network security status and updates actions in order to reconfigure the network. The SnortFlow Server is composed by four main modules. First, the SnortFlow daemon. It is responsible for gathering the alerts generated by Snort. Second, the Alert Interpreter. It processes the alert and classify the traffic as suspect or not. Third, the Rule Generator. Its goal is to, accordingly to the classification of Alert Interpreter, generate rules to insert in OpenFlow device to reconfigure the network. Finally, the fourth is a Database that exists for data analytics. In (Xing, Xiong, Huang, & Medhi, 2015) a similar work is presented.

A DDoS mitigation based on SDN/OpenFlow is presented by (Sahay, Blanc, & Zhang, 2015). This solution consists on placing two controllers: one on the ISP and one on the customer. The ISP controller is responsible for the mitigation process. The customer controller monitors the customer network, in a statistical way and detects DDoS attacks. When the controller detects anomalous flows, it generates security alerts and send them to the ISP controller. This security alerts triggers the mitigation phase. The mitigation

process is done by the ISP controller through the emission of a flow rule to *middleboxes* (i.e. specific boxes deployed in the ISP network that complements the mitigation process).

Alternatively, (Jevtic, Lotfalizadeh, & Kim, 2018) presents a network-based DDoS mitigation. It takes advantage of OpenFlow statistics collection in order to extract network features. Periodically, an application instructs the SDN controller to take the collected information in each switch. Hence, a network profile is defined. If it matches with a DDoS profile the alarms will ring.

In Scotch (a Wang, Guo, & Hao, 2014) is presented using OpenFlow vSwitch overlay network in order to improve control plane capability to help mitigating the bottleneck of the control path of SDN. They deployed OpenFlow Agents into switches across the network, that work as a proxy between the controller and switches. Nevertheless, they mitigate the attack by blocking all traffic, including benign traffic. Plus, by adding the OpenFlow Agents to the switches, goes against the idea of SDN's centralization.

A similar work to Scotch is In Content-Oriented Networking Architecture (CONA) (Suh et al., 2010). It presents an architecture where was added intermediate agents between servers and hosts. The main difference, in this architecture, is that hosts cannot freely send packets to any destination. In order to hosts send packets, they have to request contents and its agent deliver the requested contents. Hosts and servers cannot communicate directly with each other. For instance, looking to a common web page access. First, the agent has to request the content of the web page to his agent. Then, two different things can happen. One, the agent has in cache the content of the web page and respond to the host directly. Two, in case that the agent does not have it in cache, it will have to request it to the agent of the web site. The agents participate in the whole communication process. Hence, they can easily police the traffic (e.g. which contents are requested by which hosts, request rate). If an alarm is launched, then the controller will alert all agents to do not forward the content request to the victim server. Nevertheless, this feature may enhance security but increases the latency. It also is only designed for http traffic.

FlowFence (Piedrahita et al., 2016) uses the network switches to detect congestion by monitoring the bandwidth usage. If some switch is getting overloaded, it notifies a congestion state to the controller. Then, the controller sends a flow rule that limits the bandwidth of those switches. Although, this mechanism achieved a fast and simple

implementation that prevents full resources deprivation it does not stop a DDoS attack in fullness. This work is based on (X. Liu, Yang, & Xia, 2010).

Avant Guard (Seungwon Shin et al., 2013) is a framework that aims to detect and mitigate DDoS attacks on switch level. It can be separated in two modules. The first one, is called Connection Migration (CM). CM aims to be used as a proxy, over TCP SYN requests and traffic classifier, though SYN cookies. If it classifies as legit the traffic, then requests have permission and migrate to the real target. The second part is named as Actuating Triggers (ATs). In case a big volume of requests is realized, the ATs triggers an event towards the controller to react somehow to this event (e.g. insert a flow rule into the flow table). The big negative point about this one, is the reduce that inserts in performance. Besides it only works for TCP traffics.

Lastly, OrchSec (Zaalouk, Khondoker, Marx, & Bayarou, 2014) proposes an architectural solution that aims to reduce the overhead on SDN controller by splitting the communication traffic from the monitoring traffic through a manager/orchestrator. This entity, manages the controller functionalities that communicates via northbound API rather than being inside of the controller. Moreover, they also encourage the development of security applications on-top of OrchSec. For monitoring they used sFlow. Although, there were employed some techniques for mitigating network attacks (e.g. ARP Spoofing / Cache Poisoning, DoS/ DDoS), when an attack is detected, it just rate-limits or drop the traffic instead halting completely the attack traffic.

2.2.2. Anomaly-Based

Anomaly-based mitigation techniques are those which tries to perceive what is the normal behavior of the network and when it is under attack by inspection of the flows.

A mixture of an anomaly-based with signature-based is proposed by (Abubakar & Pranggono, 2017). More specifically, they used machine learning method with signature-based. The machine learning method has the intent to detect unknown attacks. The signature-based IDS is used to detect and mitigate known attacks. The mitigation procedure is made through flow modification.

A Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow is presented in (Braga et al., 2010). Summing it up, it uses a neural network for DDoS detection, that uses as input the features of the flows. Hence, it gives the classification of the traffic as

output, i.e. if it is happening a DDoS attack or not. This method brings flexibility to network management (e.g. remove/add OpenFlow Switches) and it is easily adaptable to the topology of the network. In (Gharvirian & Bohlooli, 2017) is presented a similar proposal. Their method has 5 main steps: One, count total number of packets in a flow. Two, calculate fast entropy for each flow within a defined time interval. Three, update adaptive threshold. Four, initiates a flow rate computation in determined time intervals. Five, examine flow statistics by neural network to confirm or reject existence of an attack. Nevertheless, these solutions require a training time before an appropriate use and do not mitigate a DDoS attack.

Another proposal of this nature is made by (Alshamrani et al., 2017). It consists on a Machine Learning DDoS detection that is incorporated on the controller. It takes advantage of *packet_in* messages in order to acquire predefined features. Thereby, these features are sent to a trained Machine Learning module to classify the traffic. Depending on the classification (i.e. benign or malign traffic), the controller updates the switch with new flow rule. Nevertheless, these solutions require a training time before an appropriate use.

In (Huong & Thanh, 2017) is proposed an anomaly mitigation called SDN-based Open-Packet DDoS Mitigation. It examines the statistics of the traffic through a group of incoming flows. Once a DDoS attack is detected, it reacts by deleting the malicious flows entries. Hence, the packets belonging to that flow will be dropped.

At (J. Liu et al., 2017) is proposed FL-GUARD (*Floodlight-based guard*). To the normal SDN architecture, are added more 4 modules: source IP anti-spoofing in controller and sFlow-RT Collector in control plane. The detecting and blocking attack modules are deployed in application layer. The IP anti-spoofing module is developed in order to fight the DDoS amplifications attacks by matching the switch interface and IP addresses. It uses sFlow-RT in order to gather information of network to define what is “normal” traffic in real time. The attack detection module is composed by a generic detection algorithm named C-SVM. It consists on the calculation of specific entropies and check if it goes under/over a predefined threshold. The blocking module is responsible for mitigating the attacks when the detection modules triggers an alarm.

Another proposal making use of sFlow is (Lu & Wang, 2016). It uses sFlow-RT agents deployed in switches to collect data about the network. Then, the collected data is sent to the detection module, where is analyzed according to an algorithm deployed in

Northbound API. Hence, in case of a success detection of DDoS attack, a flow rule is sent to switches for drop/block action. When the detection module defines that the network is no longer under a DDoS attack, the controller updates the flow rules for normal activity.

Lastly, an Entropy-Based Distributed DDoS Detection Mechanism in Software Defined Networks (R. Wang et al., 2015) presents a detection and response to DDoS attack, where, for each destination IP address, is calculated an entropy. If this entropy value declines more than a pre-defined value, then it is alarmed the detection of a DDoS attack. Although it achieved a reduction on overhead of the data get-together and the communication between the controller and OpenFlow switches some scalability and distribution, it cannot differentiate legit traffic from malicious traffic.

2.2.3. Other Related Techniques

Athena (Lee, Kim, Shin, Porras, & Yegneswaran, 2017) presents a framework that provides an API abstraction layer, that easily capacitate security researchers and develops to design and deploy security applications, or others of general-purpose without too much programming effort. Athena also brings itself a set of detection and mitigation algorithms, given a distributed databased and clustered computing platform (e.g. machine learning algorithms, K-Means, Logistic Regression). A similar work to Athena is Fresco that is available in (Seugwon Shin et al., 2013). An anti-DoS for OpenDayLight project was proposed by a security solutions provider (Radware, 2013). This proposal collects operational statistics and if necessary mitigates attacks by traffic redirection.

Concluding, we classified the solutions presented in three taxonomies: signature-based, anomaly-based and other related. To all signature-based techniques there are two drawbacks: its detection is as good as its “database of signatures” and the bigger the database is, the bigger the overload on the IDS is given that it will have to analyze its known signatures one by one. To all anomaly-based techniques there is one drawback: need to be trained by a human. If we think on training a neural network for every single home network we can conclude that, at this scale, it might not be usable. Therefore, we choose to go for the signature-based. Hence, in our proposal is presented a DDoS mitigation signature-based system conceived to be implemented specifically, at the client side, i.e. close to the home user’s network. Our proposal differentiates from previous literature contributions because we detect and mitigate the attack at its origin, eliminating, in this way, the attack amplification problem that could occur in other proposals.

Chapter 3 – System Design and Deployment

3.1. System Design

In this thesis, we present a DDoS attack detection and mitigation proposal by integrating an IDS with the SDN architecture, on the client side, i.e. close to user's home network.

As it is depicted in the following image (Figure 2), the system basic architecture resides on a loop with 3 main components: the network, the IDS and the controller. The network represents all the data traffic and where a potentially DDoS attack might be launched. The IDS represent our DDoS attack detection mechanism. The IDS must be fed by all the network traffic. When the IDS detect a DDoS attack it alerts the controller. The controller module closes the loop via a new flow rule emission towards the network, when needed. Hence, there are 3 crucial phases here: the detection phase, the communication phase and the mitigation phase. The detection phase consists on the capability of the system to detect if it is in the presence of a DDoS attack. The communication phase consists on alerting the controller when a DDoS attack happens. The mitigation phase consists on the reaction to the DDoS attack, once detected. This is done by sending a blocking flow rule to the switch. This blocking flow rule has no expiration time.

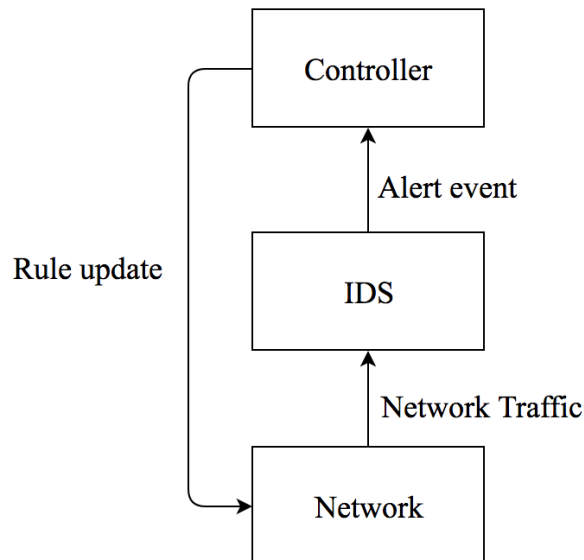


Figure 2 – System's Basic Architecture

3.2. System Deployment

The System deployment is presented in Figure 3. We can verify that the system is composed by 3 different Virtual Machines (VM) – the VM A, B and C.

VM A contains 2 modules: the Ryu controller and the IDS Snort. The Ryu controller's function is to program the performance of the network when specific events happens (e.g. receiving *Packet_in* message). These network behaviors updates are sent to the data plane via link (A). Snort is our rule-based system used in order to “fire the alarms” in the presence of a DDoS attack. Hence, when Snort detects an DDoS attack, it sends an alert packet via Unix Domain Socket – link (D).

VM B is where the emulation of a home network happens. It is where all data packets are changed and the decisions of VM A are implemented. It is composed only by Mininet. Mininet, in its turn, creates a virtual network with hosts, the OpenFlow switch and the NAT. The hosts are created to simulate infected and non-infected devices that have connection to an online resource and can potentially be DoS launchers. OpenFlow switch is created to run the client part of the OpenFlow protocol and belong to the data plane of our testbed. This switch, besides the OpenFlow protocol, was manually configured to perform port mirroring, in order to feed Snort with the packets exchanged in the network, through the link (B). The NAT element, beyond transforming the scenario in one more realistic home's network scenario, was manually configured and it is used as a gateway for the hosts having access to the online resource, through the link (C).

VM C is a virtual machine that represents an online service that can be potentially under DDoS attacks.

Therefore, we can see 4 different crucial connections between different elements (A, B, C and D).

(A) – is the Secure Channel between the OpenFlow switch and the remote controller. It is the bilateral link used to exchange traffic between the data plane and the control plane – The Southbound API.

(B) – is the unilateral link between IDS Snort and OpenFlow switch, that we have configured to perform mirroring of the all the traffic in the network.

(C) – is the bilateral link that makes the connection between the network devices and an online service.

(D) – is the unilateral link that Snort uses to, notify the controller, that the network topology is in the presence of a DDoS attack, through *alert packets* via Unix Domain Socket.

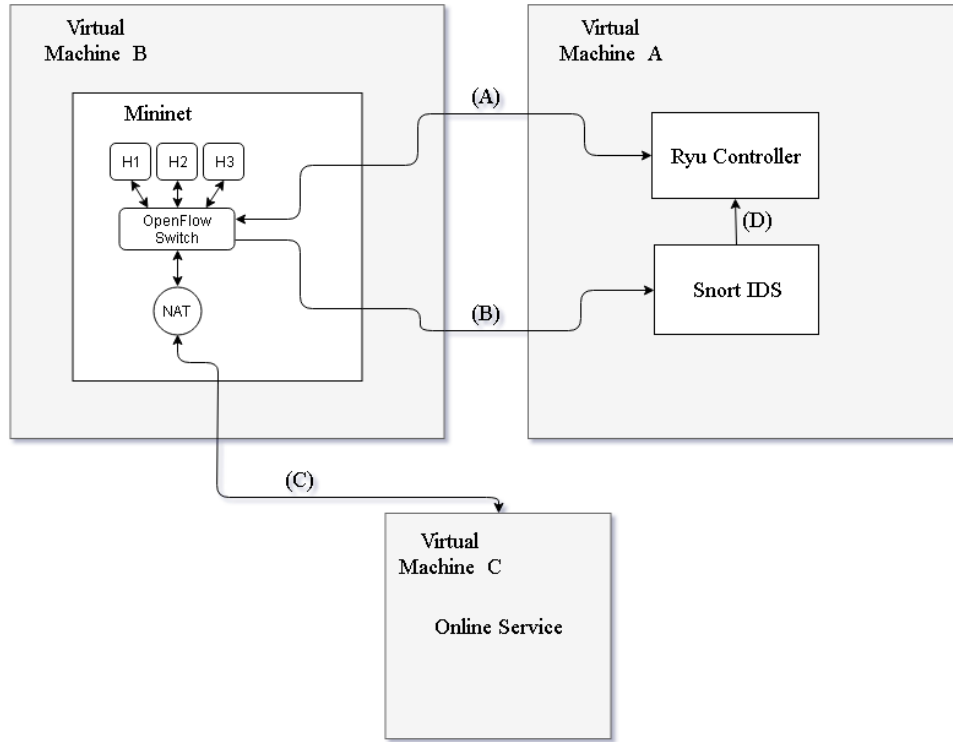


Figure 3 – System Deployment

3.3. Deployed System Components

In this subsection, we will go through all the components and its implementation aforementioned: the virtual machines, the Ryu SDN controller, Snort IDS and the network emulator designated by Mininet. In addition, we detail some Mininet components such as NAT and OpenFlow switch.

3.3.1. Virtual Machines

In this subsection, we will go through the network configurations of VM A, B and C on the VirtualBox application.

Virtual Machine A

Virtual Machine A, which contains the controller and Snort IDS, is configured to use 2 network adapters. The adapter 1 is configured as Host-Only (Figure 4). It is used for the connection between the OpenFlow switch and the SDN controller.

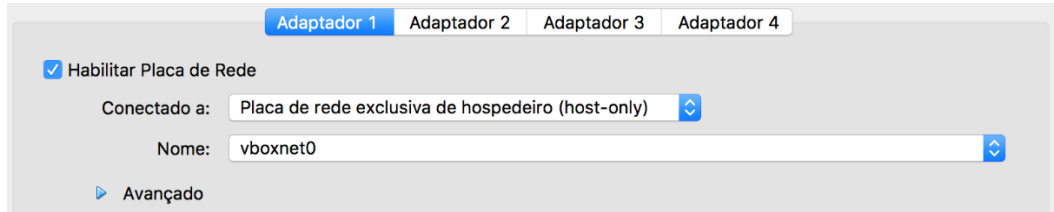


Figure 4 – VM A Network Adapter 1

The adapter 2 (Figure 5) is configured as internal network – *intnet*. Conceptually it makes the Snort IDS belong to the same network as the Mininet’s network. It is used for making the connection between the Snort IDS and network traffic.

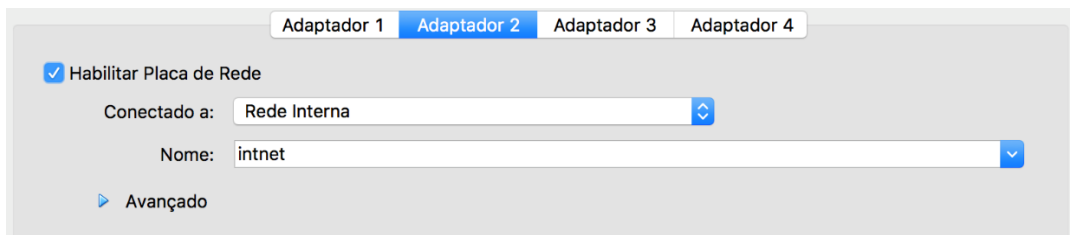


Figure 5 – VM A Network Adapter 2

The result of above configurations is shown in Figure 6, through the *ifconfig* command. The VM network adapters 1 and 2 matches, respectively, to the interfaces *enp0s3* and *enp0s8*.

```

pedro@pedro-VirtualBox:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.56.101 netmask 255.255.255.0 broadcast 192.168.56.255
    inet6 fe80::a00:27ff:fe0a:eca5 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:0a:ec:a5 txqueuelen 1000 (Ethernet)
    RX packets 2 bytes 1180 (1.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 58 bytes 7162 (7.1 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::a00:27ff:fe60:7627 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:60:76:27 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 37 bytes 5375 (5.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 197 bytes 13918 (13.9 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 197 bytes 13918 (13.9 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Figure 6 – VM A Network Interfaces

Virtual Machine B

Virtual Machine B, which contains the Mininet, is configured to use 2 network adapters. The network adapter 1 is configured as Host-Only (Figure 7). It is used for the connection between the OpenFlow switch and the SDN controller.

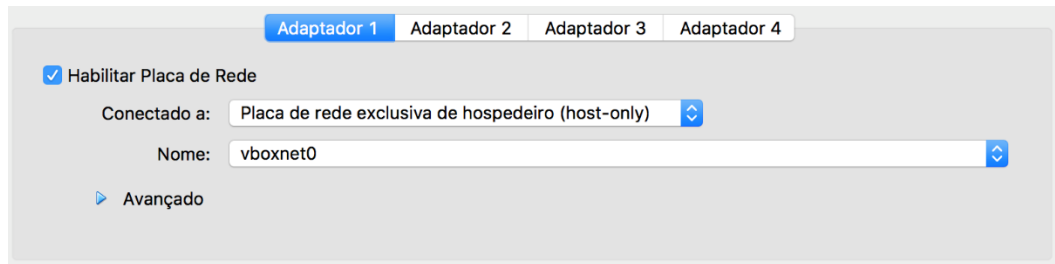


Figure 7 – VM B Network Adapter 1

The adapter 2 (Figure 8) is configured as the same internal network as VM A – *intnet*. Conceptually it makes the Snort IDS belong to the same network as the Mininet's network. It is used as an end point of the communication channel that transports the network's mirrored traffic from the OpenFlow switch to Snort IDS.

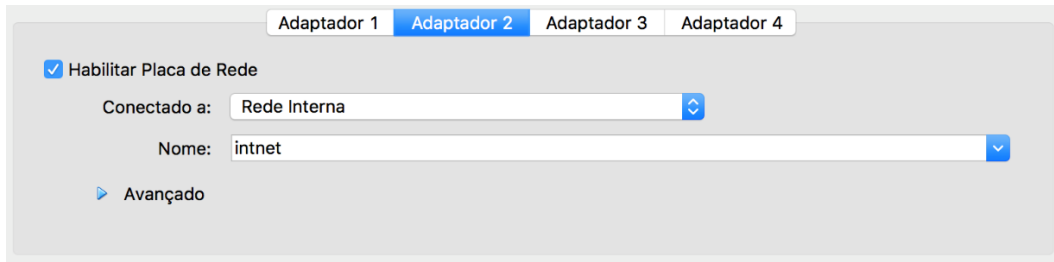


Figure 8 – VM B Network Adapter 2

The result of this configuration (Figure 7 and Figure 8) is shown in Figure 9, through the *ifconfig* command. The VM network adapters 1 and 2 matches, respectively, to the interfaces *enp0s3* and *enp0s8*.

