Queen’s University Belfast

School of Electronics, Electrical Engineering and Computer Science

ELE8095 Individual Research Project

Project Title: A Comparison of a P4-SDN DDoS Defence Solution with ML-based DDoS Defence Solutions

Student Name: Breanainn Cahill
Student Number: 40342205
Academic Advisor: Dr. Sandra Scott-Hayward

9th September 2022

Declaration of Academic Integrity

Abstract

This research delves into the topics of Software-defined Networking (SDN), the Programming Protocol-independent Packet Processors (P4) language and Distributed Denial of Service (DDoS) attacks to look at how SDN and programmable data planes can be used to combat such attacks. State-of-the-art research in these areas were examined and with inspiration from the research, a novel approach to detecting Transmission Control Protocol (TCP) SYN flooding attacks was created.

This DDoS detection approach was tested with a dataset that had also been used in testing machine learning based detections approaches. By using the same dataset the different approaches can be compared. One of the goals of this research was to use P4's programmable data plane to produce an effective DDoS mitigation system and compare the effectiveness of such a system against the more complex approach of machine learning.

This work examined the state-of-the-art research consisting of SDN, DDoS types, DDoS detection systems, machine learning detection systems and multiple P4 TCP SYN flooding specific detection solutions. This led to the creation of a novel DDoS detection system that was tested to provide results that were examined to create a report that adds to state-of-art research.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would first like to thank my supervisor Dr. Sandra Scott-Hayward for the help and guidance that was provided throughout the project. Her comprehensive explanations and quick replies to my various questions were fantastic. The help that she provided was very valuable with regards to the completion of this dissertation.

I would also like to thank my partner Larissa for her continued support and putting up with me throughout the project. I would also like to thank my family and friends for their wise words of encouragement and support throughout the project.

1. Introduction

1.1 Introduction

Each passing year, it seems more and more of us depend on the internet. From big organisations to individual people, the internet is almost a necessity to function in today's society. Alongside the growing popularity of the cyber world, is the growing popularity of cyber abuse. Everyone has grown accustomed to receiving warnings from organisations about potential phishing emails, hearing about an organisation's failure to secure data or learning their favourite web service is unavailable. In today's world every organisation should be concerned about the three pillars of the CIA triad [1].

The CIA triad stands for confidentiality, integrity and availability. Confidentiality is concerned with keeping data secure and private. Determining who has access and making sure only the correct users have access is the objective of this part of the triad. Integrity is concerned with making sure data has not been manipulated. This aspect of the triad is about the trustworthiness of the data and making sure the data is consistent and as intended.

The final pillar of the triad is availability. Availability is the idea that data should be accessible when it is needed. Organisations can lose out on profit for every moment their services are inaccessible and such an attack can result in the loss of trust from their customers. If an attack such as this becomes a regular occurance and the organisation cannot defend against the attacks then it is unlikely that this organisation will survive. This research addresses a type of availability attack known as the Distributed Denial of Service (DDoS) attack.

1.2 Distributed Denial of Service Attacks (DDoS)

A Denial of Service (DoS) attack has the aim of overwhelming the victim with requests to the point where the victim can no longer respond [2]. A DoS attack normally comes from one source which means two things, tracing the attacker is easy and one computer is simply not enough to overwhelm the servers in today's world. DDoS attacks have the same goal as DoS attacks but these attacks come from multiple sources and contain spoofed internet protocol (IP) addresses. These multiple devices are normally controlled by one individual and are called botnets. Botnets are an army of devices that a malicious actor uses to carry out an attack.

DDoS attacks can be traced back to the 90s and were originally used by hackers as a form of protest against certain changes made on the internet. One of the earliest known cases was in 1996 where a DDoS attack was used as a protest against an internet service provider (ISP) over the introduction of a spam filter to their email service [3]. Following this, it was realised that such attacks could be used to attack commercial businesses. These attacks could be performed by anyone, from a disgruntled customer to competitors looking to get ahead. These commercial attacks are commonplace in today's world with examples such as the github attack of 2018 or the amazon web service (AWS) attack of 2020 [4]. Both of these attacks involve traffic levels of over a terabyte a second with the AWS attack reaching 2.3 Tbps.

DDoS attacks have also become popular for political reasons. Since they deny access to information they are used in cyber warfare. The biggest example of this is in the case of Estonia where a DDoS attack was used to deny the country access to information [3]. DDoS attacks are also performed by hacktivists or used as an act of terror, with the distiction between the two groups becoming quite unclear as the act itself may be celebrated by one group and critized by another group.

DDoS attacks also come in many different forms, which range in how the attack is performed, the exploitation that is used, or how the botnet that is being used is built. All of these complexities are discussed in detail during the literature review. The DDoS attack has been around for nearly three decades and is still causing issues for organisations throughout the world. It is a problem that demands attention as it is an attack that is not going away anytime soon, with some companies reporting DDoS attacks occurring every 20 minutes or so [3].

DDoS attacks are a complicated form of attack that can be difficult to defend against and require complicated methods to detect and defend against. This research will provide insight into the difficulties of defending against such attacks and how some of the features found in software-defined networking (SDN) may help to provide defence.

1.3 Software-defined Networks (SDN)

In a traditional network, routers are connected and use protocols to learn the details of their neighbours so that they can operate together. Each network device could be considered to have a fixed function which means that the functions that these devices perform cannot be changed. These networks are quite difficult to manage as they require a huge amount of protocols and are slow to evolve due to each vendor making proprietary software which means that the network operator relies on the vendor for updates. At one stage, the routers used by ISPs became so complicated that they contained 100 million lines of source code to support the hundreds of protocols needed for the network to function. [5]. Years of testing were also required to ensure that these protocols can operate as expected [6]. These routers were also considered to be power hungry, unreliable, expensive and hard to secure. SDN is about giving the network operators more control over their networks.

The control that SDN gives the operator comes from one of the core concepts of SDN which is the separation of the control plane and the data plane. The control plane is concerned with managing traffic throughout the network while the data plane is concerned with the forwarding of a packet on a particular switch or router found within the network. By separating these planes the network operator no longer has to make changes on each individual device found on the network and instead can use the control plane to configure all the different devices found in the network.

The control plane provides a centralised perspective of the network which in turn gives the network operator full view of what is happening inside the network. This leads to advantages in troubleshooting issues in the network. SDN also provides advantages when it comes to scalability. SDN allows the operator to extend resources as needed, which is quite useful in defending against unknown attacks or in times when traffic levels increase.

SDN networks were originally operated using the openflow protocol which allows operators to configure the data plane. This meant that the data plane devices could still be considered to have a fixed function and the need for a programmable data plane was realised. With a programmable data plane the network operator can program their network devices to do exactly what is needed, which in turn produces performance advantages and can allow network operators to produce their own solutions to issues relating to their network. The Programming Protocol-independent Packet Processors (P4) language provides this ability and is used in this research to produce a solution to a form of DDoS attack.

1.4 Programming Protocol-independent Packet Processors (P4)

The P4 language introduces a programmable data plane to SDN networks which provides the controller of the network with complete control of how the data plane device operates. This allows the network operator to build a network that contains devices that perform the exact functions that are necessary for each particular device. This leads to a performance increase of the data plane devices.

P4, like most programming languages, requires a compiler. The P4 compiler needs two files to operate. The first of which is the architecture file. This file maps a logical pipeline to the physical pipeline to create an abstraction [7]. The second file is the program file that allows the programming of the pipeline.The logical pipeline allows the same P4 program file to be used on multiple types of hardware.

Each pipeline contains the same stages which are the parser section, ingress section, egress section and the deparser section [7]. They may also contain a checksum section for error checking to ensure the integrity of the data. The parser part of the pipeline is responsible for getting header information from the various protocols contained in the packet. For example, IP headers contain details such as source and destination IP addresses which are used to make forwarding decisions. The ingress and egress processing is concerned with making such decisions based on the information taken from the various headers. Match-action tables are used by these sections to make the forwarding decisions. A match-action table is where one of the various elements found within the headers is matched with an element in the table and that element is then associated with an action. Once a match is confirmed, the action takes place. The difference between ingress and egress is that ingress processing is done on the way into the device while egress on the way out of the device. The final element is the deparser which simply puts the headers and any changes made back into the packet before sending toward the next destination. With the flexibility and control that P4 provides in controlling the data plane, it is the obvious choice for the development of a DDoS defensive system.

1.5 Research Question

The aim of this research was to investigate P4 based DDoS mitigations and to determine the benefits and drawbacks of such a system compared to a machine learning based system. Machine learning algorithms provide high detection rate and low false positive rate which makes its use very attractive. P4 on the other hand allows the developer to create their own innovative solutions that may be useful in detecting forms of DDoS attacks.

Due to the complexity of machine learning algorithms, one of the questions that arose from this investigation was whether such complexity was necessary for network level attacks that involve protocols such as TCP or UDP. TCP SYN flooding was the attack that was chosen for this investigation.

1.6 Research Objectives

The objectives of this research were to:

- Study the state-of-the-art literature on P4 based mitigations for DDoS attacks
- Identify a P4 based solution for TCP SYN flooding attacks that is suitable for use
- Find an appropriate dataset that has been used to test machine learning algorithms to allow for comparison
- Create a test environment using SDN mininet that is suitable to perform testing on
- Implement the chosen solution inside the test environment
- Evaluate the performance of the solution inside the testbed
- Compare the solution with a machine learning based solution to understand the differences between the approach with regards to performance and security

During the testing of the chosen solution it was discovered that the solution did not work as described. Revisiting the TCP SYN flooding literature led to the creation of a new solution which led to the objective of implementing and testing the novel solution. This objective arose out of revisiting the objective of identifying another P4 solution.

1.7 Approach

A brief outline of the different sections found within this dissertation.

Introduction - Introduces the main topics of the dissertation, the research question and the research objective.

Literature Review - The literature review describes the current state-of-the-art literature on the topics of DDoS detection and P4. This section reveals how the direction of the research was formed.

Methodology - This part of the dissertation describes some of the research decisions that were made and the reasoning behind them. It contains the details of the development of the testbed, reveals how the dataset was chosen and how the DDoS detection solution was chosen.

DDoS Detection - A discussion on DDoS detection, which describes why the initial solution to the DDoS detection issue was discarded, the novel solution and the various testing that was performed to test its performance.

DDoS Defence - This section describes the implementation of the DDoS defence system, the testing of the mechanism with regards to thresholds and throughput performance. Followed by a discussion on some of the limitations of the research.

General Discussion - In the final section of the dissertation, machine learning is initially discussed in detail before making the comparison with the novel solution. Future work is also discussed here alongside the final conclusion to the work.

2. Literature Review

2.1 Effective DDoS Detection and Defence and Why it is Required

DDoS attacks are becoming more common, as the amount of devices with internet access increases and technology becomes more advanced there are more targets and new threats found in the cyber security landscape. Recent developments in technology over the last decade or so, have led to an increase in DDoS attacks. These technologies are the internet-of-things (IOT) devices and virtual machines (VMs). Both of these technologies have a huge amount of benefits to users but like everything in our world, they can be used for virtuous purposes or for evil purposes, depending on who is in control of them.

These technologies can be considered as the reason for the increase in DDoS attacks because both can be used to create a botnet. A botnet is what can only be described as an army of devices. By taking advantage of IOT devices' lack of security, malware can be produced to infect each of these devices. This malware can then sit dormant until instruction is received from the controller of these botnets, because of these sudden changes in the behaviour of the devices that are infected by this kind of malware, they are often referred to as zombies. A botnet with a size of 400,000 was discovered in 2008, which would lead one to imagine the size of botnets that could be found in today's world [8]. The creation of such a botnet is a slow process that requires a large amount of technical skill, meaning this kind of botnet would require a huge amount of planning and time before enough devices are infected. VMs on the other hand make the creation of botnets much easier.

VMs can also be used to create a huge amount of low memory machines that can operate as a botnet. These botnets can now be bought on the dark web and can be used by people with less technical skill which makes performing such an attack easier for threat actors. These purchasable botnets are commonly referred to as malware-as-service which is a play on the names of various cloud service models that are provided to customers [9]. While VMs have lowered the barrier of entry into the world of cybercrime, they also provide one of the only known ways to survive zero day attacks [8]. A zero day attack is a previously unknown attack. In the case of a DDoS attack, this could be the novel abuse of a protocol or an application fault. By using VMs to extend resources as needed, these attacks can be overcome but the cost of this high volume of traffic will be left with the organisation.

DDoS attacks come in many different forms and involve the abuse of the protocols that allow the internet to function. Firstly we will discuss two ways that DDoS attacks are performed. A direct attack is where the controller of the botnet simply tells the zombies to attack the victim [2]. This means that tracing the source of the attack is possible with the help of ISPs. Reflective attacks on the other hand are where routers are sent packets with a spoofed source ip so that the responses are sent to the victim [2]. These attacks are much more difficult to detect. The need for cooperation from the ISPs to detect the source of the attack makes it quite difficult to track down and blame a group or individual for an attack, meaning there is little repercussion for most people who perform DDoS attacks. This reveals some of the urgency that organisations should have in respect to defending their networks.

Three defence approaches can be used in the DDoS attack mitigation [8]. The first of which is the proactive defence mechanism which involves cloud computing and extending resources when needed. As discussed, this ultimately provides a defence to unknown or zero-day attacks. The second method, which is post attack analysis, is performed after the attack to gain information about the details of the attack. These details are then provided to an Intrusion Detection System for use against further attacks. The final approach is the reactive defence mechanism. This mechanism is used to detect and stop DDoS attacks as they happen. This mechanism is the most desirable but is also the most complicated.

