

# DESIGN AND IMPLEMENTATION OF A STATELESS AND STATEFUL DDOS PREVENTION SOLUTION WITH PYRETIC BASED FIREWALL ON SDN CONTROLLER

MASTER OF INTERNETWORKING CAPSTONE PROJECT  
University of Alberta  
M.Sc.(Internetworking) Program

AGBOOLA EMMANUEL OLAWALE

MARCH 2017

## **ACKNOWLEDGEMENT**

I would like to express a sincere gratitude to my supervisor, Mr. Gurpreet P. Nanda for the support during the entire time of the project.

I also would like to appreciate the efforts of Iwasam Agube and Rao Nadeem during the time of the project.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and completing this project. This accomplishment would not have been possible without them.

Thank you.

## **ABSTRACT**

We realize information as a whole need a continuous flow when it comes to terms of networking systems. With this in view, Distributed Denial of Service (DDoS) is a threat that has continually degraded this important characteristics of networking systems, thus, the need to create and build an infrastructure that detects and prevents this threats as quickly as possible.

This project proposes a DDoS prevention solution that utilizes both a stateless and stateless firewall on an SDN controller which blocks traffic from a node that has been detected as a source of a DDoS attack using explicit flows that match source addresses.

Software Defined Networking (SDN) is an emerging technology, which offers the network architecture with an ideology of physical separation of the control and data plane of the forwarding devices. The controller implements the control plane while the switches perform the forwarding operations of the data plane. Using OpenFlow protocol which is the standard of communication between the controller and the switches, SDN controllers can manage forwarding behaviors of SDN switches by managing Flow Table entries. As a result, the network becomes more dynamic, and the network resources are managed in a more effective and cost-effective manner due to the centralized control.

The detection and prevention mechanism designed here are effective for small network topologies and can also be extended to similar large domains.

# Table of Contents

|  |    |
|--|----|
| <b>ACKNOWLEDGEMENT</b> .....   | 2  |
| <b>ABSTRACT</b> .....  | 3  |
| <b>LIST OF FIGURES</b> .....   | 5  |
| <b>LIST OF TABLES</b> .....  | 6  |
| 1. INTRODUCTION .....  | 7  |
| 2. LITERATURE REVIEW .....   | 9  |
| 2.1 Distributed Denial of Service (DDoS).....  | 9  |
| 2.1.1 Effects of Design Principles of Internet Architecture and its Implication on DDoS Attacks .... | 10 |
| 2.1.2 Methods of DDoS Attacks .....  | 10 |
| 2.1.3 DDoS Defense Schemes .....   | 12 |
| 2.2 Software Defined Networking (SDN).....   | 13 |
| 2.2.1 Benefits of SDN.....   | 13 |
| 2.2.2 SDN Architecture Model .....   | 14 |
| 2.2.3 OpenFlow and Open vSwitch .....  | 15 |
| 2.3 DDoS in SDN Environment.....   | 16 |
| 2.4 Pyretic .....  | 17 |
| 2.5 Stateless and Stateful Firewall .....  | 18 |
| 2.5.1 Stateless Firewalls.....   | 18 |
| 2.5.2 Stateful Firewalls .....   | 19 |
| 3. IMPLEMENTATION OF FIREWALL USING PYRETIC.....   | 20 |
| 3.1 Controller .....   | 20 |
| 3.2 Mininet.....   | 20 |
| 3.3 Implementation .....   | 21 |
| 3.4 Simulations and Results .....  | 22 |
| 3.4.1 Network diagram.....   | 22 |
| 4. CONCLUSION.....   | 28 |
| REFERENCES .....   | 29 |
| APPENDIX: PYRETIC FIREWALL CODE.....   | 31 |

## LIST OF FIGURES

|  |    |
|--|----|
| Figure 2.1: Structure of a Typical DDoS .....                | 9  |
| Figure 2.2: TCP Three-way Handshake.....                     | 11 |
| Figure 2.3: TCP SYN attack .....                             | 11 |
| Figure 2.4: SDN Reference Model .....                        | 14 |
| Figure 2.5: Three layers that can be attacked with DDoS..... | 16 |
| Figure 3.1: Network Diagram .....                            | 22 |
| Figure 3.2: Wireshark representation of DDoS attack.....     | 26 |
| Figure 3.3: Graphical representation of DDoS attack 1 .....  | 26 |
| Figure 3.4: Graphical representation of DDoS attack 2.....   | 26 |
| Figure 3.5: Analysis of DDoS attack .....                    | 27 |

## **LIST OF TABLES**

|  |    |
|--|----|
| Table 2.1: Benefits of SDN.....                                | 13 |
| Table 2.2: Good features of SDN in defeating DDoS attacks..... | 17 |
| Table 2.3: Selected Pyretic Policies.....                      | 18 |

# 1. INTRODUCTION

A major and also popular threat that has continuously reduced the availability of targeted services on the internet is the Distributed Denial of Service (DDoS) attacks. These DDoS attacks vary from high volume of flooding on the network to the misuse of application level vulnerabilities. While detecting the availability of a service or whether there is a congestion on the network appears not to be a herculean task, identifying the attackers in a DDoS presents a bigger challenge because of the possible similarities with legitimate traffic.

Many DDoS mitigation solutions exist today. A popular choice is BGP Remote Triggered Black Hole (RTBH), which instructs routers to drop all traffic to the target in order to decrease the load on the network [1]. Also, a wide variety of in-line DDoS mitigation hardware appliance exists that filters different kind of attacks [2].

In these days, there has been complications and difficulties in innovation and policy formation as a result of the closed and proprietary networks that we have now. And also, the fast development of technologies these days requires the development and implementation of new networking strategies. Software Defined Network (SDN) which provides programmability through decoupling of the control and data planes, and ensures simple programmable network devices instead of making them more complex is an emerging technology for these innovation task. A new suitable platform for executing and testing new concepts which also boosts innovative network design is offered by SDN, using its network programmability and the fact that isolated virtual networks can be defined by the control plane. It allows a real-time centralized control due to its ability to receive current network status [3].

SDN generally works with the controllers, that implement the control plane and switches that perform the data planes operations. OpenFlow protocol serves as the present standard through which controllers and switches communicate with each other. The OpenFlow creates a platform in which SDN controllers can manage forwarding behaviors of SDN switches by managing Flow Table entries. The low-level flow table entries are used by the switch to forward packets to appropriate hosts [5].

However, OpenFlow has improved the network architecture and prevented the conventionality of Network Research, the Network Programmers have to continuously contend with several challenges like maintaining complex book-keeping in a two-tiered system architecture and avoiding race conditions, which distributed systems are vulnerable to because the software used to control the networks has a lower layer abstraction. The effect of this led to the rise of Frenetic- a new Network Programming Language, which is easier, dependable, modular, and secure and also provides a high level abstraction to programmers through a declarative model [4].

It therefore becomes difficult to guard against Distributed Denial of Service because the victim is actually in the network and thereby make the entire network a vulnerability.

This report shows and explains how the use of Pyretic which is a member of the frenetic family using the SDN platform, prevent the system from a DDoS attacks with the approach of an Intelligent firewall.



## 2. LITERATURE REVIEW

### 2.1 Distributed Denial of Service (DDoS)

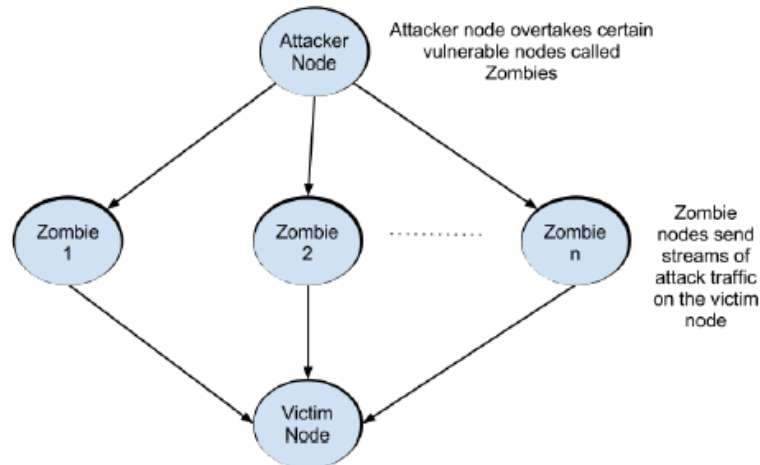


Figure 2.1: Structure of a Typical DDoS

The intention of designing the internet originally was to provide openness and scalability; and very less attention was given to securing it which has led to some vulnerabilities with respect to security [7]. The IP layer holds no support in verifying a source's authorization to access a service, therefore, the destination receives packets at any cost regardless of the packet sender and the server at the destination is left with the decision to either accept and service these requests. This has led to a class of security problem known as Denial of Service attacks where the vulnerability or the weak point of the system is the system itself [8].

A **Denial of Service** (DoS) attack targets the denial of access by legitimate users to a shared service or resource. The 'ping-of-death' is an example of a DoS attack where the attacker can cause an operating system to crash, freeze or reboot due to buffer overflow by simply sending large ICMP packets that is broken into multiple data-grams to a target system.

A Denial of Service attack can be launched in two forms. In the first form, the attacker exploits a software vulnerability of the targeted system by sending carefully fashioned packets as in the case of ICMP ping packets which eventually crashes the system. The second form is sending large volume of unusable or useless traffic to the targeted system and these type of traffic uses up all of the resources and prevents legitimate traffic from being serviced [7].