```
mininet@mininet-VirtualBox:~/mininet/examples$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:0f:22:da
        inet addr:192.168.56.102  Bcast:192.168.56.255  Mask:255.255.255.0
        inet6 addr: fe80::da8b:2715:26a7:412e/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:2 errors:0 dropped:0 overruns:0 frame:0
        TX packets:151 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:1180 (1.1 KB)  TX bytes:12856 (12.8 KB)

enp0s8  Link encap:Ethernet  HWaddr 08:00:27:50:4b:62
        inet6 addr: fe80::a00:27ff:fe50:4b62/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:52 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 B)  TX bytes:5666 (5.6 KB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:65536  Metric:1
        RX packets:1092 errors:0 dropped:0 overruns:0 frame:0
        TX packets:1092 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:83584 (83.5 KB)  TX bytes:83584 (83.5 KB)
```

Figure 9 – VM B Network Interfaces

Virtual Machine C

Virtual Machine C, which simulates an online service, is configured to use only 1 network adapter. This network adapter is configured as Host-Only (Figure 10). It is used for the connection between the Mininet network and the online service.

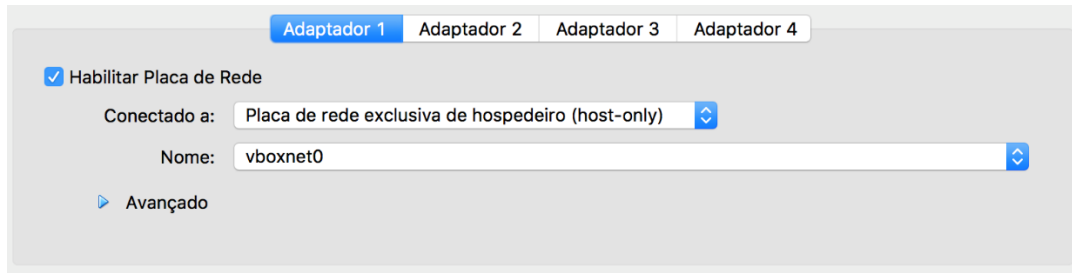


Figure 10 – VM C Network Adapter 1

The result of this configuration is shown in Figure 11, through the *ifconfig* command. The VM network adapter 1 match to the interface *enp0s3*.

```
server@server-VirtualBox:~$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:69:f9:a0
        inet addr:192.168.56.104  Bcast:192.168.56.255  Mask:255.255.255.0
        inet6 addr: fe80::90f0:9a1:c733:3018/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:3 errors:0 dropped:0 overruns:0 frame:0
        TX packets:56 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:1278 (1.2 KB)  TX bytes:7078 (7.0 KB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:65536  Metric:1
        RX packets:516 errors:0 dropped:0 overruns:0 frame:0
        TX packets:516 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:39920 (39.9 KB)  TX bytes:39920 (39.9 KB)
```

Figure 11 – VM C Network Interfaces

3.3.2. SDN Controller

Controller Ryu (RYU project team, 2013) is the entity that provides a centralized view and management through software. We choose the Ryu SDN controller over the others presented in subsection 2.1.1. because it allows the fast development of new applications. The code used to program the controller is stored the file *SimpleSwitchSnort.py*. It is partially available in Appendix A. Ryu applications are executed with *ryu-manager*. *Ryu-manager* is an executable that loads Ryu applications and run these allocating to them if necessary a default listener TCP port. Figure 12 shows our Ryu application starting.

```

pedro@pedro-VirtualBox:~/ryu-master/ryu/app$ sudo ryu-manager SimpleSwitchSnort.py
loading app SimpleSwitchSnort.py
loading app ryu.controller.ofp_handler
instantiating app None of SnortLib
creating context snortlib
instantiating app SimpleSwitchSnort.py of SimpleSwitchSnort
[snort][INFO] {'unixsock': True}
instantiating app ryu.controller.ofp_handler of OFPHandler
[snort][INFO] Unix socket start listening...

```

Figure 12 – Launching Ryu Application

The SDN controller has 5 major functions to program its behavior: *packet_in_handler*, *add_flow*, *send_blocking_flow* rule and *process_snort_alert*. The code of the controller was adopted from (Bonnet & Soenen, 2016).

The *packet_in_handler* (see Appendix A – *packet_in_handler* Code) is a handler method that runs every time a switch has a table-miss entry. Therefore, the switch will send a packet denominated by *Packet_In* message towards the SDN controller. When this happens, through the *packet_in_handler* we are able to program what we want to do to that specific packet. Usually, the SDN controller sends a forwarding rule, in order to avoid another table-miss entry.

In order to add any kind of rules to an OpenFlow switch we have the *add_flow* function (see Appendix B – *add_flow* Code). The *send_blocking_flowrule* is a particular case of *add_flow* function (see Appendix D – *send_blocking_flowrule* Code). It sends a dropping flow rule to the switch. Once this rule is updated on the switch, it never expires. This event is common to all the implementations made.

The *process_snort_alert* (Appendix C – *process_snort_alert* Code) is a function where we code what to do when a Snort alert is launched. In this case, it verifies if the alert thrown by the Snort IDS has in its content “ryu block”. If it has, it sends a blocking flow rule, otherwise it ignores it.

For a more detailed visualization of any these functions aforementioned, see Appendix A.

3.3.2.1. Controller Performance - Enhanced Version 1

In Figure 13 is presented the version 1 of the *process_snort_alert* function. The difference between this version and the version before is that we created a list that stores all IP addresses that were blocked – a blacklist. Every time that the Snort IDS alerts about a DDoS attack, the system will firstly see if that IP address was already block. If it was

not, the controller will send a blocking flow rule. Otherwise, it will ignore the alert. This way, we were able to make a significant decrease on the controller overload.

```
@set_ev_cls(snortlib.EventAlert, MAIN_DISPATCHER)
def process_snort_alert(self, ev):
    msg = ev.msg
    alertmsg = ''.join(msg.alertmsg)
    pkt = packet.Packet(array.array('B',msg.pkt))
    pkt_eth = pkt.get_protocol(ethernet.ethernet)
    pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
    src = pkt_ipv4.src

    if src not in self.blacklist:
        self.blacklist.append(src)
        self._dump_alert(ev)
        self.logger.info('Blocking this flow:{0}'.format(alertmsg))
        self.send_blocking_flowrule(msg)
```

Figure 13 – Snort Alert Handler Function Performance Enhanced Version 1

3.3.2.2. Controller Performance - Enhanced Version 2

Figure 14 presents the final version 2 of the *process_snort_alert* function. The difference between this version and the versions before is that we reduced the function's logic. In this case, we only send a flow blocking rule every time Snort alerts instead of making verifications i.e. see if the IP address is in a blacklist or see if the message thrown by snort contains the string 'ryu block'.

```
@set_ev_cls(snortlib.EventAlert, MAIN_DISPATCHER)
def process_snort_alert(self, ev):

    self._dump_alert(ev)

    msg = ev.msg
    alertmsg = ''.join(msg.alertmsg)
    self.logger.info('Blocking this flow:{0}'.format(alertmsg))
    self.send_blocking_flowrule(msg)
```

Figure 14 – Snort Alert Handler Function Performance Enhanced Version 2

3.3.3. Snort

Snort IDS (Snort, 2018) is our mechanism to detect DDoS attacks. We choose Snort as our IDS over other available software because, besides Snort being a reference in this field and having a huge community behind, it also allowed us to use it with our custom

rules in a relatively short time. Snort is a free rule-based IDS software that detects DDoS attacks with known signature. Hence, its detection bases on rules specified in a file. In order to use Snort, was necessary to configure the file *snort.conf* to consider our DDoS rules file: *ddos_detection.rules*. In this file, we specified all the rules used in this work. Figure 15 shows the command to launch Snort. We launched it with 4 arguments. Its mode of operation (1)), the interface that Snort “listens” traffic to analyze (2)), its configuration file (3)) and a file for logs (4)):

1) **-A unsock**: configures to send an alert through UNIX socket that another program can listen on;

2) **-i enp0s8**: choose the enp0s8 interface to listen traffic;

3) **-c /etc/snort/snort.conf**: choose *snort.conf* as the configuration file to launch;

4) **-l /tmp/**: write logs in the directory */tmp/* a file.

```
snort -A unsock -i enp0s8 -c /etc/snort/snort.conf -l /tmp/
```

Figure 15 – Snort Launch Command

In 1), we have a rule used in order to detect DDoS attacks. According to that rule, an alert will be generated when Snort capture more than 10 packets within 1 seconds from any source IP in *10.0.0.0/8* network and port, to *192.168.56.104* in any port.

1) *alert icmp 10.0.0.0/8 any → 192.168.56.104 any (msg:"ryu block ";*
detection_filter:track by_src, count 10, seconds 1, sid:1000001)

3.3.4. Mininet

Mininet (Mininet, 2014) is used in our work to emulate a network. It creates a network, as showed in Figure 16, through a Python script file. Our topology consists on three hosts, one OpenFlow Switch and one NAT device. Therefore, in that Python script file, there are one crucial object (i.e. *net* variable) and 4 functions: *addController*, *addSwitch*, *addNAT*, *addHost* and *addLink*. The *net* variable saves a reference for the instance of the Mininet’s constructor. We can think of it as a variable that represents all network which at that point is “empty”. The *addController* function creates an instance of a controller. In its arguments, receives 4 attributes: a name, a remote controller (i.e. a

controller that is on a different VM), an IP address, its port and the protocol. The **addSwitch** function creates an instance of a switch with 2 attributes: its name and the cls (link class). The **addNAT** function creates a NAT object with 2 attributes: its name and its IP address. The **addHost** function creates an instance of a host. It receives a name, a custom host class (“cls=Host”), IP address and its port and a **defaultRoute** via NAT IP address. Finally, **addLink** function creates a link between 2 network elements. Therefore, it receives both network elements as arguments. After creating the network, Mininet provides us with a CLI capable of executing commands.

```
net = Mininet( topo=None,
              build=False,
              ipBase='10.0.0.0/8')

info( '*** Adding controller\n' )
c0=net.addController(name='c0',
                    controller=RemoteController,
                    ip='192.168.56.101',
                    protocol='tcp',
                    port=6633)

info( '*** Add switches\n' )
s1 = net.addSwitch('s1', cls=OVSKernelSwitch)

nat0 = net.addNAT('nat0', ip='10.0.0.254')

info( '*** Add hosts\n' )
h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', mac='00:00:00:00:00:01', defaultRoute='via
10.0.0.254')
h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', mac='00:00:00:00:00:02', defaultRoute='via
10.0.0.254')
h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', mac='00:00:00:00:00:03', defaultRoute='via
10.0.0.254')

info( '*** Add links\n' )
net.addLink(h1, s1)
net.addLink(h2, s1)
net.addLink(h3, s1)
net.addLink(s1, nat0)
```

Figure 16 – Python Script for Network Creation

In Figure 17, is presented the execution of **dump** command. **Dump** command show us info about the network created. As we can verify, a network with a NAT, 3 hosts, 1 switch and 1 controller were created. One more useful command used was **xterm**. **Xterm** command allow us to open a CLI in the network elements that we want.

```
mininet> dump
<NAT nat0: nat0-eth0:10.0.0.254,nat0-eth1:None pid=2307>
<Host h1: h1-eth0:10.0.0.1 pid=2319>
<Host h2: h2-eth0:10.0.0.2 pid=2321>
<Host h3: h3-eth0:10.0.0.3 pid=2326>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None,s1-eth4:None,
s1-eth5:None pid=2302>
<RemoteController c0: 192.168.56.101:6633 pid=2295>
mininet> █
```

Figure 17 – Dump Command Output

3.3.4.1. NAT

In this subsection, the configuration of NAT is presented. To do that so, the scenario consists on a simulation of a home's network and an online service in a distinct network. The home network IP address range is *10.0.0.0/8*. The online service IP address is *192.168.56.104*. The connection between these two entities is made through a NAT. By similarity to the reality, NAT works as a gateway of our home network. This concept is shown in Figure 18.

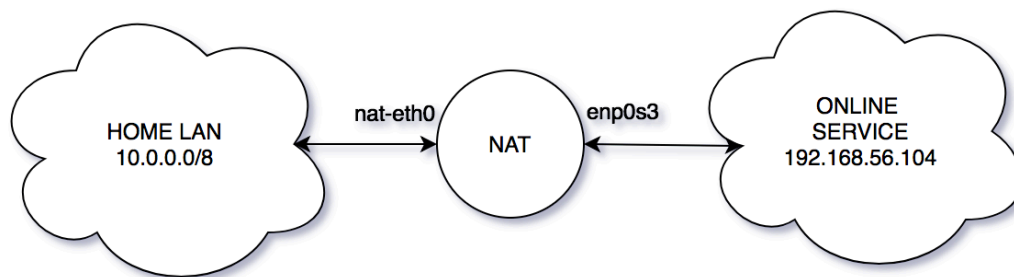


Figure 18 – NAT General Overview

NAT's routing table configuration is presented in Figure 19. That configuration is not set by default. In order to configure NAT, we accessed to the NAT CLI by the *xterm* command in Mininet CLI. To manually add **1)** or remove **2)** entries to the routing table we used the following commands:

1) `route add -net SERVER_IP_ADDRESS netmask NETMASK gw VMB_enp0s3_IP_ADDRESS`

2) `route del -net IP_ADDRESS netmask NETMASK gw GW_IP`