The question that arises is how would an effective detection and defence mechanism be described? In paper [10], a list of desirable traits for such a system is provided. The first of which is that legitimate traffic should not be affected by the mechanism, meaning the defence mechanism should only drop the DDoS traffic. It should also prevent attacks coming from both the exterior of the network and attacks within the network. The performance of the network should remain consistent and not be slowed by using the mechanism. The ability to scale the network also must not be affected by this mechanism and it must be cost effective so that many organisations can implement the defence approach.

Some further important qualities are that it is robust and adaptive which is the ability to work in multiple environments. It should feature low false positive and high detection rates which simply means that it should accurately identify attack traffic. Finally the last requirement is that such a defence mechanism should come with support from the developer to help troubleshoot issues with the mechanism.

The production of such a system to meet the various requirements presents many challenges and overall the literature provides a detailed account of the necessity of an effective DDoS defence mechanism. The stand out question here is whether there is a solution that can be easily implemented to provide protection from the most common forms of DDoS attacks.

2.2 Approach to Software-Defined networking and DDoS detection

Software-Defined Networking (SDN) is an approach to the implementation of networks, in short it is about network operators taking back control of how the network operates [5]. As discussed in the introduction, the core idea is the separation of the control and data plane. The control plane determines how the network as a whole should operate while the data plane determines how individual packets are treated as they arrive on a network device before being forwarded toward the next node or target. SDN allows the administrator to manage the network from a centralised perspective which introduces some advantages and disadvantages in securing the network from attacks. SDN’s central management provides the network operator the ability to view and control the network which in turn provides advantages in DDoS detection. Traffic analysis is one such advantage, by providing a view of the whole network the controller can analyse traffic patterns to determine if there is abnormal traffic [11]. Software-based traffic analysis can be added to the controller which can use machine learning algorithms to detect and react to attacks. Dynamically updating forwarding rules is another advantage that can provide a reactive defence to DDoS attacks.

SDN also introduces disadvantages. The first of which is a single point of failure, meaning that if the controller goes down so too does the network [5]. To combat this issue, one can distribute the centralised control plane over a number of servers [5]. The link between the channels can also be considered an attack vector. DDoS attacks where a lot of packets are sent to the controller along the southbound channel can cause the controller to fail due to CPU exhaustion or congestion on the link [12]. A solution to both these problems is to have the data plane to handle all forwarding of packets and use static calls to the controller [13]. After examining the pros and cons of SDN in the literature it led to the belief that source based DDoS detection on the data plane would be the preferred option for a DDoS defence in SDN.

A source based DDoS mechanism is where the attack is mitigated as close to the source of the attack as possible [11]. Compared with a destination based mechanism, which is where the attack is dealt with at the target, it offers more advantages. By using the source based mechanism the attack is mitigated at the point of entry of the network [14]. This has the advantage of stopping the traffic of the DDoS attack from entering your network and there should be less overhead in the southbound channel [15]. Source based defence also has the benefit of saving on cost because the traffic doesn't get the chance to enter your network. A network based solution where the controller performs an analysis on the traffic and performs deep packet inspection is also an option but the southbound channel overhead issue comes back into play in this case.

During the analysis of the literature, it was noticeable that many different framework ideas were proposed to combat DDoS attacks using fixed function pipelines [16][17][18]. These ideas were implemented using the openflow protocol. Fixed function pipelines are SDN switching chips that can be configured using the openflow protocol [19]. The issue with these frameworks is that they are platform specific and each vendor may have different specifications, meaning that these solutions do not suit all networks. To produce a source based data plane solution to DDoS attacks that can be used in many situations, there would need to be a programmable data plane, not a configurable dataplane. This is where the P4 language is a necessity.

In the more recent literature, P4 solutions are more common than openflow solutions. This is simply because by reprogramming the data plane, solutions to issues such as DDoS attacks can be created and shared. P4 solutions cannot be run on any device but can be run on any device with the same logical pipeline. As discussed, a logical pipeline maps to the physical pipeline by creating an abstraction. The idea is that eventually the same logical pipeline will be used so that P4 can work similarly to the Java virtual machine idea of write once and run anywhere [19]. With this in mind, P4 seems to be extremely useful for combating many forms of attack, due to the fact that it gives anyone who is interested the ability to deploy and test their ideas.

After the examination of the state-of-the-art literature from the previous few years, it is clear that a source based DDoS solution that has been programmed using P4 is the obvious solution for an effective DDoS defence solution.

2.3 DDoS detection methods with a focus on machine learning

There are two types of DDoS attacks that target different layers. Attacks on the network/transport layer focus on the use of protocols such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP) or Internet Control Message Protocol (ICMP). The goal of these attacks is to overload the network bandwidth of the victim. Application layer attacks on the other hand are used to overload the servers with application specific requests that require a large amount of processing [11].

Both UDP flood attacks and ICMP flood attacks can be considered as volumetric attacks [20]. A volumetric attack is a type of DDoS attack that tries to overwhelm the victim with packets. In a UDP flood attack the victim normally responds with an ICMP message to let the sender know that no such service exists. ICMP flood attacks rely on an echo message that is sent to a server with a spoofed source IP address and can be described as a reflective attack. Solutions to these forms of attacks are related to limiting ICMP packets and blocking UDP packets from certain locations.

The most popular form of attack is the TCP SYN flood attack which has been used in large scale attacks where the TCP SYN flood traffic share is 92 percent [21]. As it is used at such a high rate just defeating this form of attack should provide the network with adequate defence. A TCP SYN flooding attack is where the attacker sends a SYN message to the target server and the server responds with an SYN/ACK message, which means the connection is now half open. In normal operation of the protocol a third message is sent from the host to confirm, so that they may now start sending data using higher level protocols. In the attacker's case this ACK message is never sent, which creates a lot of open connections. Too many open connections will overload the server. As this attack is so common, the proposed solution will target this form of attack.

There are many methods used to detect DDoS attacks:

- **Entropy** is where probability distributions of attributes such as IP source address, IP destination address or port numbers are measured against an entropy value to determine if there is anomalous traffic [10].

- **Traffic Pattern Analysis** works off the basis that botnet traffic or traffic controlled by one user should mean their traffic should exhibit similar behaviour [10].
- **Connection Rate** is where the ratio of successful connections versus unsuccessful connection is measured and used to determine if there is an attack [10].
- **Machine Learning** uses algorithms to learn from datasets or real-time traffic, how to determine whether an attack is happening.

For the purposes of targeting a TCP SYN flood attack, the form of detection that seemed most appropriate is connection rate. This is as a TCP SYN flood attack should have a high level of SYN/ACK messages compared to ACK messages and lead to the detection of an attack [22]. The aim was to provide a much simpler detection method compared to machine learning. This chosen solution was discarded as testing proved that it was not as useful as described.

Throughout the literature, machine learning methods are deployed to detect DDoS attacks and provide a very high level of detection rates [23][24]. That being said they also introduce a level of complexity where the user needs to train the algorithms using datasets or real-time traffic. Deep-learning is a form of machine learning that is deemed too slow for use and requires large amounts of training data. Use of several machine learning techniques is suggested in [25], to gain a higher level of accuracy. With such complexity it seems that using machine learning to detect network level attacks such as TCP flooding is excessive. That being said, for other forms of attacks such as application layer or for economic denial of sustainability, it is much more suited as these attacks are a lot more difficult to detect.

The research gap that is notable from the literature is the use of a practical solution that works out of the box on network level attacks that has detection rates similar to machine learning detection rates. To test this idea, a solution that uses a connection rate mechanism to detect attacks coupled with a defence mechanism was tested against a dataset. Further details on the selection of such a solution is provided in the P4 solution selection section. The solution that was chosen did not prove to be useful in detecting these forms of attack so a novel solution was developed. This new solution also revolves around the connection rate of various TCP flows. It tracks the progress of each flow and determines if a TCP flow is abnormal. The details of the solution are described in detail with regards to detection and defence in the latter sections.

3. Methodology

3.1 Testbed

The tools used to create and run the testbed are mininet, python, P4 and tcpreplay. Mininet is a tool that allows the user to build a virtual network [25]. The set up of this network can be programmed in python and is what is used in this case [26]. P4 is the language that will program the switch with the DDoS defensive solution and tcpreplay is a tool that will be used to replay the traffic dataset [27].

To create a P4 based SDN mininet network, the P4 official website provides links to their github that provides python scripts which are used to create P4 specific switches and hosts [28]. P4RuntimeSwitch is used to create a switch that can receive runtime instructions from the controller such as reprogramming the data plane. The p4runtime_lib.simple_controller is used to send such instructions from the controller. These files further rely on some other scripts that have been provided to help set up a P4 specific network.

The testbed was built on a VM and is a Ubuntu linux distribution. The virtual machine image was downloaded from github as suggested by the mininet website [29]. A graphical interface was added to the operating system (OS) of the VM to help with the use of wireshark. Hping3 was installed to test the correct function of the network [30]. Hping3 allows the user to send a large amount of packets from one source to another.

The network itself contains just two hosts and a switch. The two hosts are called h1 and h2. During the testing of the solution, h1 was designated to send attack traffic while h2 responded with the victims traffic. The one switch found on the network was named s1 and was running the compiled P4 solution. To allow the traffic to flow through the network the p4 file was edited so that traffic coming from port 1 would be forwarded to port 2 and vice versa.

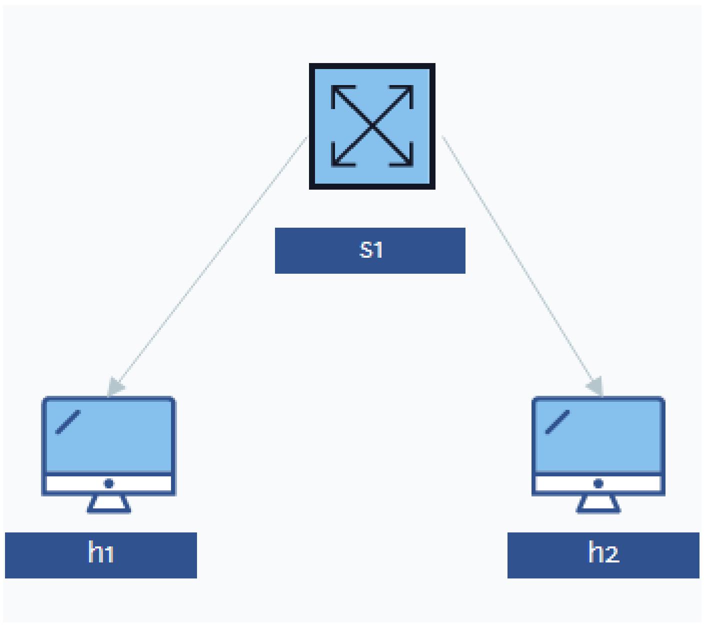


Figure 1: Network Diagram

runNetwork.py is the script that was created to build the P4-based mininet network/testbed. Before discussing the script itself, a few of the files that it depends on will be discussed first. The first of which is the P4 file itself. For the setting up of the testbed, the tutorial file, basic.p4 was used. This file was compiled into a json file alongside the creation of a P4 info file, both of which are for use in the switch and placed in the build directory of the testbed directory. This basic.p4 file was replaced by the solution. The pcaps and logs directories contain pcaps files for use with wireshark and logs files from the network.

topology.json is a file that is used in the building of the network to create two hosts and is displayed in full in Figure 2. If the testbed featured more than one switch then this file would be used to create the switches as well, but since there is only one switch found in the network, it is set up inside the runNetwork.py file. This file contains the details of the host such as IP address and mac address, and also two commands that are used once the hosts are running. One command sets up the default gateway and the other informs the switches arp table.

```
"hosts": {
"h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
       "commands":["route add default gw 10.0.1.10 dev eth0",
                   "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
"h2": {"ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
       "commands":["route add default gw 10.0.2.20 dev eth0",
                   "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]}
}
```

Figure 2: topology.json

The s1-runtime.json is a file that is used to set up the switch at runtime. It provides information such as the target device type, the P4 info file and the compiled json file which is the compiled version of the P4 program. Part of s1-runtime is displayed in Figure 3. The target type is the behavioural model version 2 (BMv2), which is a software switch that has been developed for the purposes of testing, developing and debugging [31]. It does not have the same performance as switches that are deployed in real time environments but is perfect for testing on. s1-runtime.json also informs the switch of the tables that are used in P4 for match action purposes. Match action tables are used to decide what to do with the packet that is received, for example it reads the IP address and matches it to the table to determine what port to send the packet to.

```
"target": "bmv2",
 "p4info": "build/basic.p4.p4info.txt",
 "bmv2_json": "build/basic.json",
 "table_entries": [
     {
     "table": "MyIngress.ipv4_lpm",
     "default_action": true,
     "action_name": "MyIngress.drop",
     "action_params": { }
     },
     {
     "table": "MyIngress.ipv4_lpm",
     "match": {
     "meta.port": 2
     },
     "action_name": "MyIngress.ipv4_forward",
     "action_params": {
     "dstAddr": "08:00:00:00:01:11",
     "port": 1
     }
```

Figure 3: s1-runtime.json extract

Now back to the runNetwork.py file which is the script that is used to set up the network. To start we will discuss the main function, this function does two things, creates the NetworkRunner object and following this, the object's method createNetwork() is called. The network object creates the needed variables that are used in the creation of the network and the createNetwork() method uses these variables to create the network and does some final setting up of the hosts and switches before creating the mininet command line interface (CLI).