If this attempt of denying service to legitimate users come from a group of hosts, instead of only one host, it is called **Distributed Denial of Service** (DDoS). In this case, these group of host machines are vulnerable machines which have been taken over and are being coordinated by the malicious user. To launch DDoS attacks, the attacker uses botnets – large clusters of connected devices (PCs, Laptops, and Cellphones etc.) which are infected with a malware that allows the

attacker to have remote control over them. These individual infected devices which can be called a 'bot' are weapons of mass destruction used to send large amount of packets to the victim in order to deplete its resources and make the network unavailable to legitimate users. The attacker also has the liberty to send huge volume of packets to the targeted victim using spoofed source IP addresses hereby causing huge productivity and revenue loss especially when the victims turn out to be large businesses [3][8].

### **2.1.1 Effects of Design Principles of Internet Architecture and its Implication on DDoS Attacks**

Some of the design principles of the internet architecture that has created loopholes or generally caused the rise of DDoS attacks are:

1. Simple Core Network and Complex Edge of today's internet has resulted in the core to be too naïve, which only need to deliver IP packets without needing to understand about the network layer services [4], [7].
2. Fast Core Networks and Slow Edge networks which has created a limitation that traffic from high capacity core links can overpower the low capacity edge links when many sources want to talk to the same edge node which is the case in DDoS attacks [4], [7].
3. Decentralized Internet Management has resulted in lack of central control of the internet and the DDoS Defense schemes have to be deployed at various locations in a complicated distributed manner to be effective [4], [7].

### **2.1.2 Methods of DDoS Attacks**

DDoS attacks has two major impacts. One of these impacts is the consumption of the victim's resources. The victim in this case which generally could be a web server connected to the internet has a limited amount of resources to process incoming packets but when the traffic load exceeds its resource capacity, it begins to drop the packets to notify the senders (which basically consists of both the legitimate sources and illegitimate sources) to reduce their sending rates. While the legitimate user slows down on the rate at which it is sending packets, the attack or illegitimate sources maintain or even increase the rate at which the packets are being sent to the victim which in the long run exhausts the victim's resources, such as CPU and memory, and legitimate traffic won't be able to get serviced anymore [7].

The second impact is exhausting the network bandwidth which has an overall negative effect on the entire network as a whole. In this case, the entire network could go down instead of just the victim nodes because legitimate traffic get dropped especially when there is no clear mechanism to differentiate malicious nodes from legitimate ones as it shares the same congested links with illegitimate traffic.

A DDoS is largely measured by its attack power which is the level of resources consumed at the victim by the attack and consists of two (2) parameters. The first parameter is the volume of traffic which can be denoted by the amount of packets in a given time period and the second parameter is the level of resources that is used up per packet, which can be signified by CPU time or memory needed to process the packet [7].

The methods of DDoS attacks can be:

**Protocol Based Bandwidth attacks:** These attacks are based on exploiting the weaknesses of Internet protocols. TCP SYN flood and UDP flood are good examples of these.

TCP SYN flood:

This attack is aimed at exploiting the CPU memory. In a connection from host to server, a three-way handshake protocol is established before any actual data transfer occurs. In this three-way handshake, the host sends a SYN packet to initiate the handshake, the server replies with a SYN-ACK packet and the host lastly sends an ACK packet to establish a successful connection. An attacker exploits this process by leaving the handshake half open by not sending the last ACK packet. Such half-open state is stored in the server's memory and it keeps waiting for the host to send its final packet. This way, with several half-open connections on the server, it runs out of memory and crashes and legitimate clients are denied service [10].

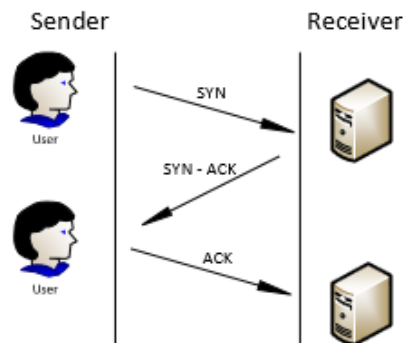


Figure 2.2: TCP Three-way Handshake

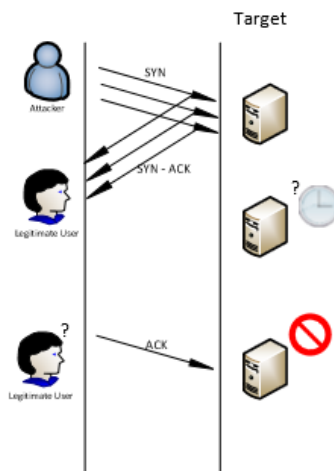


Figure 2.3: TCP SYN attack

UDP flood:

UDP flood is similar to ping flood. Instead of ping packets, UDP packets are bombarded against the server. UDP could be a lot more effective than ICMP in smaller networks as the size of the UDP packets are huge. The packet size could be set up to 65000bytes which could easily flood a given Ethernet network when several zombies are set up [11].

**Application Based attacks:** These types of attacks are focused on executing expensive operations on the targets. For example, a web site is forced to perform CPU and memory-intensive database operations by sending a large number of query to it which in turn leaves few resources to serve legitimate users. An example of this type of attack is the HTTP flood.

HTTP flood:

Generally, Hypertext Transfer Protocol (HTTP) is used over TCP port 80 in World Wide Web (WWW) applications which runs on the internet. Most firewalls, because of this, leave the TCP port 80 open to allow HTTP traffic to pass which in turns serves as prime target for the attackers. HTTP flood refers to an attack that bombards Web servers with HTTP requests and most times because a valid TCP connection has to be established for this request, attackers achieve this with the use of a bot's IP address. For example, an attackers can send HTTP requests to download a large file from the target using the botnet which can severely lead to resource consumption in the CPU, memory, and outbound internet links.

**Distributed Reflector attacks:** Here, the attacker tries to hide himself by relaying his attacks through third party nodes, called reflectors. A Distributed Reflector attack usually comprises of three stages. The first stage is related to any DDoS attack in which the attacker gains control of certain vulnerable nodes called zombies. In the second stage, the zombies are ordered to send spoofed traffic with the victim's IP address onto the third parties which are the reflector nodes. In the third stage, the reflectors are controlled to send the reply traffic to the victim nodes, which constitute the actual DDoS attack. The relay of the attacks through innocent third party reflectors makes the source trace back of such attacks extremely difficult [4].

**Infrastructure attacks:** These type of attacks aim to incapacitate the services of critical components of the Internet and possibly affect the whole of internet. An example of such attacks is the attack on DNS Root Servers [9]. DDoS attacks on such root servers can exhaust both their host and networking resources which can disrupt all the internet services which depend on such root servers.

### 2.1.3 DDoS Defense Schemes

1. **Attack Prevention:** This defense approach is aimed towards stopping the attack before they cause damage and it assumes the source IP addresses are being spoofed to hide to the real source of the attacker. It basically comes with a various packet filtering schemes which makes sure only valid source IP addresses are allowed to the network [7].

2. **Attack Detection:** Detecting attacks is critical in a system as the target is able to identify when an attack occurs and implementing the proper reaction procedures can prevent an actual damage or wastage of network bandwidth [7].
3. **Attack Source Identification:** After an attack has been detected, there is a need to trace back the IP of the attack sources whether it is being spoofed or not. This phase aims at tracing back these IP addresses irrespective of the information in the packet source address field [4], [7].
4. **Attack Reaction:** The techniques of detecting and tracing back attack sources leads us into actually trying to minimize the loss caused by DDoS attacks which means there has to be a form of reaction scheme put in place to filter or block this illegitimate traffic without affecting the legitimate traffic in transit [4], [7].

## 2.2 Software Defined Networking (SDN)

The uniqueness of SDN concept comes by the fact that it provides programmability through decoupling of control and data planes. SDN provides simple programmable network devices instead of making networking devices more complex. Furthermore, the network architecture of SDN brings about a separation of the control plane and data plane. With this model, the control of the network can be done separately on the control plane without affecting the data flows. The controller can then act as the Intelligence of the network instead of the switching devices. Meanwhile, the switches can be controlled externally by software without need of on-board intelligence. The separation of control from data planes provides not only a simpler programmable environment but a greater freedom for external software to define the behaviour of a network as well [3].

### 2.2.1 Benefits of SDN

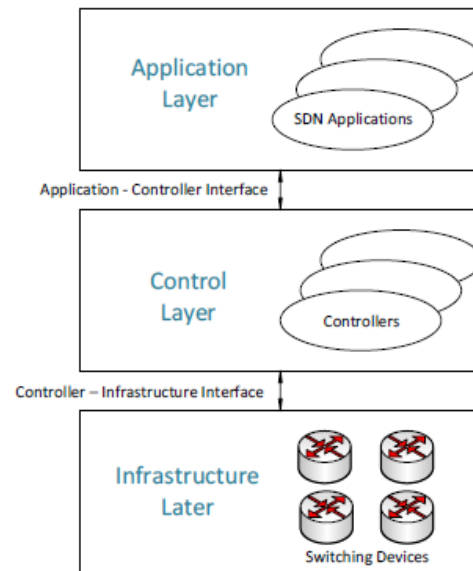
The table below shows a complete view of the benefits of SDN against conventional networks.

|               | <b>SDN</b>  | <b>Conventional Networks</b>   |
|---------------|---|--|
| Features      | - separated data and control plane<br>- programmability   | - a new protocol for every problem<br>- complex network control [14]   |
| Configuration | automated configuration with centralized validation   | error manual configuration   |
| Performance   | dynamic global control with cross layer information   | - limited information<br>-relatively static configuration  |
| Innovation    | - easy implementation of the software for innovations<br>- sufficient test environment with isolation<br>- rapid deployment using upgrade of the software | - hard hardware implementation for innovation<br>- limited testing environment<br>- long standardization process |

*Table 2.1: Benefits of SDN*

## 2.2.2 SDN Architecture Model

When it comes to differentiating between SDN and traditional networks, we can see that the physical devices become mere forwarding elements without control functions in which the intelligence of the network is attached to the control plane instead of the data plane devices. The communication between the control plane and data plane which is of utmost importance is made to function on top of open standard interfaces (OpenFlow) and this allows for the dynamic configuration of heterogeneous forwarding devices which is not found in traditional networks.