```
root@mininet-VirtualBox:~/mininet/examples# route -n
Kernel IP routing table
Destination    Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0       0.0.0.0         255.0.0.0       U        0      0        0 nat0-eth0
192.168.56.0   192.168.56.102 255.255.255.0   UG        0      0        0 enp0s3
root@mininet-VirtualBox:~/mininet/examples#
```

Figure 19 – NAT Routing Table

3.3.4.2. OpenFlow Switch

In this subsection, the **Port Mirroring** configuration that was added to the switch is presented. Initially, the switch is created by Mininet's script presented already. **Port Mirroring** consists on duplicating packets that go into/out one switch's port and forward these duplicated packets to a different port. Usually, it is used for traffic monitoring purposes as it is in our case. Figure 20 shows the concept behind of **port mirroring**. The original traffic is marked as white and the duplicated traffic as dark grey. The ports of the switch are enumerated with the numbers 1, 2 and 3. When host A sends a packet to host B (white packets) the switch will receive packets in **port 1** and will forward it to **port 2**. Furthermore, due to **port mirroring** configurations, the switch will, also, copy the packet that come in **port 1** and send that copy to **port 3**. This configuration can cause overhead over the switch.

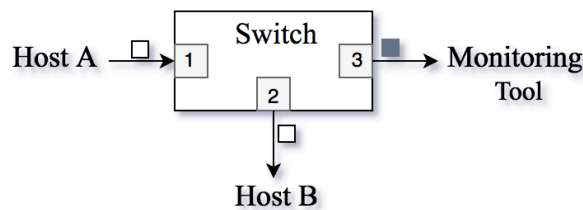


Figure 20 – Port Mirroring Concept

In order to configure **port mirroring**, we have started by removing the IP address to the interface **enp0s8** of the VM B with the command **1)**. Then, we used **ovs-vsctl tool**. **Ovs-vsctl tool** allows us to configure and query the switch. To the Open Flow switch (**s1**), we added a port that connects to the interface **enp0s8** - **2)**. With **3)**, we created a mirror and added a **bridge** to **s1**. Through **4)**, we target the traffic that is to be **mirrored**. With this configuration, all the traffic which goes out to **port s1-eth1** of **s1** will be mirrored. Hence, we complete the **mirror** configuration with the command - **5)**, which defines the destiny of the mirror traffic. In this case, all the traffic that was targeted to be mirrored, will be mirrored to the **port s1-eth4**.

- 1) – `sudo ifconfig enp0s8 0`
- 2) – `sudo ovs-vsctl add-port s1 enp0s8`
- 3) – `sudo ovs-vsctl -- --id=@m create mirror name=mymirror -- add bridge s1 mirrors @m`

4) – `sudo ovs-vsctl -- --id=@s1-eth1 get port s1-eth1 -- set mirror mymirror select_dst_port=@s1-eth1`

5) – `sudo ovs-vsctl -- --id=@s1-eth4 get port s1-eth4 -- set mirror mymirror output-port=@s1-eth4`

3.4. Our Proposal Workflow

In this section, is presented the workflow of our system (Figure 21) for each packet that arrives to the OpenFlow switch. Considering a scenario where a host tries to send an ICMP packet to the online resource. The OpenFlow switch will receive a packet and verify if there is any flow rule in its flow table that matches that packet. If there is not, the switch will “ask” to the controller for a rule to handle futures packets and the controller will respond by sending a forwarding rule. If exists a rule, the switch will forward the packet according to the rule. In this scenario, to the NAT gateway. Hence, the NAT will forward the packet to the online service. In either case, the switch will mirror every packet arrives to Snort. Then, Snort will evaluate if a DDoS attack is happening. If it is not, it does nothing. If it is, Snort will alert the SDN controller. Finally, the SDN controller send a blocking flow rule to the switch and mitigate the attack.

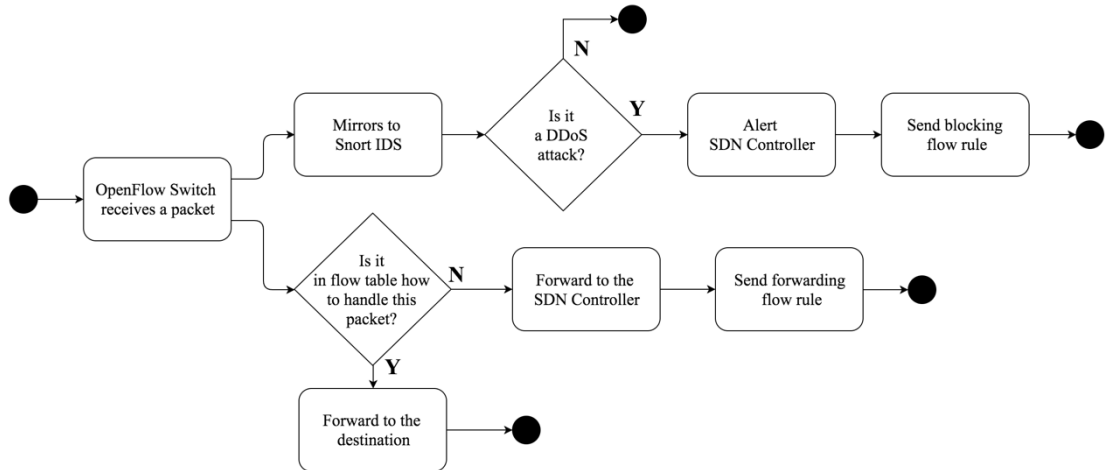


Figure 21 – System Workflow for Each Packet

Figure 22 presents us with a DDoS mitigation sequence diagram of our proposal. Considering that we have a malign host launching a DDoS attack (UDP flooding) and

that this switch has a rule to forward packets that come from that host. First, the **ICMP Echo Request** arrives to the switch. Then, the switch forwards the **ICMP Echo Request** to NAT (see Figure 22, interaction between Switch-NAT). At almost the same time, the switch will mirror the **ICMP Echo Request** to the Snort IDS (see Figure 22, interaction between Switch-Snort). While NAT forwards the **ICMP Echo Request** to the Server and waits for the **ICMP Echo Reply**, the Snort IDS will analyze the **ICMP Echo Request**. If Snort considers that the host is a malign user, it will send an alert to the SDN controller, via *Unix Domain Socket* (see Figure 22, bottom part interaction between Snort-Controller). Then, the controller will mitigate the attack by sending a blocking flow rule to the switch (Figure 22, Controller-Switch interaction). Hence, future packets sent by that Host will be dropped. All the **ICMP Echo Reply** messages are not guaranteed delivered before the mitigation process (see Figure 22, Server-NAT-Switch-Host).

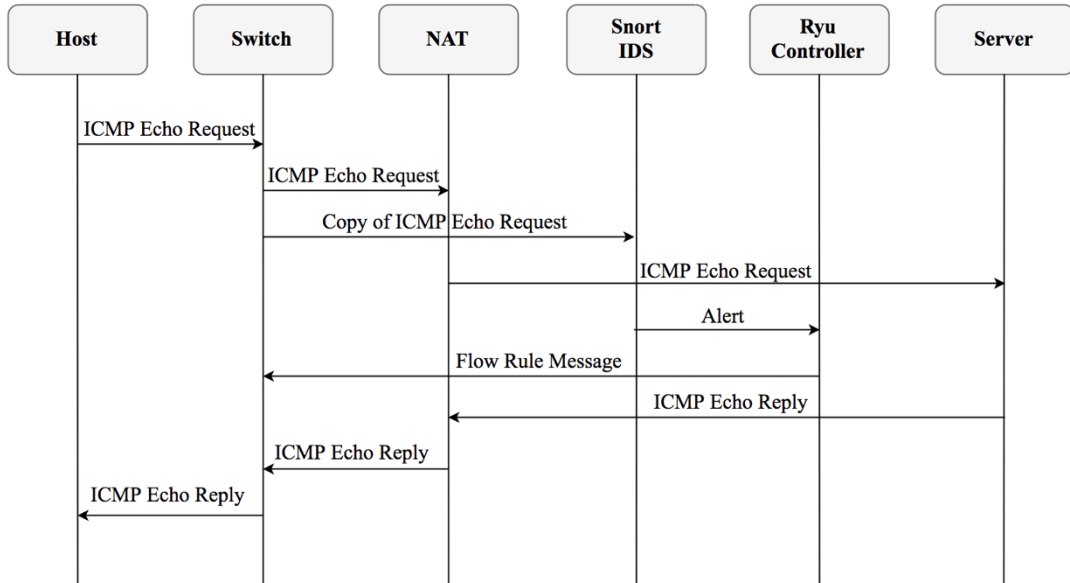


Figure 22 – System Workflow Exchanged Messages Diagram

This way, we are able to compound SDN with Snort IDS in order to mitigate a DDoS attack. Given that, we can notice two things. First, the attack was mitigated at its source, i.e. at one hop distance from its source (see Fig. 22, Host-Switch). Second, the placement of the Snort IDS does not increase overhead to the network (i.e. it does not add hops to the ICMP packets exchanged between Host/Server). Although it might increase overhead in the switch due to *mirroring* processing.

Chapter 4 – Evaluation Scenario, Performed Tests and Results Discussion

The current chapter is about our proposal evaluation, namely in the next functional aspects:

- If it has the capability to detect and mitigate a DDoS attack;
- If it preserves the *QoS* for the “benign” hosts that are using an Online Service while it is under a DDoS attack;
- The network overload induced by our proposal, through the measurement of some network metrics such as, average RTT and percentage of packet loss;
- The proposal’s efficiency, through the time that takes to detect and mitigate the DDoS attack;

Therefore, we describe the tests made (section 4.1) and present its results (section 4.2). This chapter, also does contain content about the home simulated network and for the tool used in all test scenarios – *hping3*. Therefore, we start with a description of a common characteristics to all scenarios as: the topology of the home network and the tool used for DoS simulation.

In order to simulate a home network with potential infected and benign devices, we have created a network with 3 hosts (host 1, 2 and 3), one OpenFlow switch and a NAT router, as presented in Figure 23. The topology used (e.g. number of hosts, number of switches, etc) was arbitrary and the system functionalities are independent of it. The hosts represent devices connected to an online service. These hosts may behave as a malicious botnet and launch a DDoS attack. Their IP and MAC addresses are crucial to understand the updates on the configurations of the switches flow rule table.

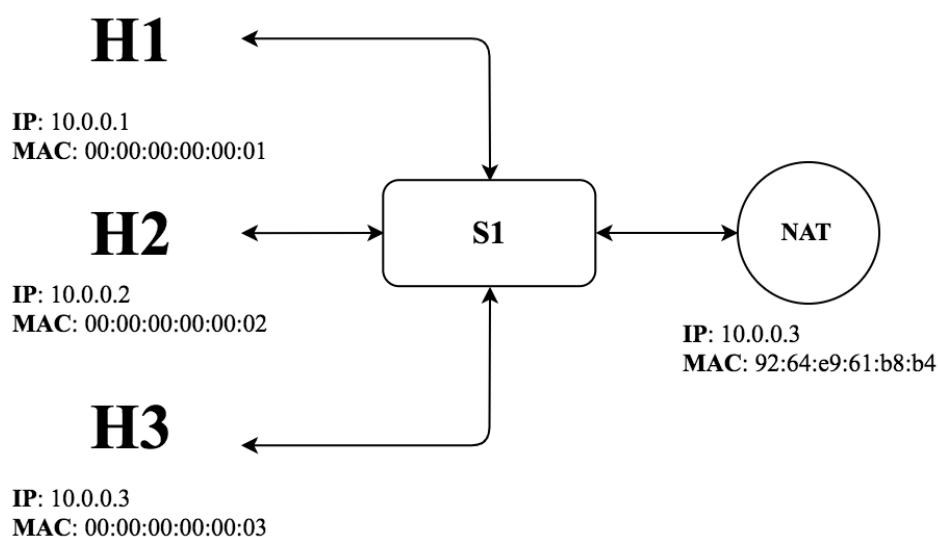


Figure 23 – Home Network Topology Simulated

In order to simulate the DoS engagement, we used the Hping3 tool. Hping3 is a network tool that is able to send custom TCP/IP packets. It was the tool chosen to launch a DoS attack in each host. In the Table 10 are shown the arguments used forward in the tests.

Table 10 – Hping3 Tool Arguments Used in our Test Scenarios

Arguments	Description
<i>-i</i>	Indicates the interface to use
<i>-flood</i>	Sends packets as fast as possible, without taking care to show incoming replies
<i>-I</i>	Uses ICMP mode
<i>-a</i>	Sets a fake IP address
<i>--rand-source</i>	Send packets with random source address
<i>-d</i>	Defines the size of each packet

4.1. Tests description

In this section, we introduce the scenarios used to test our proposal. Although the scenario I was subjected to performance improvements and divided to subtests, there are only 3 different scenarios, in total.

4.1.1. Description of Scenario I Test

The scenario I test consists on the simulation of a botnet DDoS attack. In order to do that, we simulate the botnet by launching a DoS attack from 2 devices. Hence, we have 2 infected devices (host 1 and host 3) and 1 benign host (host 2), as illustrated in Figure 24. The infected devices will launch a DoS attack aiming the online service. This attack, will consists on an extremely high rate of transmission packets per second. Therefore, we used *hping3* tool in ICMP mode to flood packets as fast as it can. The benign host will be accessing to the Online Service via the *ping* command – *ping Online_Service_IP*. This test, will be useful to evaluate if the benign host access is cut down and the impact that it might have on its usual usage, when a DDoS attack occurs. The effectiveness of our system will be measure through: if it mitigates the attack and the time it takes to mitigate it. The *Quality of Service* (QoS) will be measure with 2 variables – the average *Round Trip Time (RTT)* and the percentage of the packet loss. We will also present the results of the enhancements made.

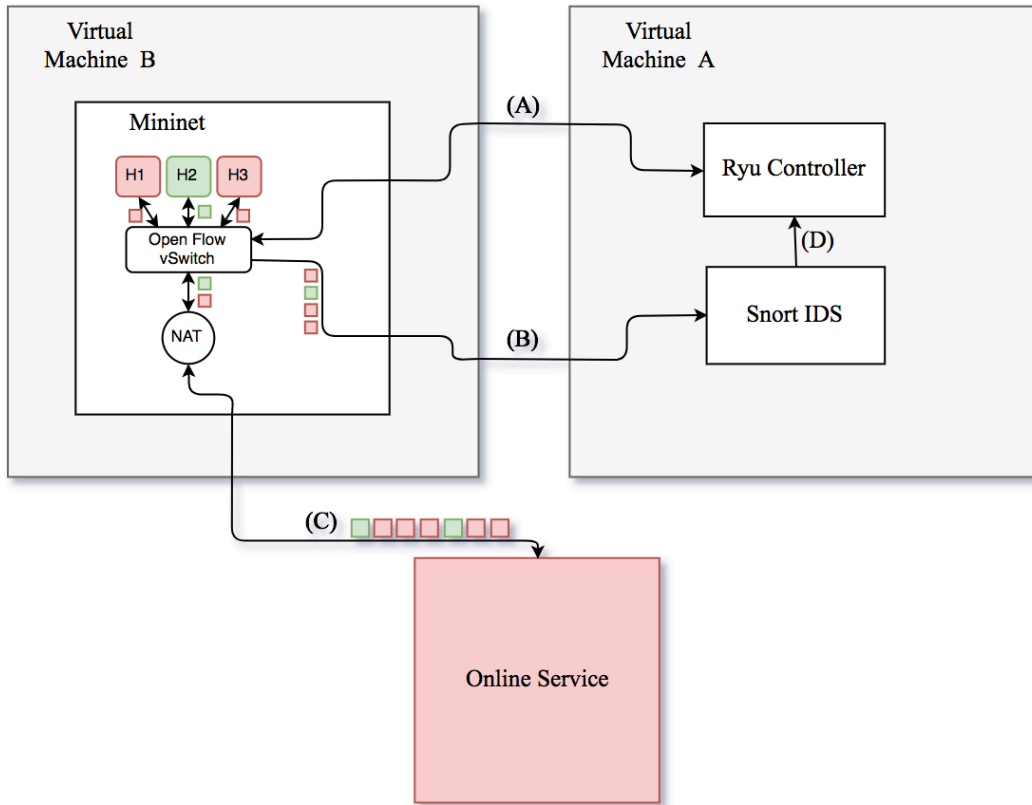


Figure 24 – Scenario I Test Diagram

4.1.2. Description of Scenario II Test

The scenario II test consists on testing our system in a DoS attack scenario with a false IP address – *spoofing* (Figure 25). In order to do that, we considered a single infected device (host 1) that will launch the attack. Host 2 and 3 will have its usual access to the server. This test on spoofing an IP address that belonging to the internal home network (i.e. $10.0.0.0/8$ network). Therefore, we used the hping3 tool in ICMP mode, flooding packets as fast as possible with a fake IP address defined by us.

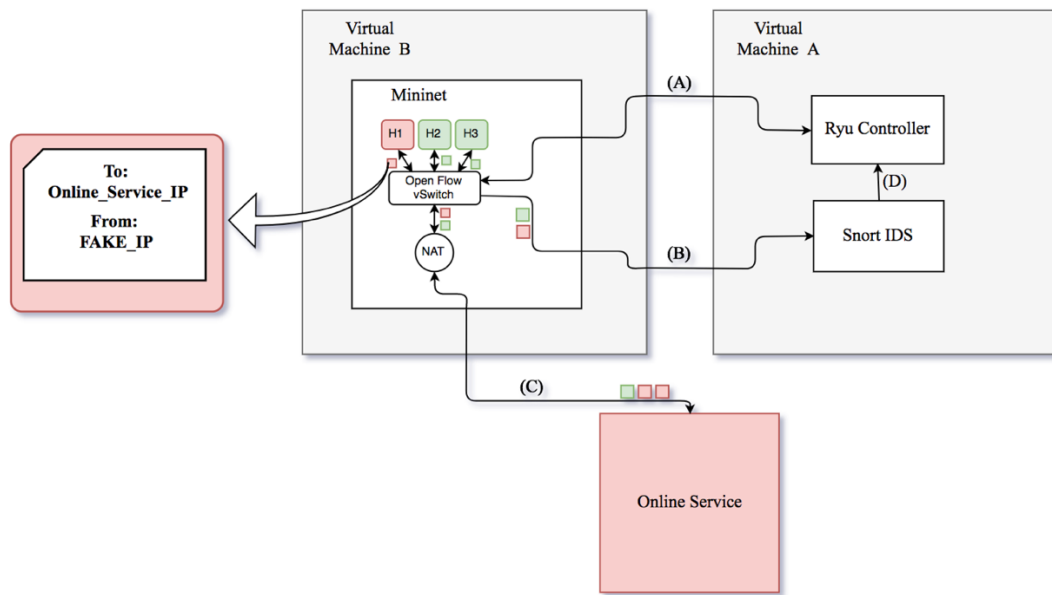


Figure 25 – Scenario II Test Diagram

4.1.3. Description of Scenario III Test

In this scenario, we test our system in a scenario where we produce a DoS attack where the packets sent by the infected device will have a size bigger than the usual (Figure 26). Host 1 will play the malicious device roll and hosts 2 and 3 are the benign hosts in this scenario. This way, we are able to evaluate the detection and mitigation process, the impact that the DoS attack have on their normal access to the online service and the versatility of our system. Once more, the attack is launched using hping3 tool in ICMP mode, flooding packets as fast as possible with a data size of 900 bytes.

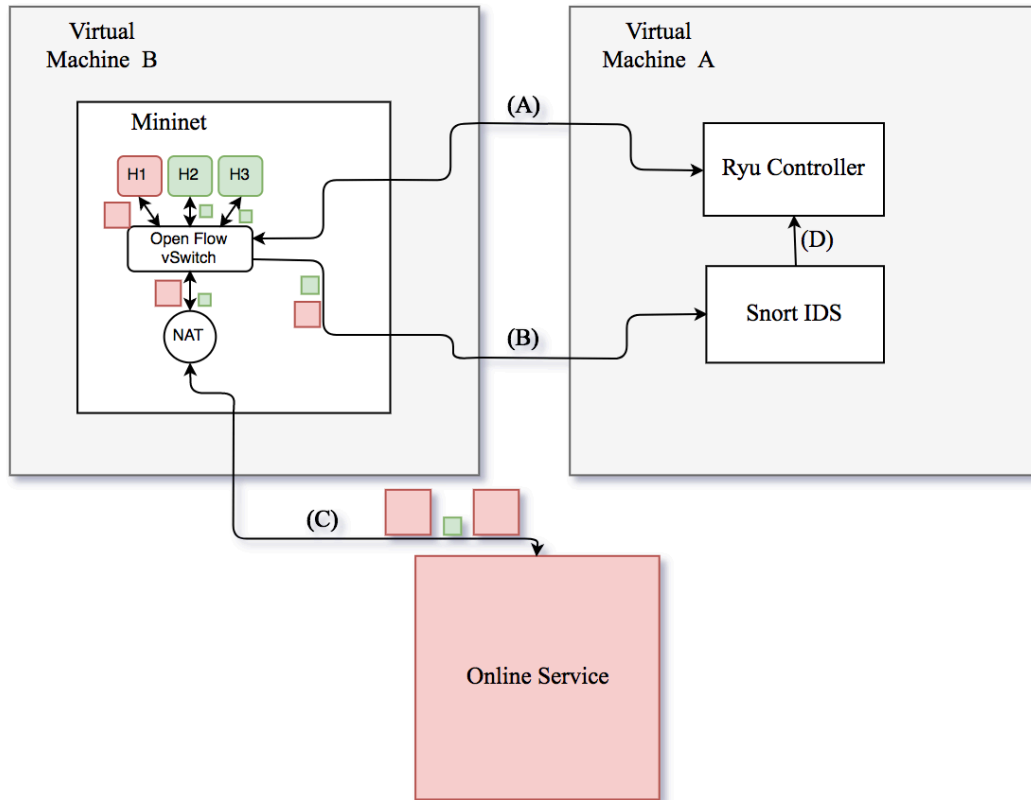


Figure 26 – Scenario III Test Diagram

A summary of the tests presented above is presented in Table 11 and 12, which contains a description, the parameters that were evaluated and the arguments used with hping3 tool for each scenario.

Table 11 – Summary of Tests Descriptions Part 1 of 2

	Description	Parameters evaluated	Hping3 arguments
Scenario I	A DDoS attack is simulated with two malign hosts, while a benign host has its benign access to the online service.	DDoS Mitigation Time; Average RTT; Packet loss;	<i>--flood</i>

Table 12 – Summary of Tests Descriptions Part 2 of 2

	Description	Parameters evaluated	Hping3 arguments
Scenario II	A DDoS attack with a spoofed IP address is simulated with one malign host while two benign hosts have their benign access to the online service.	DDoS Mitigation Time; Average RTT; Packet loss;	<i>-a, --flood</i>
Scenario III	A DDoS attack with packets strangely big is simulated with one malign host while two benign hosts have their benign access to the online service.	DDoS Mitigation Time; Average RTT; Packet loss;	<i>-d, --flood</i>

4.2. Tests Results Presentation

In this section we will present the results previous to our tests descriptions, in section 4.1. Before that, we present the state of the element's configurations before the attacks start. This will be useful to have a better understanding of the point we are at and compare it with post attack configurations. Plus, these configurations are general to all the scenario tests and work as a starting reference.

Initially, when zero packets have been exchanged in the network, the switch flow rule table only has the rule that indicates to forward the packets to the controller (Figure 27).

```
root@mininet-VirtualBox:~/mininet/examples# sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=30.117s, table=0, n_packets=72, n_bytes=8577, idle_age=0,
 priority=0 actions=CONTROLLER:65535
root@mininet-VirtualBox:~/mininet/examples#
```

Figure 27 – OpenFlow Switch Dump Flows Before Any Packets Exchange

Then, after packets have been exchanged inside the network (with Mininet command - *pingall*) the switch updates its flow rule table as shown in Figure 28. We can verify that new rules were updated with higher priority (*priority=1*). Hence, the switch has bound together the MAC address of each host with its ports. It is configured to do port forwarding (e.g. “*actions=output:1*” – forward to port 1). We can associate this behavior to our *packet_in_handler* function discussed in the SDN controller subsection (subsection 3.3.2).