Figure 4 displays the NetworkRunner object code. In short the NetworkRunner object when created, sets up variables such as the path to the pcaps and logs directory. It also sets the path to the topology.json file and the basic.json file. Using the topology file it creates a host object which contains the details needed to set up the hosts. Finally it creates the topology of the network using a constructor called SwitchTopo.

```
class NetworkRunner:
        def __init__(self):
        #Set up path variables for log, pcaps, switch type and topology
        cwd = os.getcwd()
        self.logs = os.path.join(cwd, 'logs')
        self.pcaps = os.path.join(cwd, 'pcaps')
        self.compiledP4 = os.path.join(cwd, 'build/basic.json')
        self.switchType = 'simple_switch_grpc'
        self.topology = './topology.json'
        self.logfile = self.logs + '/switch.log'
        #Read from topology file
        with open(self.topology, 'r') as file:
                top = json.load(file)
        self.hosts = top['hosts']

        self.topo = SwitchTopo(self.switchType, self.logfile, self.compiledP4, 9090,
self.pcaps, self.hosts)
```

Figure 4: NetworkRunner function extract from runnetwork.py

SwitchTopo inherits from the Topo class which is an object that mininet uses to build its network. When created it adds the switch, two hosts and the links between the hosts and the switch to the topo object, the details of this are displayed in Figure 5.

```
class SwitchTopo(Topo):
	def __init__(self, sw_path, log_file, json_path, thrift_port, pcap_dump,
hosts, **opts):
	Topo.__init__(self, **opts)
	#Add switch to topo object
	self.addSwitch('s1', sw_path = sw_path, log_file = log_file, json_path =
json_path, thrift_port = thrift_port, pcap_dump = pcap_dump)

	#Set up hosts and links with the switch
	for host_name in hosts:
		self.addHost(host_name, ip=hosts[host_name]['ip'],
mac=hosts[host_name]['mac'])
		self.addLink(host_name, 's1')
```

Figure 5: SwitchTopo class extract from runnetwork.py

The last method createNetwork which is displayed in Figure 6, is then called which creates the mininet network using the previously created variables and starts it. Following this the hosts commands are used to set up the default gateway and the arp tables on the switch. Then the switch receives the s1-runtime.json file to set up its configuration and finally the CLI is created so mininet can be interacted with.

```
def createNetwork(self):
	#Create Mininet and start it
	self.net = Mininet(topo = self.topo, host = P4Host, switch =
P4RuntimeSwitch,             controller = None)
	self.net.start()
	sleep(1)
	#Run commands from topology file on hosts
	for host_name, host_info in list(self.hosts.items()):
		current_host = self.net.get(host_name)
		if "commands" in host_info:
		for cmd in host_info["commands"]:
			current_host.cmd(cmd)

	#Use P4Runtime to do final config on switch
	switch_object = self.net.get('s1')
	grpc_port = switch_object.grpc_port
	device_id = switch_object.device_id
	runtime_json = 's1-runtime.json'
	with open(runtime_json, 'r') as sw_conf_file:
		outfile = '%s/s1-p4runtime-requests.txt' %(self.logs)
		p4runtime_lib.simple_controller.program_switch(
                     addr='127.0.0.1:%d' % grpc_port,
                     device_id=device_id,
                     sw_conf_file=sw_conf_file,
                     workdir=os.getcwd(),
                     proto_dump_fpath=outfile,
                     runtime_json=runtime_json)

        sleep(1)

        #start CLI
        CLI(self.net)
```

Figure 6: createNetwork function extract from runNetwork.py

Once the network was up and running it was tested with hping3 to determine correct functionality and the pcaps files generated were compared to check correct operation. To view the code in full of either s1-runtime.json or runNetwork.py, see Appendix A and B.

3.2 Dataset Selection

Choosing a dataset was a difficult task, as it needed to be a dataset that was used to test machine learning DDoS detection in other papers. The biggest issue that was found was that many papers used datasets that were created by the authors or the dataset that was used was not mentioned. A dataset that was used in one paper was considered for a time, but since this dataset was not found in any other relevant papers it was discarded [32].

Following this the dataset CIC-DDoS2019 was considered [33]. This dataset was also discarded as it was only found in one paper and the dataset seemed to only have attack traffic and no response traffic from the server. While this dataset could have been split to create responses from the server, it also only featured SYN messages from the one source IP. Ultimately it was not suitable for the testing of the solution.

The solution that was decided upon is the CAIDA "DDoS Attack 2007" dataset [34]. This dataset was used in multiple papers to test machine learning DDoS detection mechanisms [35][36]. It also features traffic from both the attacker and the server which should be ideal for testing the solution.

3.3 Solution Selection

To select a solution that was suitable for the test, many different TCP SYN flooding detection and defence papers were investigated. SYN cookies or SYN authentication are some examples of mitigation techniques. In paper [21], both these methods are used and compared, in both cases an allowlist is used once the user completes either test. The issue with the use of the allowlist is that requests are not dropped that come from potential malicious users, so in the case of cookies, the switch will send many cookie replies to SYN messages.

SYN cookies are also used in another use case [37]. In this case SYN cookies are used to authenticate the user and place them into an allowlist. After this process a calculation of the SYN to SYN/ACK ratio is calculated for each source IP and if it is above a predetermined threshold it is blocklisted. The last SYN/ACK message from the server in this case is edited so that it becomes a RST message. This message is then sent back to the server and used to reset the connection on the server, which frees the connection from the server.

The use case that was most desirable was similar to the previous method with its use of a ratio to determine attack traffic, but without the use of SYN cookies [22]. This case uses the tracking of FIN/ACK and SYN/ACK ratio to decide if there is an attack. Once there has been an attack it measures the ratio of FIN/ACK to SYN/ACK messages for each particular IP address and adds them to a blocklist. If the attack is coming from a set of IP addresses, once a certain threshold is reached it will create a group of IPs. For example if 140.116.245.1, 140.116.245.2 and 140.116.245.3 are considered attack sources, they will be combined into the IP subnet of 140.116.245.0/24 to block a set of IPs.

Since no source code was provided, the solution had to be built and tested by following the description in the paper closely. This solution was also discarded after testing and the reasoning behind this will be discussed in detail during the next section.

4. DDoS Detection

4.1 Initial Solution

A TCP SYN flooding attack involves the abuse of the three way handshake. The attack sends a TCP packet with the SYN flag and the victim replies with a SYN/ACK packet. In normal operation an ACK message is sent to complete the handshake. Following this, the data is transferred and the communication normally completes with the sending of a packet with the FIN/ACK flags active. In an attack situation the malicious actor will send the initial SYN message and never respond to the victims SYN/ACK message, which means the victim will send multiple messages with the SYN/ACK flags active and get no response. From the reading of this description, the idea that detecting such an attack should not require a complex algorithm arose, which in turn led to the focus on a solution that was simple to understand and implement.

The initial solution was to count the appearance of two types of flags and compare them. The idea was to use the ratio of the SYN/ACK to FIN/ACK flags to determine if there is an ongoing attack. Due to the descriptions provided on how a TCP flood occurs, one would imagine that during an attack the level of SYN/ACK packets should greatly outnumber the amount of FIN/ACK packets. This solution is described in paper [22], and it was chosen due to its simplicity and to determine if such a simple solution could produce adequate results and be used to detect and defend against TCP flood DDoS attacks. The paper describes using the overall ratio of SYN/ACK to FIN/ACK flags found in all packets to determine if there is an attack ongoing. Once it is determined there is an attack, the flags are counted for each flow and if there are many more SYN/ACK packets to FIN/ACK packets, the users IP address is placed into a blocklist. While the paper provided a great description of how this detection mechanism would work, the implementation of the mechanism was not provided.

Before implementing the entirety of the solution, the ratio differences between benign traffic and attack traffic had to be investigated by first implementing part of the solution in SYNACKCounting.p4. To test whether or not this solution was viable, the idea to count each instance of SYN/ACK and FIN/ACK packets was formed. Since the traffic sets are quite large, using wireshark was not an option. Therefore to perform this task a P4 program was developed. To perform this test two constants were added to the P4 program. These constants which are displayed in Figure 7, contained the hexadecimal value that is used to identify these flags.

```
const bit<6>  SYNACK = 0x12;
const bit<6>  ACKFIN = 0x11;
```

Figure 7: Constants extract from SYNACKCounting.p4

Following this a P4 counter was chosen as the best choice for counting these flags. Originally the variable type of register was used, but due some difficulties with its use in this particular situation, it was discarded. There are a couple of distinct differences between these data types. Firstly P4 counters only allow the program to increment its value, where registers allow all types of mathematical functions. Secondly the P4 program cannot read from P4 counters but can read from registers. P4 counters can be easily read by P4 runtime whereas registers are much more difficult to interact with using P4 runtime. So to implement this test a P4 counter of index size 4 was set up so that it could count the number of SYN/ACK and FIN/ACK messages and the direction in which the message was sent. As expected 0 SYN/ACK messages came from the attacker.

```
counter(4, CounterType.packets) AttackFlagCount;

    if(hdr.tcp.isValid()) {
      if(hdr.tcp.ctrl == SYNACK){
             if(meta.port == 1){
                    AttackFlagCount.count(0);
             }
             else{
       AttackFlagCount.count(1);
       }
     if(hdr.tcp.ctrl == ACKFIN){
            if(meta.port == 1){
             AttackFlagCount.count(2);
            }
          else{
             AttackFlagCount.count(3);
            }
```

Figure 8: Code extract from SYNACKCounting.p4

Figure 8 shows part of this P4 code implementation. The code is quite simple to follow, it basically checks if it is a TCP packet and following this, checks which flag it is and the port on which the packet arrived on. Then it simply increments the specific counter that is related to it. To read these counters a change was made to the runNetwork.py file which simply sets up a connection to the BMv2 switch before reading from the counters.

```
sw = p4runtime_lib.bmv2.Bmv2SwitchConnection(address='127.0.0.1:%d' % grpc_port,
device_id=device_id, proto_dump_file='%s/s1-p4runtime-requests.txt' %(self.logs))
 for response in sw.ReadCounters(self.p4info.get_counters_id('AttackFlagCount'),
0):
             for entity in response.entities:
             counter = entity.counter_entry
             print("SYNACK to victim:   %d" % (counter.data.packet_count))
 for response in sw.ReadCounters(self.p4info.get_counters_id('AttackFlagCount'),
1):
             for entity in response.entities:
             counter = entity.counter_entry
             print("SYNACK from victim:   %d" % (counter.data.packet_count))
```

Figure 9: Reading p4 counters extract from runNetwork.py

The code extract in Figure 9 shows the setting up of the BMv2 connection before reading from the counters. This code only displays the retrieval of two of the counters but it is the same process for the retrieval of the other counters. It was placed after the creation of the mininet command line interface (CLI) so that the details of the counters are displayed after exiting the mininet CLI and before the teardown of the network.

The traffic found within the dataset was split into 5 minute intervals with the first couple of sets containing benign traffic and the rest attack traffic. After running some tests to determine which sets of traffic contained an appropriate level of TCP packets, traffic numbers 1 and 3 were selected as the benign traffic set samples and 6,8 and 10 were selected as attack traffic set samples.

Table 1: SYN/ACK to FIN/ACK packet comparison

Traffic Number	SYN/ACK	FIN/ACK
1 (Benign Traffic)	3530	5690
3 (Benign Traffic)	1287	2156
6 (Attack Traffic)	2511	4312
8 (Attack Traffic)	0	0
10 (Attack Traffic)	1	4

After running the tests, the results in Table 1 showed that there was not much difference between the SYN/ACK and FIN/ACK ratios of the traffic. These results show that it is unlikely that this method could be used to detect an attack. Attack traffic number 8 did not even contain any packets with the either flags even though it did contain plenty of TCP attack traffic. Traffic set 10 also contained a very low amount of either packet type. Ultimately this solution was not feasible as it did not produce the desired results. This solution may work in some particular situations but overall does not seem to be effective. Appendix C displays SYNACKCounting.p4 code in its entirety.

4.2 Novel Solution

Once the initial solution was deemed unviable, the previous papers were examined. The TCP cookies solution was revisited but due to the fact that it introduces more traffic to the network and the fact the solution is quite popular with a lot of research on it, this solution was discarded. The search for a solution then led to the discovery of paper that tracked the number of unusual handshakes and used an entropy value to determine if there was an attack [38]. This detection mechanism consists of three modules. A collection module, a decision module and a monitoring module. This system works by getting information about different TCP flows by collecting packets with certain flags and other information such as connection latency. The decision module then uses the information to calculate an entropy value. This value is then compared to a threshold value to determine if there is an attack. While this solution seemed extremely promising, the fact that no implementation of the method was found and that the complexity level seemed too high for the idea of a simple TCP flood detection system led to this solution also being discarded. It did however provide inspiration for the development of the novel solution.

The paper [38], mentioned above provides a comprehensive list of the different types of unusual handshakes that may occur. This list showed that TCP flood attacks can vary and be much more complex than just tracking the ratio of SYN/ACK to FIN/ACK messages. The belief that there must be something that differentiates a legitimate flow from an attacker's flow prevailed, and led to the examination of TCP flow diagrams.