*Figure 2.4: SDN Reference Model*

This architecture consists of 3 layers, stacking over each other.

1. **Infrastructure Layer:** This layer holds the switching devices such as switches, routers, etc. in the data plane. These devices control the forwarding and data processing capabilities for the network. The packets processed by the devices are treated based on the flow rules provided by the controller [3].
2. **Control Layer:** This layer has a connection both to the application and the infrastructure layers. The SDN controller is a logical entity that receives instructions or requirements from the SDN application layer and relays them to the switching components. The controller also extracts information about the network from the hardware devices and communicates back to the SDN applications with an abstract view of the network, including statistics and events about what is happening.

The SDN architecture Application Programming Interfaces (APIs) are often referred to as Northbound and Southbound interfaces and they basically define the communication between the applications, controllers and switching components. The Northbound interface is defined as the connection between the controller and applications while the

Southbound interface is the connection between the controller and the switching components. . In case of a large administrative network domain multiple controllers will exist and they are connected via east-west interface. It is required because the controllers have to share network information and coordinate their decision-making processes [12], [13].

3. **Application Layer:** In order to meet the requirements of the users, applications are designed and placed on the application layer. These applications can access and control the infrastructure layer devices using the programmable platform of the control layer. For instance, SDN applications may include dynamic access control, seamless mobility and migration, server load balancing, and network virtualization. For example, an analytics application might be built to recognize suspicious network activity for security purposes [3].

One of the strengths of the SDN architecture is that it provides applications with the unique ability to obtain an abstracted view of the entire network by collecting information from the controller for decision-making purposes. These makes the network “smarter” by being able to analyze itself and integrate real-time information about networking activity with the applications.

### 2.2.3 OpenFlow and Open vSwitch

OpenFlow Protocol (OFP) is the first, industry specified, standardized SDN protocol which defines a way for the controller to communicate with switches. OpenFlow provides an open protocol to program the flow table in different switches and routers. OpenFlow is implemented on both ends of communication: on SDN controller and on forwarding devices.

Using OpenFlow, the forwarding states of each forwarding devices is being managed by the controller because of the ability for forwarding devices to export their network interfaces. The managed states are isolated into flow tables on such devices, which are basically set of packet header fields and the actions defined on them. The action defined on these fields are common packet operations like sending a packet to some ports or modifying protocol fields.

Deploying OFP-enabled SDN on physical or virtual networks, a network administrator can partition traffic into production and research flows. Researchers can control their own flows - by choosing the routes their packets follow and the processing they receive. In this way, researchers can try new routing protocols, security models, addressing schemes, and even alternatives to IP. On the same network, the production traffic is isolated and processed in the same way as today.

We also see that OFP enabled switches can forward packets based on matching rules as well as in traditional manner. Even in multi-vendor network infrastructure, carriers can easily deploy it ignoring vendor dependencies [5], [15].

### 2.3 DDoS in SDN Environment

We know for a fact that SDN offers a whole lot of benefits because of its ability to separate the control plane from the data plane but these doesn't still stop SDN from being a target of DDoS attacks. SDN features which includes the ability the view the network globally and dynamic updates among others facilitate the detection of DDoS attacks but the decoupling of the control plane from the data plane give rise to new type of attacks. An attacker can infiltrate the SDN system by launching attacks against the control, infrastructure and application layers [16].

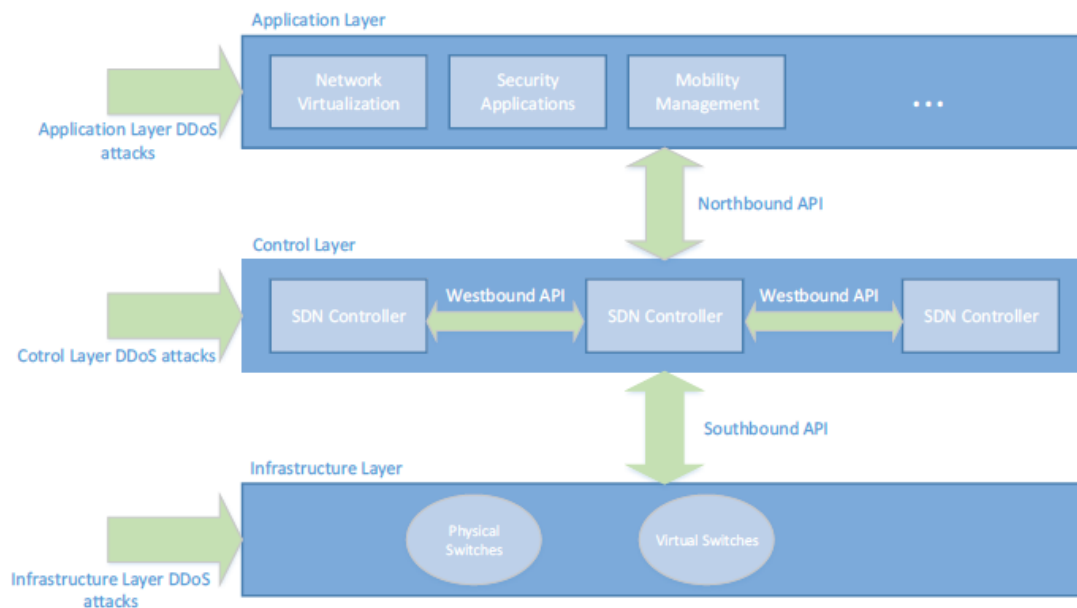


Figure 2.5: Three layers that can be attacked with DDoS

Looking at the OpenFlow architecture, the central management it provides helps in mitigating the weaknesses we see in today's internet. Using this architecture, we are able to detect, mitigate and prevent DDoS attacks more effectively as a result of the central controller which now becomes the brain of the network and manage all the switches within its range. Moreover, using the Frenetic as the Network Programming Language makes it much efficient and convenient in defeating DDoS attacks [3].

The table below shows a cross-section between the features of SDN and its benefits in mitigating DDoS attacks.

| Good features of SDN                                     | Benefits for defending DDoS attacks  |
|--|--|
| Separation from the control from data plane              | It is able to establish large scale attack and defence experiments easily. |
| A logical centralized controller and view of the network | It helps to build consistent security police                               |



|   |  |
|---|--|
| Programmability of the network by external applications   | It supports a process of harvesting intelligence from existing IDSs and IPSs |
| Software based traffic analysis                           | It improves the capabilities of a switch using any software-based technique. |
| Dynamic updating of forwarding rules and flow abstraction | It helps to respond promptly   |

Table 2.2: Good features of SDN in defeating DDoS attacks

## 2.4 Pyretic

The Idea of **Pyretic** is designing simple, reusable, high-level abstractions for programming SDNs, and efficient runtime systems that automatically generate the low-level rules on switches. Key elements in preventing a DDoS attack which includes packet-forwarding policy, monitoring network conditions, and dynamically updating policy to respond to network events are contained in these abstractions [17].

Conventionally, policies are implemented by incrementally installing physical rule after physical rule on switch after switch but in this case a pyretic policy is specified for the entire network at once, via a function from an input located packet (i.e., a packet and its location) to an output set of located packets [17].

One of the primary advantages of Pyretic's policies-as-abstract-functions approach to SDN programming is that it helps support modular programming. In traditional OpenFlow programming, the programmer cannot write application modules independently, without worrying that they might interfere with one another. Rather than forcing programmers to carefully merge multiple pieces of application logic by hand, a Pyretic program can combine multiple policies together using one of several policy composition operators, including parallel composition and sequential composition [17].

Looking at the SDN platforms that already exists, monitoring is observed as basically a side-effect of installing rules that send packets to the controller, or accumulate byte ad packet counters. As a result of this, the rules created by the programmers must be done in such a meticulous way that it monitors the network conditions and perform the right forwarding actions concurrently. Instead of these, Pyretic integrates monitoring into the policy function and supports a high- level query API. The programmer is able to merge both monitoring and forwarding using parallel composition easily [17].

With the introduction of Pyretic, the ability and chances in creating sophisticated SDN applications have increase exponentially. Pyretic comes with several examples of common enterprise and data-center network applications (hub, MAC-learning switch, traffic monitor, firewall, ARP server, network virtualization, and gate-way router). Since the initial release of Pyretic in April 2013, the community of developers has grown quickly. Some have built new

applications from scratch, while others have ported systems originally written on other platforms [17].