```

NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=76.891s, table=0, n_packets=3, n_bytes=238, idle_age=71,
  priority=1,in_port=2,dl_dst=b2:2a:30:3a:e7:f2 actions=output:1
  cookie=0x0, duration=76.889s, table=0, n_packets=2, n_bytes=140, idle_age=71,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=76.877s, table=0, n_packets=3, n_bytes=238, idle_age=71,
  priority=1,in_port=3,dl_dst=b2:2a:30:3a:e7:f2 actions=output:1
  cookie=0x0, duration=76.875s, table=0, n_packets=2, n_bytes=140, idle_age=71,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=76.870s, table=0, n_packets=3, n_bytes=238, idle_age=71,
  priority=1,in_port=4,dl_dst=b2:2a:30:3a:e7:f2 actions=output:1
  cookie=0x0, duration=76.867s, table=0, n_packets=2, n_bytes=140, idle_age=71,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=76.858s, table=0, n_packets=3, n_bytes=238, idle_age=71,
  priority=1,in_port=3,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=76.856s, table=0, n_packets=2, n_bytes=140, idle_age=71,
  priority=1,in_port=2,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=76.849s, table=0, n_packets=3, n_bytes=238, idle_age=71,
  priority=1,in_port=4,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=76.847s, table=0, n_packets=2, n_bytes=140, idle_age=71,
  priority=1,in_port=2,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=76.835s, table=0, n_packets=3, n_bytes=238, idle_age=71,
  priority=1,in_port=4,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=76.830s, table=0, n_packets=2, n_bytes=140, idle_age=71,
  priority=1,in_port=3,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=138.890s, table=0, n_packets=97, n_bytes=10017, idle_age=5, priority=0
  actions=CONTROLLER:65535

```

Figure 28 – OpenFlow Switch Dump Flows After Pingall Mininet’s Command Execution

Furthermore, we are expecting that the usual traffic follows a constant tendency when all hosts are well behaved as presented in Figure 29. In a function where the packets per second transmitted are in function of time, we can see that there are only small variations keeping a mean under 7 packets per second.

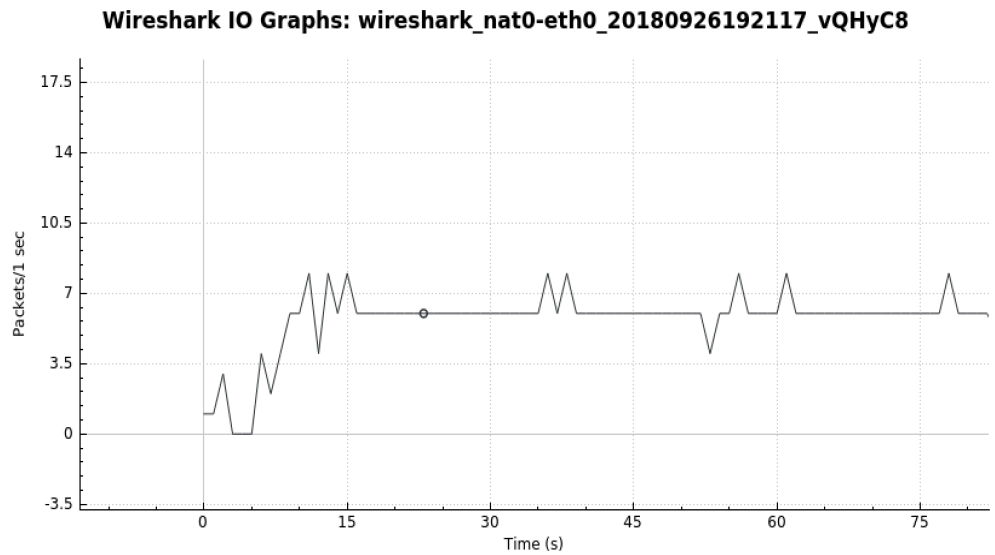


Figure 29 – Wireshark Usual System Usage

To have a numeric sensibility of the normal *Quality of Service* (QoS) of our system we present Table 13. This table will be useful to understand how much overhead our system adds and to give us a sense of the impact in QoS while our proposal is in action. These results were retrieved from the *ping* command in each host.

Table 13 – Normal Usage Latency

# tests	Host 1		Host 2		Host 3	
	Average RTT (ms)	Packet Loss (%)	Average RTT (ms)	Packet Loss (%)	Average RTT (ms)	Packet Loss (%)
1	0.588	0	0.439	0	0.765	0
2	0.756	0	1.274	0	0.712	0
3	0.493	0	0.463	0	0.593	0
4	0.766	0	0.904	0	1.37	0
5	2.004	0	0.402	0	0.447	0
Average	0.922	0	0.696	0	0.777	0

Figure 30 shows the output of the *top* command in the server machine when it is under DDoS attack. Give its output, we focus on variables as the load average, the percentage of CPU and memory for comparison and evaluation purposes. An average load of 0.59%, the 31.0% usage of the CPU and 18.5% of memory (18.5%) will be used as a reference for a DDoS attack.

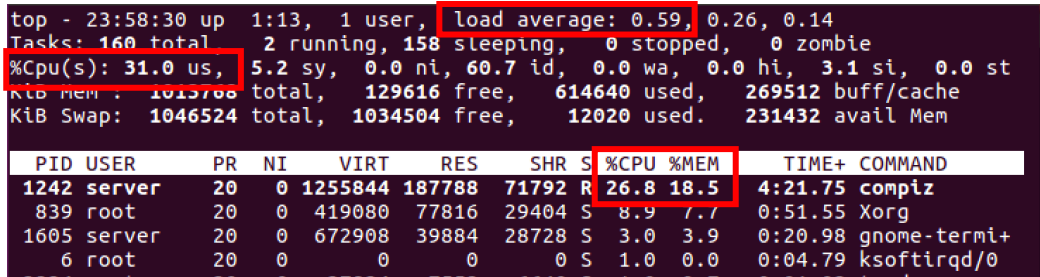


Figure 30 – Online Service Under DDoS Attack Top Command Execution

The graphical representation of what our proposal proposes to mitigate is presented in Figure 31. We can analyze, through the figure, that from the second 50 that the system was submitted to a high volume of traffic. In other words, a DDoS attack.

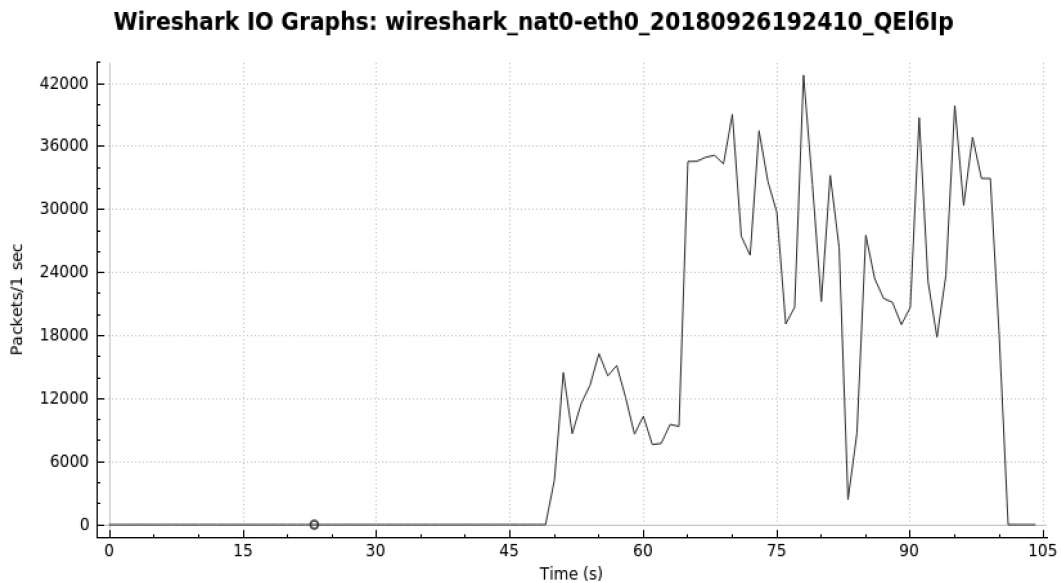
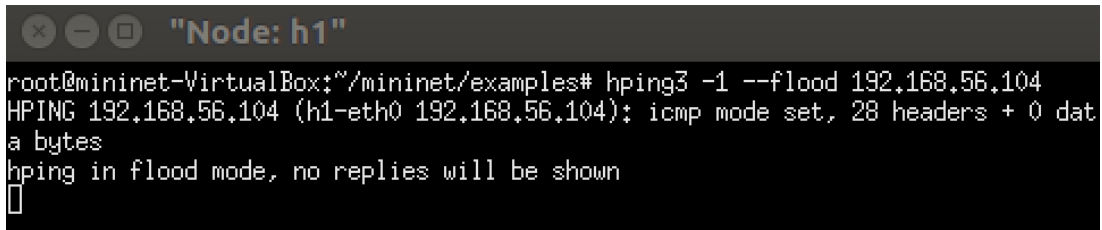


Figure 31 – Wireshark I/O Graph of DDoS Attack Without Our Defense System

4.2.1. Scenario I - Results Presentation

As described in the subsection 4.1.1, Figure 32 and Figure 33 presents the CLI of the malicious hosts (host 1 and 3) launching a DDoS attack. Therefore, the hping3 tool was used in ICMP mode, flooding packets as fast as possible to the server IP address (192.168.56.104).

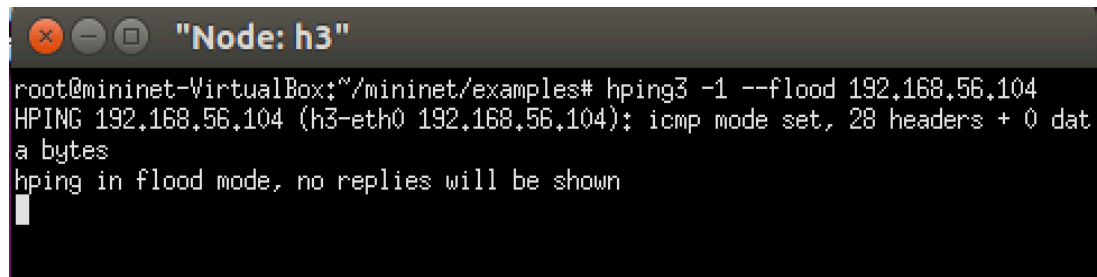


```

Node: h1
root@mininet-VirtualBox:~/mininet/examples# hping3 -1 --flood 192.168.56.104
HPING 192.168.56.104 (h1-eth0 192.168.56.104): icmp mode set, 28 headers + 0 data bytes
hping in flood mode, no replies will be shown

```

Figure 32 – Host 1 DDoS Attack Engagement



```

Node: h3
root@mininet-VirtualBox:~/mininet/examples# hping3 -1 --flood 192.168.56.104
HPING 192.168.56.104 (h3-eth0 192.168.56.104): icmp mode set, 28 headers + 0 data bytes
hping in flood mode, no replies will be shown

```

Figure 33 – Host 3 DDoS Attack Engagement

While hosts 1 and 3 launched the attack, the host 2 was simulating its normal access, as a benign host, to the server. Figure 34 shows the host 2 CLI and its normal usage of the online service through the ping command. We can verify that the ping succeeds with 10% packet loss. This implies that few packets will be lost while the attack is detected and mitigated. Once the attack is mitigated, the tendency of the ping command in host 2 is to go for 0% packet loss as we show next. In Figure 35 is presented an example of the transmission rate per second using the Wireshark I/O graph tool.

```

"Node: h2"
64 bytes from 192.168.56.104: icmp_seq=261 ttl=63 time=1.12 ms
64 bytes from 192.168.56.104: icmp_seq=262 ttl=63 time=0.539 ms
64 bytes from 192.168.56.104: icmp_seq=263 ttl=63 time=0.446 ms
64 bytes from 192.168.56.104: icmp_seq=264 ttl=63 time=0.673 ms
64 bytes from 192.168.56.104: icmp_seq=265 ttl=63 time=0.471 ms
64 bytes from 192.168.56.104: icmp_seq=266 ttl=63 time=0.864 ms
64 bytes from 192.168.56.104: icmp_seq=267 ttl=63 time=0.500 ms
64 bytes from 192.168.56.104: icmp_seq=268 ttl=63 time=0.481 ms
64 bytes from 192.168.56.104: icmp_seq=269 ttl=63 time=0.382 ms
64 bytes from 192.168.56.104: icmp_seq=270 ttl=63 time=0.805 ms
64 bytes from 192.168.56.104: icmp_seq=271 ttl=63 time=0.353 ms
64 bytes from 192.168.56.104: icmp_seq=272 ttl=63 time=0.474 ms
64 bytes from 192.168.56.104: icmp_seq=273 ttl=63 time=0.420 ms
64 bytes from 192.168.56.104: icmp_seq=274 ttl=63 time=0.450 ms
64 bytes from 192.168.56.104: icmp_seq=275 ttl=63 time=0.601 ms
64 bytes from 192.168.56.104: icmp_seq=276 ttl=63 time=0.718 ms
64 bytes from 192.168.56.104: icmp_seq=277 ttl=63 time=0.423 ms
64 bytes from 192.168.56.104: icmp_seq=278 ttl=63 time=0.417 ms
64 bytes from 192.168.56.104: icmp_seq=279 ttl=63 time=0.484 ms
^C
--- 192.168.56.104 ping statistics ---
279 packets transmitted, 249 received, 10% packet loss, time 284060ms
rtt min/avg/max/mdev = 0.218/2.263/21.135/26.602 ms
root@mininet-VirtualBox:~/mininet/examples#

```

Figure 34 – Host 2 Ping Command Output While Server is Under DDoS Attack

The host 2 usual access while the server is under attack is shown in Figure 35. We can see that it is not constant. These variations result from the server resource starvation and network congestion.

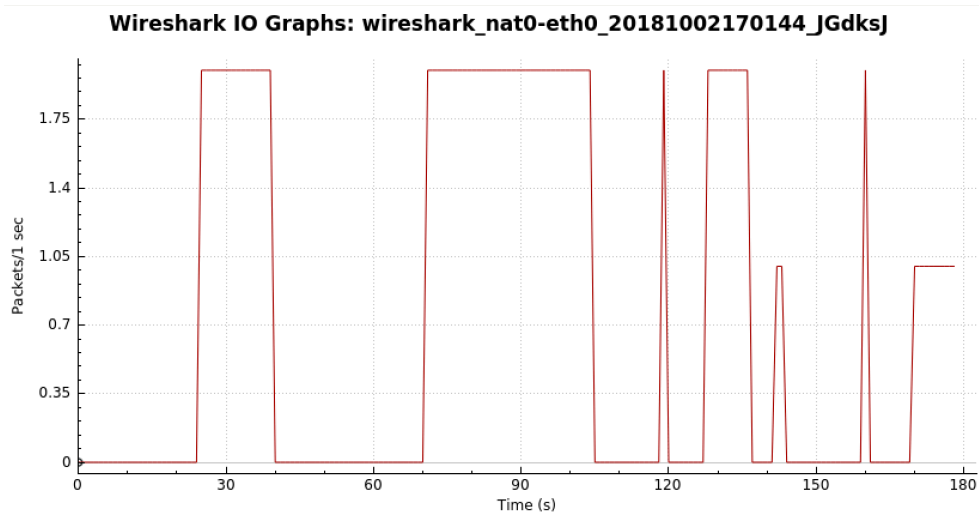


Figure 35 – Host 2 Wireshark I/O graph example While Server is Under DDoS Attack

We can visualize the detection phase succeeding through Figure 36 and Figure 37. These figures present the output generated by the controller when the attack is occurring. As we can see, the controller captured packets from the malign sources (host 1 and 3) through MAC and IP addresses ($src='00:00:00:00:00:01'$, $src='10.0.0.1'$, $src='10.0.0.03'$, $src='00:00:00:00:00:03'$). We can also notice the mitigation process

getting started through the controller's output: "*alertmsg: ryu block*" and the action that the controller will do – "*Blocking this flow:ryu block*".

```

ethernet(dst='52:ae:a8:84:08:21',ethertype=2048,src='00:00:00:00:00:01')
Blocking this flow:ryu block
received icmp packet
alertmsg: ryu block
icmp(code=0,csum=50156,data=echo(data=None,id=25105,seq=53761),type=8)
ipv4(csum=35755,dst='192.168.56.104',flags=0,header_length=5,identification=60452,offset=0,option=None,proto=1,src='10.0.0.1',tos=0,total_length=28,ttl=64,version=4)

```

Figure 36 – DDoS attack Host 1 Detection Controller's Output

```

ethernet(dst='3a:9d:41:af:36:84',ethertype=2048,src='00:00:00:00:00:03')
Blocking this flow:ryu block
received icmp packet
alertmsg: ryu block
icmp(code=0,csum=59081,data=echo(data=None,id=60681,seq=9260),type=8)
ipv4(csum=27667,dst='192.168.56.104',flags=0,header_length=5,identification=3003,offset=0,option=None,proto=1,src='10.0.0.3',tos=0,total_length=28,ttl=64,version=4)

```

Figure 37 – DDoS Attack Host 2 Detection Controller's Output

When an attack is detected it takes start the mitigation phase. One way of verifying if the attack is being mitigated as it should is through the switch flow rule table (Figure 38). We can see the action of controller sending blocking a flow to the switch taking shape, through the first 2 lines prominently displayed in the head of the file. It consists on rules with high priority to drop packets and for both malicious users, host 1 and host 3 (*priority=100, actions=drop, dl_src=00:00:00:00:00:01, dl_src=00:00:00:00:00:03, nw_src=10.0.0.1, nw_src=10.0.0.3* and *nw_dst=192.168.56.104*). Plus, as expected, there is not a rule added matching host 2, because host 2 did not break the rules specified in Snort being this way treated as a benign user.

