COMS4200/ 7200 Final Project
Exploring Suricata and DOS Protection in an SDN Context

https://images.app.goo.gl/DAuBVjhPK8brsJHp8

List of Team Members

Name	**Email**	**Course Code**
Ebrahim Awad S Alharbi	s4564841@student.uq.edu.au	COMS7200
William Cumines	s4391890@student.uq.edu.au	COMS4200
Lachlan Drury	s4478191@student.uq.edu.au	COMS4200
Harry Huang	s4530915@student.uq.edu.au	COMS4200
Kausthubram Rajesh	s4479071@student.uq.edu.au	COMS4200

Table of Contents

1. Abstract

As computer networks get more complex, Denial of Service attacks into these networks are also becoming more prominent. Hence, it is important to have systems to detect and prevent these attacks. This is where IDS and IPS applications such as Suricata come into play.

Through background research on the relevant topics and related works such as Network Security, Software Defined Network concepts, and common DOS attacks; our team was able to develop a clear scope and methodology for the project.

The overall aim of the project was to simulate various DOS attacks within an SDN environment and test the response of Suricata for detecting and preventing these attacks within tree network topologies (best topology to simulate our context of experimenting in a university setting).

Mininet was used to set up network topologies for testing. Hping3 and nmap were then used to simulate various DOS attacks on our virtual network. After running experiments multiple times, it was found that Suricata was an effective tool to log malicious network activity given the correct rulesets were defined.

Due to time constraints, our future work would look into scaling up Suricata to running in IPS mode on larger networks with different topologies.

2. Project Definition

2.1 Goals

The overall goal of this project is to evaluate Suricata's feasibility as an IDS integrated within an SDN environment. More specifically, we will investigate Suricata's capability to detect and prevent common forms of DOS attacks. From this, our goal is to make an informed recommendation as to Suricata's real world application within an SDN environment.

2.2 Scope

This will be implemented using a Mininet virtual network, allowing a range of network environments to be quickly developed and tested, whilst fully supporting SDN. The network environments implemented will best reflect a University network, with a common tree network topology along with LAN and WLAN emulation to reflect internal and external DOS attacks on a University Web Server. This project will take a Breadth first analysis approach (wide scope but not too depth), by which we will test a wide variety of DOS attacks, with an emphasis on testing capability rather than developing a full implementation. The majority of this implementation will be carried out using the Python programming language, with it being used to write scripts for simulating virtual networks, performing network traffic analysis, and installing various flow rules.

2.3 Motivation

This project is motivated by the need for network security of large enterprise/campus like networks; a common victim of both internal and external DOS attacks due to their servicing of a large number of users. For example, in 2012, a DDOS attack on six American banks crippled their operations, costing this upwards of $100K [1].

Further with the growing popularity of SDN networks, we also see the need to ensure its Security and more so feasibility within such networks. This project looks at combining these motivations with the implementation of Suricata as a suitable IDS. where we can explore Suricatas adequacy within a modern network environment; by testing its ability to mitigate common DOS attacks.

3. Background

3.1 SDN Concepts

3.1.1 SDN Breakdown

Software defined networking (SDN) is a networking architecture that aims to redefine how networks are built, to counteract the ever-growing complexity and rigidity of traditional networks [2]. Traditional networks work using a distributed algorithm, executed on every router within the network. Meaning, each router is responsible for both; forwarding all traffic that passes through it to the correct output port, and creating and managing the routing table, from which the router can query for the information to correctly forward traffic to the correct output port. The forwarding of packets to output ports is known as the Data Plane, while the creation and management of the routing table is known as the Control Plane. The primary concept behind SDN is to decouple the data and control planes, placing the control plane in a logically centralized infrastructure. Figure 1 shows this transition from traditional networking infrastructure to a SDN networking infrastructure.

SDN decouples the data and control planes first by placing an abstraction over the data plane. Currently, the main technology for this is OpenFlow, which is explored in more detail in Section 3.3 below. In brief, OpenFlow acts as the interface between the data plane and the control plane, abstracting the functionality of the data plane, so that the control plane is shielded from the vagaries and specifics of the distributed nature of the data plane. The primary advantage of this is that making changes and creating new technologies and innovations within the control plane no longer requires a detailed understanding of what happens within the data plane, as well as removing the potential need to manually configure every router and switch on the network whenever a change to that network is rolled out.

SDN opens the door to a flood of innovation in an area that has been relatively stagnant over the past several years. Some of the key advantages an SDN architecture provides are: enabling complex automation to assist with management tasks that were previously impossible to automate due to the distributed nature of the control plane, access to a multi-vendor network as previously vendor-specific hardware made this too complex to be worthwhile, increases visibility and control over the network allowing the guarantee of quality of service, security, traffic engineering, and access control being enforced consistently between multiple wireless and wired network infrastructures [2].

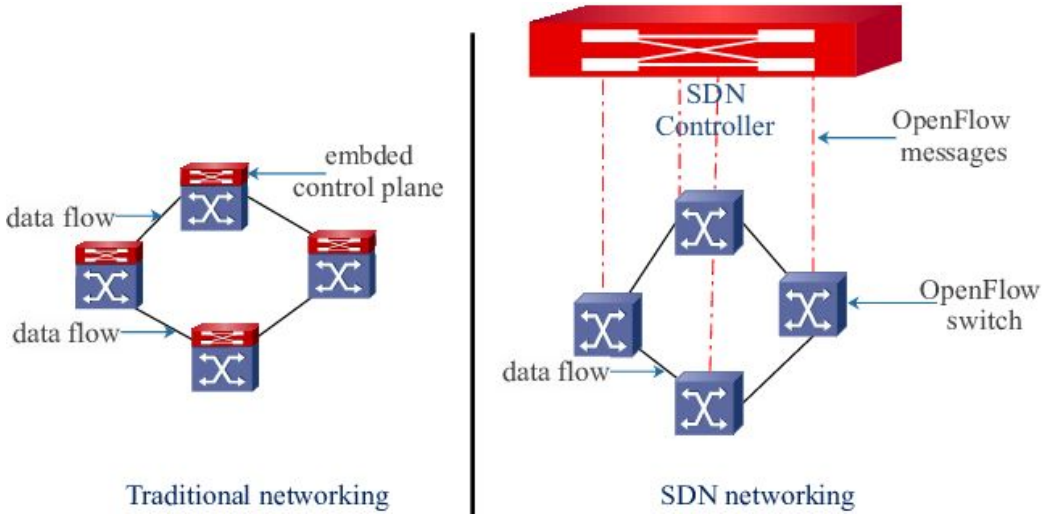

Figure 1. Traditional networking infrastructure compared to SDN networking infrastructure [3, fig. 1].

3.1.2 Openflow

Openflow is a standard protocol for communication in SDN networks between switches and controllers. It is used for communication between the control layer and data layer. These layers can be briefly summarised as the control layer being the layer which serves to make the decisions on where to route the packet, whereas the data plane is a series of switches which actually do the forwarding. In standard networks, this is done in the switch and router itself. This is really inefficient as each routing protocol needs to be implemented on top of every company's custom operating system and effectively leads to duplication of the effort. A core concept in software defined networks is the fact there is a separation of the control and data plane with the switches being 'dumb' devices that communicate with controllers which manage the Control plane of the entire network. However, this means that there needs to be a common language that is reliable between these switches and controllers. OpenFlow is this language.

OpenFlow allows you to directly tell switches different actions based on the packets that they have received in a reliable manner. This reliability stems from the fact that it is written on top of TCP which means that efforts will be made by both sides to ensure the otherside has received their query.

The SDN and OpenFlow paradigm has many key advantages over traditional routing frameworks as it allows for central control over switches regardless of which company had manufactured them, more granular control and better user experience. This granular control will be important in the experiments that we intend to run and would not be as simple in traditional networking architectures [4].

3.1.3 Mininet

Mininet is a network emulator for several purposes, such as testing, research, and learning [5]. It creates virtual hosts, switches, controllers, and links and supports OpenFlow. It can run up to 4096 virtual hosts and switches. Its main advantage is providing a cheap and straightforward network testbed, so there is no need to deal with the physical network.

3.2 Network Security

Network Intrusion Detection Systems (NIDS) detect and prevent intrusion into computer systems and networks by providing information on susceptible activities [6]. They have two types: host-based systems, which collect information about a particular system or host; and network-based systems, which collect information from its whole network rather than a separate host [7]. An Intrusion Detection System (IDS) is the physical device, or software application that monitors a network system for malicious activity or policy violations. Primary assumptions of using an IDS is that the system activities are observable with normal and intrusive activities having distinct evidence. IDS use two main types of detection techniques, anomaly (behaviour-based) and signature (knowledge-based) detection. The anomaly-based IDS establishes a normal profile and generates alerts when different behaviour was observed. Signature-based IDS detect threats and generate alerts through using predefined attack signatures.

Intrusion Prevention System (IPS) is an extension of an IDS that includes the capability to block or prevent detected malicious activity. IPS can be host-based, network-based, or distributed/ hybrid. IPS uses anomaly detection to identify behaviour that is not that of legitimate users, and signature detection to identify known malicious behaviour to block traffic similar to a firewall; it makes use of the algorithms developed for IDSs to determine when to do so [8].

Overall, in network security, IDS and IPS are important as they provide the peace of mind that when configured properly, a network will be protected from known threats with limited resource requirements.

3.3 DOS Attacks

3.3.1 Overview

A Denial Of Service (DOS) attack is a network based attack intended to negatively affect legitimate users' availability to access a system by saturating the system's resources. There are two primary methods of a DOS attack, one being through system vulnerabilities, which by malformed network packets utilise such vulnerabilities, "causing excessive memory consumption, extra CPU processing, system reboot, or general system slowing" [6]. On the other hand there are flooding attacks, which aim to overload a network or system by sending large amounts of network traffic to the intended victim. This attack can be inflicted by a range of protocols, namely, ICMP, UDP, and TCP [9]. Additionally this can be extended to a Distributed DOS (DDOS) where multiple hosts coordinate together to launch the attack [6].

The detection of DOS-like attacks requires the distinction between a malicious and a legitimate packet to ensure no service interruption is made to any legitimate user. As such most detection methods define an attack as an "abnormal and noticeable deviation of some statistic" when monitoring a real-time network workload. The discerning difference in methods being the statistical metric used. One such approach is to create activity profiles by correlating network traffic by inspecting packet headers for similarities in address, port, protocol, etc. From this, flows are defined as the packet rate for each profile; in which an increase in a flow or a cluster of similar flows could indicate an attack [6].

An attacker can execute a DOS attack several different ways, as there are a variety of protocols, and network structures available for them to exploit, as well as several different possible outcomes that may be desired. The following are common structures for DOS attacks and network weaknesses that aim to exploit.

3.3.2 ICMP Flooding

ICMP flooding is the most common and simple DOS attack. The attack consists of an attacker attempting to overwhelm the victim device with ICMP echo requests or pings. This can result in the victim device becoming inaccessible to normal traffic, as well as heavily taxing the victims CPU, potentially crashing the device [10].

Detecting an ICMP flood can be done as mentioned above, by monitoring the network, searching for an "abnormal and noticeable deviation of some statistic". The statistic mentioned is generally based on historical data from the network. Once detected, the simplest method of prevention is to disable the ICMP functionality of the victim device [10].

3.3.3 IP Fragmentation Attack

An IP Fragmentation attack is another common form of a denial of service attack which aims at leveraging the fragmentation mechanisms set by networks when a receiving IP datagram is too large. Generally speaking, every network has a limit known as the maximum transmission unit (MTU); indicating the maximum size datagram it will accept before it must be fragmented. While this attack comes in multiple forms, its key aim is to leverage the reassembly mechanisms present on the target device in order to overwhelm the target network. Two most common forms of this attack are the UDP/ICMP fragmentation attack and the TCP fragmentation attack.

The UDP/ ICMP fragmentation attack involves flooding the target with UDP/ICMP packets larger than the MTU. Furthermore, these packets are fake and will be impossible to reassemble by the target device, hence quickly consuming the device's system resources.

The TCP fragmentation attack, also known as a teardrop attack, targets the TCP/IP fragmentation reassembly mechanism. The attacker sends fragmented packets which overlap with each other in order to confuse and hence overwhelm the target device. However, as of today, most operating systems have patched the vulnerabilities leveraged by the TCP fragmentation attack, and hence the UDP/ICMP fragmentation attack will be focused on in this project.