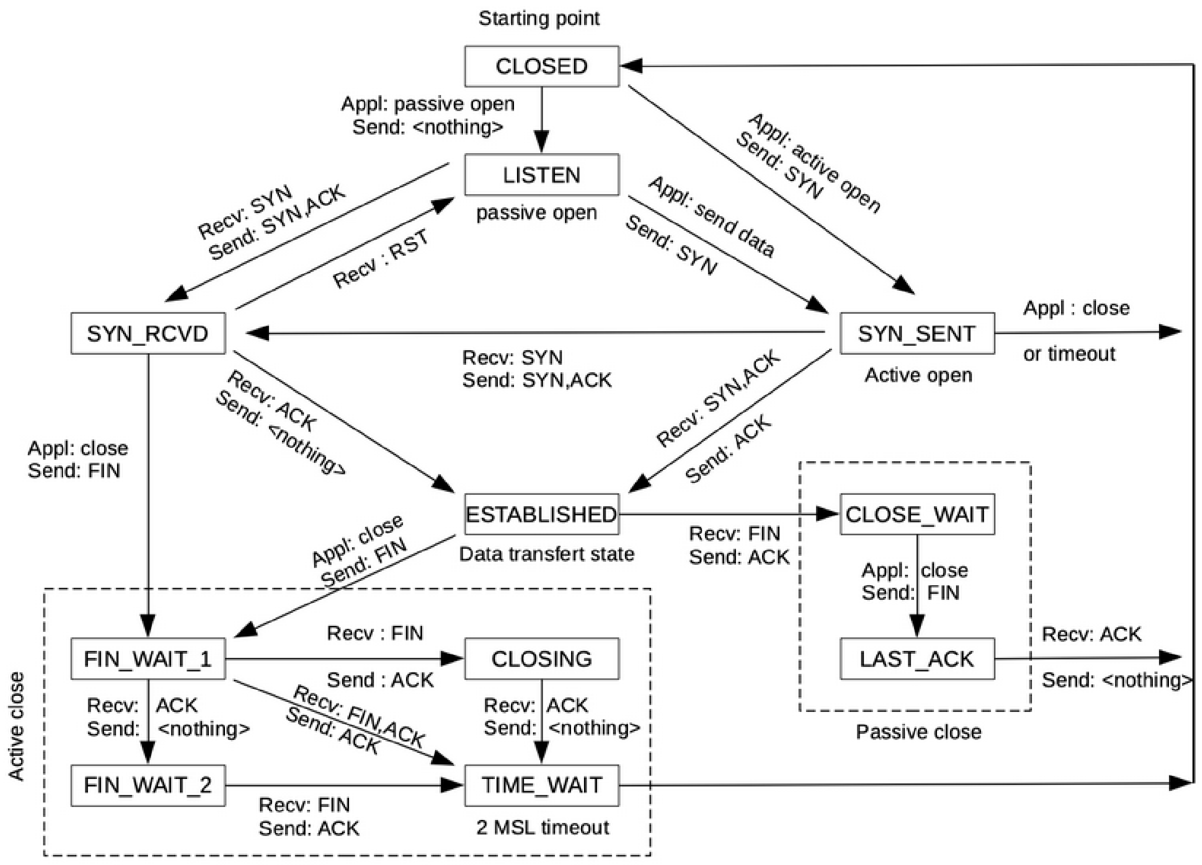


Figure 10: TCP state diagram from IBM website [39]

The main focus of the examination was to identify patterns that legitimate connections exhibit. One of the noticeable features of a legitimate connection was plenty of ACK packets being sent and received. During the data transfer section of TCP it was also noticeable that the only messages that are sent are either ACK or PSH/ACK. Therefore a legitimate connection should use various flags to set up the connection and then use a lot of ACK or PSH/ACK flags during the data transfer. Conversely an abnormal connection should contain a lot of packets with flags such as SYN or SYN/ACK. After this realisation the idea of a scoring system was developed.

The system revolves around the idea that each TCP flow should be given a score. The higher the score the more likely it is to be considered abnormal. The score for each flow will start with 0. When a packet such as SYN or SYN/ACK is sent, the score increments. When an ACK or a PSH/ACK frame is sent the score decrements unless that score is 0. If it is 0 the score will remain the same. Furthermore, if one of the various packets that relates to resetting the connection is sent the score returns to 0. This resetting should help legitimate connections who have a bad connection.

Table 2: Scoring system

Flags	Score
FINACK, RST, RSTACK, FINPSHACK	Returns to 0
ACK, PSHACK	-1
All other flags	+1

Once the score of a connection reaches a certain value such as 8 for example, the connection will be considered abnormal. The amount of abnormal connections that are found are then stored and once the amount of abnormal connection reaches a certain threshold, an attack is detected.

4.3 Detection Implementation

To test whether or not this scoring system was feasible another p4 program was developed called DDoSDetect.p4. To start, constants were added to the program to hold the values of the various flags such as ACK, PSH/ACK or RST. To set up the variable that stores the score of each individual flow, firstly the value of *MAX_ADDRESS* was defined with the value 32w4194304 which is 2 to the power of 22 stored in 32 bits. This value was used as the index size for the *IPScores* variable. While this number is quite a bit less than the amount of possible I.P addresses, it provides enough space to test the validity of the solution. This value was then used to set up register *IPScores* with each individual value having a size of 4 bits, which are used to store the individual scores of the TCP flows. Following this a counter was set up to store the amount of abnormal connections that have been found. After checking if the packet is a valid TCP packet, the scoring system was placed.

The values of individual variables such as the individual flow scores, are placed into meta values in the P4 program as the registers need to be read from and written to, and don't allow directly performing mathematical functions on the variable. The first event that happens in relation to the scoring system is determining which port the packet arrived on the switch. This is done to determine the IP that the flow is tied to, which is the IP that is found outside the network. Following this a hash function is performed to get the flows index which is where the score is stored in the register. Figure 11 displays part of the code that does so.

```
if(hdr.tcp.isValid()) {
       if(meta.port == 1){
        hash(meta.index, HashAlgorithm.crc32, 32w0, {hdr.ipv4.srcAddr},
MAX_ADDRESS);
       }
       else{
        hash(meta.index, HashAlgorithm.crc32, 32w0, {hdr.ipv4.dstAddr},
MAX_ADDRESS);
       }
```

Figure 11: Hashing of ip address based on port extract from DDoSDetect.p4

Following this in Figure 12, the packet's flags are checked and depending on the flags that are found within the packet the score is incremented, decremented or reset to zero. Once the score of a flow reaches a certain threshold the connection is considered abnormal. These abnormal connections are then stored in a P4 counter and are printed to the terminal after exiting the CLI.

```
       if(hdr.tcp.ctrl == FINACK || hdr.tcp.ctrl == RST || hdr.tcp.ctrl == RSTACK
|| hdr.tcp.ctrl == FINPSHACK){
        IPScores.write(meta.index, 0);
       }
       else if(hdr.tcp.ctrl == ACK || hdr.tcp.ctrl == PSHACK){
        IPScores.read(meta.score, meta.index);
        if(meta.score != 0){
              meta.score = meta.score - 1;
              IPScores.write(meta.index, meta.score);
        }
       }
       else{
        IPScores.read(meta.score, meta.index);
        if(meta.score != 7){
              meta.score = meta.score + 1;
              IPScores.write(meta.index, meta.score);
              if(meta.score == 7){
               AbnormalConnections.count(0);
              }
        }
```

Figure 12: Implementation of scoring system extract from DDoSdetect.p4

Initially the score of 7 was used to determine whether or not the flow was abnormal and to test if this solution was viable. After initial testing of benign and attack traffic it was revealed that far more abnormal connections were found in the attack traffic which showed that this solution is successful in identifying attack and benign traffic. The code in full can be found in Appendix D. Following the production of this solution, there are two thresholds that need to be tested. The first of which is the connection score. Once this connection score reaches a certain number the connection is considered abnormal. Following this once a certain amount of abnormal connections is detected, an ongoing attack is detected. The following section delves into this testing.

4.4 Threshold Testing

Different score thresholds were tested with different parts of the traffic dataset to determine the best score threshold. The same two benign traffic sets and three attack traffic sets from the initial solution were used during these tests. Score thresholds between 6 and 12 were tested with the five sets of traffic. Table 3 shows the abnormal connections found in each set using different score thresholds.

Table 3: Threshold testing results

Traffic Num.	Score 6	Score 7	Score 8	Score 9	Score 10	Score 11	Score 12
T1 (Benign)	21	13	8	8	6	5	4
T3 (Benign)	7	7	7	7	7	4	4
T6 (Attack)	1069	1044	1033	1019	988	977	965
T8 (Attack)	1087	1067	1055	1035	1012	991	975
T10 (Attack)	888	854	832	796	785	768	733

These initial results show that this scoring system seems to work quite well and during attack traffic there are a lot more abnormal connections. While these results show that with lower score thresholds more abnormal connections will be detected and vice versa, this table does not display the percentage of abnormal connections compared to total connections found in each case. Figure 13 contains the information that was found.

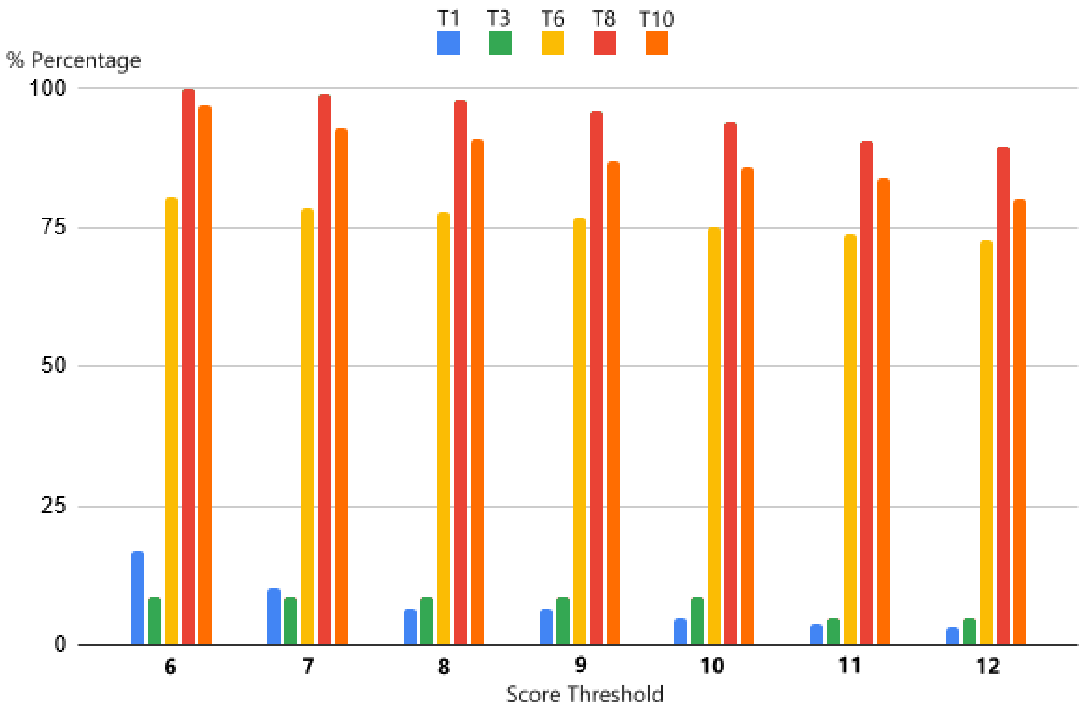


Figure 13: Graph of percentages of traffic deemed to be attack traffic

The blue and the green sections of the graph represent the benign traffic. As expected there is a low percentage of abnormal connections found in benign traffic. Using a score threshold of 6 or 7 results in more than 10 percent of the benign traffic being reported as abnormal in the first traffic set. On the other end of the spectrum the use of higher scores such as 11 and 12 will yield lower than 5 percent of traffic being determined as abnormal but less abnormal traffic is found during the attack traffic.

The yellow, red and orange sections represent the attack traffic. In each of the attack traffic sets when a score threshold is set at 10 or lower, over 75 percent of TCP traffic is reported as abnormal traffic. In dataset T8 using a threshold score of 6, 100 percent of TCP connections are considered abnormal. The results of the testing show any of these score thresholds would work and can be used to detect an attack as the difference between abnormal connections in benign and attack traffic is huge.

Due to scores 6 and 7 detecting over 10 percent of the benign traffic as abnormal, both these scores were decided to be too strict and ultimately would produce too many false positives. On the other end of the scale both the thresholds of 11 and 12 detect a low level of false positives in the benign traffic but detect a lower level of abnormal connections in the attack traffic. Therefore the best threshold score seems to be between 8 and 10, because 8 captures the most attack traffic it was deemed the best choice.

As displayed in Figure 13, there is a huge difference between the level of abnormal connections found in benign and attack traffic. Therefore one could track the amount of abnormal connections found in a certain time frame or after receiving a set amount of TCP packets. Once this period is over, the amount of abnormal connections can be simply compared with a set abnormal connection number or the percentage of total abnormal connections can be calculated. Following this the abnormal connection counter would be reset to record the next time period. This will be discussed further in the defence section of this document.

4.5 Detection Throughput Performance

Network throughput, which is the amount of data that can pass through the switch and the speed in which it can pass through, was determined using the tool iperf. After measuring the baseline by using a simple P4 program that forwards packets from port 1 to port 2 and packets arriving from the other direction from port 2 to port 1, the P4 program that performs the detection was tested. After running the tests up to 5 times each, it was revealed that there was a drop in throughput. Using the Simple.p4 program the bandwidth of the client was 203 Mbit/s and 198 Mbit/s on the server side. The detection program DDoSDetect.p4, had a bandwidth of 140 Mbit/s and a bandwidth of 137 Mbit/s.

The drop in speed is related to the hash function. The hashing function that is used is considered to be the fastest. An improvement in throughput could involve keeping track of previously seen IPs and possibly using a match action table to improve upon the speed. This would involve the controller updating the match action tables regularly but for our purposes we have decided to deploy a data plane only mechanism, the reasoning behind this decision is discussed in the following section. Further improvements are discussed in the defence throughput performance section of the research.