```

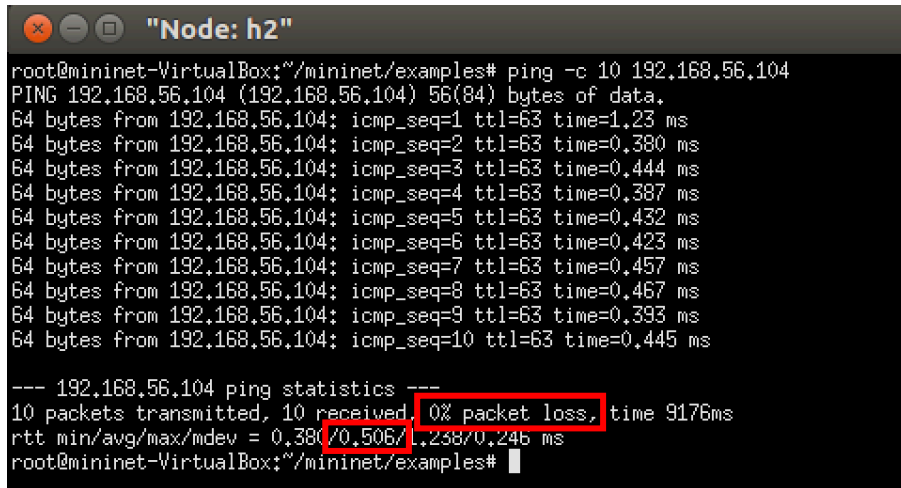
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=25.998s, table=0, n_packets=6370225, n_bytes=267549450, idle_age=105,
  priority=100,icmp,dl_src=00:00:00:00:00:01,nw_src=10.0.0.1,nw_dst=192.168.56.104 actions=drop
  cookie=0x0, duration=0.344s, table=0, n_packets=0, n_bytes=0, idle_age=25,
  priority=100,icmp,dl_src=00:00:00:00:00:03,nw_src=10.0.0.3,nw_dst=192.168.56.104 actions=drop
  cookie=0x0, duration=900.309s, table=0, n_packets=6, n_bytes=364, idle_age=126,
  priority=1,in_port=2,dl_dst=92:04:e3:01:b8:b4 actions=output:1
  cookie=0x0, duration=900.307s, table=0, n_packets=3344, n_bytes=140504, idle_age=126,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=900.303s, table=0, n_packets=3, n_bytes=238, idle_age=895,
  priority=1,in_port=3,dl_dst=92:64:e9:61:b8:b4 actions=output:1
  cookie=0x0, duration=900.301s, table=0, n_packets=2, n_bytes=140, idle_age=895,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=900.294s, table=0, n_packets=20232, n_bytes=849856, idle_age=118,
  priority=1,in_port=4,dl_dst=92:64:e9:61:b8:b4 actions=output:1
  cookie=0x0, duration=900.292s, table=0, n_packets=20231, n_bytes=849758, idle_age=118,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=900.286s, table=0, n_packets=3, n_bytes=238, idle_age=895,
  priority=1,in_port=3,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=900.284s, table=0, n_packets=2, n_bytes=140, idle_age=895,
  priority=1,in_port=2,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=900.277s, table=0, n_packets=3, n_bytes=238, idle_age=895,
  priority=1,in_port=4,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=900.276s, table=0, n_packets=2, n_bytes=140, idle_age=895,
  priority=1,in_port=2,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=900.268s, table=0, n_packets=3, n_bytes=238, idle_age=895,
  priority=1,in_port=4,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=900.266s, table=0, n_packets=2, n_bytes=140, idle_age=895,
  priority=1,in_port=3,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=902.955s, table=0, n_packets=209, n_bytes=28917, idle_age=17, priority=0
  actions=CONTROLLER:65535

```

Figure 38 – OpenFlow Switch's Flow Table with Mitigation Rules

The rule used was the same as the presented in the subsection 3.3.3 (Snort section, header *I*). It works as a detection trigger and classifies that anyone from the home network (i.e. *10.0.0.0/8*) that sends more than 10 packets per second to the server IP address (*192.168.56.104*) as a DoS attack.

After the DDoS attack mitigation process host 2 have its usual access to the server with the normal QoS (Figure 39). Looking to the values of *avg RTT* (0.506 ms) and to the percentage of packet loss (0%) we can conclude that might exist a decrease on QoS while the server is under attack.



```

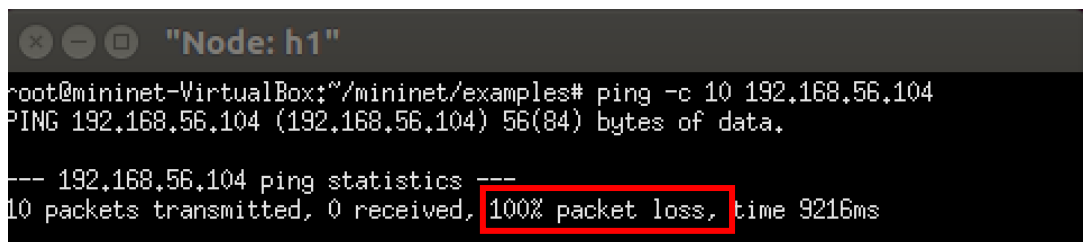
root@mininet-VirtualBox:~/mininet/examples# ping -c 10 192.168.56.104
PING 192.168.56.104 (192.168.56.104) 56(84) bytes of data:
64 bytes from 192.168.56.104: icmp_seq=1 ttl=63 time=1.23 ms
64 bytes from 192.168.56.104: icmp_seq=2 ttl=63 time=0.380 ms
64 bytes from 192.168.56.104: icmp_seq=3 ttl=63 time=0.444 ms
64 bytes from 192.168.56.104: icmp_seq=4 ttl=63 time=0.387 ms
64 bytes from 192.168.56.104: icmp_seq=5 ttl=63 time=0.432 ms
64 bytes from 192.168.56.104: icmp_seq=6 ttl=63 time=0.423 ms
64 bytes from 192.168.56.104: icmp_seq=7 ttl=63 time=0.457 ms
64 bytes from 192.168.56.104: icmp_seq=8 ttl=63 time=0.467 ms
64 bytes from 192.168.56.104: icmp_seq=9 ttl=63 time=0.393 ms
64 bytes from 192.168.56.104: icmp_seq=10 ttl=63 time=0.445 ms

--- 192.168.56.104 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9176ms
rtt min/avg/max/mdev = 0.380/0.506/1.238/0.246 ms
root@mininet-VirtualBox:~/mininet/examples#

```

Figure 39 – Host 2 Successful Ping After DDoS Attack

In contrast, after the DDoS mitigation, both malicious hosts have lost their access to the online service with a rate of packet loss of 100% (Figure 40 and Figure 41).



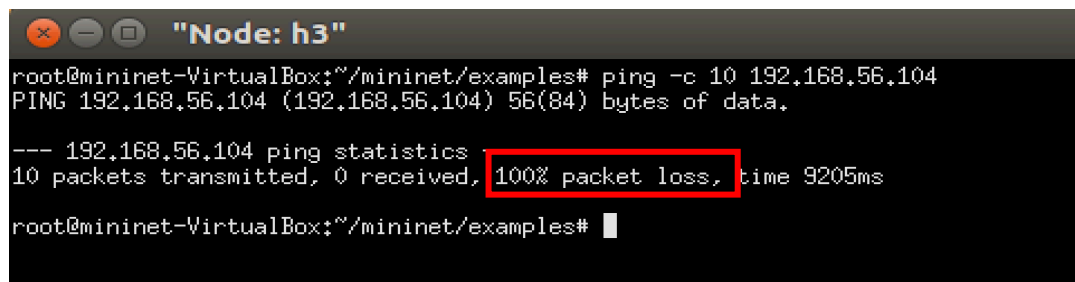
```

root@mininet-VirtualBox:~/mininet/examples# ping -c 10 192.168.56.104
PING 192.168.56.104 (192.168.56.104) 56(84) bytes of data.

--- 192.168.56.104 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9216ms

```

Figure 40 – Host 1 Unsuccessful Ping After DDoS Attack Mitigation



```

root@mininet-VirtualBox:~/mininet/examples# ping -c 10 192.168.56.104
PING 192.168.56.104 (192.168.56.104) 56(84) bytes of data.

--- 192.168.56.104 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9205ms
root@mininet-VirtualBox:~/mininet/examples#

```

Figure 41 – Host 3 Unsuccessful Ping After DDoS Attack Mitigation

Although our proposal achieves the DDoS attack mitigation, it is taking too long. In Figure 42 we can see the traffic generated as a function of time. From the second 0 to second 44 is represented the usual usage of the system. Then, host 1 launch the DoS attack

(peak of traffic). At almost same time, we can see host 3 launch another DoS. Host 1 is almost instantly blocked. From then on, we only see host 3 traffic. We verify that the host 3 DoS attack persists for 170 seconds. Finally, after 170 seconds, the host 3 DoS attack is mitigated and the system goes back to its “usual” traffic loading. The reason why it takes so long to mitigate host 3 attack is due to the controller overloading. Given that, during the time of processing all packets of host 1 and start to processing host 3 packets there is a window of 170 seconds.

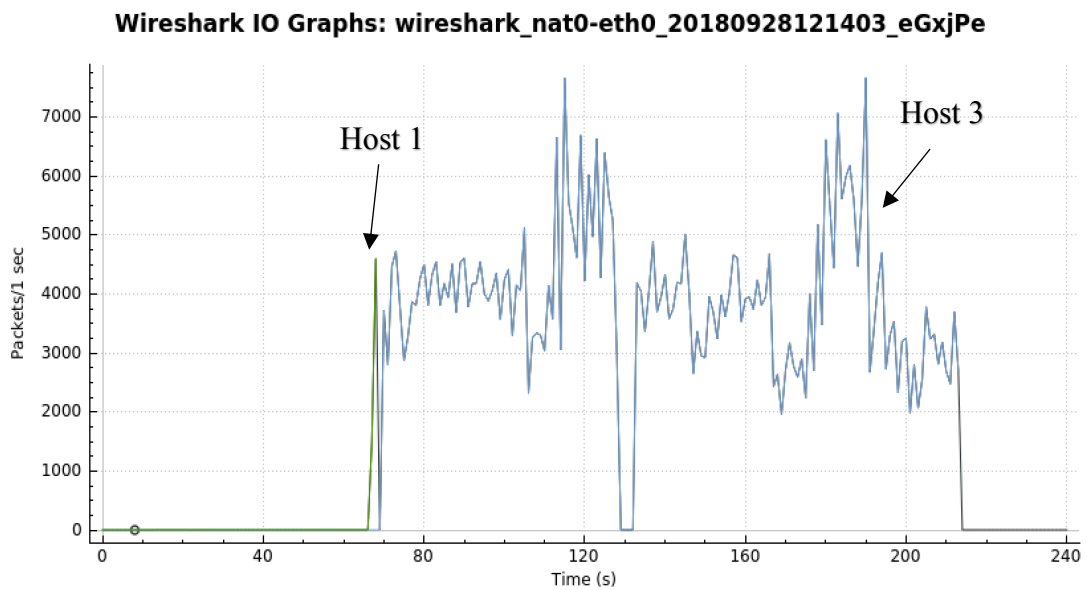


Figure 42 – DDoS Attack and Mitigation

In Figure 43 we have the output of *top* command in the server. Comparing to Figure 30 we can notice that the rate of usage of CPU (26.8% to 0.3%), memory (18.5% to 7.5%) has decreased and load average (0.59 to 0.09).

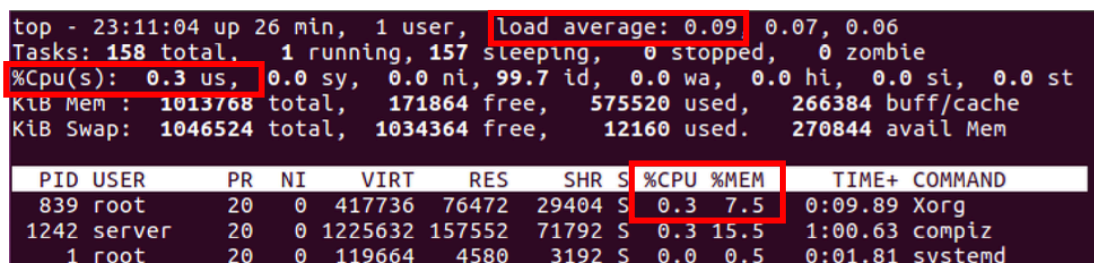


Figure 43 – Server *top* Command after DDoS attack Mitigation

Table 14 shows our results to the variables: DDoS mitigation time, the average *Round Trip Time (RTT)* and the percentage of the packet loss. The results of the *average RTT* and the *percentage of packet loss* were retrieved from the host 2 *ping* command while the server was under attack. The DDoS mitigation time was retrieved from the Wireshark I/O graph.

Table 14 – Scenario I Results

# tests	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
1	170	2.269	10
2	138	0.624	0
3	254	8.375	13
4	260	1.763	6
5	147	0.53	0
Average	193.8	2.712	5.8

a. System Performance - Enhanced Version 1

On this version, besides a slight improvement on the performance, our proposal is still taking too much time for DDoS mitigation. In Figure 44 is represented a graphic of the whole process, i.e. the DDoS engagement and mitigation. From the second 0 to the second 50 is represented the usual usage of the system. Then, host 1 launch the DoS attack (first peak of packets transmitted per second). At almost same time, we can see host 3 (second longer peak of packets transmitted per second) launch another DoS. Host 1 is almost instantly blocked. From then on, we only see host 3 traffic. We verify that the host 3 DoS attack persists for 50 seconds. Finally, after 50 seconds, the host 3 DoS attack is mitigated and the system goes back to its “usual” traffic loading. The reason why it takes so long to mitigate host 3 attack is due to the controller overloading.

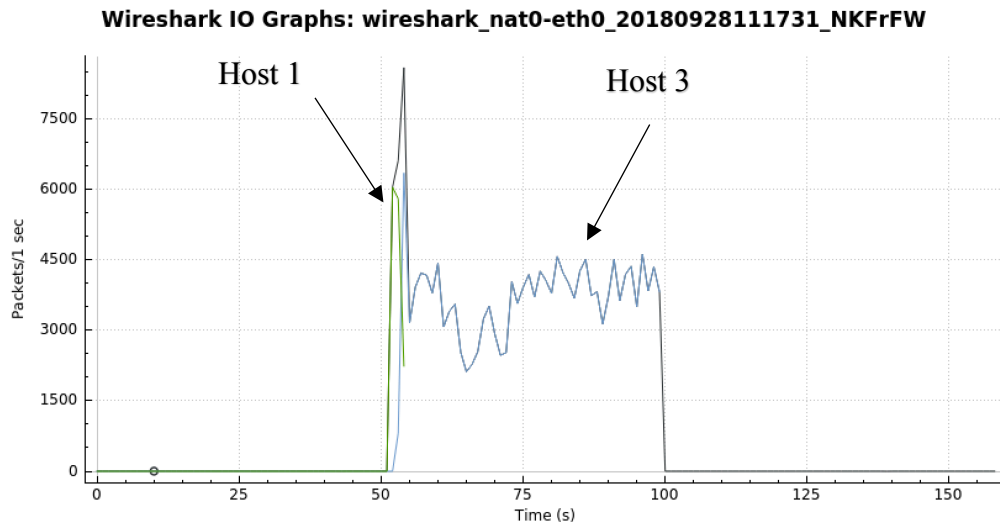


Figure 44 – Graphic of DDoS Attack Mitigation with System Version 1 Performance Enhanced

Table 15 shows our results to the variables: DDoS mitigation time, the average *Round Trip Time (RTT)* and the percentage of the packet loss. The results of the average RTT and the percentage of packet loss were retrieved from the host 2 ping command. The DDoS mitigation time was retrieved from the Wireshark I/O graph.

Table 15 – Scenario I Performance Enhanced Version 1 Results

# tests	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
1	50	0.596	0
2	19	0.651	0
3	29	0.647	0
4	24	0.633	0
5	50	0.683	0
Average	34.4	0.642	0

b. System Performance - Enhanced Version 2

In this solution, in order to avoid the overloading of the controller, we aimed to blocked the malicious mirrored traffic once one an attack was detected. Therefore, we configured Snort to limit the logging events for 60 seconds (Figure 45). This way, the controller will only process one alert per 60 seconds at time. This configuration was set up in a filed named *threshold.conf*.

```
# Global Threshold - Limit to logging 1 event per 60 seconds per IP triggering
# any alert for any event generator:
#
event_filter gen_id 0, sig_id 0, type limit, track by_src, count 1, seconds 60
```

Figure 45 – Snort Extra Configuration

In this version, we achieved the “graphical view” wished. In Figure 46 we can see the traffic generated as a function of time. From the second 0 to the second 60 is represented the usual usage of the system. Then, host 3 launch a DoS attack (first peak of packets transmitted per second). Next, host 2 launch another DoS attack (second peak of packets transmitted per second), simulating the DDoS attack. The DDoS attack was mitigated nearly at the second 64. Hence, from then on, we can verify the system going back to its “usual” traffic loading representation.

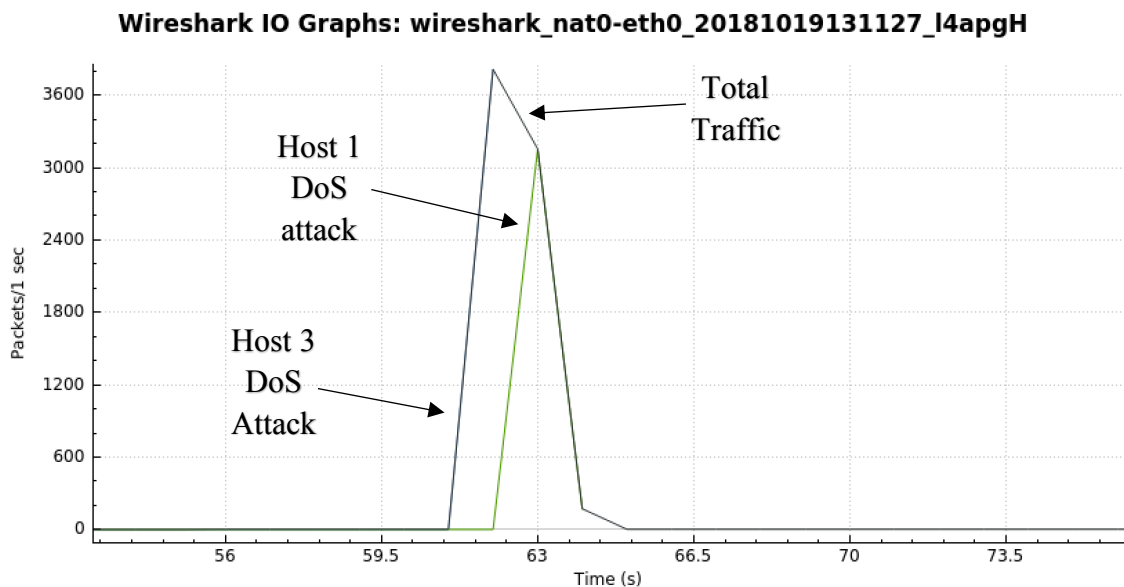


Figure 46 – DDoS Attack and Mitigation Performance Enhanced Version 2

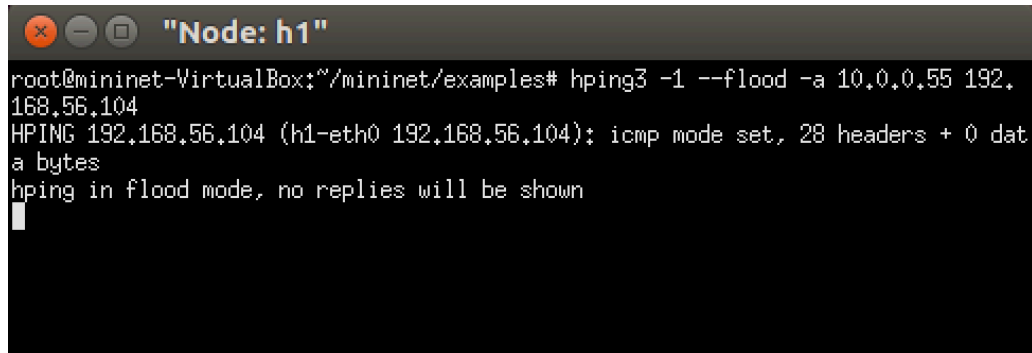
Table 16 shows our results to the variables: DDoS mitigation time, the average *Round Trip Time (RTT)* and the percentage of the packet loss. The results of the average RTT and the percentage of packet loss were retrieved from the *ping* command of host 2. The DDoS mitigation time was retrieved from the Wireshark I/O graph.

Table 16 – Scenario I Performance Enhanced Version 2 Results

# tests	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
1	3	0.619	0
2	4	0.547	0
3	5	0.561	0
4	5	0.484	0
5	7	0.767	0
Average	4.8	0.596	0

4.2.2. Scenario II - Results Presentation

As described in the subsection 4.1.2, in Figure 47 is presented the CLI of the malicious host launching a DDoS attack. In this scenario, the malicious host is initiating a DDoS spoofed IP attack. Therefore, the hping3 tool was used in ICMP mode, flooding packets as fast as possible to the server IP address (*192.168.56.104*) with a spoof IP address *10.0.0.55*.



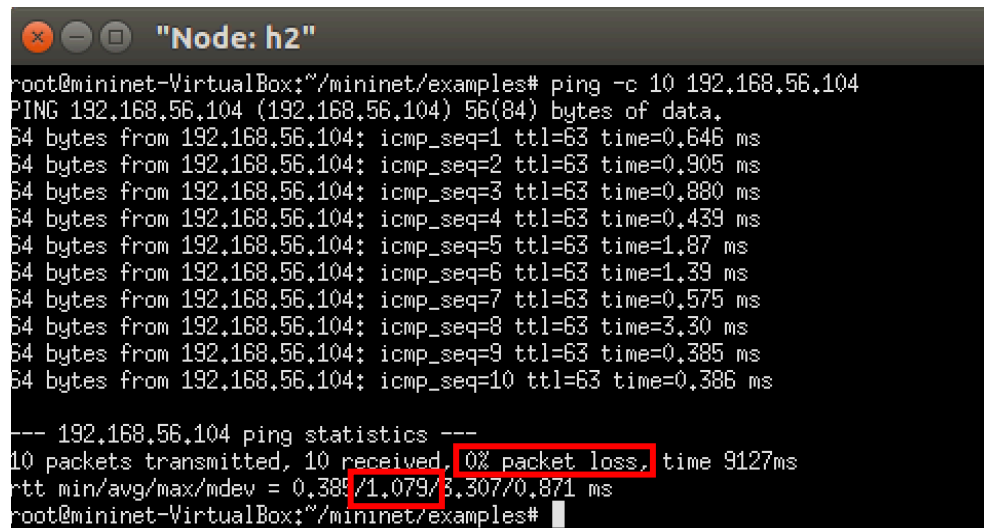
```

root@mininet-VirtualBox:~/mininet/examples# hping3 -1 --flood -a 10.0.0.55 192.168.56.104
HPING 192.168.56.104 (h1-eth0 192.168.56.104): icmp mode set, 28 headers + 0 data bytes
hping in flood mode, no replies will be shown

```

Figure 47 – Host 1 DDoS Engagement

While the server was under attack host 2 was able to have its normal access to it as illustrates Figure 48. The values of *QoS* measurement were retrieved from this action (*average RTT* and *percentage of packet loss*) while the system is under attack.