Detection and mitigation of an IP fragmentation attack can be achieved similarly to any form of DOS flooding attack by monitoring each packet's sender and blocking any senders that generate an unreasonable amount of traffic. However, IP fragmentation attacks easily implement various methods of IP spoofing in order to mask the sender's identity and as such make it difficult to block the offender. This will be discussed later in the report.

3.3.4 Asymmetric Routing and SYN Packet Flooding

Asymmetric routing occurs in almost all networks, and is often an unwanted network behaviour. It occurs when a packet that is travelling through the network takes one route to its destination, and then a different route returning. This however is not an issue for most routing functions, and is beneficial in terms of efficiency and redundancy within the network. Asymmetric routing can become an issue when an attacker uses the asymmetric nature of the network to bypass security function, in order to send malicious packets to targets within the network. A common type of attack that utilises this network flaw is the SYN flood attack [11].

A SYN flood attack is an attack on a TCP server, where that attacker bombards the server with SYN packets. As mentioned previously, this can be used as a probing attack, however, if sent at a high enough rate, a SYN flood attack will disable the TCP servers ability to receive packets, as the receive buffer for half open connections will be full of fake SYN packets. Even after the flood stops, the server will still require time to recover, as it must process each packet (assuming there are no SYN flood countermeasures), replying to each one with an SYN-ACK response, and wait for half open connection to timeout [12].

From the perspective of the server, there are several measures that can be used to prevent or lessen the effect of a SYN flood attack. These include: recycling the oldest half open connection; increasing the buffer for half open connections; reducing the timeout for half open connections; and SYN cookies, a strategy that involves the server dropping the half open connection immediately after sending the SYN-ACK packet, and if the SYN request is legitimate, a final ACK packet will be received, and from which the server will reconstruct the SYN and SYN-ACK requests from the data within the TCP succession number [12].

Alternatively, a SYN flood attack, or any other similar DOS attack that might take advantage of the asymmetric nature of a network, can be detected and/or prevented by various network threat detection engines, such as Suricata, which detects statistical changes in network traffic with the ability to drop malicious packets. This method of prevention will continue to be explored alongside DOS attack protection.

3.3.5 Port Scanning Attacks

Whilst not exactly a DOS attack, probing attacks, such as Port Scanning Attacks, send a large number of packets to a network, however, not to cripple functions within that network, rather to learn which ports are open in a target host through the use of a listening device . The sole purpose of these attacks is to gain information about the state of the victim’s network. For example, a Port Scanning Attack may send a large amount of SYN packets to the victim, attempting a TCP connection, to find out which ports the host has open. Once this information is known, the attacker is able to listen in on open ports and even send malicious packets to the victim on these ports [13].

3.3.6 Additional Strategies

In addition to the attacks mentioned, an attacker can employ a number of additional strategies to increase the effectiveness of the attack. Such strategies include Internal Reflection, IP Spoofing and DDOS.

IP Spoofing is referred to when a sender modifies the Source Address in an IP packet in order to conceal their true network address. In terms of DOS attacks, this can prevent the target from identifying the true source of a DOS attack, and further an attacker can continue to spoof another source address should one be blocked by the target network.

A Distributed DOS or DDOS attack is a method by which an attacker will use multiple hosts in order to conduct the attack, rapidly increasing the inbound network traffic on the target network. As the attack is being sourced from multiple hosts, the target may struggle to identify and block each individual attacker, rendering the attack much more effective.

3.4 Anomaly Based Detection

Anomaly based detection is a more complex form of detecting DOS attacks or the like. There are many different implementations of anomaly-based network detection systems (A-NIDS), with the three main implementations being: statistical based, knowledge based, and machine learning based. A statistical based system utilises the data generated by network traffic, relying on the statistical analysis of that data to gain insight into potential anomalies. On the other hand, a knowledge based system aims to capture the context in which the network is operating in, absorbing the claimed behaviour of various systems, such as protocol specifications, and network traffic instances. Finally, a machine learning based system relies on the categorisation of patterns analysed by an explicit or implicit model, that the system aims to establish [14].

In general, all three implementations follow a similar basic process:

- **Parameterisation**: The target system is observed and is represented as a set of parameterised instances.
- **Training**: The normal (or abnormal) behaviour that occurs within the system is characterised. With this, a corresponding model is built, which may be done very differently, depending on the implementation of the A-NIDS. The statistical and machine learning based methods will require some manual input, but are predominantly automatically generated. Whereas the knowledge based method requires manual building of the model.
- **Detection**: The created model is compared against the parameterised instances of the observed traffic. An alert will be triggered when traffic deviates greater than or less than a given threshold [14].

3.5 Suricata

Monitoring network activity is crucial because it is the first step to detect and prevent threats. This is where the use of Suricata is effective. Suricata is a free and open-source network threat detection engine made by the Open Information Security Foundation (OISF). Suricata is capable of real time Intrusion Detection, inline Intrusion Prevention, Network Security Monitoring (NSM) and offline packet capturing. On a high level, Suricata inspects the network traffic using extensive rulesets and signature language through the support of several standard formats, such as JSON and YAML [6].

The first version of Suricata was released in 2010 [7]. Although Suricata code is original, its developers borrowed from the Snort, which is open source NIDS and was released in 1998 [7]. OISF claims the purpose of Suricata is to bring new ideas and technologies to the field not to replace the existing tools [10]. Although Suricata uses similar rules as Snort, it brings new technologies and ideas such as using multi-threading for IDS [10]. Therefore, it analyzes the network traffic concurrently, so it achieves high scalability to detect network threats [14].

Suricata can detect and protect several types of attacks such as DOS attacks with SYN flooding and FTP brute force [15]. Suricata mainly can be used as host-based IDS to monitor the traffic of a single host. It can also be used as a passive IDS, which alerts the network administrators, or as an active inline IPS, which prevents the threat and alerts the administrators.

Suricata gets one packet at a time from the network system of the nfnetlink_queue from the kernel. These packets are then pre-processed, and passed to the detection engine. Suricata works with rules and can either accept or drop packets. Suricata IPS introduces the actions of 'drop', 'sdrop' (silent drop) and 'reject' for rejecting packets.

4. Literature Review

4.1 Suricata

Nowadays, the Denial-of-Service (DOS) attacks, which are malicious attempts to overwhelm the targets (e.g. servers or networks) with a flood of traffic, are overgrowing. For example, in the first half of 2020, this kind of attacks increased by 151 per cent [16]. Therefore, it is essential to detect and prevent those attacks or even to mitigate them.

There are several tools to detect and prevent many attacks. Those tools are called Network Intrusion Detection Systems (NIDS) such as Snort and Suricata. Therefore it is vital to explore and test their capabilities to use them with an informed decision and to improve them.

This project focuses on exploring the Suricata capabilities in terms of the accurate detection of (DOS) attacks so that the previous research will be discussed in this section.

Although the majority of previous work is on Snort, there is some research about the Suricata. Most of the purposes of work on Suricata are to measure the Suricata performance in terms of the speed and memory requirements. Then the researchers compared against the performance of other systems like Snort and Bro. For example, a thesis by Mauno Pihelgas compared Suricata, Snort, and Bro in terms of their performance (e.g., CPU, memory usage, dropped packages) [17]. Another example, White et al. [18] tested the performance of Suricata and Snort with considering the high scalability of system resources like the number of CPU cores. They found the Suricata has higher performance than Snort.

However, this previous purpose is out of the scope of this project which is the exploration of Suricata in terms of the accurate detection of (DOS) attacks.

A thesis by Eugene Albin [19] has a part about the evaluation of the accuracy of detection in Suricata and comparing it with Snort. The researcher used two virtual machines and one a physical host (i.e., laptop) as the server to test the accurate detection of the two systems, as shown in Figure 2.

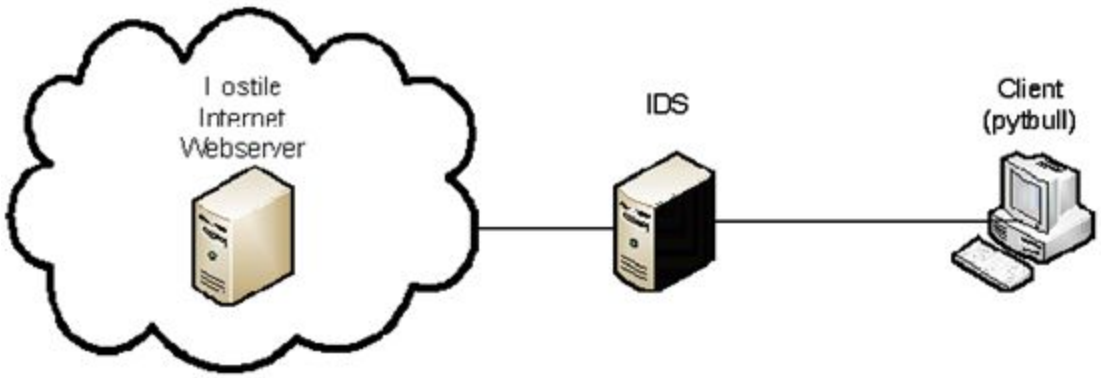

Figure 2. The used network topology [19, Fig. 4]

The researcher conducted 54 tests in 9 categories of attacks: malformed traffic, client-side attacks, packet fragmentation, denial of service, intrusion-detection system evasion, malware identification, failed authentication, common rule testing, and shellcode. His testing and findings for two categories (packet fragmentation and denial of service) which are related to this project:

- Fragmented packets: the researcher tested two attacks. The first one was a Ping-of-Death attack in which the packet fragmentations are larger than allowed one when they are reassembled [19]. He found that Suricata just identified this attack. Another attack was Nestea attack (teardrop) which is when the fragmented packets are out of the sequence order when they are reassembled. He found that Suricata and Snort were unable to detect this attack.
- Denial of Service: the researcher was able to launch just one attack, which was a DoS-specific attack against MSSQL application because he used Pytbull, which provides limited tests. He found that both Suricata and Snort detected the attack, but they could not identify it as DoS-specific attack, so they just considered it as suspicious traffic. Also, the researcher could not conduct an ICMP ping flood attack.

His findings show that both Suricata and Snort have false positives and false negatives, but the researcher thinks that happened because of the weakness of the used rules. Although he recommended using the Suricata, he could not decide which better in terms of the accuracy of detections.

In terms of what is similar between this thesis and our project, it has tested the ability of Suricata to detect various attacks. However, in terms of difference, it has not considered the SDN environment. Also, it has not focused on DoS attacks. We will aim to build upon this thesis with our experiments and implementation.

This project will focus more on the accuracy of detection of DoS attacks, including ping attack, and it will use the current version of Suricata than that used in that thesis.

In 2019, a conference paper by Ernawati et al. [20] analysed the detection accuracy and the speed of detection of three IDS (Suricata, Port Scan Attack Detector (PSAD), Portsentry). The researcher used physical hosts, so they did not use the simulation method.

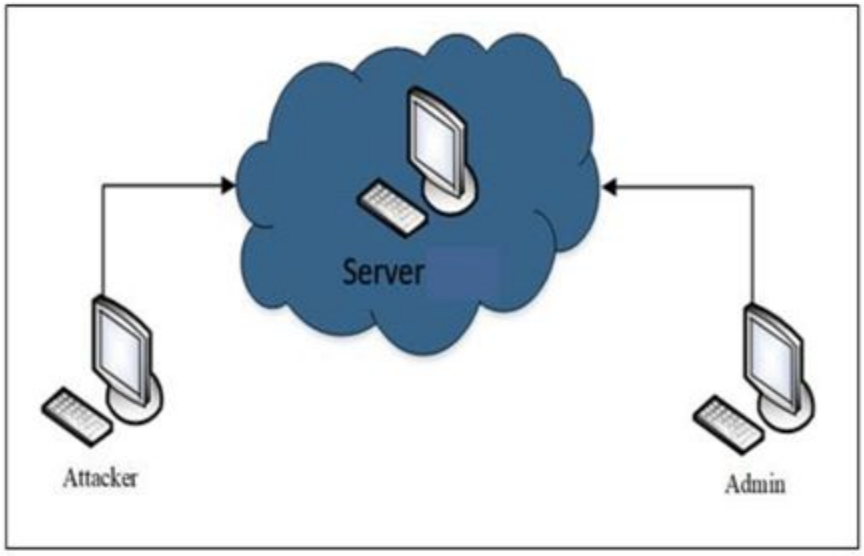

Figure 3. The used network topology [20, Fig. 2]

They created a small scale network topology between the server which works as IDS and the attacker. Also, there was an admin host for network administration, see Figure 3. Then the attacker can launch attacks against the server and the IDS system will monitor and detect any suspicious activities on the server or the network. The researchers tested three types of attacks including Port scanning, which the attacker tries to find out any open ports, DDoS(Distributed Denial of Service) through SYN Flood, and Brute force attack. Their results showed that Suricata could detect all the tested attacks, see Figure 4 and 5.