Table 4: Detection Throughput

Program	**Client Bandwidth**	**Server Bandwidth**
Simple.p4	203 Mbit/s	198 Mbit/s
DDoSDetect.p4	140 Mbit/s	137 Mbit/s

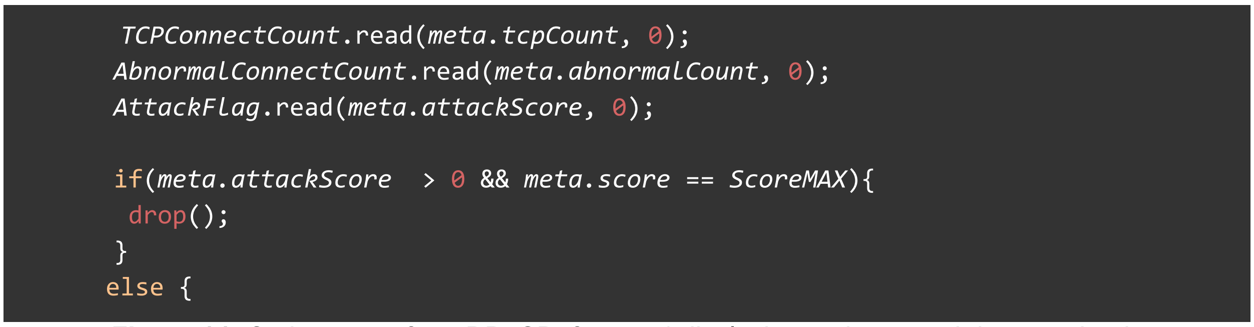
5. DDoS Defence

5.1 Defence Implementation

When implementing the defence, the main thought was to develop the defence to run on the data plane with no interaction from the control plane. This would mean that TCP SYN flood attacks would be completely taken care of by the data plane and this solution could be deployed on a device that sits on the network edge.This defence implementation was built on top of the DDoSDetect.p4. So the DDoSDetect.p4 file was copied to a file DDoSDefense.p4 and changes were made to implement the defence.

The main challenge in implementing the defence was figuring out how to implement the interval in which the abnormal connection count would be checked. Initially a timeframe was considered but it was then decided that after every 50 new connections the amount of abnormal connections would be checked and if there has been a certain amount of abnormal connections an attack is detected to be ongoing. By using this method, one can use a percentage of the new connections that are abnormal to decide if there is an attack.

To implement the defence three variables were added to the program. Figure 14 displays an extract of the program and displays the variables. Each of these were of register type so that the contents of each could be read from and written to. The first of these variables was *TCPConnectCount* which is used to store the number of new TCP connections. *AbnormalConnectCount* was also created to track the amount of abnormal connections. The final variable was the *AttackFlag* variable which is used to activate the defence mechanism when an ongoing attack is detected.



```
         TCPConnectCount.read(meta.tcpCount, 0);
        AbnormalConnectCount.read(meta.abnormalCount, 0);
        AttackFlag.read(meta.attackScore, 0);

        if(meta.attackScore  > 0 && meta.score == ScoreMAX){
         drop();
        }
       else {
```

Figure 14: Code extract from DDoSDefense.p4 displaying registers and drop mechanism

As seen in Figure 14, after each of the registers are read from, they are placed into the meta variables for use later in the program. The first if statement checks if the attack flag variable is above 0 and if that particular TCP flow score has the same value as the score threshold, which is a constant called *ScoreMAX*. If so, the packet is dropped. As discussed in the threshold testing part of the detection section of this research, *ScoreMAX* was decided to be of the value of 8. The reason that the *AttackFlag* value is checked if it is above 0 is due to the fact that each time an attack is detected the *AttackFlag* value increments. When an attack is not detected the score will decrease if it is not 0, this allows for a cool down time period after an attack happens.

If there is no attack ongoing or the flow score has not reached the threshold limit of the value of *ScoreMAX*, the packet continues to the else statement. The first event that happens inside this block of code is a check of the flow's score. If the score is 0, then it is considered to be a new connection and the *TCPConnectCount* value increments.

Following this the packet's flags are checked, first a check if it is a reset flag is performed and if so, the score is set back to 0 if so. Then the ACK or PSH/ACK flags are considered and if they are found, the score decreases if the score is above 0. Finally the score increments if other flags are detected and the *abnormalConnectCount* increments if the threshold value of the *scoreMAX* constant is reached. These statements are basically unchanged from the detection program, it is the next statement where changes were made to implement the defence.

```
if(meta.tcpCount > 50){
        if(meta.abnormalCount > AbnrmLMAX){
              meta.attackScore = meta.attackScore + 1;
              AttackFLag.write(0, meta.attackScore);
        }
        else{
              if(meta.attackScore != 0){
               meta.attackScore = meta.attackScore - 1;
               AttackFLag.write(0, meta.attackScore);
              }
        }
        TCPConnectCount.write(0, 0);
       }
```

Figure 15: Code extract from DDoSDefense.p4 displaying attack detection window

Figure 15 shows that once 50 new connections have been made, the amount of abnormal connections is compared with the *AbnrmlMAX* threshold to determine if there is an ongoing attack. The *AbnrmlMAX* threshold is discussed further within the next section. Following this comparison the *AttackFlag* is incremented or decreased depending on whether an TCP SYN flood attack is detected. Finally the *TCPConnectCount* is reset to 0 to start the next detection phase. Appendix E provides the source code of the mechanism.

5.2 Abnormal Connection Threshold Testing

To test the different abnormal connection thresholds (the value of *AbnrmlMAX*), it was decided that a percentage of the new connections would be used to test if there is an ongoing attack. The values of 8, 7, 6 and 5 were used to test the best choice for the *AbnrmlMAX*. They were chosen because the number of abnormal connections found during the period of 50 new connections should be a percentage that sits between 10 and 20 percent of the total new connections. This was decided because the chosen score threshold of 8 detects less than 10 percent of benign traffic as abnormal. The values chosen represent 16, 14, 12 and 10 percent of the total new connections.

Each of the 14 different sets of traffic were tested with each value to understand the percentage of packets that was dropped by the defence mechanism. Benign traffic is found in the first 5 sets of traffic with a low level of TCP packets found in each of these sets. During the benign traffic none of the traffic was dropped by the mechanism.

To find the percentage of traffic that was dropped, the pcaps files that were created by the P4 switch were viewed. This folder consists of four pcap files, two for each port. One file contains packets entering the switch from the port and the other contains packets that are being sent to that port. So by using tshark to print the contains of the files and coupling it with the pipeline command of “ | *grep -c ‘TCP’* ” the amount of TCP packets found in each file was revealed. While this method was not perfect as there were some discrepancies between the numbers found in the attack traffic, it provided enough information to understand how successful this novel solution is.

```
Tshark -r s1-eth1_in.pcap | grep -c 'TCP'
```

Figure 16: Command used to get the number of TCP packets

In the case of all the various *AbnrmlMAX* thresholds, in most of the attack traffic sets only around 2 percent of the attack traffic made it through the defence mechanism. Traffic sets 6, 9 and 12 reported vastly different results to the other sets with a high percentage of attack traffic getting through. Using a lower *AbnrmlMAX* value did improve the results but over 40 percent of attack traffic still made it through using an *AbnrmlMAX* value of 5 in traffic sets 9 and 12. The explanation for this relates to another form of TCP flooding attack that is used in the dataset which is mentioned in paper [35]. TCP ACK flooding is where a huge amount of ACK messages are sent to the victim. The victim then must process each of these ACK requests which leads to the victim becoming overwhelmed. Due to ACK messages decreasing the score of a flow in the novel solution, this form of attack circumvents the defence mechanism. The possible solution of this problem is discussed in the future work section. To view the results of these tests which use various *AbnrmlMAX* thresholds and each individual traffic set, see Appendix F.

Most of the victim’s response traffic during the attack made it through the mechanism with over 90 percent of the response traffic making it through. Of course due to the response traffic being what the victim could respond to during the orginal attack, this may differ during a real world situation where the traffic is being actively blocked by this mechanism. In a real world situation when this mechanism is blocking illegitimate flows, the response traffic should still remain reasonably high.

The percentage of total attack traffic that made it through the mechanism is displayed in Table 5. The high amount of traffic that made it through the mechanism in traffic sets 6, 9 and 10 had a big effect on the total attack traffic percentages. Without those sets the total attack traffic detected would be around 98 percent in all cases of the various AbnrmlMAX thresholds.

Table 5: Detection Rate Table

AbnrmlMAX	Attack Traffic Detected	Attack Traffic Undetected
8	74%	26%
7	78%	22%
6	81%	19%
5	87%	13%

5.3 Defence Throughput Performance

Once again iperf was used to test the performance of the solution. This time the bandwidth reported on the client side was 108 Mbits/sec and around 103 Mbits/sec on the server side.

Ultimately this means that there is around a 32 Mbit/sec drop in throughput from the implementation of this TCP flooding defence. To improve the speed, the code was examined to see if there were any sections that could be improved upon. After examining the code, it was realised that the program's performance could be improved.

The performance could be improved by simply rearranging one of the if statements found within the program. Within the program the reset flags are checked initially before checking for the ACK or PSH/ACK flags. By changing the order of these statements the performance improved. The reason behind this is that there should be a lot more packets with ACK and PSH/ACK flags and these packets no longer have to be checked against the various reset flags.

```
if(hdr.tcp.ctrl == ACK || hdr.tcp.ctrl == PSHACK){
                    if(meta.score != 0){
                            meta.score = meta.score - 1;
                            IPScores.write(meta.index, meta.score);
                     }
                     }
else if(hdr.tcp.ctrl == FINACK || hdr.tcp.ctrl == RST || hdr.tcp.ctrl == RSTACK ||
hdr.tcp.ctrl == FINPSHACK){
                    IPScores.write(meta.index, 0);
             }
```

Figure 17: Changes made to DDoSDetect.p4

After this change the bandwidth improved to 110 Mbits/s on the client side and 109 Mbits/s on the server side. While it is a small improvement, it does improve the performance slightly and would have a big effect in a real world situation. A further improvement would relate to separating the solution into the ingress and egress parts of the P4 program. Ingress and

Egress use separate threads so by implementing this change, two packets could be handled at once but due to time constraints this change was not implemented.

Table 6: Defence Throughput

Program	Client Bandwidth	Server Bandwidth
Simple.p4	203 Mbit/s	198 Mbit/s
DDoSDetect.p4	140 Mbit/s	137 Mbit/s
DDoSDefense.p4 (Initial)	108 Mbit/s	103 Mbit/s
DDoSDefense.p4 (Improvement)	110 Mbit/s	108 Mbit/s

5.4 Limitations

The biggest limitation of this novel solution is with regard to testing. First off this solution has only been tested with the CAIDA 2007 dataset [34]. To perform further analysis on this solution would have to be tested using multiple types of datasets. This would provide more information regarding the accuracy of the solution against TCP SYN flooding attacks.

Another two forms of testing that could have potentially been performed is testing of the window size and the use of a time interval. The window size refers to the amount of new connections that are made before checking the amount of abnormal connections that have been made. Initially a window size of 25 new connections was briefly tested but seemed too small to be useful. To further test the window size, different sizes could be tested, although 50 did seem to work quite well. Further tests could be performed by replacing the new connection method with a time period that checks the amount of abnormal connections that were found in a time frame. The new connections method was chosen due to it being more closely tied to the traffic.

The register that stores the scores of the IP addresses has a size of 2 to the power of 22 while the total number of possible IP addresses sits at 2 to the power of 32. This means there is a small chance of 2 IP addresses having the same hash value. To improve the chances that this would not happen, the register size could be increased but the performance of the solution would suffer. Conversely the register size could be decreased to improve performance. Trying to determine the best size for the registers is a difficult problem and would require real world testing to determine which size is the most practical.

What is meant by real world testing? Real world testing is the idea that there are several VMs or devices performing different functions. For example, one device would play the attack traffic, another the response traffic, one device for the controller and a production grade software switch would operate on its own VM or device. By performing this kind of testing, it would reveal the true throughput and latency of using this solution.

6. General Discussion

6.1 Machine Learning

Machine learning (ML) is a subfield of computer science that relies on the use of a dataset and a training algorithm that produces a statistical model [40]. There are various subcategories found within this subfield. Supervised learning is a form of machine learning that works with a dataset of labelled examples and unsupervised learning works with unlabelled data. Semi-supervised learning works with both forms of data and reinforcement learning works within the environment and is capable of perceiving changes in its environment.

ML itself is a subcategory of artificial intelligence which can be defined as the performing of tasks that humans would have traditionally performed. Artificial intelligence is considered as mimicking human behaviour and has the goal of ultimately producing a general intelligence that is indistinguishable from a human. Machine learning is a part of this goal and is used to learn from previous data to make decisions.

Systems using ML have produced many advantages to humans in a wide range of fields [41]. Humans have benefited from these systems from advances that have been made in the field of medicine. The field of genetics has used such systems to discover the various genes that are associated with diseases. ML systems have been used to create prescription drugs and to interpret complicated medical scans. ML's skills in pattern recognition facilitate these functions.

There are further developments in various fields such as automation with regards to transportation and industries. Anomaly detection is also performed by ML systems to discover fraudulent activities found in the banking world. The world of security has also been affected by these systems with regards to facial recognition or the detection of cyber attacks.

Ultimately machine learning has proved to be a quite useful tool to humanity in a large number of fields. The field of focus here is of course cybersecurity, specifically its use in the detection of DDoS attacks. In the following sections, machine learning algorithms that are used to detect TCP SYN flooding attacks are discussed before comparing them with the novel P4 solution found within this research.

6.2 TCP-SYN Flood Detection and Machine Learning

There are two machine learning detection systems that have been tested using the dataset "CAIDA 2007" [34]. These systems are quite similar in their TCP SYN flood detection methods. Each TCP flow stores the information of source IP, destination IP, source port and destination port. They both use these same parameters to create an entropy value. To calculate an entropy value the number ports per source IP address is used. This decision was made as during the attack the number of open ports was reported as 240 times the number of source IPs. This entropy value is compared with a threshold value after a specific monitoring time period.