```

root@mininet-VirtualBox:~/mininet/examples# ping -c 10 192.168.56.104
PING 192.168.56.104 (192.168.56.104) 56(84) bytes of data:
64 bytes from 192.168.56.104: icmp_seq=1 ttl=63 time=0.646 ms
64 bytes from 192.168.56.104: icmp_seq=2 ttl=63 time=0.905 ms
64 bytes from 192.168.56.104: icmp_seq=3 ttl=63 time=0.880 ms
64 bytes from 192.168.56.104: icmp_seq=4 ttl=63 time=0.439 ms
64 bytes from 192.168.56.104: icmp_seq=5 ttl=63 time=1.87 ms
64 bytes from 192.168.56.104: icmp_seq=6 ttl=63 time=1.39 ms
64 bytes from 192.168.56.104: icmp_seq=7 ttl=63 time=0.575 ms
64 bytes from 192.168.56.104: icmp_seq=8 ttl=63 time=3.30 ms
64 bytes from 192.168.56.104: icmp_seq=9 ttl=63 time=0.385 ms
64 bytes from 192.168.56.104: icmp_seq=10 ttl=63 time=0.386 ms

--- 192.168.56.104 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9127ms
rtt min/avg/max/mdev = 0.385/1.079/3.307/0.871 ms
root@mininet-VirtualBox:~/mininet/examples#

```

Figure 48 – Host 2 Ping Output while DDoS Attack

The same procedure was realized with host 3. To retrieve the values of *QoS* (*RTT* average and percentage of packet loss) we used *ping* command (Figure 49), while the system was under attack.

```

"Node: h3"
root@mininet-VirtualBox:~/mininet/examples# ping -c 10 192.168.56.104
PING 192.168.56.104 (192.168.56.104) 56(84) bytes of data:
64 bytes from 192.168.56.104: icmp_seq=1 ttl=63 time=0.529 ms
64 bytes from 192.168.56.104: icmp_seq=2 ttl=63 time=0.371 ms
64 bytes from 192.168.56.104: icmp_seq=3 ttl=63 time=0.430 ms
64 bytes from 192.168.56.104: icmp_seq=4 ttl=63 time=0.726 ms
64 bytes from 192.168.56.104: icmp_seq=5 ttl=63 time=1.58 ms
64 bytes from 192.168.56.104: icmp_seq=6 ttl=63 time=0.540 ms
64 bytes from 192.168.56.104: icmp_seq=7 ttl=63 time=0.406 ms
64 bytes from 192.168.56.104: icmp_seq=8 ttl=63 time=0.335 ms
64 bytes from 192.168.56.104: icmp_seq=9 ttl=63 time=0.526 ms
64 bytes from 192.168.56.104: icmp_seq=10 ttl=63 time=1.32 ms

--- 192.168.56.104 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9192ms
rtt min/avg/max/mdev = 0.335/0.676/1.583/0.407 ms
root@mininet-VirtualBox:~/mininet/examples#

```

Figure 49 – Host 3 Ping Output while DDoS Attack

To give a better perception of the impact on the QoS in the hosts 2 and 3, while the system is under attack, we present Figure 50. Even though we simulated the attack with only one host, we can verify that the host's debits remained constant.

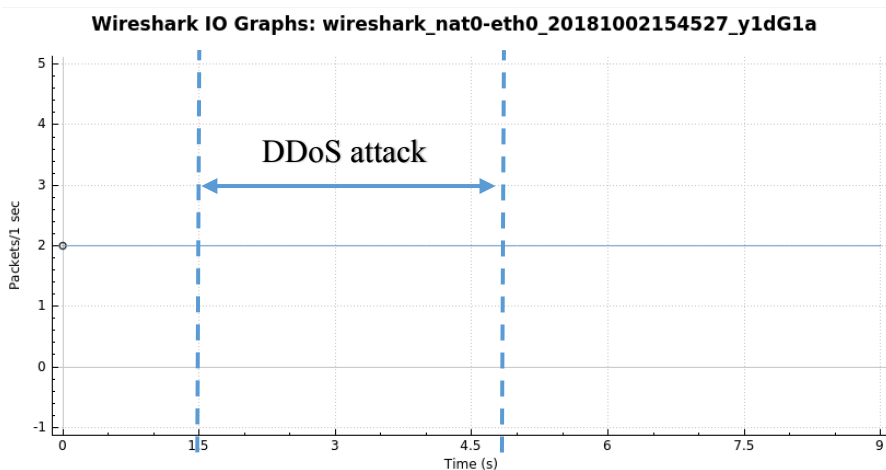


Figure 50 – Host 2 and 3 Wireshark I/O Graph while DDoS Attack

After the attack started, the system detects it and gives start to the mitigation phase. The following example (Figure 51) displays the output that is generated by the controller when the attack is detected. Then, the controller will start the mitigation phase for the malign host ($src='00:00:00:00:00:01'$, $src='10.0.0.55'$).

```

ethernet(dst='7a:02:17:a7:a9:b2',ethertype=2048,src='00:00:00:00:00:01')
Blocking this flow:ryu block
received icmp packet
alertmsg: ryu block
icmp(code=0,csum=21470,data=echo(data=None,id=9490,seq=32527),type=8)
ipv4(csum=35523,dst='192.168.56.104',flags=0,header_length=5,identification=6063
0,offset=0,option=None,proto=1,src='10.0.0.55',tos=0,total_length=28,ttl=64,vers
ion=4)

```

Figure 51 – DDoS Attack Host 1 Detection Controller's Output

We can confirm that the mitigation process is taking over by checking the switch's flow rule table (Figure 52). We can verify that controller mitigated the attack through the first 2 lines of the file. It consists on a new rule with higher priority to drop packets for the malign host (*priority=100, actions=drop, dl_src=00:00:00:00:00:01, nw_src=10.0.0.55* and *nw_dst=192.168.56.104*).

```

NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=0.383s, table=0, n_packets=1402250, n_bytes=58894500, idle_age=294,
  priority=100,icmp,dl_src=00:00:00:00:00:01,nw_src=10.0.0.55,nw_dst=192.168.56.104 actions=drop
  cookie=0x0, duration=385.724s, table=0, n_packets=38013, n_bytes=1625522, idle_age=91,
  priority=1,in_port=2,dl_dst=7a:02:17:a7:a9:b2 actions=output:1
  cookie=0x0, duration=385.732s, table=0, n_packets=24, n_bytes=2128, idle_age=91,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=385.727s, table=0, n_packets=3, n_bytes=238, idle_age=380,
  priority=1,in_port=3,dl_dst=7a:02:17:a7:a9:b2 actions=output:1
  cookie=0x0, duration=385.725s, table=0, n_packets=2, n_bytes=140, idle_age=380,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=385.713s, table=0, n_packets=3, n_bytes=238, idle_age=380,
  priority=1,in_port=4,dl_dst=7a:02:17:a7:a9:b2 actions=output:1
  cookie=0x0, duration=385.711s, table=0, n_packets=2, n_bytes=140, idle_age=380,
  priority=1,in_port=1,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=385.701s, table=0, n_packets=3, n_bytes=238, idle_age=380,
  priority=1,in_port=3,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=385.699s, table=0, n_packets=3, n_bytes=238, idle_age=380,
  priority=1,in_port=2,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=385.688s, table=0, n_packets=3, n_bytes=238, idle_age=380,
  priority=1,in_port=4,dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=385.680s, table=0, n_packets=2, n_bytes=140, idle_age=380,
  priority=1,in_port=2,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=385.653s, table=0, n_packets=3, n_bytes=238, idle_age=380,
  priority=1,in_port=4,dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=385.651s, table=0, n_packets=2, n_bytes=140, idle_age=380,
  priority=1,in_port=3,dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=391.873s, table=0, n_packets=143, n_bytes=15399, idle_age=10, priority=0
  actions=CONTROLLER:65535

```

Figure 52 – OpenFlow Switch's Flow Table with Mitigation Rules

The rules used in this test were the same as the test before (subsection 3.3.3 - Snort) plus the rule specified in *I*). The detection filter is the same but for all IP addresses that are not in 10.0.0.0/8 network. This way, a wider range of IP addresses is under “surveillance”.

*I) alert icmp !10.0.0.0/8 any → 192.168.56.104 any (msg:"ryu block";
detection_filter:track by_src, count 10, seconds 1, sid:1000002)*

A graphical visualization of the whole process (i.e. detection and mitigation phase) is presented in the Figure 53. As we can see, it illustrates the traffic generated as a function of time. From the second 0 to 2 seconds is represented the usual usage of the system. Then, host 3 launched a DDoS attack. This attack was blocked after 2 seconds. From second 4 to the end we see the traffic stabilizing and going back to the “usual” traffic loading.

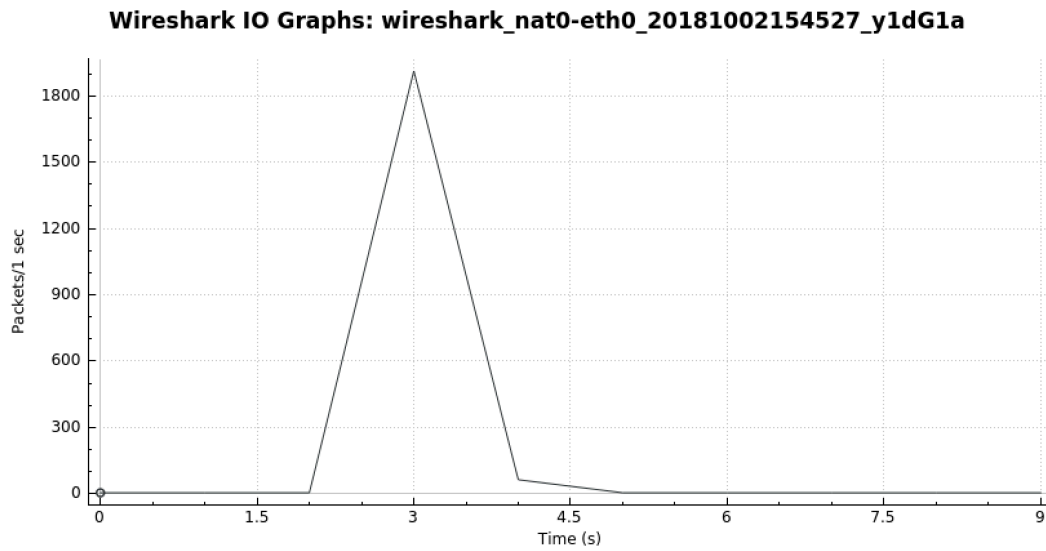


Figure 53 – Wireshark I/O Graph of DDoS Mitigation

Table 17 and 18 shows our results to the variables: DDoS mitigation time, the average *Round Trip Time (RTT)* and the percentage of the packet loss. The results of the average RTT and the percentage of packet loss were retrieved from the host 2 ping command. The DDoS mitigation time was retrieved from the Wireshark I/O graph.

Table 17 – Scenario II Performance Results Part 1 of 2

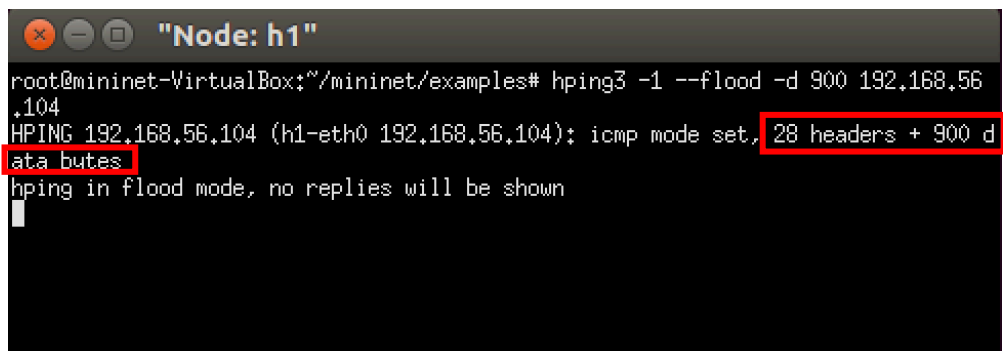
# tests	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
1	3	0.504	0
2	2	0.460	0

Table 18 – Scenario II Performance Results Part 2 of 2

# tests	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
3	2	0.557	0
4	2	0.513	0
5	2	0.523	0
Average	2.2	0.511	0

4.2.3. Scenario III - Results Presentation

In Figure 54 is presented malicious host's CLI launching a DDoS attack. Therefore, the hping3 tool was used in ICMP mode, flooding packets as fast as possible to the server IP address (*192.168.56.104*) plus 900 data bytes. The choice of 900 extra data bytes was totally arbitrary. Our only restriction is to be above 800 bytes in order to be detectable by the Snort IDS.



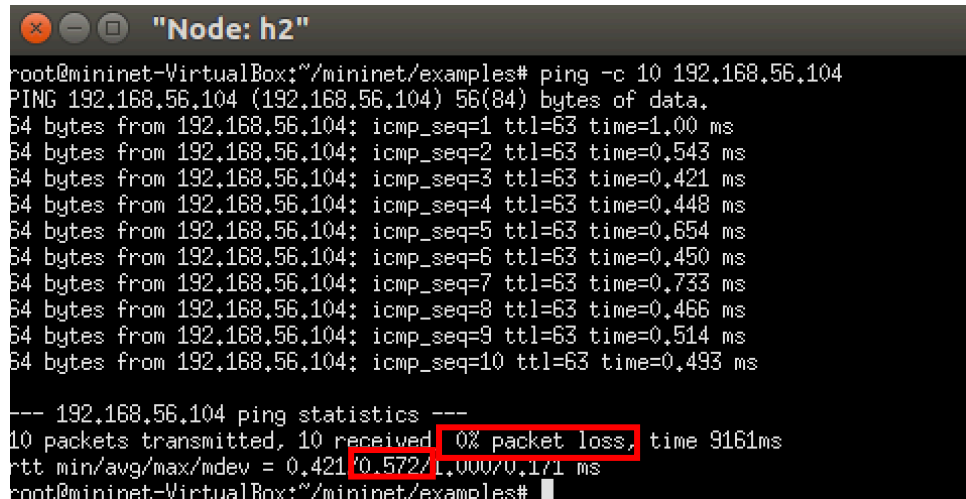
```

Node: h1
root@mininet-VirtualBox:~/mininet/examples# hping3 -1 --flood -d 900 192.168.56.104
HPING 192.168.56.104 (h1-eth0 192.168.56.104): icmp mode set, 28 headers + 900 data bytes
hping in flood mode, no replies will be shown

```

Figure 54 – Host 1 DDoS Engagement

As the scenario before, host 2 has its usual access to the server while the it is under a DDoS attack (Figure 55), with 0% of packet loss and an average RTT of 0.572 milliseconds.



```

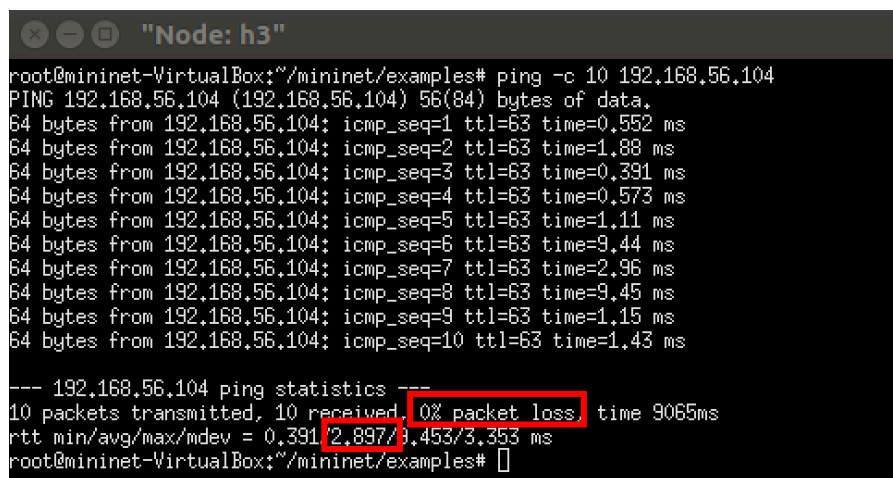
root@mininet-VirtualBox:~/mininet/examples# ping -c 10 192.168.56.104
PING 192.168.56.104 (192.168.56.104) 56(84) bytes of data:
64 bytes from 192.168.56.104: icmp_seq=1 ttl=63 time=1.00 ms
64 bytes from 192.168.56.104: icmp_seq=2 ttl=63 time=0.543 ms
64 bytes from 192.168.56.104: icmp_seq=3 ttl=63 time=0.421 ms
64 bytes from 192.168.56.104: icmp_seq=4 ttl=63 time=0.448 ms
64 bytes from 192.168.56.104: icmp_seq=5 ttl=63 time=0.654 ms
64 bytes from 192.168.56.104: icmp_seq=6 ttl=63 time=0.450 ms
64 bytes from 192.168.56.104: icmp_seq=7 ttl=63 time=0.733 ms
64 bytes from 192.168.56.104: icmp_seq=8 ttl=63 time=0.466 ms
64 bytes from 192.168.56.104: icmp_seq=9 ttl=63 time=0.514 ms
64 bytes from 192.168.56.104: icmp_seq=10 ttl=63 time=0.493 ms

--- 192.168.56.104 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9161ms
rtt min/avg/max/mdev = 0.421/0.572/1.000/0.171 ms
root@mininet-VirtualBox:~/mininet/examples#

```

Figure 55 – Host 2 Ping Command Output while the Server is under DDoS Attack

In Figure 56 is illustrated the normal access of host 3 to the server. We can notice that it 0% of packet loss and an average RTT of 2.897 milliseconds.