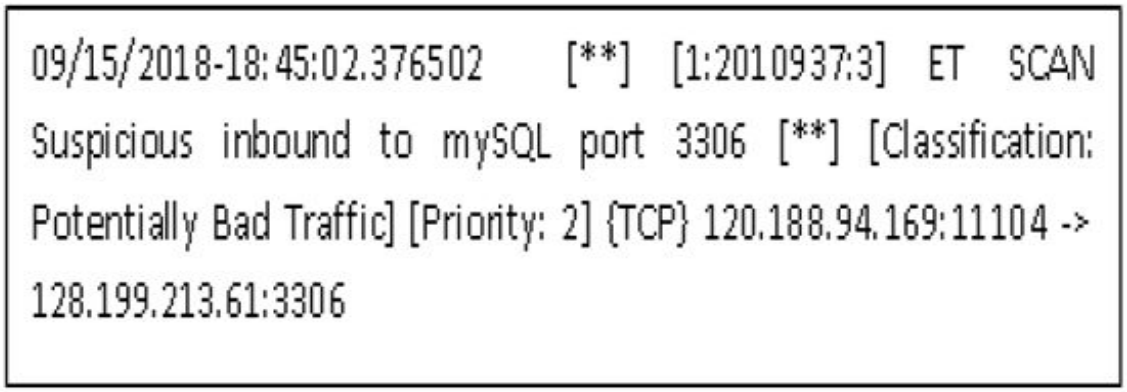

Figure 4. The response of Suricata to port scan attacks [20, Fig. 9].

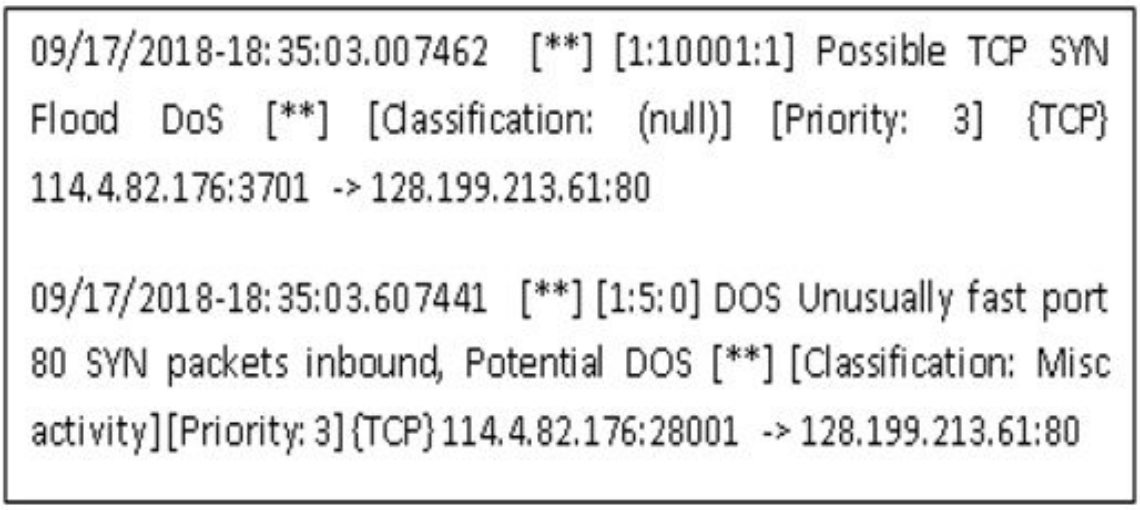

Figure 5. The response of Suricata to DDoS SYN Flood attacks [20, Fig. 10].

They found that Suricata and PSAD could detect all the used attacks and PSAD is faster at detection of attacks on average. However, the Suricata is better at detection speed of DDoS attacks, see Table 1. Also, they concluded that both Suricata and PSAD are good to use as IDS.

	Metric	PSAD				Portsentry				Suricata			
		Port Scanning	DDoS SYN Flood	Brute Force Attack	Average (all attacks)	Port Scanning	DDoS SYN Flood	Brute Force Attack	Average (all attacks)	Port Scanning	DDoS SYN Flood	Brute Force Attack	Average (all attacks)
Speed of detection	second	2.18	6.94	3.51	4.21	2.09	6.14	-	-	2.56	3.44	14.54	6.85

Table 1. The detection speed of three IDS [20, Tab. 10].

In terms of what is similar between this conference paper and our project, it has tested the ability of Suricata to detect various attacks. However, in terms of difference, it has not considered the SDN environment. Also, it has not focused on DoS attacks. We will aim to build upon this paper with our experiments and implementation.

Although the researchers used a recent version of Suricata, this project will test more DoS attacks like ping attack and create more complicated virtual network topology.

A peer-reviewed conference paper by Bouziani et al. [21] tested and compared the capabilities of three IDS(Suricata, Snort, and Bro) to detect known attacks, malware and evasion techniques.They collected their general functionalities of the literatures, see Table 2 below.

Functionality	**Bro**	**Suricata**	**SNORT**
Rule	BRO rules, Bro script similar to c++ language structure, Snort rule by using the snort2bro script	Using rules from emerging threat and VRT, Lua script, possibility of using pulled pork for rule management	Using rules from VRT and doesn't support all rule from emerging threat possibility of using pulledpork for rules management
RAM usage	Low.	High	Medium
Dropped Packets	Low.	Medium	High in the large networks
Multi CPU	Bro use multiple processors by providing a "worker" based.	Suricata use multiple CPU	Single CORE processor
IPS functionality	BRO provides only limited intrusion Prevention functionality, Its scripts can execute arbitrary programs, which are used operationally to terminate misbehaving TCP connections.	Intrusion Prevention with NFQUEUE	SNORT Inline supports new rule keywords to allow traffic matching a rule to be dropped, thus turning SNORT into an Intrusion Prevention System (IPS)
IPV6 Support	Yes	Yes	Yes
Installation and configuration	User may have problems during the installation due to	Easy to install and configure	Complete documentation in www.SNORT.com

	Missing, incompatible packages. -Documentation incomplete.		
Compatibility with OSs.	standard Linux, FreeBSD, and MacOS platforms.	Windows, UNIX, macOS	Portable (Linux, Windows, macOS X, Solaris, BSD, IRIX, Tru64, HP-UX)

Table 2. The general functionalities of Bro,Suricata and Snort [21, tab. 1].

They used a simple network topology which are three hosts (the target, the attacker, and IDS) to simulate the attacks manually without any defence techniques such as firewalls, as shown in Figure 6.

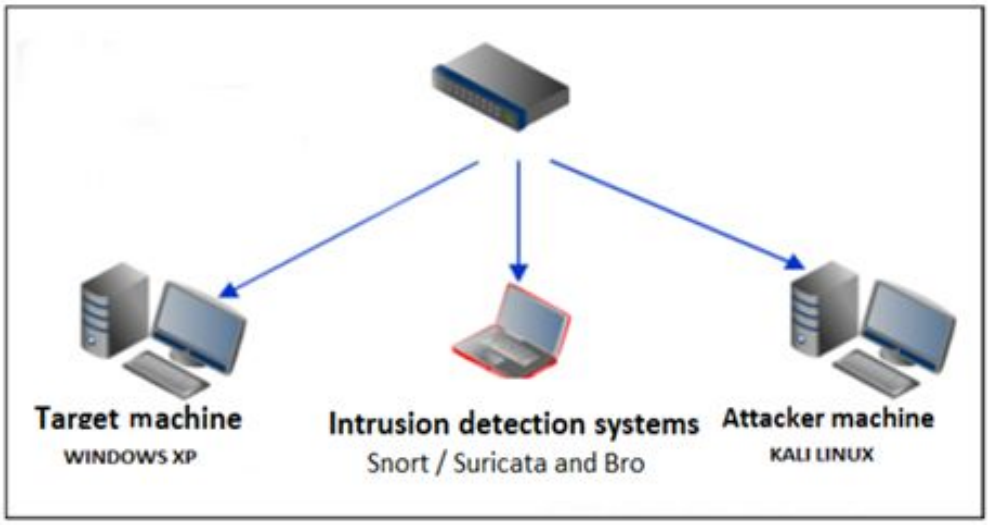

Figure 6. The network topology [21, Fig.1]

They tested several attacks, including:

- Detection of known attacks:
    - SQL injection: Suricata could detect this attack.
    - Local File Inclusion (LFI) which allows the attacker to include files on the server. Suricata could detect this attack.
- Persistence against evasion techniques.
- Payload mutation:
    - JavaScript obfuscation: Suricata could not detect this attack.
    - Encoding payload: Suricata was able to detect this attack.
- Packet splitting: Suricata was able to detect this attack.
- Denial of Service (DoS).
- Ping of death which the attacker sends larger packets than allowed ones, so they can crash the target machine because the TCP/IP systems can not handle the exceeded packet size: none of them was able to detect this type of attack.
- Hping SYN Flood: Suricata was able to detect this attack.
- FTP brute force Suricata was able to detect this attack.
- Bad traffic: Suricata and Snort were able to detect some of those attacks.
- Malware:
    - Trick bot infection with tor traffic: which is a malware to redirect the target user on a malicious page to enter his/her credential information. Suricata was able to detect this attack. Suricata was able to detect this attack.
    - Word docs push Herms ransomware: Suricata was able to detect this attack.
- Shellcode mutation:
    - Shellcode encoded by Shikata Ga Nai (SGN): Suricata was able to detect some of those attacks.
    - Shellcode encoded with base 64: Suricata was able to detect this attack.

In terms of what is similar between this peer-reviewed conference paper and our project, it has tested the ability of Suricata to detect various attacks. However, in terms of difference, it has not considered the SDN environment. Also, it has not focused on DoS attacks. We will aim to build upon this paper with our experiments and implementation. Furthermore, We will aim to compare the results of this article with our experiments' results to understand deeper the Suricata abilities.

Their results show that every IDS has different abilities. Therefore, this project will explore Suricata further and compare its results with their ones.

4.2 Security in SDN

An article by Bawany et al.[22] provides an in-depth survey about SDN-based DDoS attacks detection to detect- DDoS attacks through using the new network paradigm, which is Software-defined networking (SDN). The researchers described the following mechanisms to detect the attacks:

- Entropy: It is used in IDS to detect anomaly because it can compute the randomness of the various network features.Therefore entropy-based algorithms are used in both traditional network and SDN-based one to detect DDoS attacks.
- Machine learning: Intrusion Detection Systems (IDS) uses machine learning algorithms to detect DDoS attacks. However, their performance depends on their training datasets.
- Traffic Pattern Analysis: It has an assumption which is that the behavioural patterns of the infected hosts are similar to other infected ones, but they are different from normal ones. Therefore it is used to detect DDoS attacks.
- Connection Rate: Connection rate-based mechanisms to detect the anomaly have two classes. The first class is the connection success ratio which means the normal host has more successful connection attempts than the infected one. The second one is the number of connections established, which means the infected hosts have many attempts to connect with too many different hosts in a short time. Therefore those mechanisms are used in SDN controllers to detect DDoS attacks.
- Integration of SNORT and OpenFlow: There are systems which combine OpenFlow and SNORT such as Snortflow so that the system can detect any intrusion and deploy countermeasures on the fly. Although the Suricata is more efficient than Snort, they did not find any SDN-based solution which uses Suricata. Therefore this project will explore Suricata in-depth in an environment which supports SDN to detect DoS attacks.

In terms of what is similar between this article and our project, it provides several mechanisms to detect DDoS attacks in an SDN environment. However, in terms of difference, it has not considered the Suricata in this environment. We will aim to build upon this paper with our experiments and implementation through testing the abilities of Suricata For detection DDoS attacks in the SDN environment.