The table below shows some of the policies used in Pyretic

|                          | <b>Summary</b>  |
|--------------------------|---|
| <code>identity</code>    | returns original packet   |
| <code>none</code>        | returns empty set   |
| <code>match(f=v)</code>  | identity if field <code>f</code> matches <code>v</code> ,<br>none otherwise |
| <code>modify(f=v)</code> | returns packet with field <code>f</code> set to <code>v</code>              |
| <code>fwd(a)</code>      | <code>modify(port=a)</code>   |
| <code>flood()</code>     | returns one packet for each local port<br>on the network spanning tree      |

*Table 2.3: Selected Pyretic Policies*

## 2.5 Stateless and Stateful Firewall

Generally, firewalls are either software components or hardware devices that use a set of filtering rules to enforce certain security policies in order to prevent unauthorized access into and from the network. Filtering rules related to TCP and UDP bi-directional services (such as http, ftp, and telnet) have to allow both traffic directions to cross the firewall. A typical TCP or UDP session has a client which is the computer that initiates the session, and a server which hosts the service to be provided [18].

### 2.5.1 Stateless Firewalls

Stateless firewalls watch network traffic and restrict or block packets based on source and destination addresses or other static value. In a stateless firewall, the pattern or flow of the traffic are not known to the firewall. In a case where there is a disguised ‘correct’ packet in the flow, the stateless firewall is unable to account for it because it is control by a simple set of rules.

The filtering of a stateless firewall is also known as an Access Control List which does not inspect traffic in state but assess packet content statically and does not keep track of the state of the network connections.

A Stateless firewall filter basically helps to improve security through the use of packet filtering which permits the inspection of the inbound and outbound packet components and as well perform the necessary actions that has been specified for these packets. A typical use of a stateless firewall is to protect the Routing Engine processes and resources from malicious or untrusted packets. Stateless firewalls are typically faster and perform better under heavier traffic loads [18], [19].

### 2.5.2 Stateful Firewalls

In a Stateful firewall, the connection state of network are tracked (such as TCP streams or UDP communication) and substantial information (which include IP addresses, ports involved in the connection and the sequence numbers of the packets traversing the connection) about each connection state are stored in memory.

The monitoring of the connection state as well as incoming and outgoing packets by a stateful firewall is done over-time and the data is stored in a dynamic table. These data stored ensures that accurate filtering decisions are been carried out in such a way that decisions are not based only on the administrator defined set rules, but also on a framework that has been built by previous connections and packets belonging to the same connection.

It is discovered that at the connection setup time, it utilizes a lot CPU processes because entries that gratify a defined security policy are being created for TCP connections or UDP streams. After this process, packets in the same session are processed rapidly because of the pre-known knowledge of its existing pre-screened session and are allowed to pass through the firewall whereas sessions or packets that have no matching with the policy or existing table entry are denied entry.

It could be a hard task for a firewall to maintain the state of a connection depending on the protocol of the connection. For example, TCP is characteristically a stateful protocol as connections are established with a three-way handshake ("SYN, SYN-ACK, ACK") and ended with a "FIN, ACK" exchange. This means that all packets with "SYN" in their header received by the firewall are interpreted to open new connections. If the service requested by the client is available on the server, it will respond with a "SYN-ACK" packet which the firewall will also track. Once the firewall receives the client's "ACK" response, it transfers the connection to the "ESTABLISHED" state as the connection has been authenticated bi-directionally. This allows tracking of upcoming packets through the established connection. Concurrently, the firewall drops all packets which are not linked with an existing connection recorded in its state table (or "SYN" packets), preventing unwanted connections [20].

Stateful firewalls are better at identifying unauthorized and forged communications as well as implementing various IP Security (IPsec) function such as tunnels and encryption.

### 3. IMPLEMENTATION OF FIREWALL USING PYRETIC

In order to prevent DDoS attacks effectively, I have tried to implement both a stateless and stateful firewall using pyretic. I used mininet which is a virtual network simulator to test the implementation.

#### 3.1 Controller

There are several controllers available for use but for the case of our solution here, the POX controller is preferably used because POX and Pyretic both use python programming language.

POX is widely used for experiments, it is fast, lightweight and designed as a platform, so a custom controller can be built on top of it. The structure which allows for the communication with SDN switches is provided in POX. It is a development version of its ancestor NOX, and both are running on Python. POX can be immediately used as a basic SDN controller by using the stock components that come bundled with it. If need be to build a more complex SDN controller, it can be developed by creating new POX components [4].

For totality, we have other controllers like Floodlight which is open-source and written in Java. It's widely used also and has an advantage of allowing third parties to basically adjust or change the software and develop applications and Representational state transfer (REST) APIs. Another controller we have is the OpenDaylight which is also written in Java and has features that is able to collect information about the network, run algorithms to conduct analytics and also create new rules throughout the network.

#### 3.2 Mininet

This is the Network Simulator which is for testing in this project. It creates scalable and customizable Software Defined Networks (SDN) on a single PC by using Linux Processes in network namespaces. Mininet has an advantage in which complex topology testing can be done in a simple, efficient and inexpensive manner. Moreover, the code developed and tested on the mininet network can be moved to a real system with no or minute changes, for real-world testing, performance evaluation and deployment [3].

Mininet takes advantage of the Network namespaces which gives personal processes with their own network interfaces, routing tables and Address ARP tables. It uses process-based virtualization to run hosts and switches on a single operating system (OS) kernel. It allows large networks (up to 4096 host on a single operating system) with different topologies can be tested and emulated [3], [4].

Building a network in Mininet is as basically entering the command *mn*, which springs up a network that has one switch connected to two hosts and one reference controller. NOX is the default controller of Mininet.

### 3.3 Implementation

In the case here, the pyretic code running on the POX controller acts as both a stateless and stateful firewall.

The code functionality is as described below: Initially, the firewall checks a base .csv data file to see if a packet matches the rules defined in it. Narrowing down to this implementation, the data file contains specific rules that drop packets that matches it while packets that are contained in this data file rules are captured by the firewall.

The firewall tracks the flow of the traffic either from 'inside' or 'outside' and determines whether the traffic is TCP or UDP. After the firewall captures packets that are not defined in the data file, it inspects these packets to determine if it's a SYN packet in the case of TCP.

For this test, different hosts are used to simulate a DDoS attack as an implementation of a botnet using same ports on the hosts for each connection. When the firewall captures a SYN packet from a particular node up to a count of **10**, it automatically detects it as a DDoS attack and blocks further traffic originating from it.

Below is a description of some functions used in the implementation

#### **Firewall ():**

This class adds policy rules to pass or drop the packets.

add\_policy():

This method adds the policies to the dictionary and sets the value to True if it was not specified in the policy file.

drop\_rule():

Add flow to drop the matching packets on firewall

#### **PacketCapture ():**

Class for Capturing Packets on switch ports

packet\_capture():

Captures the packets and registers a callback for the captured packets.

packet\_inspection():

Parse the received packet to get the required fields

#### **PktParserFactory ():**

Factory class for parsing Ether Packets and returns either TCP or UDP tracker objects.

get\_ip\_info():

Extracts the ip header fields.

#### **TCPConnTrack ():**

Class for analysing TCP connections

track\_inside\_network():

Filter and install the flows for inside network traffic

track\_outside\_network():

Checks and installs the flows for outside network traffic

**UDPConnTrack ( ):**

Class for analysing UDP connections

track\_inside\_network():

Checks and installs the flows for insider network traffic

track\_outside\_network():

Checks and installs the flows for outside network traffic

### 3.4 Simulations and Results

The simulation carried out was done in Virtual box using a virtual machine that contained Mininet, POX controller, Pyretic and Wireshark

#### 3.4.1 Network diagram

A switch connected with 6hosts was used for the simulation. 3hosts (h1, h5, h6) were assumed to be the bots used in simulating DDoS traffic. Host 3 served as the node to be attacked.