```

root@mininet-VirtualBox:~/mininet/examples# ping -c 10 192.168.56.104
PING 192.168.56.104 (192.168.56.104) 56(84) bytes of data:
64 bytes from 192.168.56.104: icmp_seq=1 ttl=63 time=0.552 ms
64 bytes from 192.168.56.104: icmp_seq=2 ttl=63 time=1.88 ms
64 bytes from 192.168.56.104: icmp_seq=3 ttl=63 time=0.391 ms
64 bytes from 192.168.56.104: icmp_seq=4 ttl=63 time=0.573 ms
64 bytes from 192.168.56.104: icmp_seq=5 ttl=63 time=1.11 ms
64 bytes from 192.168.56.104: icmp_seq=6 ttl=63 time=9.44 ms
64 bytes from 192.168.56.104: icmp_seq=7 ttl=63 time=2.96 ms
64 bytes from 192.168.56.104: icmp_seq=8 ttl=63 time=9.45 ms
64 bytes from 192.168.56.104: icmp_seq=9 ttl=63 time=1.15 ms
64 bytes from 192.168.56.104: icmp_seq=10 ttl=63 time=1.43 ms

--- 192.168.56.104 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9065ms
rtt min/avg/max/mdev = 0.391/2.897/9.453/3.353 ms
root@mininet-VirtualBox:~/mininet/examples#

```

Figure 56 – Host 3 Ping Output while Server is under the DDoS Attack

The transmission rate of hosts 2 and 3 had no variations (Figure 57). We can see the debit of these hosts before and while the server is under a DDoS attack with a constant transmission rate during the time.

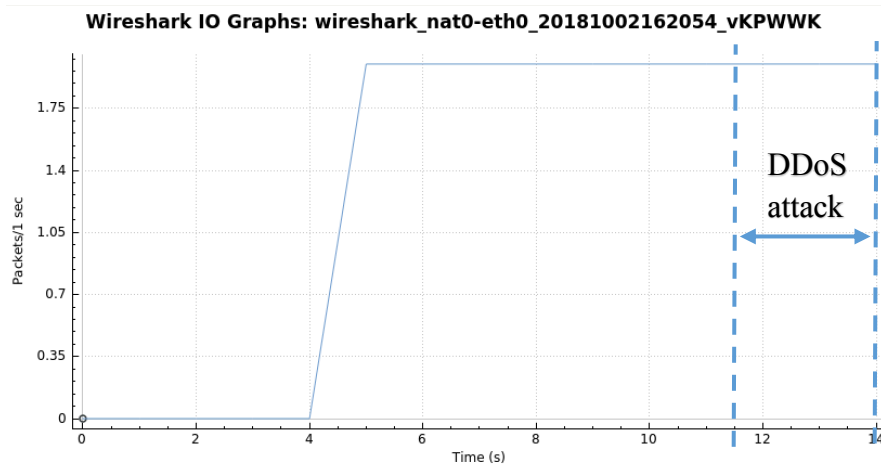


Figure 57 – Hosts 2 and 3 Wireshark I/O Graph while Server is under the DDoS Attack

Through the output of the controller (Figure 58), we can verify that the attack was detected with packet's total length of 928 bytes. (*src='00:00:00:00:00:01', src='10.0.0.1', total_length=928*).

[illegible]

Figure 58 – Controller's Output

Once more the attack was mitigated (Figure 59). Through the switch's flow rule table, we verify that the SDN controller mitigated successfully the attack through the first line of the file. It consists on a new rule with higher priority to drop packets for host 1 (*priority=100, actions=drop, dl_src=00:00:00:00:00:01, nw_src=10.0.0.1 and nw_dst=192.168.56.104*).

```

NXST FLOW reply (xid=0x4):
  cookie=0x0, duration=69.740s, table=0, n_packets=9049469, n_bytes=8524579542, idle_age=71,
  priority=100, icmp, dl_src=00:00:00:00:00:01, nw_src=10.0.0.1, nw_dst=192.168.56.104 actions=drop
  cookie=0x0, duration=268.081s, table=0, n_packets=19506, n_bytes=18362556, idle_age=79,
  priority=1, in_port=2, dl_dst=5a:2b:35:b6:01:2d actions=output:1
  cookie=0x0, duration=268.080s, table=0, n_packets=4512, n_bytes=4239052, idle_age=79,
  priority=1, in_port=1, dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=268.063s, table=0, n_packets=14, n_bytes=1260, idle_age=119,
  priority=1, in_port=3, dl_dst=5a:2b:35:b6:01:2d actions=output:1
  cookie=0x0, duration=268.061s, table=0, n_packets=13, n_bytes=1162, idle_age=119,
  priority=1, in_port=1, dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=268.055s, table=0, n_packets=14, n_bytes=1260, idle_age=133,
  priority=1, in_port=4, dl_dst=5a:2b:35:b6:01:2d actions=output:1
  cookie=0x0, duration=268.053s, table=0, n_packets=13, n_bytes=1162, idle_age=133,
  priority=1, in_port=1, dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=268.047s, table=0, n_packets=3, n_bytes=238, idle_age=262,
  priority=1, in_port=3, dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=268.045s, table=0, n_packets=2, n_bytes=140, idle_age=262,
  priority=1, in_port=2, dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=268.040s, table=0, n_packets=3, n_bytes=238, idle_age=262,
  priority=1, in_port=4, dl_dst=00:00:00:00:00:01 actions=output:2
  cookie=0x0, duration=268.037s, table=0, n_packets=2, n_bytes=140, idle_age=262,
  priority=1, in_port=2, dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=268.027s, table=0, n_packets=3, n_bytes=238, idle_age=262,
  priority=1, in_port=4, dl_dst=00:00:00:00:00:02 actions=output:3
  cookie=0x0, duration=268.022s, table=0, n_packets=2, n_bytes=140, idle_age=262,
  priority=1, in_port=3, dl_dst=00:00:00:00:00:03 actions=output:4
  cookie=0x0, duration=271.650s, table=0, n_packets=129, n_bytes=16280, idle_age=14, priority=0
  actions=CONTROLLER:65535

```

Figure 59 – OpenFlow Switch's Flow Table with Mitigation Rules

The rule used in this test is presented in **1)**. In this case, Snort will detect a DDoS attack if a packet that comes from the *10.0.0.0/8* network towards the server IP address has a size bigger than 800 bytes (*dsiz>800*).

1) *alert icmp 10.0.0.0/8 any → 192.168.56.104 any (msg:"ryu block"; dsiz>800; sid:1000003)*

A graphical representation of the system's load during the detection and mitigation phase (Figure 60). From the second 0 to 11 seconds is represented the usual usage of the system. Then, host 1 launch the DoS attack. This attack is mitigated and blocked nearly in 3 seconds. Therefore, from second 14 until to the end we see the traffic stabilizing and going back to the "usual" traffic loading.

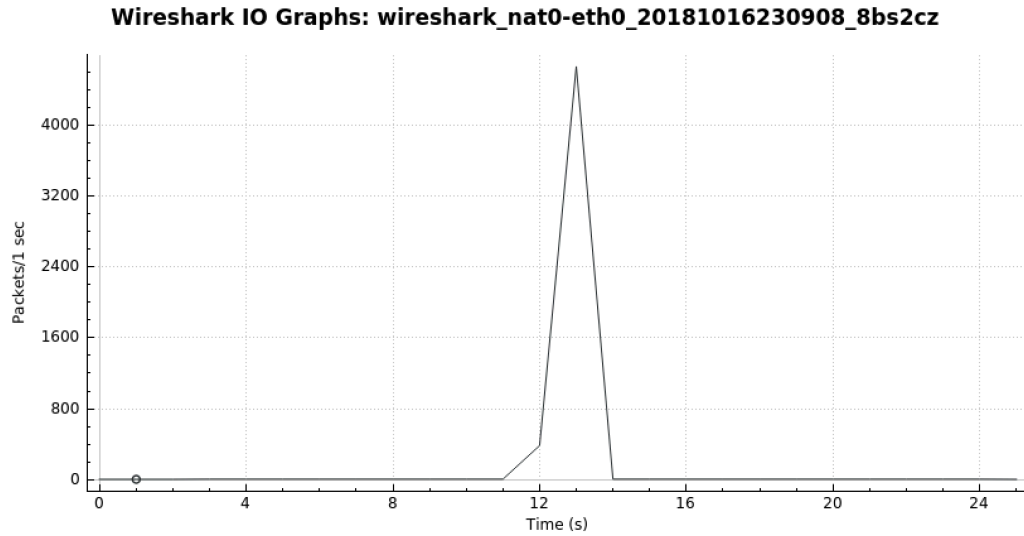


Figure 60 – Wireshark I/O Graph of DDoS Attack Mitigation

In Figure 61 is presented the output of *tcpdump* command on the server. We can verify that packets with length of 908 bytes stop being processed (i.e. packets from host 1 was blocked) and that the other packets (i.e. from hosts 2 and 3) with 64 bytes length, continue its normal request/reply process.