4.3 A Study on SDN security enhancement using open source IDS/ IPS Suricata

This paper details preliminary research into defending against security attacks in an SDN paradigm using Suricata as its primary IDS. It is a white paper presenting no results or discussion beyond the theory, however it provides useful methodologies of attack prevention that we will be able to use in our project. The architecture that this paper bases these ideas upon is shown in Figure 7 and is something that we will look to implement in our project with the appropriate (discussed in section 5) tools. The security issues that this paper theoretical explores include:

- Firewall Implementation
- Network Scan Detection
- Abnormal Traffic Detection
- Brief discussion into Attack Prevention using an OpenFlow tool [23]

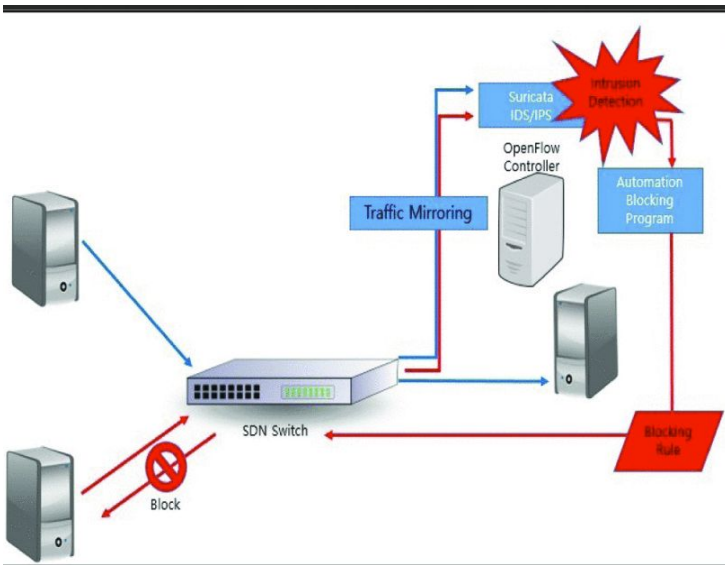

Figure 7. SDN architecture with Switch, Controller, Hosts and Suricata as IDS/ IPS [23, Fig. 3].

In terms of what is similar between this short paper and our project, it has effectively laid the theoretical groundwork for implementing advanced security detection patterns in the SDN paradigm as shown in Figure 7. However, in terms of difference as stated previously it hasn't provided any proper implementation details or experimental data. It also hasn't looked at how this system scales as hosts get larger and when there are many different sdn switches leading into the hosts of the system. We have built upon this paper with our experiments and implementation.

5. Methodology

5.1 Tools

5.1.1 Mininet

We have used Mininet as our experimental testbed for the experiments that we will run. Mininet is useful as it provides an instantly configurable virtual network whose API allows it to be easy to interact with and modify. Mininet also allows Suricata (through port mirroring) to be integrated within it and thus allows you to run various experiments with varying link parameters and topology design. For the purposes of this project, a custom tree topology was built with Mininet to which variations of this topology was used in the team's testing based upon the nature of these experiments.

5.1.2 Suricata

As explained previously, Suricata is an open-source intrusion detection system tool that much like the match-action paradigm of SDN switches, matches packets to rules and logs them in a file [13]. One important point about using Suricata is that it cannot be directly used on top of an SDN switch. Rather a port mirroring scheme like Figure 7. was used where Suricata is connected as another host to the switch and all the traffic coming inbound in the switch is mirrored (copied) to the Suricata as well through the network topology. Furthermore, due to this port mirroring, running Suricata in intrusion prevention mode did not work as it does not sit in the front of the network.

5.1.3 ONOS

The ONOS project is an open source SDN controller that is one of the most widely used in both personal and commercial use cases. It has been used in this project as an avenue for exploration to see if it is possible to integrate a port mirroring paradigm / Suricata with it. This exploration served to give this project and the team's experiments legitimacy as if it is already integrable with one of the most popular SDN controllers, people/ researchers will be more willing to adopt the framework. Integration is possible as ONOS provides a REST API which serves as a northbound interface to which an external scripting language can use to manage flows on switches. This is possible on any controller which offers a sufficient REST API which can perform flow management actions. However, ONOS was chosen as it is one of the most.

5.1.4 Python

A scripting language was required to manage flows on SDN switches as Suricata cannot be run in prevention mode. Python was chosen as it contains the most useful libraries and is a language the team is familiar with. Important libraries that were used in the methodology include:

1. requests - as it is the easiest way to communicate with the exposed REST API of the ONOS controller
2. json – allows the program to parse the raw json data which will be converted from raw text into a json object
3. os - Allows the program to open files and read text as well as well as run system commands
4. socket - allows the program to generate a socket UNIX file which Suricata can connect to

Furthermore, this python tool was also utilized to do more advanced analysis beyond the simple match action paradigm including techniques such as anomaly detection.

5.1.5 HPing3

HPing3 is a Linux based command line tool that was used to simulate DOS/Probing attacks. It features a wide variety of packet flooding and analysis options that the team used to benchmark the implementation with [24]. This tool was also used to simulate different types of DOS attacks including things such as ICMP flooding, SYN flooding as well as more sophisticated methods such as IP spoofing as well generating normal traffic.

5.1.6 Nmap

Nmap is a free and open source utility for network discovery and security auditing. The main use of Nmap in this project will be to simulate Port Scanning attacks. Nmap is easy to use as there are a number of command line options allowing the user to specify which ports of a network they want to attempt to probe. For example the user can try to scan 100 of the most commonly used ports (-F) or all ports (-p-) or just one specific port, e.g. port 80 (-p 80). When performing Port Scanning with Nmap, SYN packets are sent to the specified port(s) of the host. After a scan is complete, a set of results will inform the user whether the scanned ports are open (SYN ACK packets returned), closed (failure to connect will have RST packets returned) or filtered (unsure status due to blocking by network obstacle) [15].

5.1.7 Ovs - ofctl

Ovs-ofctl is an open flow switch management and configuration tool that works from the command line. It mainly works as administering and managing any open flow switch currently on the network. We have used this tool in this project to create port mirroring links, installing and removing rules and viewing the currently installed flows on the tools.

5.2 Methods and Approach

A standard methodology for testing and documenting results was developed to have consistency in the results. Firstly, a standard protocol for launching Suricata and setting up port mirroring was created and the team was briefed on how to launch and set up Suricata. The methodology that was used for iterating and generating a progressively more sophisticated system is presented here with the results discussed in Section 6.

1. **Setup and Connect Tools Together**

As discussed in Section 4.3 Kiho et al. provided a good foundational architecture for us to implement defense mechanisms, however it failed to elaborate on how the tools should integrate together. Thus, work was done exploring the best way to allow for the architecture presented in Figure 7 to be implemented. The Suricata option file allows for it to connect to a file socket and send alerts through said socket. This was a far better option than continuously polling a log file, in terms of computing resources and thus was used to connect Suricata to the Automated Block Program (ABP) written in Python.

Architecture Component	**Implementation**
Overall Network Virtual Environment	Mininet and ovs-vsctl
Traffic Mirroring	Using ovs-vsctl tool to delete ports and create a mirror on the appropriate interface
Suricata IDS Engine	Suricata IDS
Automated Blocking Program (ABP)	Scripted in Python
Blocking Rule / ONOS	Ovs - ofctl , onos

Table 3. Ways of Integrating tools for the project.

2. **Implement Topology**

As a modification from the progress report the topologies used have been simplified as it was decided to focus more on Suricata and the ABP itself rather than creating various different topologies. So overall in the results, some variation of the tree topology presented below in Figure 8 was used. However, the system and the ABP was built such that it was not contingent on one topology but so that it can be used on any network topology.

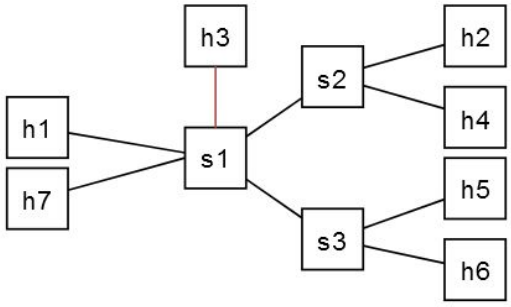

Figure 8. General Topology Used

3. Implement Attack Technique

Good attack simulation was a vital part of developing the system to defend against DOS attacks. There are many attack types that were simulated using HPing3 and Nmap. Briefly, these include:

a) **Syn Flooding** - where the attacker sends the server a series of SYN requests to use up resources on the host [25].
b) **ICMP Flooding** - This attack aims to overwhelm the server with ICMP or 'ping' packets. Excessive amounts of these packets in turn overload the server and cause a denial of service where the machine is unable to function normally [25].
c) **Teardrop Attacks** - This attack sends fragment IP packets with oversized payloads which may cause a crash of the host machine's operating system [25].
d) **Port Scanning Attacks** - This attack includes the attacker using a probe tool (e.g. nmap) to scan for open ports in a host. This process involves sending TCP SYN packets to each port of the host trying to be probed.

4. Implement Defense Technique/ Improve Integrability

In conjunction with our attack simulations, defence methods were also implemented. Detection of these were done with Suricata rulesets coupled with the ABP. The defense techniques implemented include:

a) **Rate Limiting** - Only allowing packets into the system at a certain rate. Useful for ip spoof attacks
b) **Signature Based Detection** - Using a match-action scheme to search for exact or approximate packet headers.
c) **Internal Spoof Detection** - Used to stop reflection attacks.
d) **Anomaly Based Detection**- which utilizes some statistical method to detect deviations from normal traffic to allow for predictions when anomalous packets arise.

On top of this, features that enhance the applicability of Suricata in real world contexts, have also been explored. This includes things such as integration with ONOS, and allowing the ABP to manage multiple instances of Suricata. While these are not directly related to defense techniques, they formed a key part of feasibility and strengthened the applicability of this method.

5. **Testing / Running Experiments and Collecting Results with Steps 3 and 4**

Basic experiments have also been run throughout the implementation. Each of these experiments examine their own set of metrics, and thus are each explained in their relevant sections.

6. Implementation and Results

6.1 Overview

In our project our major overarching outcome was bringing the theoretical concepts in the literature described in Section 4.3 to life. At a high level, we were able to set up a virtual SDN network in the form of a tree topology in mininet and perform various experiments. Suricata was then able to be integrated into the SDN framework as the IDS by running it on a specific host in the network. This host was hanging off the main switch of the network, in which all traffic passed throught. Hence, Suricata rulesets were able to be used to monitor and match traffic; block rules were installed on the main switch when these Suricata rules were matched.

6.2 Size of Topologies

Since real-life networks vary in size, we thought it was important that we allowed the variation of the network topology sizes we created for testing. As mentioned, in the Scope of the Project Definition, we narrowed the type of topology we looked at the tree topologies, as this best reflected a simulation of university networks. In building these topologies for experimentations, a python script was written to accept a custom number of hosts the user desired. With this input, a binary tree was built with hosts as leaf nodes and switches above it. While the functionality of this script allows for us to test various topologies due to the complexity of our other features tests were mainly done on one size of topology. However, these can be readily scaled with this tool greatly assisting in that effort.

6.3 ONOS Integration

Importance of the Contribution

As aforementioned, ONOS is one of the most popular SDN controllers used today. Thus a key goal of this project was to prove that integration with ONOS is feasible and possible in real world contexts. This is in comparison to the team's original and simpler approach of just using the ovs-ofctl to manage and dictate different flows and switches.

Outcomes and Implementation

To implement this, it was first evaluated whether ONOS' REST API was sufficient, secure and efficient. After reading through the documentation, the team deemed that it was sufficient in nature as it allows for adding and deleting flows, getting devices, and viewing current installed flows. ONOS also has the advantage of requiring authentication on call to its REST API which is embedded in each request, so with a secure password and username no unauthorized application can install flows. Allowing ONOS to manage switches and flows also adds the additional benefit of permanent flows that cannot be deleted. This makes addition of the reflected spoof protection discussed in 6.5.4, quite straightforward as these flows cannot be deleted . The implementation overall was successful and allowed for the same functionality that we had using pure system calls with the 'os' library. Some of the challenges in integrating our ABP included getting the right format to install and change flows as well as connecting to the REST API endpoints. However, these were able to be overcome and the implementation thus can be considered successful and feasible.