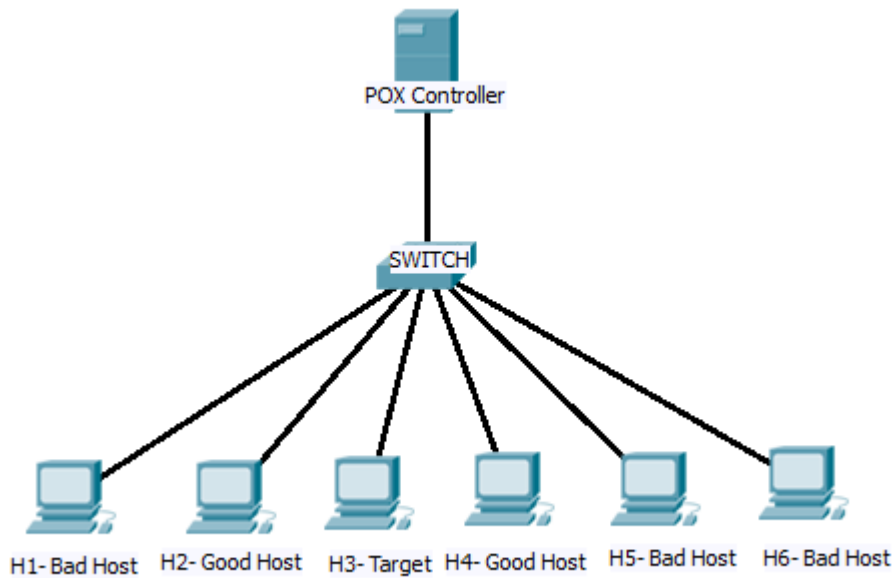
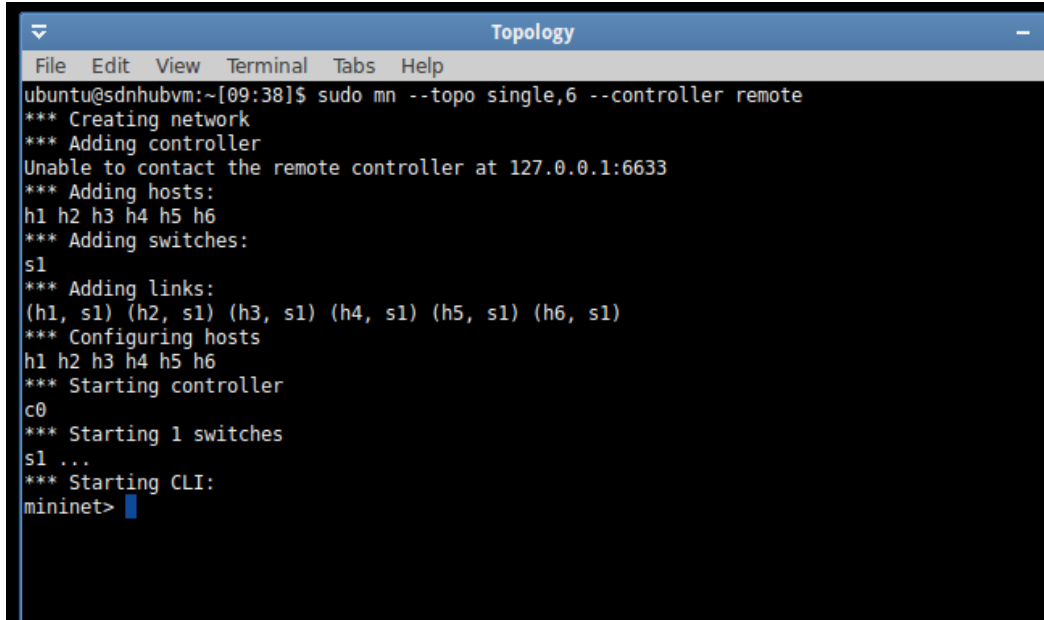


Figure 3.1: Network Diagram

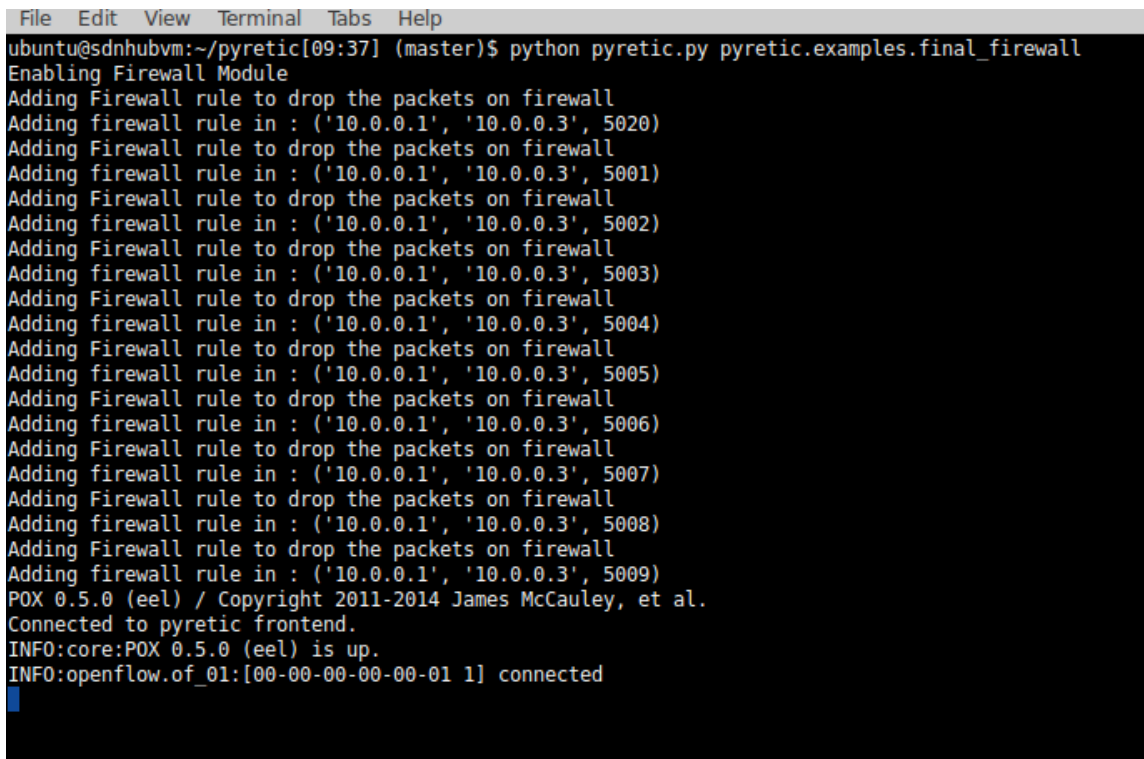
We can see that at the start-up of the topology, there was no controller running.



```
Topology
File Edit View Terminal Tabs Help
ubuntu@sdnhubvm:~[09:38]$ sudo mn --topo single,6 --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1) (h5, s1) (h6, s1)
*** Configuring hosts
h1 h2 h3 h4 h5 h6
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

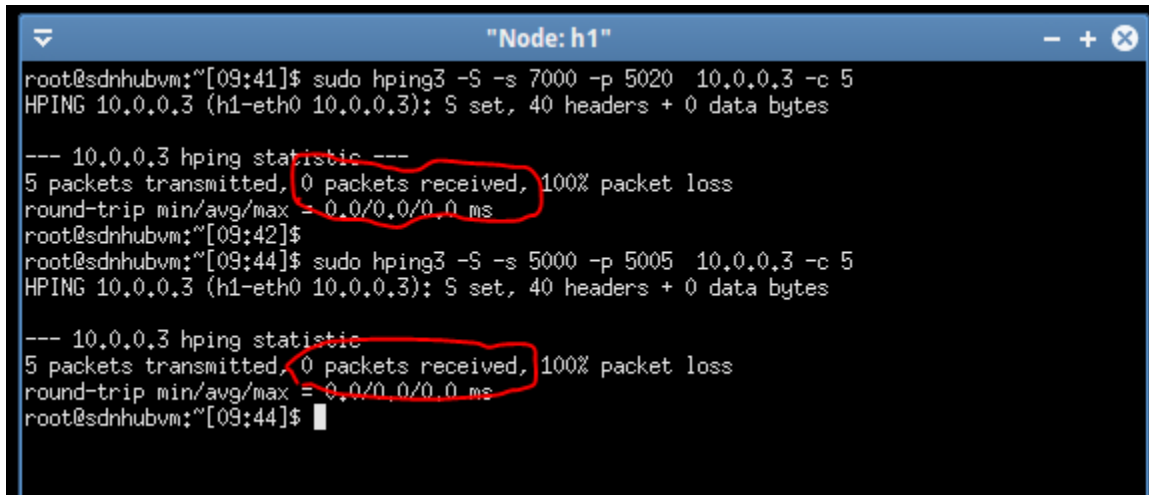
The POX controller was installed on the same virtual machine. On the next picture we started the POX controller which was running the pyretic.

Based on the rules in the base .csv file, the controller add these firewall rules as we can see in the picture below. The controller is started and the switch is connected. These firewall rules match a particular host and a port together.



```
File Edit View Terminal Tabs Help
ubuntu@sdnhubvm:~/pyretic[09:37] (master)$ python pyretic.py pyretic.examples.final_firewall
Enabling Firewall Module
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5020)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5001)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5002)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5003)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5004)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5005)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5006)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5007)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5008)
Adding Firewall rule to drop the packets on firewall
Adding firewall rule in : ('10.0.0.1', '10.0.0.3', 5009)
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
Connected to pyretic frontend.
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
```

The picture below demonstrates an initial experiment of the rules added to the firewall at start-up of the controller which is set to drop packet matching it. We can observe that the packets are dropped (0 packets received) because they match rules in the firewall.

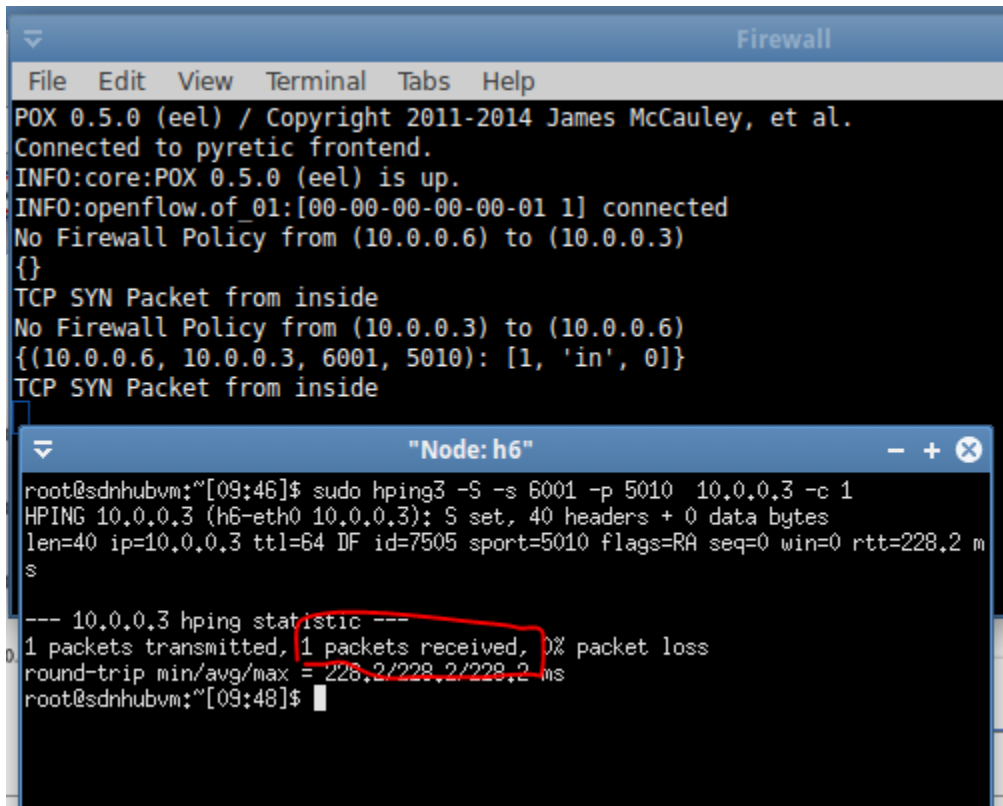