```
server@server-VirtualBox: ~
eq 20130, length 908
16:44:14.858361 IP 192.168.56.102 > 192.168.56.104: ICMP echo request, id 19209,
seq 20386, length 908
16:44:15.204381 IP 192.168.56.102 > 192.168.56.104: ICMP echo request, id 2381,
seq 1, length 64
16:44:15.204412 IP 192.168.56.104 > 192.168.56.102: ICMP echo reply, id 2381, se
q 1, length 64
16:44:16.205027 IP 192.168.56.102 > 192.168.56.104: ICMP echo request, id 2381,
seq 2, length 64
16:44:16.205061 IP 192.168.56.104 > 192.168.56.102: ICMP echo reply, id 2381, se
q 2, length 64
16:44:17.197172 IP 192.168.56.102 > 192.168.56.104: ICMP echo request, id 2382,
seq 1, length 64
16:44:17.197205 IP 192.168.56.104 > 192.168.56.102: ICMP echo reply, id 2382, se
q 1, length 64
16:44:17.212602 IP 192.168.56.102 > 192.168.56.104: ICMP echo request, id 2381,
seq 3, length 64
16:44:17.212630 IP 192.168.56.104 > 192.168.56.102: ICMP echo reply, id 2381, se
q 3, length 64
16:44:18.198533 IP 192.168.56.102 > 192.168.56.104: ICMP echo request, id 2382,
seq 2, length 64
16:44:18.198565 IP 192.168.56.104 > 192.168.56.102: ICMP echo reply, id 2382, se
q 2, length 64
16:44:18.265124 IP 192.168.56.102 > 192.168.56.104: ICMP echo request, id 2381,
```

Figure 61 – Server *tcpdump* Command Output

Table 19 shows our results to the variables: DDoS mitigation time, the average *Round Trip Time (RTT)* and the percentage of the packet loss. The results of the average RTT

and the percentage of packet loss were retrieved from the host 2 ping command. The DDoS mitigation time was retrieved from the Wireshark I/O graph.

Table 19 – Scenario III Results

# tests	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
1	3	0.674	0
2	2	0.480	0
3	2	0.430	0
4	2	0.449	0
5	2	0.547	0
Average	2.2	0.516	0

4.2.4. Scenario I - Discussion of Results

In this test, we have shown that both the detection and mitigation process succeeded. Table 20 presents the summary of results of the variables we purpose to evaluate in subsection 4.1.3 obtained. In the first Scenario I (the original), our system was able to detect and mitigate a DDoS attack. Although, the numbers were alarming. The effectiveness of our system mitigating DDoS attacks was measured with DDoS mitigation time variable (193.8 s). The *QoS* that our system provide for benign hosts was measured with both average RTT (2.712 ms) and the percentage of packet loss (5.8%) variables. These numbers gave us the motivation to seek for better. We have found that our controller was getting overloading due to the mirror configurations. Therefore, we advanced to the Scenario I enhanced version 1. In this version, we realized code optimizations, specified in the subsection 3.3.2.1. The effectiveness of our system in DDoS attacks mitigation process was measured with DDoS mitigation time variable (34.4 s). The *QoS* that our system provide for benign hosts was measured with both average RTT (0.642 ms) and the percentage of packet loss (0%) variables. The DDoS mitigation

time improved 82%, the average *RTT* 76% and the packet loss turned to 0%. Although, 34.4 seconds was just not good enough. Therefore, we seek for better. On Scenario I enhanced version 2, we decided to make an optimization in the Snort IDS. Being the controller overloading and the Snort IDS the only element between the network and the controller we look to limit the alerts generated by Snort to one. Detailed information about the configurations added for this optimization is present in the subsections 3.3.2.2 and 4.2.1 heading *b*). The DDoS mitigation time improved 86% and the average *RTT* 7%.

Table 20 – Scenario I Summary Results

Scenario I	DDoS Mitigation time (s)	AVG RTT (ms)	Packet Loss (%)
Original	193.8	2.712	5.80
Version 1	34.4	0.642	0
Version 2	4.80	0.596	0

4.2.5. Scenario II - Discussion of Results

In this scenario, we were able to study a DDoS attack with IP spoofing. We have showed that our system is able to detect and mitigate this kind of attacks. The results are expressed in Table 21. As a measurement of effectiveness, we have got to an average of 2.20 seconds to mitigate a DDoS attack. As a measurement of *QoS*, we have got an average *RTT* of 0.511 milliseconds and 0% of packets loss. In this test, we used the enhanced version 2 of Scenario I (subsection 4.2.1 heading *b*). For the *QoS* measurement, we calculated the average *RTT* and the percentage of the packet loss as the mean of the average *RTT* of one benign host (host 2). The DDoS mitigation time was also calculated using the mean of the values presented in the section 4.2.

Table 21 – Scenario II Summary Results

	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
Average	2.20	0.511	0

4.2.6. Scenario III - Discussion of Results

In this scenario, we were able to study a DDoS attack with packet size manipulation. We have showed that our system is able to detect and mitigate this kind of attacks. The results are exposed in Table 22. We have got to an average of 2.20 seconds to mitigate a DDoS attack, as a measurement of effectiveness. As a measurement of *QoS*, we have got an average *RTT* of 0.516 milliseconds and 0% of packets loss. In this test, we used the version 2 of Scenario I. For the *QoS* measurement, we calculated the average *RTT* and the percentage of the packet loss as the mean of the average *RTT* of one benign host (host 2). The DDoS mitigation time was also calculated using the mean of the values presented in the section 4.2.

Table 22 – Scenario III Summary Results

	DDoS mitigation time (s)	Average RTT (ms)	Packet loss (%)
Average	2.20	0.516	0

Chapter 5 – Conclusions and Future Work

In this chapter we will present the conclusions of our work and some future research work directions.

5.1. Main Conclusions

The current thesis presents a security system based on the SDN paradigm, at the client side, to ensure, in a centralized way, the “normal operation” of a home network. This system is composed by 3 main phases: the detection, the communication and the mitigation. The detection phase consists on a rule-based file, defining the rules that the system will have to comply. Therefore, an attack is detected and identified through the Snort IDS when the rules predefined in the file are broken. Hence, all the network traffic is mirrored for Snort analysis. The communication phase consists on making the connection between the Snort IDS and Ryu controller via Unix Domain Socket. This phase allows us to compound the detection with the mitigation phase. The mitigation phase consists on the reaction to a DDoS attack. This is accomplished by sending a blocking flow rule to the switch, only for the malicious hosts, through the controller - the developed controller was based on the work proposed by (Bonnet & Soenen, 2016). This system detects a DDoS attack scenario and mitigates it, at its origin, limiting the impact caused by devices with a detected malicious behavior. As discussed in section 4.3, our system was also able to maintain the connection between the benign hosts and the server while an attack was ongoing and maintain the expected *QoS*. These features are some of the major challenges in DDoS attacks mitigations. Instead of dropping all the packets, it differentiates the benign from malicious ones. A limitation of the developed system is that it is based on a rule-based mechanism. Therefore, we are only able to detect known attack signatures. Any unknown attack that is capable of evading the existing rules, would result on a bypass of the developed system. Hence, from an information security point of view, the rules database would have to be constantly updated.

All the attacks produced were conducted using the UDP protocol in different scenarios: DDoS attack (Scenario I v2), DDoS attack with IP spoofing (Scenario II) and DDoS with IP packet size manipulation (Scenario II). Our system was able to detect and mitigate all of them. In fact, according to the Table 23, making an evaluation of the system in the three test-case scenarios, we got an average DDoS mitigation time of 3.07 seconds, an average RTT of 0.541 milliseconds and a an average 0% of packets loss. Comparing the

average RTT of the normal usage with our system in usage, we can conclude that our system introduces a very small overhead given that the values in the three scenarios are not higher.

The major limitation of system is on the detection phase. Our rules that define a DDoS are predefined and static. Therefore, it can produce a significative number of false alarms. To solve this, we propose some improvements as future work (subsection 5.2). Nevertheless, we can conclude that the implementation and design of a security system based on SDN paradigm, at the client side, to ensure in a centralized way the normal operation of the potentially affected home network, was accomplished.

Table 23 – Scenarios Results Comparation Summary

Scenario	DDoS Mitigation time (s)	AVG RTT (ms)	Packet Loss (%)
Normal Usage	-	0.798	0
Scenario I v2	4.80	0.596	0
Scenario II	2.20	0.511	0
Scenario III	2.20	0.516	0
Average	3.07	0.541	0

5.2. Future Work

As a future work, we propose to firstly add to the system the capability to “forget” ex-malicious users. In other words, the system must be transparent for the user, i.e. if a user does not notice that belongs to a botnet, he must not notice that his access was blocked. In other words, a user must not be permanently “blocked” when a DDoS attacks happens. Second, to increase the home network simulated specified in the chapter 4.1.1, with more benign and infected devices. Given that our system is to be implemented in the client side, it will be worthful to study it with as many hosts as it is predictable that will be in the future, per home network. Last but not least, we propose to compound the Snort IDS tool with a machine learning tool detection in order to complement our rule-based detection mechanism. This way, we will increase the capabilities of our proposal in order to detect newer and sophisticated attacks.

Bibliography

- A. Doria, E., J. Hadi Salim, E., H. Khosravi, Ed W. Wang, E., L. Dong, R. Gopal, & J. Halpern. (2010). *Forwarding and Control Element Separation (ForCES) Protocol Specification*. Kenchiku Setsubi Iji Hozen Suishin Kyōkai. Retrieved from <https://tools.ietf.org/html/rfc5810>
- Abubakar, A., & Pranggono, B. (2017). Machine Learning Based Intrusion Detection System for Software Defined Networks. In *2017 Seventh International Conference on Emerging Security Technologies (EST)* (pp. 138–143). <https://doi.org/10.1109/EST.2017.8090413>
- Alshamrani, A., Chowdhary, A., Pisharody, S., Lu, D., & Huang, D. (2017). A Defense System for Defeating DDoS Attacks in SDN based Networks. *Proceedings of the 15th ACM International Symposium on Mobility Management and Wireless Access - MobiWac '17*, 83–92. <https://doi.org/10.1145/3132062.3132074>
- Alsmadi, I., & Xu, D. (2015). Security of Software Defined Networks: A Survey. *Computers & Security*, 53, 79–108. <https://doi.org/10.1016/j.cose.2015.05.006>
- Anderson, C. J., Foster, N., Kozen, D., & Walker, D. (2014). NetKAT : Semantic Foundations for Networks. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 113–126. <https://doi.org/10.1145/2535838.2535862>
- Anderson, T., & Khosravi, E. (2003). Requirements for Separation of IP Control and Forwarding. Retrieved from <https://tools.ietf.org/html/rfc3654>
- Arghire, I. (2018). “Wicked” Variant of Mirai Botnet Emerges. Retrieved June 27, 2018, from <https://www.securityweek.com/wicked-variant-mirai-botnet-emerges#>
- Atlas, A., Halpern, J., Hares, S., Ward, D., & Nadeau, T. (2016). *An Architecture for the Interface to the Routing System*. <https://doi.org/10.17487/RFC7921>
- Berde, P., Snow, W., Parulkar, G., Gerola, M., Hart, J., Higuchi, Y., ... Radoslavov, P. (2014). ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14* (pp. 1–6). <https://doi.org/10.1145/2620728.2620744>
- Bierman, A., & Bjorklund, M. (2012). *Network Configuration Protocol (NETCONF)*. Kenchiku Setsubi Iji Hozen Suishin Kyōkai. <https://doi.org/10.17487/rfc6536>
- Bonnet, J., & Soenen, T. (2016). Ryu Controller Code. Retrieved from https://github.com/sonata-nfv/son-examples/blob/master/vnfs/sonata-ryu-snort-vnf-docker/simple_switch_snort_sonata.py
- Botelho, F., Bessani, A., Ramos, F. M. V., & Ferreira, P. (2014). SMaRtLight: A Practical Fault-Tolerant SDN Controller, (July). <https://doi.org/10.1109/JPROC.2014.2371999>
- Braga, R., Mota, E., & Passito, A. (2010). Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow. *Proceedings - Conference on Local Computer Networks, LCN*, 408–415. <https://doi.org/10.1109/LCN.2010.5735752>
- Cai, Z., Cox, A., & T. S. Ng, E. (2011). Maestro: A System for Scalable OpenFlow Control. *Cs.Rice.Edu*, 10. <https://doi.org/Tech.Rep.TR10-08>
- Chowdhary, A., Pisharody, S., Alshamrani, A., & Huang, D. (2017). Dynamic Game based Security framework in SDN-enabled Cloud Networking Environments. *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization - SDN-NFVSec '17*, 53–58. <https://doi.org/10.1145/3040992.3040998>
- Cisco. (2016). *VNI Complete Forecast Highlights Global Internet Users: % of Population Devices and Connections per Capita Average Speeds Average Traffic*

- per Capita per Month Global -Device Growth Traffic Profiles 2021 Forecast*. Retrieved from https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_Device_Growth_Traffic_Profiles.pdf
- Darien Hirotsu. (2016). REST APIs in SDN: An introduction for network engineers. Retrieved July 24, 2018, from <https://searchsdn.techtarget.com/tip/REST-APIs-in-SDN-An-introduction-for-network-engineers>
- Erickson, D. (2013). The Beacon OpenFlow Controller. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking - HotSDN '13*, 13. <https://doi.org/10.1145/2491185.2491189>
- Floodlight. (2014). Floodlight OpenFlow Controller -Project Floodlight. Retrieved from <http://www.projectfloodlight.org/floodlight/>
- Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., & Walker, D. (2011). Frenetic: A Network Programming Language. *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming - ICFP '11*, 46(9), 279. <https://doi.org/10.1145/2034773.2034812>
- Gharvirian, F., & Bohlooli, A. (2017). Neural Network Based Protection of Software Defined Network Controller Against Distributed Denial of Service Attacks. *International Journal of Engineering*, 30(11), 1714–1722.
- Gorbunov, A., & Architect, N. (2013). OpenContrail as SDN controller for NFV infrastructure in AT&T network. Retrieved from <https://github.com/Juniper/contrail-controller>
- Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., & Shenker, S. (2008). NOX: Towards an Operating System for Networks. *SIGCOMM Computer Communication Review*, 38(3), 105–110. <https://doi.org/10.1145/1384609.1384625>
- Haleplidis, E., & Halpern, J. (2014). *Forwarding and Control Element Separation (ForCES)*. <https://doi.org/10.17487/rfc7409>
- Heller, B. (2009). OpenFlow Switch Specification. *Current*, 0, 1–36. <https://doi.org/10.1002/2014GB005021>
- Hinrichs, T. L., Gude, N. S., Casado, M., Mitchell, J. C., & Shenker, S. (2009). Practical Declarative Network Management. *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking - WREN '09*, 1. <https://doi.org/10.1145/1592681.1592683>
- HP Enterprise. (2013). HP SDN Controller Architecture. *Technical Solution Guide Hewlett-Packard Development Company, L.P.*, (September), 1–14.
- Huong, T. T., & Thanh, N. H. (2017). Software Defined Networking-based One-Packet DDoS Mitigation Architecture. *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication - IMCOM '17*, 1–7. <https://doi.org/10.1145/3022227.3022336>
- Ieee, F., Rothenberg, C. E., Ieee, M., Azodolmolky, S., Ieee, S. M., Uhlig, S., & Ieee, M. (2015). Software-Defined Networking : A Comprehensive Survey. *Proceedings of the IEEE*, 103(1), 14–76. <https://doi.org/10.1109/JPROC.2014.2371999>
- IETF RFC 6373. (2011). MPLS Transport Profile (MPLS-TP) Control Plane Framework. Retrieved from <https://tools.ietf.org/html/rfc6373>
- Internet World Stats. (2017). World Internet Users Statistics and 2018 World Population Stats. Retrieved June 27, 2018, from <https://www.internetworldstats.com/stats.htm>
- Jakaria, A. H. M., Rashidi, B., Rahman, M. A., Fung, C., & Yang, W. (2017). Dynamic DDoS Defense Resource Allocation using Network Function Virtualization. *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 37–42. <https://doi.org/10.1145/3040992.3041000>

- Jevtic, S., Lotfalizadeh, H., & Kim, D. S. (2018). Toward Network-based DDoS Detection in Software-defined Networks. In *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication - IMCOM '18* (pp. 1–8). New York, New York, USA: ACM Press. <https://doi.org/10.1145/3164541.3164562>
- Kia, M. (2015). Early Detection and Mitigation of DDoS Attacks In Software Defined Networks. Retrieved from http://digital.library.ryerson.ca/islandora/object/RULA%3A4357/datastream/OBJ/download/Early_detection_and_mitigation_of_DDos_Attacks_in_software_defined_networks.pdf
- Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., ... Shenker, S. (2010). Onix: A Distributed Control Platform for Large-Scale Production Networks. *9th USENIX Conference on Operating Systems Design and Implementation*, 1–6. <https://doi.org/10.1.1.186.3537>
- Lee, S., Kim, J., Shin, S., Porras, P., & Yegneswaran, V. (2017). Athena: A Framework for Scalable Anomaly Detection in Software-Defined Networks. *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 249–260. <https://doi.org/10.1109/DSN.2017.42>
- Lim, S., Ha, J., Kim, H., Kim, Y., & Yang, S. (2014). A SDN-Oriented DDoS Blocking Scheme for Botnet-Based Attacks, 63–68.
- Liu, J., Lai, Y., & Zhang, S. (2017). FL-GUARD: A Detection and Defense System for DDoS Attack in SDN. *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, 107–111. <https://doi.org/10.1145/3058060.3058074>
- Liu, X., Yang, X., & Xia, Y. (2010). NetFence: Preventing Internet Denial of Service from Inside Out. *Proceedings of the ACM SIGCOMM 2010 Conference on SIGCOMM - SIGCOMM '10*, 255. <https://doi.org/10.1145/1851182.1851214>
- Lu, Y., & Wang, M. (2016). An Easy Defense Mechanism Against Botnet-based DDoS Flooding Attack Originated in SDN Environment Using sFlow. *Proceedings of the 11th International Conference on Future Internet Technologies - CFI '16*, 14–20. <https://doi.org/10.1145/2935663.2935674>
- Matsumoto, S., Hitz, S., & Perrig, A. (2014). Fleet : Defending SDNs from Malicious Administrators. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking - HotSDN '14*, 103–108. <https://doi.org/10.1145/2620728.2620750>
- McCauley, M. (2011). POX. Retrieved July 11, 2018, from <https://github.com/noxrepo/pox>
- McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., ... Turner, J. (2008). OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 69. <https://doi.org/10.1145/1355734.1355746>
- Mininet. (2014). Mininet: An Instant Virtual Network on your Laptop (or other PC). *Mininet.Org*, www.mininet.org. Retrieved from <http://mininet.org/>
- Monsanto, C., Foster, N., Harrison, R., & Walker, D. (2012). A Compiler and Run-time System for Network Programming Languages. *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '12*, 217. <https://doi.org/10.1145/2103656.2103685>
- Mousavi, S. M. (2014). Early Detection of DDoS Attacks in Software Defined Networks Controller Early Detection Of DDoS Attacks in Software Defined Networks Controller.

- Mousavi, S. M., & St-Hilaire, M. (2015). Early Detection of DDoS Attacks against SDN Controllers. *2015 International Conference on Computing, Networking and Communications, ICNC 2015*, 77–81. <https://doi.org/10.1109/ICCNC.2015.7069319>
- Nec. (2012). Helios.
- NETSCOUT. (2018). Frequency and Complexity of DDoS Attacks is Rising; Defenders Turn to Automation and Managed Services for Support | Business Wire. Retrieved June 27, 2018, from <https://www.businesswire.com/news/home/20180123005628/en>
- Newman, L. H. (2016). What We Know About Friday’s Massive East Coast Internet Outage. Retrieved June 27, 2018, from <https://www.wired.com/2016/10/internet-outage-ddos-dns-dyn/>
- Newman, L. H. (2018). Github Survived The Biggest DDoS Attack Ever Recorded. Retrieved June 27, 2018, from <https://www.wired.com/story/github-ddos-memcached>
- Nordrum, A. (2016). Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated. Retrieved June 27, 2018, from <https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated>
- Ohyama, H. (2012). Jaxon:Java-based {OpenFlow} Controller. Retrieved from <http://jaxon.onuos.org/>
- Ombase, P. M., Scholar, P. G., Bagade, S. T., Kulkarni, N. P., & Mhaisgawali, A. V. (2017). DoS Attack Mitigation Using Rule Based and Anomaly Based Techniques in Software Defined Networking, (Icici), 469–475.
- Open vSwitch. (2009). Open vSwitch Manual. Retrieved August 2, 2018, from <https://manned.org/ovs-controller/46dcde74>
- Peffer, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3), 45–77. <https://doi.org/10.2753/MIS0742-1222240302>
- Pfaff, B., & Davie, B. (2013). The Open vSwitch Database Management Protocol. Retrieved July 16, 2018, from <https://tools.ietf.org/html/rfc7047>
- Pham, M., & Hoang, D. (2016). SDN applications - the intent-based Northbound interface realisation for extended applications. *SDN Controller : Intent-Based Northbound Interface Realization for Extended Applications*, 372–377. <https://doi.org/10.1109/NETSOFT.2016.7502469>
- Phemius, K., Bouet, M., & Leguay, J. (2014). DISCO: Distributed Multi-domain SDN Controllers. *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*. <https://doi.org/10.1109/NOMS.2014.6838330>
- Piedrahita, A., Rueda, S., Mattos, D., & Duarte, O. (2016). FlowFence: A Denial of Service Defense System for Software Defined Networking. *Journal of Infectious Diseases* (Vol. 167). <https://doi.org/10.1093/infdis/167.3.759>
- Poonam, V. T. D. V. (2016). The Northbound APIs of Software Defined Networks, 5(10), 501–513.
- Radware. (2013). Defense4All: Anti-DoS for OpenDaylight.
- RYU project team. (2013). *Ryu SDN Framework*. Retrieved from <https://osrg.github.io/ryu/>
- Sahay, R., Blanc, G., & Zhang, Z. (2015). Towards Autonomic DDoS Mitigation using Software Defined Networking. *Sent 2015: Ndss*, (February).

- <https://doi.org/10.14722/sent.2015.23004>
- Saint-Andre, P. (2011). *Extensible Messaging and Presence Protocol (XMPP)*.
Kenchiku Setsubi Iji Hozen Suishin Kyōkai. <https://doi.org/10.17487/rfc6122>
- Scott-Hayward, S.a , Natarajan, S.b , Sezer, S. . A. (2016). Survey of Security in Software Defined Networks. *Surveys & Tutorials*, 18(1), 623–654.
<https://doi.org/10.1109/COMST.2015.2474118>
- Sezer, S., Scott-Hayward, S., Kaur Chouhan, P., Fraser, B., Lake, D., Systems Jim Finnegan, C., ... Layout, S. (2013). Are We Ready for SDN? Implementation Challenges for Software-Defined Networks. *Future Carrier Networks*, 51(7), 36–43. <https://doi.org/10.1109/MCOM.2013.6553676>
- Shin, S., Porras, P., Yegneswaran, V., Fong, M., Gu, G., Tyson, M., ... Park, M. (2013). FRESKO: Modular Composible Security Services for Software-Defined Networks.
- Shin, S., Yegneswaran, V., Porras, P., & Gu, G. (2013). AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* (pp. 413–424).
<https://doi.org/10.1145/2508859.2516684>
- SNAC. (2012). OpenFlow Controller : SNAC (Simple Network Access Control). Retrieved from <http://archive.openflow.org/wp/snac>
- Snort. (2018). Snort - Network Intrusion Detection & Prevention System. Retrieved August 16, 2018, from <https://www.snort.org/>
- Suh, J., Choi, H., Yoon, W., You, T., Kwon, T. T., & Choi, Y. (2010). Implementation of Content-Oriented Networking Architecture (CONA): A Focus on DDoS Countermeasure. *1st European NetFPGA Developers Workshop*, 1–5. Retrieved from
https://mmlab.snu.ac.kr/publications/docs/2010_EU_netfpga_workshop_jhsuh.pdf
- Tootoonchian, A., & Ganjali, Y. (2010). HyperFlow: A Distributed Control Plane for OpenFlow. *Proceedings of the 2010 Internet Network ...*, 3–3.
<https://doi.org/10.1109/JPROC.2014.2371999>
- Trema. (2015). Trema: Full-Stack OpenFlow Framework in Ruby and C. Retrieved July 11, 2018, from <https://trema.github.io/trema/>
- Ujcich, B. E., & Rausch, M. (2014). Responding to Malicious Traffic with an Intrusion Detection System in a Software-Defined Network.
- Vasseur, J., & Le Roux, J. (2009). Path Computation Element (PCE) Communication Protocol (PCEP). Retrieved from <https://tools.ietf.org/html/rfc5440#page-8>
- Voellmy, A., Kim, H., & Feamster, N. (2012). Procera: A Language for High-Level Reactive Network Control. *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, 43–48. <https://doi.org/10.1145/2342441.2342451>
- Wang, a, Guo, Y., & Hao, F. (2014). Scotch: Elastically Scaling up SDN Control-Plane using vSwitch based Overlay. ... *of the 10th ACM ...*, 403–414.
<https://doi.org/10.1145/2674005.2675002>
- Wang, R., Jia, Z., & Ju, L. (2015). An Entropy-Based Distributed DDoS Detection Mechanism in Software-Defined Networking.
<https://doi.org/10.1109/Trustcom.2015.389>
- White, R., & Zandi, S. (2016). Cloudy-Eyed: Complexity and Reality with Software-Defined Networks, 19(3), 31–40. Retrieved from <http://ipj.dreamhosters.com/wp-content/uploads/issues/2016/ipj19-3.pdf>
- Xing, T., Huang, D., Xu, L., Chung, C., & Khatkar, P. (2013). SnortFlow: A OpenFlow-based Intrusion Prevention System in Cloud Environment.

- <https://doi.org/10.1109/GREE.2013.25>
- Xing, T., Xiong, Z., Huang, D., & Medhi, D. (2015). 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* (pp. 308–311). <https://doi.org/10.1109/CNSM.2014.7014181>
- Yap, K.-K., Huang, T.-Y., Dodson, B., Lam, M. S., & McKeown, N. (2010). Towards Software-Friendly Networks. *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems - APSys '10*, 49. <https://doi.org/10.1145/1851276.1851288>
- You, X., Feng, Y., & Sakurai, K. (2018). Packet_In message based DDoS attack detection in SDN network using OpenFlow. In *Proceedings - 2017 5th International Symposium on Computing and Networking, CANDAR 2017* (Vol. 2018–Janua, pp. 522–528). <https://doi.org/10.1109/CANDAR.2017.93>
- Zaalouk, A., Khondoker, R., Marx, R., & Bayarou, K. (2014). OrchSec: An Orchestrator-Based Architecture For Enhancing Network-Security Using Network Monitoring And SDN Control Functions. In *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*. <https://doi.org/10.1109/NOMS.2014.6838409>
- Zhou, W., Li, L., Luo, M., & Chou, W. (2014). REST API Design Patterns for SDN Northbound API. *Proceedings - 2014 IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, IEEE WAINA 2014*, 358–365. <https://doi.org/10.1109/WAINA.2014.153>

Appendixes

Appendix A – *packet_in_handler* Code

The *packet_in_handler* is a handler method that runs when a switch receives an OpenFlow *packet_in* message.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    # self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

Appendix B – *add_flow* Code

The *add_flow* is a method that add new rules to the OpenFlow switch.

```
def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]

    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                             match=match, instructions=inst)
    datapath.send_msg(mod)
```

Appendix C – *process_snort_alert* Code

The *process_snort_alert* is a handler method that runs every time that Snort launch an alert.

```
@set_ev_cls(snortlib.EventAlert, MAIN_DISPATCHER)
def process_snort_alert(self, ev):
    self._dump_alert(ev)

    msg = ev.msg
    alertmsg = ''.join(msg.alertmsg)
    if 'ryu block' in alertmsg:
        self.logger.info('Blocking this flow:{0}'.format(alertmsg))
        self.send_blocking_flowrule(msg)
```

Appendix D – *send_blocking_flowrule* Code

The *send_blocking_flowrule* method that sends an “empty” (i.e. to drop traffic) rule to the switch.

```
def send_blocking_flowrule(self, msg):

    datapath = self.datapath

    if datapath is None:
        self.logger.info('no switch detected yet, ignoring alert...')
        return

    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    pkt = packet.Packet(array.array('B',msg.pkt))
    pkt_eth = pkt.get_protocol(ethernet.ethernet)

    mac_src = pkt_eth.src
    eth_type = pkt_eth.ethertype

    pkt_ipv6 = pkt.get_protocol(ipv6.ipv6)
    if pkt_ipv6:
        self.logger.info('received ipv6 packet, blocking ipv6 rules is not supported yet...')
        return

    pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
    if pkt_ipv4:
        dst = pkt_ipv4.dst
        src = pkt_ipv4.src
        ip_proto = pkt_ipv4.proto

        pkt_tcp = pkt.get_protocol(tcp.tcp)
        pkt_udp = pkt.get_protocol(udp.udp)
        pkt_icmp = pkt.get_protocol(icmp.icmp)

        elif pkt_icmp:
            self.logger.info('received icmp packet')
            match = parser.OFPMatch(eth_src=mac_src,
                                    eth_type=eth_type, ipv4_dst=pkt_ipv4.dst, ipv4_src=pkt_ipv4.src,
                                    ip_proto=ip_proto)

        else:
            self.logger.info('received other packet')
            match = parser.OFPMatch(eth_src=mac_src,
                                    eth_type=eth_type, ipv4_dst=pkt_ipv4.dst, ipv4_src=pkt_ipv4.src,
                                    ip_proto=ip_proto)

    priority = 100
    # only send out of snort port, to keep on monitoring
    #actions = [parser.OFPActionOutput(self.snort_port)]
    # empty actions = DROP packets
    actions = []
    #actions = [parser.OFPActionOutput(OFPPS_BLOCKED)]
    self.add_flow(datapath, priority, match, actions)
```