One of the key limitations currently in our approach is that ONOS device identifiers must be manually mapped to the switches. This is currently not an issue in smaller networks however having automatic device mapping would greatly improve the robustness of the ABP and ONOS integration.

Performance Evaluation

In order to verify that there is no sizable drop in performance (time to install rules) from using system calls (ovs-ofctl) to install flows vs using the ONOS REST API, a small experiment was devised between these two approaches. The experiment was set up as follows:

- Topology

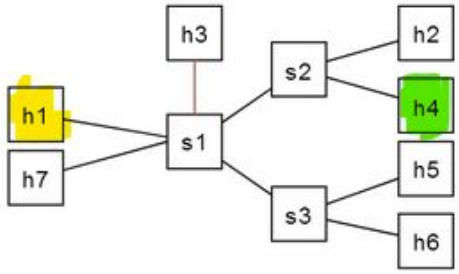

Figure 9. Mininet Topology Used For Onos Integration Testing

- Attacker (h1)
- Host (h4)
    - Using HPing3 , various speeds , 500 packets
- Testing : The performance rule blocking using ONOS vs System Calls
- Suricata Threshold : Sends an alert to the scripting language once it detects 100 packets within 10 seconds
- Measures: The delay between Python receiving the alert from Suricata and installing the rule (number of extra packets let through)
- Variables : Different rates of speed, on hping3, one packet every millisecond, 100 microseconds, 10 microseconds, as well the two different approaches for installing rules
- (#) of Times Experiment was Run: 10 times

Results and Discussion

In terms of the discussion, the results (see Table 4) are promising with the ONOS controller being on par with the system commands in terms of the speed of installing rules.

	Avg Packets Let Through After Threshold is Reached (#)	
(One Packet Every)	ONOS	System Calls
1000 microseconds	107	138
100 microseconds	157	140
10 microseconds	171	162

Table 4. Results For Different Rule Installation Schemes

The ONOS controller and the system calls have little difference even as the rates speed up to fast and faster. This is an important finding for the credibility of our system and means that ONOS integration is feasible without any drastic dip in the performance of installing, which could compromise the system.

One flaw (red cells in Table 4) that was found in the method using system calls is that to install a rule all previous rules had to be deleted on that switch. This is because the default mininet controller installs rules to switches at the max priority. So, when a drop rule had to be installed all other flows from the switch had to be deleted so that there are no conflicting instructions on the switch. However, this causes a problem as the rates get sped up, wherein if a packet arrives as the ABP deletes the flows but before the drop rule is installed then the blocking rule doesn't work as a flow rule is already installed. This is not an issue in the ONOS controller as it by default installs rules with a priority of 40000. Therefore, using a higher priority such as 41000, means that our blocking rule will always take precedence over the flow rule. This is a key advantage among others for using ONOS with our system. Furthermore, all other experiments found in this paper will also use an ONOS controller to ensure a working drop rule.

6.4 Scaling Suricata

Importance of the Contribution

Getting multiple instances of Suricata up and running is something that is important for the purpose of showcasing the feasibility of using Suricata in this SDN context. This is because most networks have more than one switch or interface to monitor that are external facing to the wider internet where possible. A key constraint of this approach before the following exploration was that it could only sit on one interface and monitor all traffic through only that interface. Thus it was imperative that we could get Suricata to scale to multiple instances and to do a preliminary test to ensure that having multiple instances of Suricata managed by the ABP did not hamper performance.

Implementation

A key advantage of connecting Suricata to the ABP through a socket is the fact that multiple connections can be achieved through that very same socket. So having concurrency through multithreading was the best way to manage these multi connections. In a similar way to how TCP servers spin-off new threads for incoming connections to handle new connections without blocking, the ABP is now able to handle multiple instances of Suricata running multiple threads. It does this by accepting an incoming connection and then starting a thread which handles this connection. This means while this thread is reading the data of Suricata the main python program is able to wait for another incoming connection. This can be seen in Figure 10 which showcases the implementation of this feature.

```
# Sets up unix file socket in eve.sock which is what suricata expects to connect to in the suricata.yaml file
# Sockets gives this program json which is parsed
server_address = "eve.sock"
sock = socket.socket(socket.AF_UNIX,  socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(server_address)
sock.listen(5)

threads = []
#Amount of suricatas before the main thread stops accepting new connectionSS
suricatasNo = 4

for i in range(suricatasNo):
    conn, addr = sock.accept()

    # Manges the suricata connected to the conn socket in this thread
    thread = SuricataConnection(i, "Thread-"+str(i), conn)
    # Start new Threads
    thread.start()
    # Add threads to thread list
    threads.append(thread)

for t in threads:
    t.join()

sock.close()
```

Figure 10. Implementation of the Multithreading in the ABP

Evaluation

As this project has mainly been about feasibility, a small scale feasibility analysis for this feature where we want to detect if there is any noticeable drop in performance while Suricata is scaling. Any drastic performance decline could be a potential constraint for this system. So it was decided to test this on a tree topology with 1, 2, and 4 instances of Suricata and with 1, 2, 4 attack all attacking the same host to see the average time it takes for the last attacker to be blocked from the system.

The experiment set up was as follows:

- **Topology**:

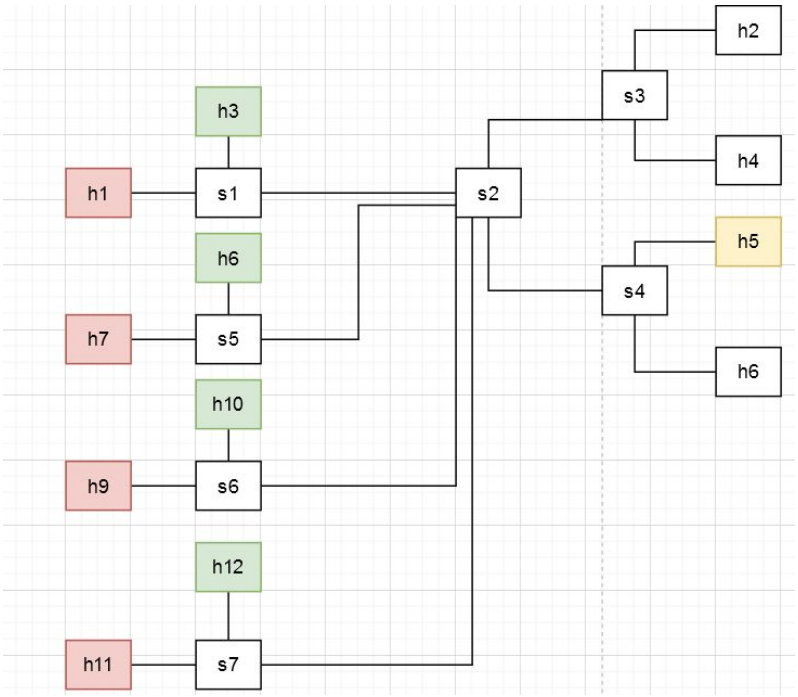

Figure 11. Mininet Topology

  - Red are attackers
  - Green are the hosts where Suricata is being run
  - Yellow is host being attacked
- **Attacker(s)** : h1, h7, h9 h11(with 4 Suricata); h1 and h7(with two Suricata), h1 (with one Suricata)
- **Host being attack** : h5
- **Measuring** : Difference between attack start time and time it takes to block the final attacker
- **Method:**
  - Have Suricata(s) connected to the ABP
  - Launch attack from relevant attacks simultaneously on 1, 2 or 4 of the hosts
  - Have ABP output when it installs a rule onto the switches
  - Record difference between attack start and attack end
- **(#) of Times Experiment was Run**: 10 times

Results and Discussion

The results after average for the 10 tests are given in the following Table 5:

Average Time to Block the last rule (milliseconds)		
1 Suricata (on h3)	**2 Suricatas (on h6, h10)**	**4 Suricatas (on h3, h6, 10, 12)**
91.2	**98.4**	**107.4832**

Table 5. Results for the Scaling Experiment

The results indicate that is a minor drop in performance as Suricata scales however it is unclear how much of this is the delay in actually trying to launch the attacks simultaneously with a single core virtual machine. As there is no clear statistically significant drop in performance as we have scaled this up in a small scale, we can conclude at least in small cases that Suricata scales on many different switches, with only one ABP required.

6.5 Implementation of Attack Prevention Techniques

6.5.1 Blacklisting and Suricata Rulesets

Overview

With Suricata, we ran it in IDS mode with given rulesets on when to alert if the network was under attack. There are various fields that can be filled out when specifying a Suricata rule; e.g. source/ destination IP, port, protocol, etc. see Figure 12. In our project we mainly stuck to rules with a threshold. This is where an alert will be created if within a specified time limit, the maximum number of packets in that threshold is reached. Since it was run in IDS mode, any attack that matched our rulesets would only be logged and alerted. When this alert is picked up with the traffic monitoring script, a drop rule then can be installed on the main switch of the network to blacklist the IP of the host sending those packets. Subsequently, in the future, this host is deemed as an attacker and any packets from its source is blocked. Suricata's rule sets are easy to define and can be added on demand and initiated through the suricat.yaml file.

```
alert icmp any any -> any any (msg:"Threshold Reached";sid:0; threshold:type threshold, track by_src, count 10, seconds 10;)
alert tcp any any -> any any (msg:"SYN Flood"; sid:1; threshold:type threshold, track by_src, count 10, seconds 10;)
```

Figure 12. Example Suricata Rulesets

Furthermore, as an extension, example datasets can be retrieved online to be used. These are datasets from previously run experiments specifying abnormal traffic. If these were used, further attacks can be detected if they happened to match the signatures of these datasets.

6.5.2 Anomaly Detection

Importance of the Contribution

Anomaly detection is a more complex form of attack detection that is based on statistics to determine whether a packet is malicious or legitimate. These sorts of models in actual networking security scenarios require a sizable set of sample data as well as time and resources to generate machine and deep learning models. However, the development of a simplistic anomaly detector implies more complex ones are able to be developed. Thus this is what is explored in this section.

As explained above the framework for this project incorporates the use of a scripting language to create the ABP which analyses the messages from Suricata and installs flow rules into the switch. This naturally extends to allow for anomaly detection as you have all the capability of the Python programming language. In this project, a three sigma rule anomaly detector was written, with some arbitrary sample mean and deviations. The three sigma states that if an observation is greater than 3 standard deviations above the mean then it should be flagged as an anomaly.

Implementation and Recommendations

In terms of the implementation of the three sigma rule with a hardcoded median and std deviation, the program counts the Suricata alert messages from the previous threshold (variable set in the code) . If it satisfies the three sigma then a blocking rule is installed. Initial testing of this system yielded a working prototype.

The success of the simple initial exploration implementation yields the fact that more sophisticated analysis tools are able to be in the ABP. Looking toward the future,models in sklearn and tensor flow can be developed as Suricata logs all messages in the .log file. This means that packet data is able to be obtained through this switch and models are able to be trained in machine learning. From here, more sophisticated models can be trained. This has the implication of being really powerful and flexible, much like the ethos of SDN itself.

6.5.3 Rate Limiting

Importance of Contribution

In instances where an IDS is not able to detect the source/sources of an attack, whether it be through IP fragmentation, IP spoofing or DDOS, it is still vital that a solution is made to mitigate the effects of the attack. One possible solution is rate limiting, which involves restricting the bandwidth to a targeted service to ensure it does not become overwhelmed with inbound network traffic. The drawback of such a solution is that it may affect normal non-malicious traffic by restricting the bandwidth or dropping packets. This may be affected to varying degrees depending on how it is implemented.