```
root@sdnhubvm:~# hping3 -S -s 7000 -p 5020 10.0.0.3 -c 5
HPING 10.0.0.3 (h1-eth0 10.0.0.3): S set, 40 headers + 0 data bytes

--- 10.0.0.3 hping statistic ---
5 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@sdnhubvm:~# hping3 -S -s 5000 -p 5005 10.0.0.3 -c 5
HPING 10.0.0.3 (h1-eth0 10.0.0.3): S set, 40 headers + 0 data bytes

--- 10.0.0.3 hping statistic ---
5 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@sdnhubvm:~#
```

The next picture demonstrates the flow of a non-matching firewall rule added. This TCP packet is captured by the firewall and passed on for inspection.



```
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
Connected to pyretic frontend.
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{}
TCP SYN Packet from inside
No Firewall Policy from (10.0.0.3) to (10.0.0.6)
{(10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0]}
TCP SYN Packet from inside

root@sdnhubvm:~# hping3 -S -s 6001 -p 5010 10.0.0.3 -c 1
HPING 10.0.0.3 (h6-eth0 10.0.0.3): S set, 40 headers + 0 data bytes
len=40 ip=10.0.0.3 ttl=64 DF id=7505 sport=5010 flags=RA seq=0 win=0 rtt=228.2 ms

--- 10.0.0.3 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 228.2/228.2/228.2 ms
root@sdnhubvm:~#
```



Looking closely at the picture below, we are able to observe and see the function of the firewall inspection. The firewall starts an incremental count of each packet that is not defined in its policy to drop.

```
File Edit View Terminal Tabs Help
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 0], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 1]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 1], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 1]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 1], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 2]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 2], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 3]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 3], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 3]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 3], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 4]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 4], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 4]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 4], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 5]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 5], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 5]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 5], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 6]}
ERROR PUSHING MESSAGE [ 'packet', { 'raw': '\xaa\x98\x02\x9a\x05\x03\xec\x9b\x08\x06\x00\x01\x08\x00\x06\x04\x00\x02\x9a\x05\x03\xec\x9b\n\x00\x00\x03\xaa\x98\x02\x9a\x05\x06', 'switch': '1, 'import': 3}]
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 6], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 6]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 6], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 7]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 7], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 7]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 7], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 8]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 8], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 8]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 8], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 9]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 9], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 9]}
No Firewall Policy from (10.0.0.6) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 9], (10.0.0.3, 10.0.0.6, 5010, 6001): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 5010): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 10]}

```

As a result of a defined rule in the firewall functionality that blocks a source that sends a packet up to a count of 10, we can see this functionality being implemented in the picture below. Host 6 which is part of the botnet sends packets and it was received until a count of 10 which the firewall detected as a DDoS attack and then the node gets blocked.

```
File Edit View Terminal Tabs Help
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 9], (10.0.0.3, 10.0.0.6, 5020, 7000): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 7000, 5020): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 9]}
No Firewall Policy from (10.0.0.3) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 9], (10.0.0.3, 10.0.0.6, 5020, 7000): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 7000, 5020): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 10]}
No Firewall Policy from (10.0.0.6) to (10.0.0.3)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 10], (10.0.0.3, 10.0.0.6, 5020, 7000): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 7000, 5020): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 6001, 80): [1, 'in', 10]}
DDoS attack detected!
Adding firewall rule in : ('10.0.0.6', '10.0.0.3', 80)
negate:
union:
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5009)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5005)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5008)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5001)
match: ('srcip', IPv4Network('10.0.0.6/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 801)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5004)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5007)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5003)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5006)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5020)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5002)
No Firewall Policy from (10.0.0.3) to (10.0.0.6)
{(10.0.0.3, 10.0.0.6, 80, 6001): [1, 'in', 10], (10.0.0.3, 10.0.0.6, 5020, 7000): [1, 'in', 0], (10.0.0.6, 10.0.0.3, 7000, 5020): [1, 'in', 0]}
DDoS attack detected!
Adding firewall rule in : ('10.0.0.3', '10.0.0.6', 6001)
negate:
union:
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5009)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5005)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5008)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5001)
match: ('srcip', IPv4Network('10.0.0.6/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 801)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5004)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5007)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5003)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5006)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5020)
match: ('srcip', IPv4Network('10.0.0.1/32')) ('dstip', IPv4Network('10.0.0.3/32')) ('dstport', 5002)

```