In paper [35], the algorithm K-nearest neighbour (KNN) is used. This algorithm is used to compare the entropy value with the threshold value. The two possible outcomes of this comparison are attack state or normal state. KNN uses euclidean distance to determine if the entropy value is nearest to either state. Euclidean distance can be defined as the distance between two points. In paper [36], the algorithm that is used is the c.45 decision tree. A decision tree is a model that describes possible outcomes of decisions, which got its name due to its tree-like structure. This algorithm uses the same information as the previous example but instead uses a decision tree to decide if there is an ongoing attack.

Each of the algorithms work in a SDN environment and use the openflow protocol. A table of flows is stored on the controller which collects information about the flows during the time frame window. Following this a decision module which is running one of the algorithms makes a decision based on the entropy and threshold values. If the flow is considered to be an attack, the source IP is placed in a block list. This rule is then received by the switch so that any packet coming from that IP source is dropped.

Both of these algorithms are extremely successful with accuracy rates of both being over 95 percent. They also boast of low latencies or low response times using these mitigation mechanisms. They both require a considerable amount of optimising to ensure performance and accuracy. The window size must be considered, as too large a time period would lead to south bound channel issues resulting in the controller being adversely affected, whereas too short a window results in poor accuracy.

6.3 P4 solution and Machine Learning: A Comparison

The accuracy, performance and effectiveness of the machine learning TCP SYN flood defence mechanisms is quite hard to beat. The KNN algorithm claims to mitigate over 96 percent of the attack traffic [35]. Legitimate traffic is also barely affected by the mechanisms and the latency remains low when the right window size has been determined. If the traffic sets of 6, 12 and 9 are not considered the results of the novel P4 solution are similar. A comparison between the performance (bandwidth/latency) of the ML systems and the P4 solution is difficult because in paper [35], their KNN algorithm is tested in a real world environment with a seperate devices used for the switch, controller, attacker and victim. Paper [36] does not mention the details of their testbed but does display an architecture diagram that displays a similar set up. The P4 solution does lead to a significant drop in throughput but the VM is playing the part of the switch, victim and attacker. Furthermore, the BMv2 switch is considered to be considerably slower with relation to throughput and latency than that of a production grade software switch [31].

A considerable amount of time must go into optimising and training the ML mitigation system to get the mechanism to function correctly. The threshold values must be calculated and may differ from network to network. The novel P4 solution needed some initial testing but is deployable from the outset once the threshold values and scoring system has been optimised, as the solution should work on a variety of networks as it is based on the correct functioning of the TCP protocol. The complexity of the ML strategies also increases the difficulty of understanding the mechanism and troubleshooting may require a third party.

The window/time period of the ML mechanisms needs to be at the correct value, if too large there is too much congestion on the controllers channel. This south bound channel issue needs to be carefully managed to ensure that the congestion does not degrade the network. Conversely the novel solution completely runs on the data plane and does not need any interaction from the controller to perform its functioning, which eliminates the risk of the south bound channel becoming overwhelmed.

ML systems provide great value and can be used to detect issues quickly with great degrees of accuracy. They provide many benefits in dealing with problems that traditional software cannot overcome. In paper [40], a list of use cases for the use of ML algorithms is found. In cases where the problem is too complex, constantly changing, an unstudied phenomenon or is cost-effective, ML algorithms are recommended. Conversely in situations where the problem can be solved using traditional software or a simple heuristic would work well, it is recommended against. One could argue for either in the case of TCP SYN attacks but it seems that an approach such as the novel solution found in this research can provide defence against TCP SYN flooding attacks.

The P4 programming language is still relatively new but the control that it gives network operators and the potential for innovation is high. As the popularity of the language increases so too should learning resources which in turn should lead to innovative solutions to various problems that could be shared and used to fight against all kinds of network attacks. The language allows users to strip away any unnecessary or unused protocols from their network which improves the network's performance and simplifies troubleshooting of their network.

In the future it may be possible for the production of a P4 solution that can deal with network/transport layer attacks that can be combined with ML solutions that run on the controller that can be used to detect application layer attacks. For example while the P4 solution deals with network/transport layer flows and the legitimate flows could be further scrutinised and packets could be sampled by looking at packet size or randomly sampled, to allow ML systems to detect whether this flow is performing unusual requests.

Ultimately P4 programming introduces an exciting new area of research to network security with the potential for cost effective solutions to be created for many forms of attack. That being said, ML solutions are not going anywhere and the combination of the two may provide a new level of defence that attackers may struggle to overcome.

6.4 Future Work

With regards to future work there are a number of possibilities. From further testing to the incorporation of various other forms of attack. We will discuss a number of different research directions that are possible. The first major areas of further testing include using other datasets to get a more complete understanding of the solution's accuracy in detecting attacks. The other form of testing would be to spend more time testing the various thresholds and testing the performance of the test when using a time window rather than the amount of abnormal connections.

The performance testing of this solution was performed on a BMv2 switch and as discussed this switch is not meant for use in production situations. By setting up the solution in a dedicated switch, the performance with regards to bandwidth and latency would be a lot more accurate with regards to real world situations. Another further research direction for performance testing would be to investigate the code and make improvements so the solution would be optimised. As mentioned separating the code into ingress and egress section would allow multithreading thus allowing multiple packets to be processed at once. Other potential areas of improvement are in relation to details such as the hash function or the size of the register that stores the IP addresse’s scores. With regard to the hash function, it is possible that the active TCP flow hashes could be stored for easier access or the possible elimination of the hash function and replacing it so that just a substring of the IP address would be stored. In the case of eliminating the hash function it may be possible just to store the last 22 bits of the address which could help with the speed of solution. Alternatively, maybe after hashing the function maybe the hash could be stored in a match-action table or elsewhere. Although storing hash and IP in a match-action table would need interaction from the controller so it may introduce south bound issues but maybe an alternative is possible.

Further future work is found in relation to incorporating other forms of attacks into the scoring system. Other forms of network level attack could be incorporated into the scoring system to improve the range of DDoS attacks that can be detected. One example which was briefly discussed is a form of TCP flooding attack that just uses ACK packets. In TCP ACK flooding the attacker sends a huge amount of ACK packets. To combat an attack such as this a couple of changes could be done to provide sufficient defence. Firstly ACK and PSH/ACK packets scoring could be edited so that the lowest score that could be achieved by these packets sits at 1 meaning these packets could no longer return the score to 0. Following this when checking if it is a new connection (where the score = 0) a small check would be performed to see if it was an ACK packet. If it is determined to be an ACK packet and the flow score is 0 then the packet would be dropped. This solution would need further testing against attacks such as these but these code changes could provide the means to do so.

During the research the solution was compared with two ML solutions. Both of these solutions involve an SDN environment but use openflow rather than P4. For future work, the development of a machine learning solution could be investigated. The research may reveal whether using P4 rather than openflow could produce better results when using ML solutions. The ML detection module would more than likely still need to be performed by the controller but the gathering of flow parameters could be done by the P4. This could potentially reduce the amount of data sent along the southbound channel.

The final area of future work would be developing P4 programs to target other forms of cyber attacks. It may be possible to build a P4 program that targets data extraction or to stop unencrypted data from leaving your network to stop eavesdropping attacks. P4 could be used to drop protocols that are deemed unnecessary from entering your network in the first place, so P4 could be used to completely block protocols that are not required from entering your network in the first place.

6.5 Conclusion

During the course of this research SDN and P4 solutions were investigated in relation to DDoS attacks. An understanding of SDN and P4 was developed by researching state-of-the-art literature. After some deliberation the form of DDoS attack that was decided upon was the TCP SYN flood. This was due to it being used in a high percentage of attacks with some saying that by defending against this form of attack would prevent a large amount of attacks.

Initially the solution that was chosen was a form of defence that measured the amount of SYN/ACK to FIN/ACK packets and compared them. This solution was disregarded as it was found to be ineffective. Following this further research was performed and a novel solution was created. This solution was outlined before testing its validity. The detection part of the mitigation system was first created to test whether the solution would be successful. Following the success of the first round of testing, various threshold values were tested to understand the effects of using different values. Once the threshold value had been determined, some performance testing was performed to understand the solution's effect on throughput.

Once the detection had been deemed to be successful, the defence mechanism was built on top of the detection code. This introduced the dropping of packets that were deemed to be malicious. Once again, threshold values were tested and the performance of the solution was measured. A short discussion on the limitations of our work was then had.

ML was discussed generally before discussing the ML systems that perform mitigation against TCP-SYN attacks. A comparison between the novel solution and the ML solutions was had, to understand the differences. Finally some ideas about the direction of future work was had.

The novel solution that has been created here works quite well against TCP SYN attacks. Its performance will need to be further tested to understand its true ability with relation to throughput. It does however provide defence against this form of attack and could be used as an alternative to ML TCP SYN flood mitigation systems. Other novel solutions may be created in the future that may provide an even higher level of accuracy. ML solutions provide solutions which are very successful in detecting TCP SYN attacks and are just as accurate as the novel solution. ML systems ability to quickly determine if traffic is malicious or normal is outstanding. This research argues that ML systems may be more useful if the focus of its abilities were on application layer attacks that are more difficult to detect. The possibility of combining P4 and ML to develop a solution that can mitigate more forms of DDoS attacks is clear.

This research reveals some insights into the world of SDN, P4, DDoS detection and DDoS mitigation. It was found that P4 has the ability to overcome some forms of DDoS attacks that have been pestering networks for 30 years. P4 is still in its infancy and there is not a huge amount of learning resources for people looking to learn and understand the language. As time goes on, it seems that P4 will be used more and more, which hopefully will lead to the sharing of innovations. It is an exciting time in the world of networking because of P4’s unique ability for network operators to get creative and develop their own solutions to various problems that they face. DDoS attacks have become extremely common in today's world and the world of network defence has received a new tool in the form of the P4 language to combat the attacks.

References

[1] Lundgren, Björn, and Niklas Möller. “Defining Information Security.” *Science and engineering ethics* vol. 25,2 (2019): 419-441. doi:10.1007/s11948-017-9992-1
[2] B. Nagpal, P. Sharma, N. Chauhan and A. Panesar, "DDoS tools: Classification, analysis and comparison," *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2015, pp. 342-346.
[3] R. R. Brooks, L. Yu, I. Ozcelik, J. Oakley and N. Tusing, "Distributed Denial of Service (DDoS): A History," in *IEEE Annals of the History of Computing*, vol. 44, no. 2, pp. 44-54, 1 April-June 2022, doi: 10.1109/MAHC.2021.3072582.
[4] S. Gupta, D. Grover and J. Singla, "Characterization of Flash events and DDoS Attacks: A Survey," *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2021, pp. 1442-1447, doi: 10.1109/ICACCS51430.2021.9441793.
[5] Larry Peterson, Carmelo Cascone, Brian O’Connor, Thomas Vachuska, and Bruce Davie. “Software-Defined Networks: A Systems Approach - Chapter 1: Intro” [Online] https://sdn.systemsapproach.org/intro.html Date Accessed: [1 June 2022]
[6] Feamster, Nick, Jennifer Rexford, and Ellen Zegura. "The road to SDN: an intellectual history of programmable networks." *ACM SIGCOMM Computer Communication Review* 44.2 (2014): 87-98.
[7] Gao, Ya, and Zhenling Wang. "A Review of P4 Programmable Data Planes for Network Security." *Mobile Information Systems* 2021 (2021).
[8] Holl, Patrick. ”Exploring DDoS defense mechanisms.” *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 25.1 (2015).
[9] Yan, Qiao, and F. Richard Yu. ”Distributed denial of service attacks in software-defined networking with cloud computing.” *IEEE Communications Magazine 53.4* (2015): 52-59
[10] Bawany, Narmeen Zakaria, Jawwad A. Shamsi, and Khaled Salah. ”DDoS attack detection and mitigation using SDN: methods, practices, and solutions.” *Arabian Journal for Science and Engineering* 42.2 (2017):425-441.
[11] Q. Yan, F. R. Yu, Q. Gong and J. Li, "Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges," in *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 602-622, Firstquarter 2016, doi: 10.1109/COMST.2015.2487361
[12] B. Mladenov, ”Studying the DDoS Attack Effect over SDN Controller Southbound Channel,” *2019 X NationalConference with International Participation (ELECTRONICA)*, 2019, pp. 1-4, doi:10.1109/ELECTRONICA.2019.8825601.
[13] K. Friday, E. Kfoury, E. Bou-Harb and J. Crichigno, "Towards a Unified In-Network DDoS Detection and Mitigation Strategy," *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, 2020, pp. 218-226, doi:10.1109/NetSoft48620.2020.9165336.
[14] A. Febro, H. Xiao and J. Spring, ”Distributed SIP DDoS Defense with P4,” *2019 IEEE Wireless Communications and Networking Conference (WCNC),* 2019, pp. 1-8, doi: 10.1109/WCNC.2019.8885926.
[15] J. E. Varghese and B. Muniyal, ”Trend in SDN Architecture for DDoS Detection- A Comparative Study,” *2021 IEEE International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, 2021, pp. 170-174, doi: 10.1109/DISCOVER52564.2021.9663589.
[16] B. Pande, G. Bhagat, S. Priya and H. Agrawal, ”Detection and Mitigation of DDoS in SDN,” *2018 Eleventh International Conference on Contemporary Computing (IC3)*, 2018, pp. 1-3, doi:10.1109/IC3.2018.8530551.
[17] W. Sun, Y. Li and S. Guan, ”An Improved Method of DDoS Attack Detection for Controller of SDN,” *2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology (CCET),* 2019, pp. 249-253, doi: 10.1109/CCET48361.2019.8989356.
[18] N. I. G. Dharma, M. F. Muthohar, J. D. A. Prayuda, K. Priagung and D. Choi, ”Time-based DDoS detection and mitigation for SDN controller,” *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2015, pp. 550-553, doi:10.1109/APNOMS.2015.7275389.