Implementation and Recommendations

A simple implementation of this solution would be to drop all inbound packets when the inbound traffic rate exceeds a certain threshold. This may result in a high amount of normal traffic impacted however it will ensure the targeted service is protected. In an SDN context, this can be implemented by setting an ingress policing policy on an OVS interface using the following commands shown in Figure 13.

```
$ ovs-vsctl set interface vif1.0 ingress_policing_rate=10000
$ ovs-vsctl set interface vif1.0 ingress_policing_burst=8000
```

Figure 13. OVS Policies

A more complex solution would be to implement a QOS policy, whereby flows are sorted into various classifications as set by the user. From this, a queue can be instantiated for each classification allowing for a max bandwidth to be set for each. This can allow traffic that is known to be legitimate to be unaffected while abnormal/suspicious traffic may be restricted but allows false-positive malicious traffic to still be transmitted. OVS allows for QOS by setting a QOS policy on each interface; each classification is sent to a queue with the ability to set individual max-rates. This solution is more effective as it does not drop any packets when the max-rate is exceeded. An example of this can be seen below in Figure 14.

```
$ ovs-vsctl -- \
  add-br br0 -- \
  add-port br0 eth0 -- \
  add-port br0 vif1.0 -- set interface vif1.0 ofport_request=5 -- \
  add-port br0 vif2.0 -- set interface vif2.0 ofport_request=6 -- \
  set port eth0 qos=@newqos -- \
  --id=@newqos create qos type=linux-htb \
      other-config:max-rate=1000000000 \
      queues:123=@vif10queue \
      queues:234=@vif20queue -- \
  --id=@vif10queue create queue other-config:max-rate=10000000 -- \
  --id=@vif20queue create queue other-config:max-rate=20000000
```

Figure 14. OVS QOS

Complexities arise when trying to implement QOS policies as sorting the network traffic into its classifications can be quite difficult and beyond the scope of this project. As a result, the solution to apply ingress policing policies was used to demonstrate such functionality. Although further work beyond this project could look at implementing a QOS policy solution.

6.5.4 Internal Reflection DOS Prevention

Importance of the Contribution

A reflected DOS attack is an attacked method where packets get spoofed to an internal IP in the network. Thus in the case of a flood all packets will get reflected to the IP to the spoofed internal source in its response. It is this where the name reflected attacks comes from as packets are reflected from one host and flooded to the next host. This is tricky to detect once the attack has begun as from the target of the reflected host it will look as if another host within the network is attacking it rather than from an external source.

Due to this reason, building this protection into our framework as a general prevention tool is easy to implement in our framework requiring little. It is important in protection that due to the flexibility of the ABP, openflow and onos, flow rules can automatically be installed at the start of the network lifespan.

Outcomes

The implementation of this mechanism was successful with the python code in Figure 15 and integration with Figure 16 with ONOS controller illustrating this success. The one caveat to this implementation is that it assumes internal IP's are known and all external facing ports are also known. However, this knowledge can be easily discovered through the ONOS API in a real world context or manually found by a network administrator.

In terms of crudely using command line tools with this reflected spoof protection is done in the manner below in Figure 15, where at the start of the program's launch it iterates through all outward facing ports on particular switches and installation drop rules, such that when a packet has an internal ip from an external port, it is dropped. The implementation below has been done using the Python's 'os' library running command line tools fig. However, this can also be integrated with an ONOS controller using the REST API described in section 6.3. The code for this is present in Figure 16 . ONOS also allows for flows to be permanently installed on switches which means that the rules only have to be installed once. The function described in Figure 16 does a similar enumeration to that of Figure 15, however it needs to use the particular POST format described in REST API documentation. This is done in the generateBlockingRule function where all the flows are added to the payload to post JSON structure. All these flows are bulk sent to the controller and installed on the corresponding switches ids.

```
internalIPs = {"10.0.0.2", "10.0.0.3", "10.0.0.4", "10.0.0.5","10.0.0.6"}
ingressPort = {'s1' : [1, 3]};
outFacingSwitches = ['s1'];

def reflectedSpoofProtection():
    for switch in outFacingSwitches:
        for port in ingressPort[switch]:
            for key in internalIPs:
                cmd = "sudo ovs-ofctl add-flow " + switch +
                    " hard_timeout=0,priority=65535,in_port="+str(port)+
                    ",dl_type=0x0800,nw_src="+str(key)+",actions=drop"
                os.system(cmd)
```

Figure 15. Implementation 1 of Reflected Spoof Protection Using Ovs-Ofctl

```
internalIPs = {"10.0.0.2", "10.0.0.3", "10.0.0.4", "10.0.0.5", "10.0.0.6"}
ingressPort = {"of:0000000000000001"] : [1, 3]}
outFacingSwitches = ["of:0000000000000001"]
def ingressPortRules():
    print("HERE")
    pay_load_to_post = {
    "flows": [

    ]
    }
    print("Installing Reflection DOS Protection")

    for switch in outFacingSwitches:
        for port in ingressPort[switch]:
            for ip in internalIPs:
                pay_load_to_post["flows"].append(generateBlockingRule(ip, permanent=True, id=switch, port= port))

    result = requests.post("http://127.0.0.1:8181/onos/v1/flows/", data = json.dumps(pay_load_to_post) , auth=('onos','rocks'))
```

Figure 16. Implementation 2 is when an ONOS controller is used

Evaluating This Methodology

As this method is something that is done at the start of the network lifespan there is no need to evaluate it beyond simple spoofed ping requests to ensure that the packets do not reach the spoofed target. Figure 17 shows the mininet topology that has been used in this short evaluation with Figure 18 show the wireshark logs of what happens at h4 when an attacker from h1 spoofs (changes its ip address) as h4 sends an ICMP packet to h5, h4 receives the flood of reply packets from h5. This has been tested through command ‘hping3 -1 -a 10.0.0.4 10.0.0.5’ launched from the h1 host. The -a flag does the spoof with the -1 flag making the packet an ICMP ping rather than a regular TCP packet. Figure 19 shows the same command launched after the reflected spoof protection function has been run on startup. Both the target and the reflected ip do not receive any packets meaning that the implementation is successful.

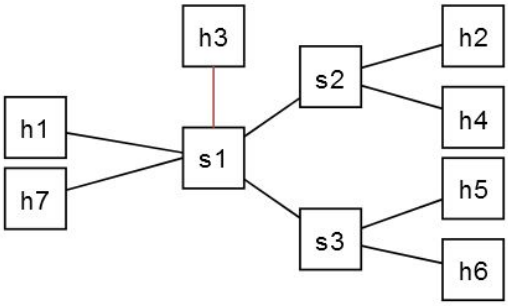

Figure 17. Mininet Topology used for ONOS vs Sys call testing

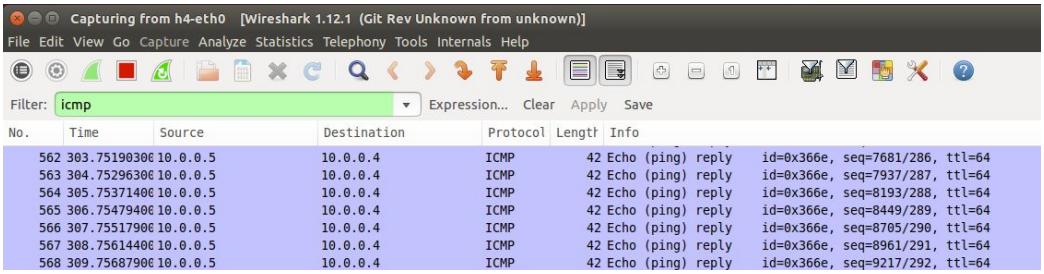

Figure 18. Wireshark capture without protection

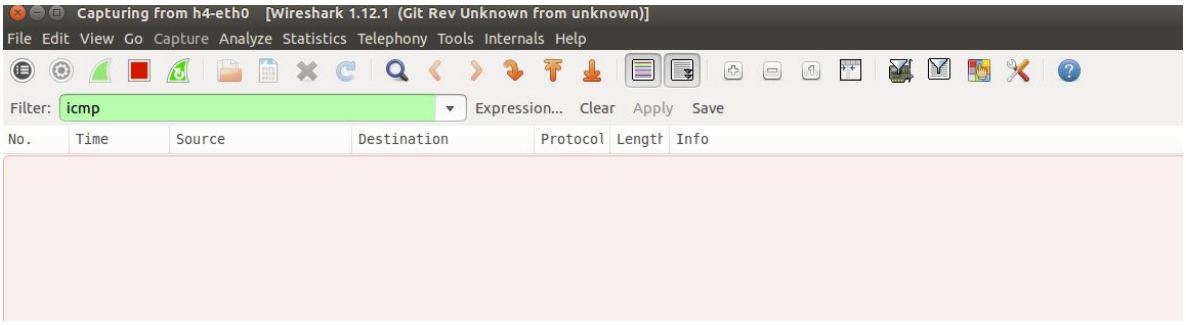

Figure 19. Wireshark capture with protection

6.6 Simulating Attacks, Results and Discussion

6.6.1 ICMP Flooding Prevention

An ICMP flood is the most basic for DOS attack where the attack continuously sends ping packets to the host filling the buffer and causing a performance reduction at best, and causing the host to crash in the worst case. In order to evaluate the usefulness of the using a blocking rule and to see how drastic of an effect it has on the performance of a network, the following experiment was devised:

- **Topology**:
  - Red are attackers
  - Green are the hosts where Suricata is being run
  - Yellow is host being attacked
- **Attacker(s)** : h1
- **Host being attacked** : h5
  - Using hping3, the host will spam packets using the flood option (as fast as hping3) possible can
- **Measuring** : The average of 25 pings from h4 (internal host) to h5, indicative of the responsiveness of the host and amount of network bandwidth available.
- **Method:**
  - Have Suricata on (h3) connected to the ABP
  - Have the attack set up
  - Have the ping set up from h4
  - Launch both simultaneously
- **(#) of Times Experiment was Run**: 10 times

The original mininet links were also changed to have a much smaller bandwidth (5 mbits) rather than a larger set of attack hosts(due computer resource restrictions), as they both serve the purpose of reducing the amount of available bandwidth for a request.

Results and Discussion

First as a baseline, the experiment was performed with an attack but no defense mechanisms. The average ping time for the first 25 packets was recorded. In addition to this a baseline was recorded of the average ping time from h4 to h5, when no ICMP flood is currently occurring. This is how in the blue line of Figure 20.

Figure 21 shows the response time increasing and increasing as the packets from hping3 overwhelm the attacker, causing an in. This line can feasibly be extrapolated and until the response time becomes so slow the host becomes unresponsive, unable to clear the backlog. This is further emphasized with the high average response times in each of 10 tests conducted in the grey line where no protection of the network exists. Furthermore, looking at the results when the Suricata rule is installed onto the attack, the Suricata

line and control line are almost identical. This indicates that the internals of the network have successfully been protected from the malicious ICMP taking up bandwidth. This is another important finding for this system as it shows that it can successfully defend against basic DOS attacks with a small amount of the effort. The ABP can also be much more sophisticated than just installing rules as discussed in the anomaly based detection section of this paragraph..