```
Node: h6
round-trip min/avg/max = 138.0/138.0/138.0 ms
root@sdnhubw1:~# sudo hping3 -S -s 6001 -p 80 10.0.0.3 -c 1
HPING 10.0.0.3 (64-eth0 10.0.0.3): S set, 40 headers + 0 data bytes
len=40 ip=10.0.0.3 ttl=64 DF 167/511 sport=80 flags=RA seq=0 win=0 rtt=179.4 ms
--- 10.0.0.3 hping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 138.0/138.0/138.0 ms
root@sdnhubw1:~# sudo hping3 -S -s 6001 -p 80 10.0.0.3 -c 1
HPING 10.0.0.3 (64-eth0 10.0.0.3): S set, 40 headers + 0 data bytes
--- 10.0.0.3 hping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@sdnhubw1:~#
```

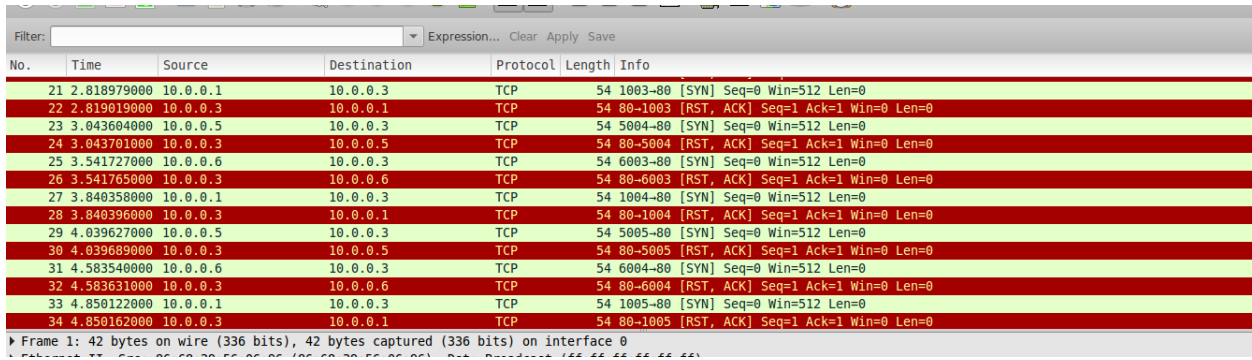


Figure 3.2: Wireshark representation of DDoS attack

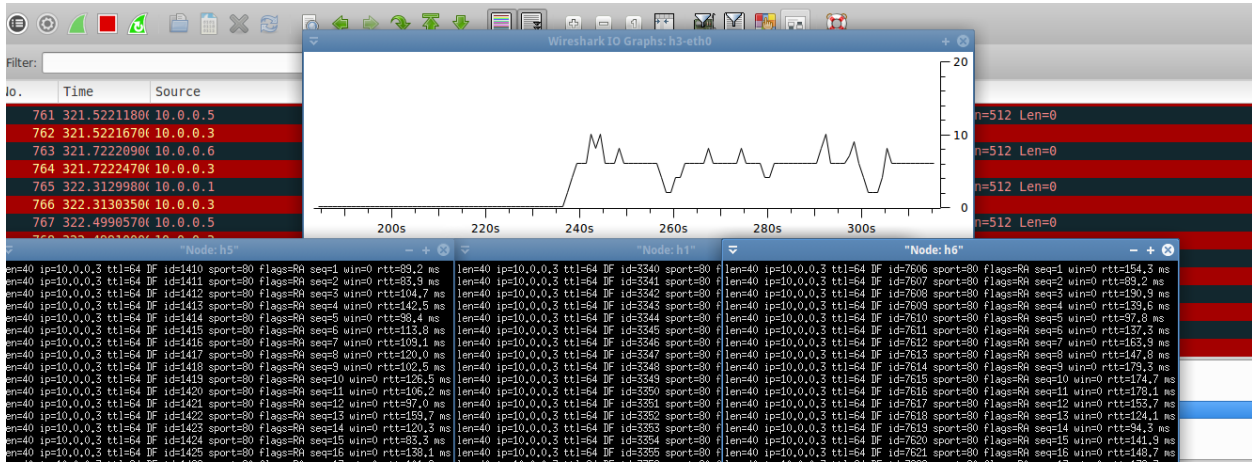


Figure 3.3: Graphical representation of DDoS attack 1

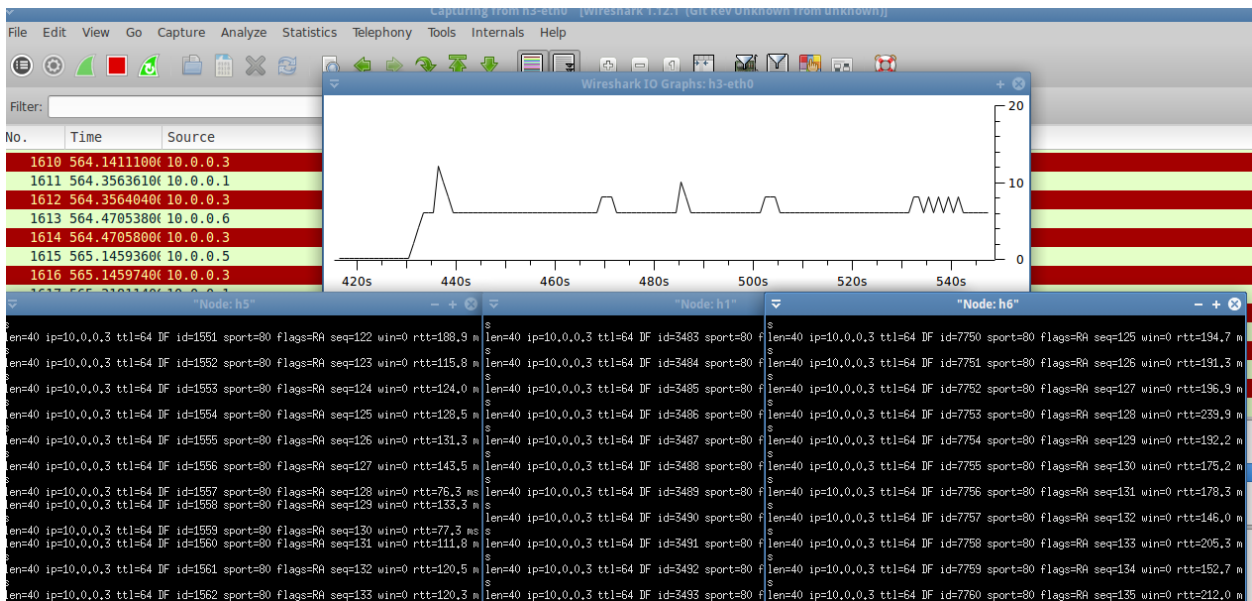


Figure 3.4: Graphical representation of DDoS attack 2

Basically, the picture below shows how DDoS attack use up the resources of the target. We can see that h1, h5 and h6 send SYN packets to Host 3 in which they get a SYN-ACK packet reply from H3, but they never respond with an ACK. This particular operation after some point depletes the resources of the target; in this case Host 3.

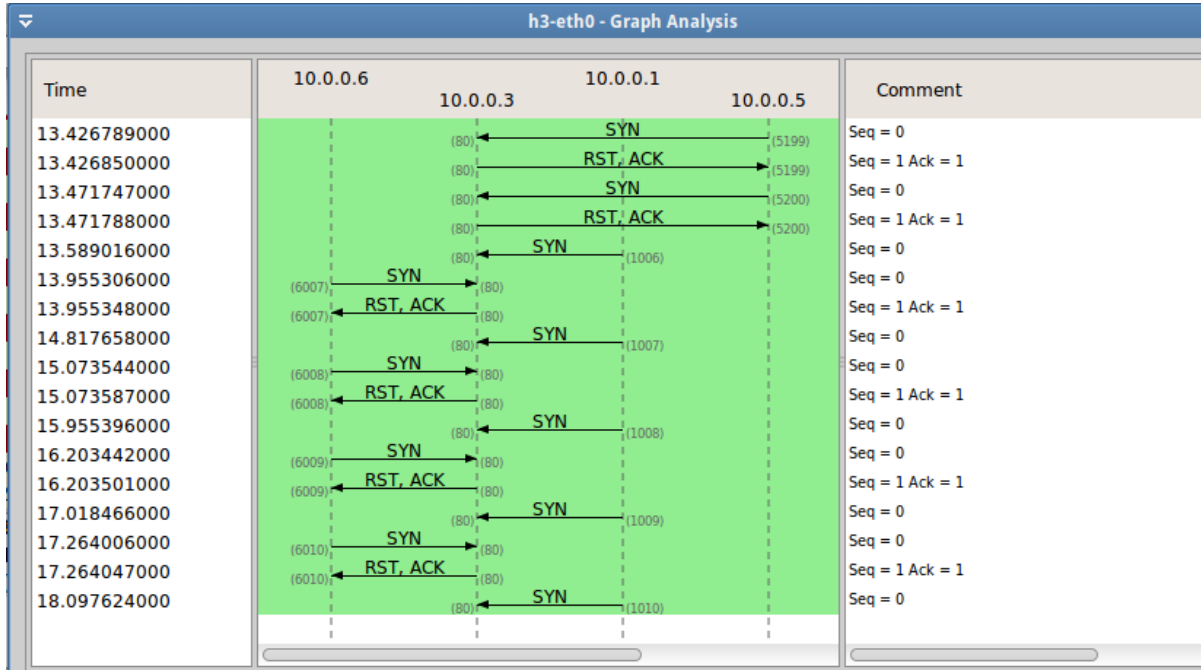


Figure 3.5: Analysis of DDoS attack

## 4. CONCLUSION

During this course of this project, we looked at how a stateless and stateful pyretic firewall is used in preventing the event of a DDoS attack which is a key threat in the networking industry today. Early detection was of utmost importance in the architecture to prevent the SDN controller from being compromised which could lead to entire network failure.

The identification of attackers is performed by capturing and counting the packet originating from the source. On positive detection, blocking flows will be pushed to the switch to prevent such nodes from sending more packets that could cause congestion in the network.

The use of Pyretic based on its swiftness in detection and prevention of DDoS attacks gives an idea on how it could be utilized furthermore to build an efficient, robust, and secure future networks.

## REFERENCES

- [1] W. Kumari, D. McPherson, RFC 5635: *Remote Triggered Black Hole Filtering with Unicast Reverse Path Forwarding (uRPF)*, 2009
- [2] C. Dillon, M. Berkelaar, *OpenFlow (D)DoS Mitigation*, 9<sup>th</sup> February, 2014
- [3] Martina Stoyanova Todorova and Stamelina Tomova Todorova, *DDoS Attack Detection in SDN-based VANET Architectures*, June 2016.
- [4] Sumanth M. Sathyanarayana, *Software Defined Network Defense*, 2011-2012
- [5] Jay Shah, *Implementation and Performance Analysis of firewall on Open vSwitch*, 29<sup>th</sup> April, 2015
- [6] Alauddin Shieha, University of Colorado Boulder, *Application Layer Firewall Using OpenFlow*, 24<sup>th</sup> June, 2014
- [7] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao, *Survey of network-based defense mechanisms countering the DoS and DDoS problems*. ACM COMP. SURV, 39(1), 2007
- [8] Virgil D. Gligor. A note on denial-of-service in operating systems. IEEE Trans Software Eng., 10(3):320-324, 1984
- [9] S. Cheung, *Denial of service against the domain name system*, Security Privacy, IEEE, 4(1):40-45, Jan.-Feb, 2006
- [10] Khaled M. Elleithy, *Denial of Service Attack Techniques: Analysis, Implementation and Comparison*, IJISCI Journal, vol. 3, no.1, pp. 66-71
- [11] Subramani rao Sridhar rao, *Denial of Service attack and Mitigation techniques: Real time implementation with detailed analysis*, SANS Institute InfoSec Reading Room, 2011
- [12] H. Yin et al, *SDNi: A message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains*, Jun. 2012
- [13] <https://www.sdxcentral.com/sdn/definitions/inside-sdn-architecture/>
- [14] *Software-Defined Networking: The New Norm for Networks*, ONF White Paper, Open Networking Foundation, April 2012
- [15] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner, *OpenFlow: Enabling Innovation in Campus Networks*, SIGCOMM Comput. Commun. Rev., 38(2):69-74, March 2008.
- [16] Q. Yan; F. R. Yu, *Distributed denial of service attacks in software-defined networking with cloud computing*, IEEE Communications Magazine, 2015, Volume: 53, Issue: 4, Pages: 52 - 59
- [17] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker, *Modular Programming with Pyretic*, 2013

- [18] Zouheir Trabelsi, *Teaching Stateless and Stateful firewall filtering: A hands-on Approach*, UAE University, Proceedings of the 16th Colloquium for Information Systems Security Education, June 11-13, 2012
- [19] <https://www.linkedin.com/pulse/stateful-vs-stateless-firewalls-utkarsh-bhargava>
- [20] [https://en.wikipedia.org/wiki/Stateful\\_firewall](https://en.wikipedia.org/wiki/Stateful_firewall)
- [21] <http://www.cs.princeton.edu/~jrex/papers/pyretic-login13.pdf>
- [22] <https://www.youtube.com/watch?v=hq7L4davQy8&feature=youtu.be>
- [23] <https://github.com/nampho2/CS6250#assignment-7-sdn-and-firewalls>
- [24] <https://www.pacificsimplicity.ca/blog/reading-packet-hex-dumps-manually-no-wireshark>
- [25] [https://github.com/jms30/SDN\\_Firewall](https://github.com/jms30/SDN_Firewall)

## APPENDIX: PYRETIC FIREWALL CODE