[19] Larry Peterson, Carmelo Cascone, Brian O'Connor, Thomas Vachuska, and Bruce Davie. "Software-Defined Networks: A Systems Approach - Chapter 4: Bare-Metal Switches" [Online] https://sdn.systemsapproach.org/switch.html Date Accessed: [2 June 2022]
[20] R. Kumar, S. P. Lal and A. Sharma, "Detecting Denial of Service Attacks in the Cloud," *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, 2016, pp. 309-316, doi: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.70.
[21] Scholz, Dominik, et al. "SYN flood defense in programmable data planes." *Proceedings of the 3rd P4 Workshop in Europe.* 2020.
[22] T. -Y. Lin *et al.*, "Mitigating SYN flooding Attack and ARP Spoofing in SDN Data Plane," *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2020, pp. 114-119, doi: 10.23919/APNOMS50412.2020.9236951.
[23] F. Musumeci, V. Ionata, F. Paolucci, F. Cugini and M. Tornatore, "Machine-learning-assisted DDoS attack detection with P4 language," *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1-6, doi: 10.1109/ICC40277.2020.9149043.
[24] S. S. Priya, M. Sivaram, D. Yuvaraj and A. Jayanthiladevi, "Machine Learning based DDOS Detection," *2020 International Conference on Emerging Smart Computing and Informatics (ESCI)*, 2020, pp. 234-237, doi: 10.1109/ESCI48226.2020.9167642.
[25] Mininet, Version 2.3.0, Bob Lantz, Brandon Heller, and Nick McKeown, "Mininet - an instant virtual network." [Online]. http://mininet.org/ Date Accessed: [3 September 2022]
[26] Python3, Version 3.8.5, Python Software Foundation. Python Language Reference, [Online] http://www.python.org Date Accessed: [3 September 2022]
[27] tcpreplay, Version 4.3.2, Appneta, tcpreplay manual, [Online] http://tcpreplay.appneta.com/ Date Accessed: [3 September 2022]
[28] GitHub - p4lang/tutorials: P4 language tutorials, *GitHub*, [Online] https://github.com/p4lang/tutorials Date Accessed: [15 June 2022]
[29] Releases · mininet/mininet, *GitHub*, [Online] https://github.com/mininet/mininet/releases/ Date Accessed: [4 July 2022]
[30] hping3, version 3.0.0-alpha-2, Salvatore Sanfilippo, [Online] http://www.hping.org/ Date Accessed: [3 September 2022]
[31] P4Lang. "GitHub - P4lang/Behavioral-Model: The Reference P4 Software Switch." *GitHub*, [Online] https://github.com/p4lang/behavioral-model Date Accessed 1 Sept. 2022.
[32] Derya Erhan, "Boğaziçi University DDoS Dataset", *IEEE Dataport*, October 9, 2019, doi: https://dx.doi.org/10.21227/45m9-9p82 Date Accessed: [4 July 2022]
[33] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing realistic distributed denial of service (DDoS) attack dataset and taxonomy," in Proc. 53rd International Carnahan Conference on Security Technology, Chennai, India, 2019.
[34] The CAIDA UCSD "DDoS Attack 2007" Dataset, *CAIDA UCSD*, 2007, [Online] https://www.caida.org/catalog/datasets/ddos-20070804_dataset Date Accessed: [13 July 2022]
[35] N. N. Tuan, P. H. Hung, N. D. Nghia, N. Van Tho, T. V. Phan and N. H. Thanh, "A Robust TCP-SYN Flood Mitigation Scheme Using Machine Learning Based on SDN," *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, 2019, pp. 363-368, doi: 10.1109/ICTC46691.2019.8939829
[36] Sudar, K. Muthamil, et al. "TFAD: TCP flooding attack detection in software-defined networking using proxy-based and machine learning-based mechanisms." *Cluster Computing* (2022): 1-17.
[37] Z. -Y. Shen, M. -W. Su, Y. -Z. Cai and M. -H. Tasi, "Mitigating SYN Flooding and UDP Flooding in P4-based SDN," *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2021, pp. 374-377, doi: 10.23919/APNOMS52696.2021.9562660.

[38] M. Bellaiche and J. -C. Gregoire, "SYN Flooding Attack Detection Based on Entropy Computing," *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*, 2009, pp. 1-6, doi: 10.1109/GLOCOM.2009.5425454.
[39] Flowchart of TCP connections and their definition, IBM.com [Online] https://www.ibm.com/support/pages/flowchart-tcp-connections-and-their-definition Date Accessed: [22 July 2022]
[40] Burkov, Andriy. "Chapter 1: Introduction." Machine Learning Engineering, *True Positive Inc*, Québec, Canada, 2020.
[41] A. Nayak and K. Dutta, "Impacts of machine learning and artificial intelligence on mankind," *2017 International Conference on Intelligent Computing and Control (I2C2)*, 2017, pp. 1-3, doi: 10.1109/I2C2.2017.8321908.

Appendices:

Appendix A: s1-runtime.json

The source code of s1-runtime.json

```
{
  "target": "bmv2",
  "p4info": "build/basic.p4.p4info.txt",
  "bmv2_json": "build/basic.json",
  "table_entries": [
       {
       "table": "MyIngress.ipv4_lpm",
       "default_action": true,
       "action_name": "MyIngress.drop",
       "action_params": { }
       },
       {
       "table": "MyIngress.ipv4_lpm",
       "match": {
       "meta.port": 2
       },
       "action_name": "MyIngress.ipv4_forward",
       "action_params": {
       "dstAddr": "08:00:00:00:01:11",
       "port": 1
       }
       },
       {
       "table": "MyIngress.ipv4_lpm",
       "match": {
       "meta.port": 1
       },
       "action_name": "MyIngress.ipv4_forward",
       "action_params": {
       "dstAddr": "08:00:00:00:02:22",
       "port": 2
       }
       }
  ]
}
```

Appendix B: runNetwork.py

The source code of runNetwork.py

```
#!/usr/bin/env python3
from mininet.net import Mininet
from mininet.topo import Topo
from mininet.log import setLogLevel, info
from mininet.cli import CLI

from p4_mininet import P4Switch, P4Host
from p4runtime_switch import P4RuntimeSwitch
import p4runtime_lib.simple_controller
import p4runtime_lib.helper
import p4runtime_lib.bmv2
import os
import json
from time import sleep
import sys

class SwitchTopo(Topo):
	def __init__(self, sw_path, log_file, json_path, thrift_port, pcap_dump,
hosts, **opts):
	Topo.__init__(self, **opts)
	#Add switch to topo object
	self.addSwitch('s1', sw_path = sw_path, log_file = log_file, json_path =
json_path, thrift_port = thrift_port, pcap_dump = pcap_dump)

	#Set up hosts and links with the switch
	for host_name in hosts:
		self.addHost(host_name, ip=hosts[host_name]['ip'],
mac=hosts[host_name]['mac'])
		self.addLink(host_name, 's1')

class NetworkRunner:
	def __init__(self):
	#Set up path variables for log, pcaps and topology
	cwd = os.getcwd()
	self.logs = os.path.join(cwd, 'logs')
	self.pcaps = os.path.join(cwd, 'pcaps')
	self.compiledP4 = os.path.join(cwd, 'build/basic.json')
	self.switchType = 'simple_switch_grpc'
	self.topology = './topology.json'

	self.logfile = self.logs + '/switch.log'
	#Read from topology file
	with open(self.topology, 'r') as file:
		top = json.load(file)
	self.hosts = top['hosts']

	self.topo = SwitchTopo(self.switchType, self.logfile, self.compiledP4, 9090,
self.pcaps, self.hosts)

        self.p4info =
p4runtime_lib.helper.P4InfoHelper('./build/basic.p4.p4info.txt')

        def createNetwork(self):
        #Create Mininet and start it
        self.net = Mininet(topo = self.topo, host = P4Host, switch =
P4RuntimeSwitch, controller = None)
        self.net.start()
        sleep(1)

        #Run commands from topology file on hosts
        for host_name, host_info in list(self.hosts.items()):
                current_host = self.net.get(host_name)
                if "commands" in host_info:
                for cmd in host_info["commands"]:
                        current_host.cmd(cmd)

        #Use P4Runtime to do final config on switch
        switch_object = self.net.get('s1')
        grpc_port = switch_object.grpc_port
        device_id = switch_object.device_id
        runtime_json = 's1-runtime.json'
        with open(runtime_json, 'r') as sw_conf_file:
                outfile = '%s/s1-p4runtime-requests.txt' %(self.logs)
                p4runtime_lib.simple_controller.program_switch(
                        addr='127.0.0.1:%d' % grpc_port,
                        device_id=device_id,
                        sw_conf_file=sw_conf_file,
                        workdir=os.getcwd(),
                        proto_dump_fpath=outfile,
                        runtime_json=runtime_json)

        sleep(1)

        #start CLI
        CLI(self.net)

#This is where the reading of the P4 counters happens for SYNACKCounting.p4 and
#DDoSDetect testing

        self.net.stop()

if __name__ == '__main__':

        network = NetworkRunner()
        network.createNetwork()
```

Appendix C: SYNACKCounting.p4

The source code of SYNACKCounting.p4

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;
const bit<6>  SYNACK = 0x12;
const bit<6>  ACKFIN = 0x11;

/*************************************************************************
*********************** H E A D E R S  ***********************************
*************************************************************************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
       macAddr_t dstAddr;
       macAddr_t srcAddr;
       bit<16>   etherType;
}

header ipv4_t {
       bit<4> version;
       bit<4> ihl;
       bit<8> diffserv;
       bit<16>   totalLen;
       bit<16>   identification;
       bit<3> flags;
       bit<13>   fragOffset;
       bit<8> ttl;
       bit<8> protocol;
       bit<16>   hdrChecksum;
       ip4Addr_t srcAddr;
       ip4Addr_t dstAddr;
}

header tcp_t {
       bit<16> srcPort;
       bit<16> dstPort;
       bit<32> seqNo;
       bit<32> ackNo;
       bit<4>  dataOffset;
       bit<3>  res;
       bit<3>  ecn;
       bit<6>  ctrl;
       bit<16> window;
```

```
	bit<16> checkSum;
	bit<16> urgentPtr;
}

struct metadata {
	bit<8>	count;
	bit<8>	compare;
	egressSpec_t  port;
}

struct headers {
	ethernet_t	ethernet;
	ipv4_t		ipv4;
	tcp_t		tcp;
}

/*************************************************************************
*********************** P A R S E R  ***********************************
*************************************************************************/

parser MyParser(packet_in packet,
		out headers hdr,
		inout metadata meta,
		inout standard_metadata_t standard_metadata) {

	state start {
	transition parse_ethernet;
	}

	state parse_ethernet {
	packet.extract(hdr.ethernet);
	transition select(hdr.ethernet.etherType) {
		TYPE_IPV4: parse_ipv4;
		default: accept;
	}
	}

	state parse_ipv4 {
	packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol){
	8w0x6: parse_tcp;
		default: accept;
	}
	}

	state parse_tcp {
    packet.extract(hdr.tcp);
    transition accept;
	}
```

```
}

/*************************************************************************
************   C H E C K S U M    V E R I F I C A T I O N   *************
*************************************************************************/

control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
      apply {  }
}


/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************

*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

      counter(4, CounterType.packets) AttackFlagCount;

      action drop() {
      mark_to_drop(standard_metadata);
      }
      action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
      }
      table ipv4_lpm {
    key = {
      meta.port: exact;
    }
    actions = {
      ipv4_forward;
      drop;
      NoAction;
    }
    size = 1024;
    default_action = drop();
      }

      apply {
      if(hdr.ipv4.isValid()) {
      meta.port = standard_metadata.ingress_port;
             ipv4_lpm.apply();
```

```
        }
    if(hdr.tcp.isValid()) {
        if(hdr.tcp.ctrl == SYNACK){
                if(meta.port == 1){
                        AttackFlagCount.count(0);
                }
                else{
         AttackFlagCount.count(1);
         }

                }
                if(hdr.tcp.ctrl == ACKFIN){
         if(meta.port == 1){
                AttackFlagCount.count(2);
                }
                else{
                AttackFlagCount.count(3);
                }

                }

        }
        }
}

/*************************************************************************
****************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
        apply {  }
}

/*************************************************************************
*************   C H E C K S U M   C O M P U T A T I O N   **************
*************************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
        apply {
        update_checksum(
        hdr.ipv4.isValid(),
                { hdr.ipv4.version,
                hdr.ipv4.ihl,
                hdr.ipv4.diffserv,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
              hdr.ipv4.hdrChecksum,
              HashAlgorithm.csum16);
       }
}

/*************************************************************************
***********************  D E P A R S E R  *******************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
       apply {
       packet.emit(hdr.ethernet);
       packet.emit(hdr.ipv4);
    packet.emit(hdr.tcp);
       }
}

/*************************************************************************
***********************  S W I T C H  *******************************
*************************************************************************/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

Appendix D: DDoSDetect.p4

The source code of DDoSDetect.p4

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

#define MAX_ADDRESS 32w4194304

const bit<16> TYPE_IPV4 = 0x800;
const bit<6>  ACK         = 0x10;
const bit<6>  FINACK      = 0x11;
const bit<6>  PSHACK      = 0x18;
const bit<6>  FINPSHACK = 0x19;
const bit<6>  RST         = 0x04;
const bit<6>  RSTACK      = 0x14;