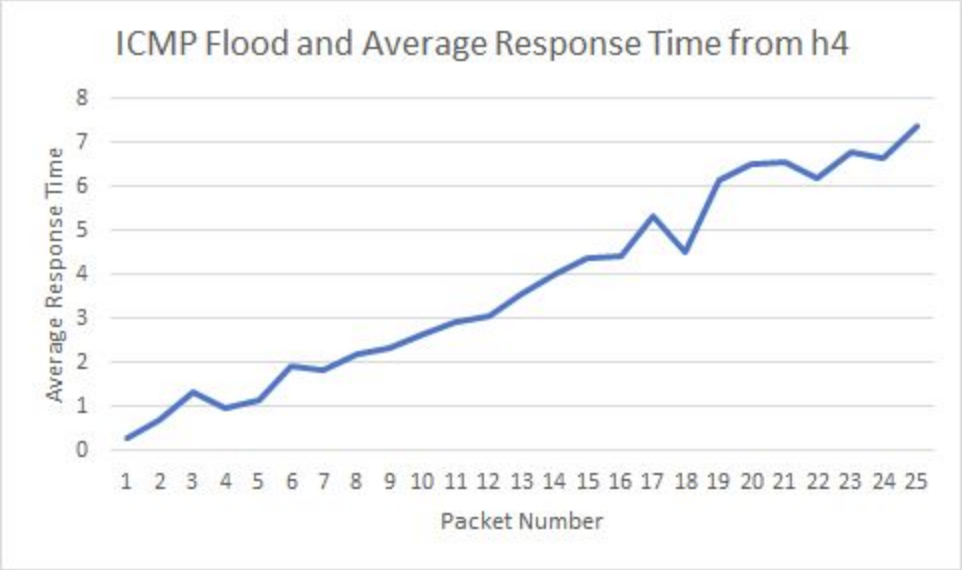

Figure 20. ICMP Flood Results with no Protection

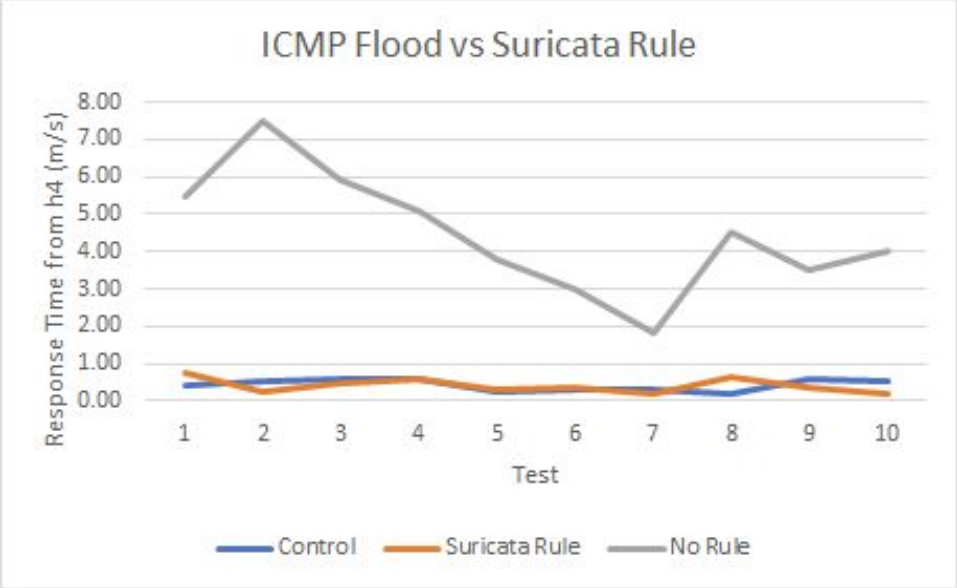

Figure 21. ICMP Flood vs. Suricata Rule

6.6.2 SYN Flood Recovery

The SYN flood attack works in a very similar way as an ICMP flood attack. However the goal of a SYN flood attack is often to block other external TCP requests from being processed by a host running a TCP protocol, most commonly HTTP. Most SYN flood attacks utilise IP spoofing, meaning if the network is to simply block the incoming packets from an attack, it would block packets from legitimate TCP requests too. Solutions to this problem exist, such as recycling of the oldest half open connection and SYN cookies, however these are implementations within the TCP server. For the application of Suricata as a defence against SYN Flooding, a focus on recovery speed after an attack was chosen. For the following experiment, the server is a simple TCP server with no such defence against SYN flood attack. This experiment aims to find the difference between the response time of the server to a legitimate TCP request, directly after a SYN flood attack, with and without Suricata using threshold blacklisting.

- **Topology**:

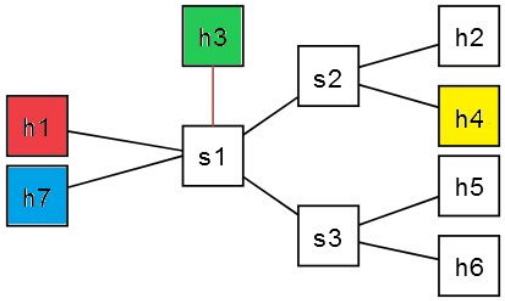

Figure 22. Mininet Topology

    - Red is the external attacker
    - Green is the host where Suricata is being run (the timeout for triggering the rule was set to 4 seconds)
    - Yellow is TCP host being attacked
    - Blue is the external client sending a legitimate TCP request
- **Attacker(s)** : h1
    - Using hping3, this host will send packets using the flood option (as fast as hping3) possible can, sending a minimum of 10000 SYN request packets.
- **Host being attacked** : h4
    - Running SimpleHTTPServer.py.
- **Host sending legitimate TCP request** : h7
    - Running a bash script seen in Figure 23.
- **Measuring** : The bash script seen in Figure 23 continuously queries the server, and records the times before and after each query.

- **Method:**
    - **Variable step**: Have Suricata on (h3) connected to the ABP.
    - Have SimpleHTTPServer.py running on (h4).
    - Start the bash script on (h7).
    - Start the SYN flood attack using hping3 on (h1).
    - Once the attack has completed, and the legitimate host is able to send and receive TCP requests from the server, extract the times correlating to the attack from the log.txt file.
- **(#) of Times Experiment was Run**: 20 times with Suricata running and 20 without

```
while :
do
        now=$(date +"%H:%M:%S.%N")
        curl 10.0.0.4
        now2=$(date +"%H:%M:%S.%N")
        echo "$now to $now2"
        echo "$now to $now2" >> log.txt
        sleep 0.5
done
```

Figure 23. TCP Request Bash Script.

Results and Discussion

The results for the experiment are as shown in Figure 24, with the following averages:

- Without Suricata rulesets and blacklisting: 65.28 seconds
- With Suricata rulesets and blacklisting: 23.82 seconds

As shown, it can be suggested that the process of blacklisting incoming TCP SYN packets, after the detection of a SYN flood attack, greatly decreases the time taken to resume normal operation of a TCP server.

This experiment was extremely small scale, with the SYN flood running for roughly only one second. However even on this small scale, the time difference in recovery between a network running Suricata as an IPS and a network without an IPS is substantial. It can be assumed that scaling this experiment up, would not significantly increase the recovery time of the network running an IPS, but would increase, likely in a logarithmic trend, the recovery time of a network not running an IPS. Unfortunately a larger scale experiment requires too much computing power, which is out of the realm of Mininets capabilities. The roughly 90 second variance in response time for the control tests (seen in Figure 24) is also likely due to the restricted capabilities of Mininet. However, a real, physical network would likely also have a similarly large variance, as network traffic is highly volatile, and subject to rapid changes.

Real world applications of this IPS setup include a network hosting a large number of individual, simple servers, which do not have the processing power required to conduct essential operations alongside defensive processes such as SYN cookies. By having this centralised IPS, whilst not allowing for continued operation during a SYN flood attack, guarantees a swift recovery, and minimises the likelihood that a server crashes as a result of the flood.

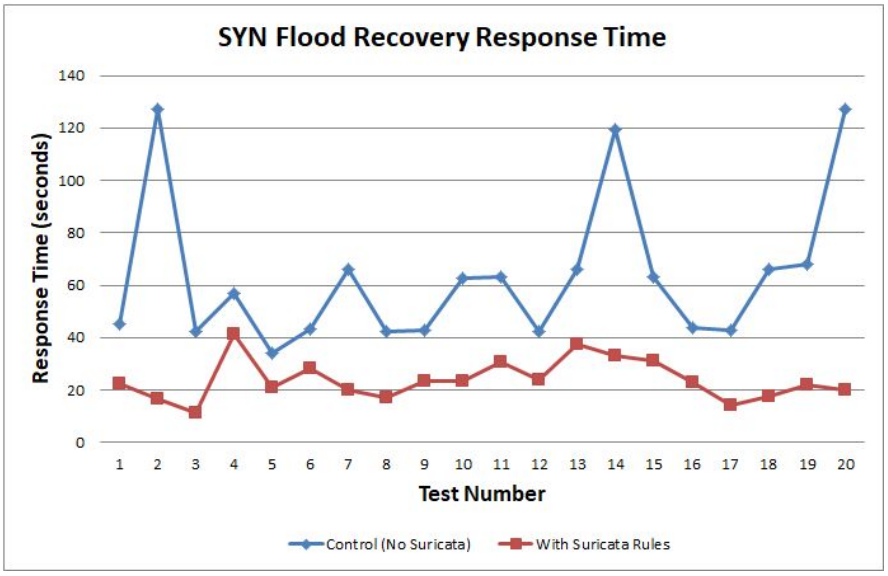

Figure 24. SYN Flood Recovery Times

6.6.3 Port Scanning

Nmap flag -F was used to simulate the port scanning attacks. This flag option was chosen as it attempts to probe 100 of the most common ports in a host to find out their open/ closed status. On average 100 ports scanned on our tree network topology with 6 hosts was around 14.6 seconds, see Figure 25. During this process, the attacker sent 2 SYN packets to the victim on the port attempting to be probed, see Figure 26. In response, due to protection of the SDN host, a RST packet was returned to the attacker indicating that the connection failed due to the port being closed. After this, the Suricata TCP threshold rule will be matched alerting the port mirroring monitor program; hence, a drop rule was installed on the main switch from accepting further TCP packets from the attacker's IP in the future. The average time for the drop rule to be installed was about 1.3 seconds.

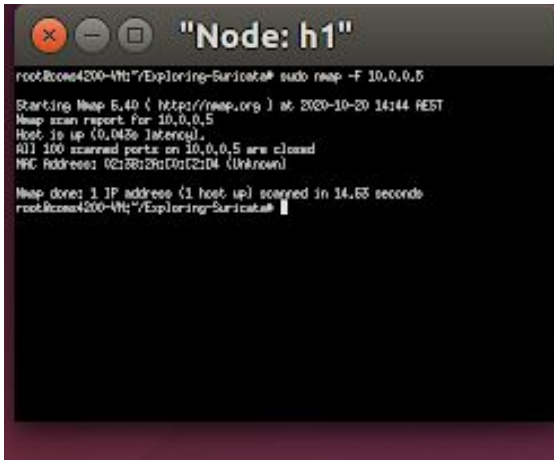

Figure 25. Running the Port Scanning Attack

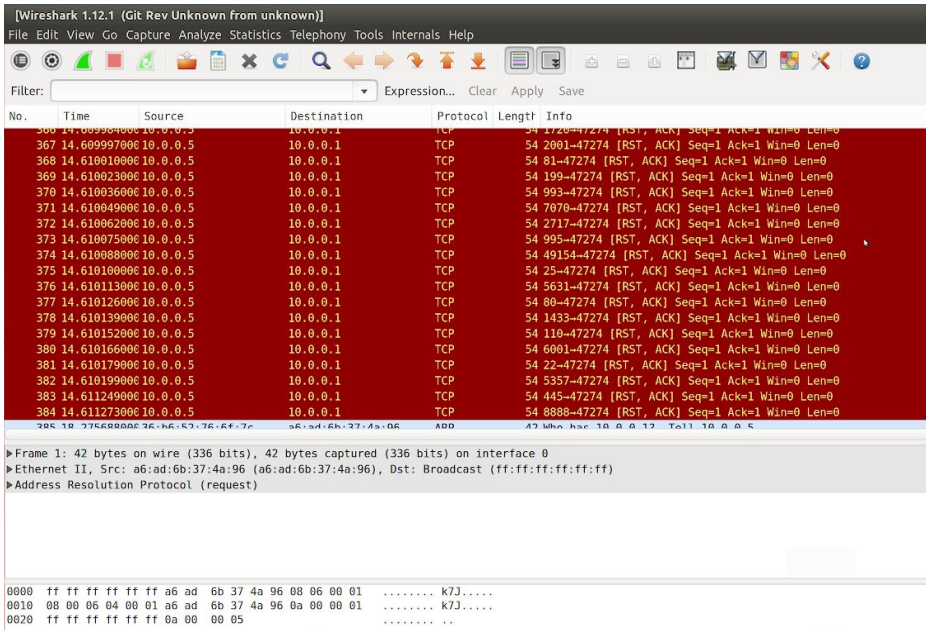

Figure 26. Port Scanning Attack with no protection

6.6.4 IP Fragmentation Attacks

The IP fragmentation attack was conducted as an extension of the ICMP flood attack, whereby the packets sent by the attacker will be larger than the network's MTU(1500 Bytes). This will be able to demonstrate Suricatas ability to identify the malicious fragmented packets and act accordingly.

Topology:
This experiment will use a simple topology as shown in Figure 27 below. Host h1 will be the targeted host, h2 will be the host running Suricata, h3 will be the attacking host and h4 will be the legitimate host. The Switch s2 is used to ensure the IP datagrams are fragmented prior to being received by s1 and inherently the suricata agent on h2.

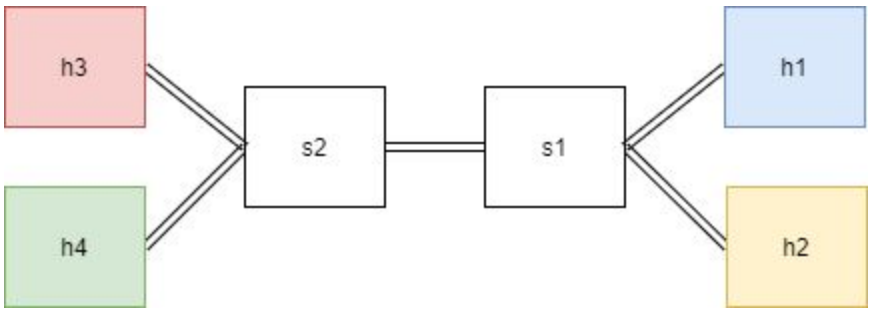

Figure 27. Topology used in Mininet for IP Fragmentation

Methods:

1. Instantiate h2 with Suricata
2. Begin a series of pings from h4 to h2 to simulate normal traffic and to observe a baseline for the experiment, this will be measured using the RTT of the pings
3. On h3 begin an IP fragmentation attack using hping 3, observing the change in the RTT of the pings from h4 to h2

Results and Discussion:
Through much testing, no significant experimental data was able to be obtained, this was due to the fact that when hping3 would transmit large IP datagrams it required a considerable amount of time. This in turn restricted the maximum throughput of the malicious traffic, hence rendering the attack ineffective. The results of this experiment would be considered as a limitation of using mininet on a low performance device and would require a more physical set up to conduct further experimentation

6.6.5 DDOS

The following experiment was used to simulate multiple hosts DOS attacking a single web server.

Topology:
As shown in Figure 28 below, there are 4 hosts (h2 - h5), each being an attacker. These are each connected to their own switch in order to provide isolation; simulating the attack residing from multiple networks. The host h7 will have suricata running on it and h1 will be the targeted web server. Host h6 will act as a legitimate user attempting to access the web server.

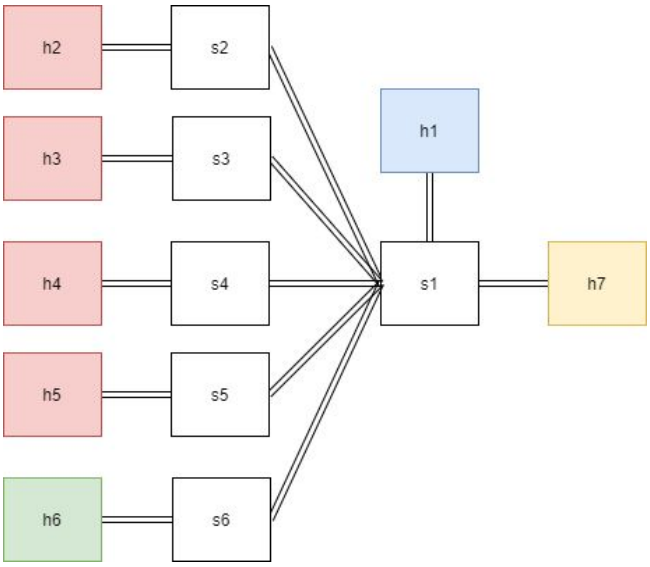

Figure 28. Topology Used for DDOS Attack Simulation

Methods:

1. Instantiate h0 with Suricata and h1 with a simple http web server
2. Generate intermittent HTTP GET requests to h1 from h2-h7 to observe normal traffic metrics, measuring the response time of the server to h7.
3. Stop the GET requests from h2-h7 and instead begin a TCP flood attack using hping3 from each host, again measuring the response time of the server to h7.

Results and Discussion:
It was identified in this experiment that hping3 would not be able to run concurrently on multiple hosts. And as such would require a sequential locking mechanism to gain access. Although this appeared to be a viable solution, the sequential mechanism heavily restricted the throughput of the attacker, rendering this experiment ineffective. It would be recommended that further experiments are made on a physical system to account for such limitations of using a virtual network.

6.7 Summary

Overall, what we discovered during our project can be easily applied to a legitimate real-world context. Since we operated in an SDN environment, any changes to the network in terms of size and functionality can be easily configured and programmed. Furthermore, with using Suricata, rulesets can also be quite easily added or changed to suit the situations and attacks wanting to be detected and prevented. As there is a lack of research for security measures and attack prevention in OpenFlow, our set up is innovative and can easily be adapted and scaled up where Suricata could be running on multiple hosts in a large network to monitor mass traffic and protect important enterprise networks from common attacks. Furthermore, the security defense mechanisms that can be developed with the use of a general purpose scripting language is also a huge advantage for this system, as it is flexible and adaptable. As we found in our breadth-first analysis this language is able to connect to any controller, perform rate limit and blacklisting based on Suricata rulesets as well more sophisticated attack prevention tools such as anomaly detection with machine learning methods being very feasible.

7. Conclusion

The main goal of the project was to investigate the use of Suricata as an IDS/ IPS in an SDN context. The motivation for the project mainly came from researching the evolution of SDN networks, the prominence of attacks and hence, implementing measures to mitigate such threats on these networks. As this project was carried in a university environment, we critically decided that it was best to limit our project scope to investigating common DOS attacks on tree network topologies to best simulate the surrounding situation.

Overall, the main outcome and contribution of the implementation side for this project was bringing to life the theoretical concepts in the literature presented in Section 4.3. This process mainly included, setting up an SDN tree topology in mininet. Suricata was then run on a host in IDS mode in the topology. With a port mirroring Python script running, all traffic passing through the main switch on the network was able to be analysed. Rulesets were utilised to specify the type of traffic that was deemed as potential attacks. Once all the set-ups were complete, various DOS attacks were performed on the network using 2 hosts with one as the attacker and one as the victim respectively. Tools such as ping, hping3 and nmap were used to send packets and simulated attacks (SYN flooding, ICMP flooding, teardrop fragment attacks, port scanning/ probing). As mentioned, from the output of the port mirroring of the main switch, when traffic matched the rulesets defined in Suricata, an alert was received. With this alert, flow rules were installed on the main switch in the network to block future packets from the attacker. This was done with command line tool ovs-ofctl and the ONOS interface. The results from the experimental simulation showed that Suricata was an effective tool to use for SDN network management and protection.

A limitation of the project was that the architecture that we used was quite low level and required high proficiency in Python programming; however, over time with further adoption more general and higher level tools can be developed. Another limitation was false-positive detection. In order for this to be done properly it requires high level anomaly based detection methods, which is only viable given a sizable dataset is gathered; this was beyond the scope of this feasibility experiment due to time constraints and the small size of the experiments carried out.

Although this project brought many theoretical concepts to life, there is much future work that could be undertaken. Mainly, providing abstractions to allow for more widespread adoption. This would ensure the set-up used in this project would be able to be adapted to any network; including being able to customise IDS/ IPS behaviour on larger networks; hence addressing the limitations mentioned previously. A lot of the work that we have done is quite static and a lot of hard coded parameters exist in our code. This makes modification from one topology to the next difficult as there are many parameters that need updating. Making our northbound interface program able to automatically detect switches and links as well setting up Suricata automatically in future before widespread adoption can feasibly occur. Moreover, due to time constraints, we were not able to look deeply into running Suricata in IPS mode. This would definitely be looked at for future work, in regards to applying scalability of our system on similar large scale university/ enterprise networks. Doing this could potentially save resources on the port mirroring side of our project implementation where Suricata would automatically drop malicious packets when rules are matched. Running this along with IDS logging, would create an extensive dataset to be continuously used in the future.

8. References

[1] N. Perlroth, “Attacks On 6 Banks Frustrate Customers”. NY Times, 2012. Available at: https://www.nytimes.com/2012/10/01/business/cyberattacks-on-6-american-banks-frustrate-customers.html

[2] O. N. Foundation, "Software-defined networking: The new norm for networks," ONF White Paper, vol. 2, no. 2-6, p. 11, 2012.

[3] R. Maaloul, R. Taktak, L. Chaari, and B. Cousin, "Energy-aware routing in carrier-grade ethernet using sdn approach," IEEE Transactions on Green Communications and Networking, vol. 2, no. 3, pp. 844-858, 2018.

[4] N. McKeown et al., "OpenFlow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69-74, 2008.

[5] M. Team, "Mininet Overview - Mininet", Mininet.org, 2020. [Online]. Available: http://mininet.org/overview

[6] "Use offense to inform defense. Find flaws before the bad guys do.", Cyber-defense.sans.org. [Online]. Available: https://cyber-defense.sans.org/resources/papers/gsec/host-vs-network-based-intrusion-detection-systems-102574

[7] J. White, T. Fitzsimmons and J. Matthews, "Quantitative Analysis of Intrusion Detection Systems: Snort and Suricata", in The International Society for Optical Engineering, 2013.

[8] "Intro to Information Security", Udacity.com, 2020. [Online]. Available: https://www.udacity.com/course/intro-to-information-security--ud459

[9] V. Nagadevara, "Evaluation of Intrusion Detection Systems under Denial of Service Attack in virtual Environment," Faculty of Computing, Blekinge Institute of Technology, Karlskrona Sweden, 2017.

[10] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher, “Understanding a Denial of Service Attack”, Internet Denial of Service: Attack and Defense Mechanisms, 2005. [Online]. Available: http://www.informit.com/articles/article.aspx?p=386163&seqNum=3

[11] Mesalab.cn. 2020. [online] Available at: https://mesalab.cn/managercode/achievements/2013%20Defending%20Against%20SYN%20Flood%20Attack%20under%20Asymmetric%20Routing%20Environment.pdf

[12] T. Gangte, “SYN Flood Attacks- How to protect?”, Hakin9, 2020. [Online]. Available: https://hakin9.org/syn-flood-attacks-how-to-protect-article

[13] "What is probe? - Definition from WhatIs.com", SearchSecurity, 2020. [Online]. Available: https://searchsecurity.techtarget.com/definition/probe

[14] García-Teodoro, P, Díaz-Verdejo, J, Maciá-Fernández, G, & Vázquez, E. “Anomaly-based network intrusion detection: Techniques, systems and challenges”, Computers & Security, vol. 28(1-2), pp. 18–28, 2009.

[15] "Nmap: the Network Mapper - Free Security Scanner", Nmap.org, 2020. [Online]. Available: https://nmap.org/

[16] "DDoS attacks increase by 151 percent in first half of 2020", Continuitycentral.com, 2020. [Online]. Available: https://www.continuitycentral.com/index.php/news/technology/5557-ddos-attacks-increase-by-151-percent-in-first-half-of-2020

[17] M. Pihelgas and R. Vaarandi, "A comparative analysis of open-source intrusion detection systems," Tallinn: Tallinn University of Technology & University of Tartu, 2012.

[18] J. S. White, T. Fitzsimmons, and J. N. Matthews, "Quantitative analysis of intrusion detection systems: Snort and Suricata," in Cyber sensing 2013, 2013, vol. 8757: International Society for Optics and Photonics, p. 875704.

[19] E. Albin, "A comparative analysis of the snort and suricata intrusion-detection systems," NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 2011.

[20] T. Ernawati, M. Fachrozi, and D. Syaputri, "Analysis of Intrusion Detection System Performance for the Port Scan Attack Detector, Portsentry, and Suricata," in IOP Conference Series: Materials Science and Engineering, 2019, vol. 662, no. 5: IOP Publishing, p. 052013.

[21] O. Bouziani, H. Benaboud, A. S. Chamkar, and S. Lazaar, "A Comparative study of Open Source IDSs according to their Ability to Detect Attacks," in Proceedings of the 2nd International Conference on Networking, Information Systems & Security, 2019, pp. 1-5.

[22] N. Z. Bawany, J. A. Shamsi, and K. Salah, "DDoS attack detection and mitigation using SDN: methods, practices, and solutions," Arabian Journal for Science and Engineering, vol. 42, no. 2, pp. 425-441, 2017.

[23] K. Nam and K. Kim, "A study on sdn security enhancement using open source ids/ips suricata," in 2018 International Conference on Information and Communication Technology Convergence (ICTC), 2018: IEEE, pp. 1124-1126.

[24] "hping3", Tools.kali.org, 2020. [Online]. Available: https://tools.kali.org/information-gathering/hping3

[25] "Types of Network Topology in Computer Networks | Studytonight", Studytonight.com, 2020. [Online]. Available: https://www.studytonight.com/computer-networks/network-topology-types