/*************************************************************************
*********************** H E A D E R S  ***********************************
*************************************************************************/

register<bit<4>>(MAX_ADDRESS) IPScores;

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
      macAddr_t dstAddr;
      macAddr_t srcAddr;
      bit<16>   etherType;
}

header ipv4_t {
      bit<4> version;
      bit<4> ihl;
      bit<8> diffserv;
      bit<16>   totalLen;
      bit<16>   identification;
      bit<3> flags;
      bit<13>   fragOffset;
      bit<8> ttl;
      bit<8> protocol;
      bit<16>   hdrChecksum;
      ip4Addr_t srcAddr;
      ip4Addr_t dstAddr;
}

header tcp_t {
      bit<16> srcPort;
```

```
	bit<16> dstPort;
	bit<32> seqNo;
	bit<32> ackNo;
	bit<4>  dataOffset;
	bit<3>  res;
	bit<3>  ecn;
	bit<6>  ctrl;
	bit<16> window;
	bit<16> checkSum;
	bit<16> urgentPtr;
}

struct metadata {
	bit<8>  count;
	bit<8>  compare;
	bit<32> index;
	bit<4>  score;
	egressSpec_t  port;
}

struct headers {
	ethernet_t   ethernet;
	ipv4_t       ipv4;
	tcp_t        tcp;
}
/*************************************************************************
*********************** P A R S E R  ***********************************
*************************************************************************/

parser MyParser(packet_in packet,
		out headers hdr,
		inout metadata meta,
		inout standard_metadata_t standard_metadata) {

	state start {
	transition parse_ethernet;
	}

	state parse_ethernet {
	packet.extract(hdr.ethernet);
	transition select(hdr.ethernet.etherType) {
		TYPE_IPV4: parse_ipv4;
		default: accept;
	}
	}

	state parse_ipv4 {
	packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol){
        8w0x6: parse_tcp;
              default: accept;
        }
        }

        state parse_tcp {
    packet.extract(hdr.tcp);
    transition accept;
        }

}
/*************************************************************************
************   C H E C K S U M    V E R I F I C A T I O N   *************
*************************************************************************/

control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
        apply {  }
}
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                   inout metadata meta,
                   inout standard_metadata_t standard_metadata) {

        counter(1, CounterType.packets) AbnormalConnections;

        action drop() {
        mark_to_drop(standard_metadata);
        }
        action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
        }
        table ipv4_lpm {
    key = {
        meta.port: exact;
    }
    actions = {
        ipv4_forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = drop();
        }
```

```
       apply {
       if(hdr.ipv4.isValid()) {
       meta.port = standard_metadata.ingress_port;
             ipv4_lpm.apply();
       }
    if(hdr.tcp.isValid()) {
       if(meta.port == 1){
        hash(meta.index, HashAlgorithm.crc32, 32w0, {hdr.ipv4.srcAddr},
MAX_ADDRESS);
       }
       else{
        hash(meta.index, HashAlgorithm.crc32, 32w0, {hdr.ipv4.dstAddr},
MAX_ADDRESS);
       }

       if(hdr.tcp.ctrl == FINACK || hdr.tcp.ctrl == RST || hdr.tcp.ctrl == RSTACK
|| hdr.tcp.ctrl == FINPSHACK){
        IPScores.write(meta.index, 0);
       }
       else if(hdr.tcp.ctrl == ACK || hdr.tcp.ctrl == PSHACK){
        IPScores.read(meta.score, meta.index);
        if(meta.score != 0){
             meta.score = meta.score - 1;
             IPScores.write(meta.index, meta.score);
        }
       }
       else{
        IPScores.read(meta.score, meta.index);
        if(meta.score != 7){
             meta.score = meta.score + 1;
             IPScores.write(meta.index, meta.score);
             if(meta.score == 7){
              AbnormalConnections.count(0);
             }
        }
       }

       }
       }
}
/*************************************************************************
****************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyEgress(inout headers hdr,
             inout metadata meta,
             inout standard_metadata_t standard_metadata) {
       apply {  }
```

```
}
/*************************************************************************
*************   C H E C K S U M   C O M P U T A T I O N   **************
*************************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
      apply {
      update_checksum(
      hdr.ipv4.isValid(),
             { hdr.ipv4.version,
             hdr.ipv4.ihl,
             hdr.ipv4.diffserv,
             hdr.ipv4.totalLen,
             hdr.ipv4.identification,
             hdr.ipv4.flags,
             hdr.ipv4.fragOffset,
             hdr.ipv4.ttl,
             hdr.ipv4.protocol,
             hdr.ipv4.srcAddr,
             hdr.ipv4.dstAddr },
             hdr.ipv4.hdrChecksum,
             HashAlgorithm.csum16);
      }
}
/*************************************************************************
***********************  D E P A R S E R  *******************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
      apply {
      packet.emit(hdr.ethernet);
      packet.emit(hdr.ipv4);
    packet.emit(hdr.tcp);
      }
}
/*************************************************************************
***********************  S W I T C H  *******************************
*************************************************************************/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

Appendix E: DDoSDefense.p4

The source code of DDoSDefense.p4

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

#define MAX_ADDRESS 32w4194304

const bit<16> TYPE_IPV4 = 0x800;
const bit<6>  ACK         = 0x10;
const bit<6>  FINACK      = 0x11;
const bit<6>  PSHACK      = 0x18;
const bit<6>  FINPSHACK = 0x19;
const bit<6>  RST         = 0x04;
const bit<6>  RSTACK      = 0x14;
const bit<4>  ScoreMAX  = 0x08;
const bit<6>  AbnrmlMAX = 0x08;

/*************************************************************************
*********************** H E A D E R S  ***********************************
*************************************************************************/

register<bit<4>>(MAX_ADDRESS) IPScores;
register<bit<6>>(1) TCPConnectCount;
register<bit<6>>(1) AbnormalConnectCount;
register<bit<6>>(1) AttackFlag;

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
       macAddr_t dstAddr;
       macAddr_t srcAddr;
       bit<16>   etherType;
}

header ipv4_t {
       bit<4> version;
       bit<4> ihl;
       bit<8> diffserv;
       bit<16>   totalLen;
       bit<16>   identification;
       bit<3> flags;
       bit<13>   fragOffset;
       bit<8> ttl;
       bit<8> protocol;
       bit<16>   hdrChecksum;
       ip4Addr_t srcAddr;
```

```
	ip4Addr_t dstAddr;
}

header tcp_t {
	bit<16> srcPort;
	bit<16> dstPort;
	bit<32> seqNo;
	bit<32> ackNo;
	bit<4>  dataOffset;
	bit<3>  res;
	bit<3>  ecn;
	bit<6>  ctrl;
	bit<16> window;
	bit<16> checkSum;
	bit<16> urgentPtr;
}

struct metadata {
	bit<32> index;
	bit<4>  score;
	bit<6>  tcpCount;
	bit<6>  abnormalCount;
	bit<6>  attackScore;
	egressSpec_t  port;
}

struct headers {
	ethernet_t   ethernet;
	ipv4_t       ipv4;
	tcp_t        tcp;
}

/*************************************************************************
*********************** P A R S E R  ***********************************
*************************************************************************/

parser MyParser(packet_in packet,
		out headers hdr,
		inout metadata meta,
		inout standard_metadata_t standard_metadata) {

	state start {
	transition parse_ethernet;
	}

	state parse_ethernet {
	packet.extract(hdr.ethernet);
	transition select(hdr.ethernet.etherType) {
		TYPE_IPV4: parse_ipv4;
```

```
             default: accept;
      }
      }

      state parse_ipv4 {
      packet.extract(hdr.ipv4);
   transition select(hdr.ipv4.protocol){
      8w0x6: parse_tcp;
             default: accept;
      }
      }

      state parse_tcp {
   packet.extract(hdr.tcp);
   transition accept;
      }

}

/*************************************************************************
************   C H E C K S U M    V E R I F I C A T I O N   *************
*************************************************************************/

control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
      apply {  }
}


/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {


      action drop() {
      mark_to_drop(standard_metadata);
      }
      action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
   standard_metadata.egress_spec = port;
   hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
   hdr.ethernet.dstAddr = dstAddr;
   hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
      }

      table ipv4_lpm {
   key = {
      meta.port: exact;
   }
   actions = {
      ipv4_forward;
      drop;
      NoAction;
   }
   size = 1024;
   default_action = drop();
      }

      apply {
      if(hdr.ipv4.isValid()) {
      meta.port = standard_metadata.ingress_port;
             ipv4_lpm.apply();
      }
   if(hdr.tcp.isValid()) {
      if(meta.port == 1){
       hash(meta.index, HashAlgorithm.crc32, 32w0, {hdr.ipv4.srcAddr},
MAX_ADDRESS);
      }
      else{
       hash(meta.index, HashAlgorithm.crc32, 32w0, {hdr.ipv4.dstAddr},
MAX_ADDRESS);
      }

      IPScores.read(meta.score, meta.index);
      TCPConnectCount.read(meta.tcpCount, 0);
      AbnormalConnectCount.read(meta.abnormalCount, 0);
      AttackFlag.read(meta.attackScore, 0);

      if(meta.attackScore  > 0 && meta.score == ScoreMAX){
       drop();
      }
      else {
       if(meta.score == 0){
                        meta.tcpCount = meta.tcpCount + 1;
                  TCPConnectCount.write(0, meta.tcpCount);
             }

                    if(hdr.tcp.ctrl == ACK || hdr.tcp.ctrl == PSHACK){
                  if(meta.score != 0){
                         meta.score = meta.score - 1;
                         IPScores.write(meta.index, meta.score);
                    }
                    }
       else if(hdr.tcp.ctrl == FINACK || hdr.tcp.ctrl == RST || hdr.tcp.ctrl ==
RSTACK || hdr.tcp.ctrl == FINPSHACK){
                  IPScores.write(meta.index, 0);
```

```
            }
                   else{
                  if(meta.score != ScoreMAX){
                         meta.score = meta.score + 1;
                         IPScores.write(meta.index, meta.score);

             if(meta.score == ScoreMAX){
                        meta.abnormalCount = meta.abnormalCount + 1;
                        AbnormalConnectCount.write(0, meta.abnormalCount);
                  }
            }
                   }
      }
      if(meta.tcpCount > 50){
       if(meta.abnormalCount > AbnrmlMAX){
            meta.attackScore = meta.attackScore + 1;
            AttackFlag.write(0, meta.attackScore);
       }
       else{
            if(meta.attackScore != 0){
             meta.attackScore = meta.attackScore - 1;
             AttackFlag.write(0, meta.attackScore);
            }
       }
       TCPConnectCount.write(0, 0);
        AbnormalConnectCount.write(0, 0);
      }

      }
      }
}

/*************************************************************************
****************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyEgress(inout headers hdr,
             inout metadata meta,
             inout standard_metadata_t standard_metadata) {
      apply {  }
}

/*************************************************************************
*************   C H E C K S U M   C O M P U T A T I O N   **************
*************************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
      apply {
      update_checksum(
        hdr.ipv4.isValid(),
                { hdr.ipv4.version,
                hdr.ipv4.ihl,
                hdr.ipv4.diffserv,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags,
                hdr.ipv4.fragOffset,
                hdr.ipv4.ttl,
                hdr.ipv4.protocol,
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr },
                hdr.ipv4.hdrChecksum,
                HashAlgorithm.csum16);
        }
}

/*************************************************************************
***********************  D E P A R S E R  *******************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
        apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
     packet.emit(hdr.tcp);
        }
}

/*************************************************************************
***********************  S W I T C H  *******************************
*************************************************************************/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

Appendix F: Abnormal Connection Threshold Table

Table displaying results abnormal connections by the drop percentage of attack traffic

Traffic Set	**Traffic Drop Rate AbnrmlMAX= 8**	**Traffic Drop Rate AbnrmlMAX= 7**	**Traffic Drop Rate AbnrmlMAX= 6**	**Traffic Drop Rate AbnrmlMAX= 5**
T1 Benign Traffic	0%	0%	0%	0%
T1 Response Traffic	0%	0%	0%	0%
T2 Benign Traffic	0%	0%	0%	0%
T3 Response Traffic	0%	0%	0%	0%
T3 Benign Traffic	0%	0%	0%	0%
T3 Response Traffic	0%	0%	0%	0%
T4 Benign Traffic	0%	0%	0%	0%
T4 Response Traffic	0%	0%	0%	0%
T5 Benign Traffic	0%	0%	0%	0%
T5 Response Traffic	0%	0%	0%	0%
T6 Attack Traffic	67%	70%	70%	73%
T6 Response Traffic	1%	1%	1%	1%
T7 Attack Traffic	98%	98%	98%	98%
T7 Response Traffic	0%	0%	0%	0%
T8 Attack Traffic	98%	98%	98%	98%
T8 Response Traffic	0%	0%	0%	0%
T9 Attack Traffic	22%	26%	33%	56%
T9 Response Traffic	2%	2%	2%	1%
T10 Attack Traffic	98%	98%	98%	98%
T10 Response Traffic	0%	0%	0%	0%
T11 Attack Traffic	98%	98%	98%	98%
T11 Response Traffic	0%	0%	0%	0%
T12 Attack Traffic	14%	41%	51%	59%
T12 Response Traffic	1%	1%	1%	1%
T13 Attack Traffic	98%	98%	98%	98%
T13 Response Traffic	0%	0%	0%	0%
T14 Attack Traffic	94%	94%	94%	94%
T14 Response Traffic	0%	0%	0%	0%