```
from pyretic.lib.corelib import *
from pyretic.lib.std import *
from pyretic.lib.query import *
from netaddr import IPNetwork, IPAddress
from types import *
import os
import csv
# Policy file for Firewall Rules
PolicyFile = "%s/pyretic/firewall-policies.csv" % os.environ[ 'HOME' ]

class Firewall(DynamicPolicy):
    """
    Class to add and remove Firewall rules
    """

    def __init__(self):
        """
        Initializes the Firewall class
        """
        self.firewall = {}
        super(Firewall,self).__init__(true)
        print "Enabling Firewall Module"
        self.register_rules()

    def register_rules(self):
        """
        Register Policy rules to filter traffic
        """
```

```

"""
with open(PolicyFile, 'rb') as f:
    reader = csv.reader(f)
    for row in reader:
        if row[1] != "srcip":
            self.add_policy(srcip=row[0], dstip=row[1], port=int(row[2]), value=self.str_bool(row[3]))

def str_bool(self, policy):
    """
    This method convert a list of words to either True or False (Boolean)
    @args:
        policy: yes or no argument with different possible options
    """
    if str(policy).lower() in ("yes", "y", "true", "t", "allow", "permit", "1"):
        return True
    if str(policy).lower() in ("no", "n", "false", "f", "block", "deny", "0", ""):
        return False

def add_policy(self, srcip="*", dstip="*", port=0, value=True):
    """
    This method adds the policies to the dictionary and sets the value to True if it was not
    specified in the csv file.
    @args:
        srcip: Source IP address of packet
        dstip: Destination IP address of packet
        port: Destination port of packet
        value: Boolean variable to apply the firewall rules
    """
    if value:

```



```

    print "Adding Firewall rule to drop the packets on firewall"
        self.drop_rule((srcip,dstip,port))
    else:
    print "Adding Firewall rule to forward/bypass the firewall"
        self.forward_rule((srcip,dstip,port))

def drop_rule(self, match_tuple):
    """
    Add flow to drop the matching packets on firewall
    @args:
        match_tuple: (srcip, dstip, srcport, dstport) Tuple
    """
    if match_tuple in self.firewall:
        print "Firewall rule for :%s already exists" % str(match_tuple)
        return
    self.firewall[match_tuple]=True
    print "Adding firewall rule in : %s" % str(match_tuple)
    self.update_policy()

def forward_rule(self, match_tuple):
    """
    Delete flow to forward the matching packets from firewall
    @args:
        match_tuple: (srcip, dstip, srcport, dstport) Tuple
    """
    try:
        del self.firewall[match_tuple]
        print "Deleting firewall rule in : %s" % str(match_tuple)
        self.update_policy()
    except:

```

```

        pass
    try:
        del self.firewall[match_tuple]
        print "Deleting firewall rule in %s: %s" % str(match_tuple)
        self.update_policy()
    except:
        pass

def update_policy (self):
    """
    Adds the policy flows to match the packets
    """
    self.policy = ~union([match(srcip=tup[0], dstip=tup[1], dstport=tup[2]) for tup in
self.firewall.keys()])

class PacketCapture(object):
    """
    Class for Capturing Packets on switch ports
    """

    def __init__(self, firewall_obj):
        """
        Initializes the class
        @args:
            firewall_obj: Firewall class object passed for reference
        """
        self.flow_table = {}
        self.firewall_obj = firewall_obj

        self.insideNetwork = ["10.0.0.0/24"] # Modify it according to your switch internal network. In
mininet this is the default one

```

```

def packet_capture(self):
    """
    Captures the packets and registers a callback for the captured
    packets
    """
    pkt = packets()
    pkt.register_callback(self.packet_inspection)
    return pkt

def packet_inspection(self, pkt):
    """
    Parse the received packet to get the required fields
    @args:
        pkt = Captured packet on switch port
    """
    if pkt['ethype'] == IP_TYPE:
        parser = PktParserFactory(self, pkt) # Returns the TCP/UDP class handles
        tracker = parser.get_conn_track_obj()
        if tracker:
            tracker.track_network()

def checkIPinside(self, ip):
    """
    Check if the IP is from inside the network or not.
    """
    for network in self.insideNetwork:
        if IPAddress(str(ip)) in IPNetwork(network):
            return True
    return False

```

```

class PktParserFactory(object):
    """
    Base class for parsing Ether Packets
    """

    def __init__(self, cap_obj, pkt):
        """
        Initializes the class with packet raw data
        @args:
            cap_obj: PacketCapture class object for references
            pkt: Received packet raw dump
        """
        self.cap_obj = cap_obj
        self.frwl_obj = self.cap_obj.firewall_obj
        self.pkt = pkt
        self.raw_bytes = [ord(c) for c in self.pkt['raw']]
        self.get_ip_info()

    def get_eth_payload(self):
        """
        Returns the ether payload
        """
        self.eth_payload_bytes = self.raw_bytes[self.pkt['header_len':]]

    def get_ip_info(self):
        """
        Returns the ip fields like ip version, payload and proto

```

```

"""
self.get_eth_payload()
ihl = (self.eth_payload_bytes[0] & 0b00001111)
ip_header_len = ihl * 4
self.ip_version = (self.eth_payload_bytes[0] & 0b11110000) >> 4
self.ip_payload_bytes = self.eth_payload_bytes[ip_header_len:]
self.ip_proto = self.eth_payload_bytes[9]

def get_conn_track_obj(self):
    """
    Returns the object of TCP or UDP conn tracking classes
    depending on the proto type
    """
    if self.ip_proto == 0x06:
        return TCPConnTrack(self)
    elif self.ip_proto == 0x11:
        return UDPConnTrack(self)
    elif self.ip_proto == 0x01:
        print "ICMP packet"
        print self.frw_obj.policy

def check_policy(self, srcip="", dstip="", port=0):
    """
    This method checks the src ip, dst ip, and port number of the packet against the rules and
    return the value of the corresponding policy.
    @args:
        srcip: Source IP address of packet
        dstip: Destination IP address of packet
        port: Destination Port of packet
    """

```

```

key = (srcip, dstip, port)
if key in self.frwl_obj.firewall:
    print "Policy From (%s) to (%s) found." % (srcip, dstip)
    return self.frwl_obj.firewall[key]
else:
    print "No Firewall Policy from (%s) to (%s)" %(srcip, dstip)
    return False

```

```

class TCPConnTrack(object):
    """
    Class for analysing TCP connections
    """

    def __init__(self, parser_obj):
        """
        Initializes the TCP class
        @args:
            parser_obj: PktParserFactory class object for references
        """
        self.parser_obj = parser_obj
        self.cap = self.parser_obj.cap_obj
        self.set_flags()

    def set_flags(self):
        """
        Extracts and Sets the TCP flags (SYN, ACK, PSH)
        """
        flags_byte = self.parser_obj.eth_payload_bytes[33]
        decode = '{0:08b}'.format(flags_byte)

```

```

self.SYN = decode[-2]
self.PSH = decode[-4]
self.ACK = decode[-5]

def track_network(self):
    """
    Tracks inside and outside network traffic
    """
    if self.cap.checkIPinside(self.parser_obj.pkt['srcip']):
        self.track_inside_network()
    else:
        self.track_outside_network()

def track_inside_network(self):
    """
    Filter and install the flows for inside network traffic
    """
    if self.parser_obj.check_policy(self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'],
self.parser_obj.pkt['dstport']) == True:
        print "Packet matched the rule and dropped! "
        return True

    key = (self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'], self.parser_obj.pkt['srcport'],
self.parser_obj.pkt['dstport'])

    if key in self.cap.flow_table:
        print self.cap.flow_table
        if self.cap.flow_table[key][:2] == [1, "out"] and self.SYN and self.ACK:
            self.cap.flow_table[key][0] += 1
            self.cap.flow_table[key][1] = "in"
            print "TCP SYN ACK packet from inside"
            return True

        elif self.cap.flow_table[key][:2] == [2, "out"] and self.ACK:

```

```

self.cap.flow_table[key][0] += 1

self.cap.flow_table[key][1] = "in"

print "TCP ACK Packet from inside "

return True

elif self.cap.flow_table[key][0] == 3 and self.PSH:

self.cap.flow_table[key][0] += 1

print "TCP handshacke is done. First packet of connection from inside.MOD is done "

return True

elif self.cap.flow_table[key][0] == 4:

print "TCP handshacke is done. Second packet of connection from inside.MOD is done "

del self.flow_table[key] # Removes the key from flow table not to get it overflow

return True

elif self.cap.flow_table[key][2] < 10: # It's a check to prevent from DDOS

self.cap.flow_table[key][2] += 1

return True

elif self.cap.flow_table[key][2] >= 10:

del self.cap.flow_table[key]

print "DDOS attack detected!"

# Adding Rule to drop the packets from this source next time

self.cap.firewall_obj.drop_rule((str(self.parser_obj.pkt['srcip']), str(self.parser_obj.pkt['dstip']),

int(self.parser_obj.pkt['dstport']))) #drop the pkt

print self.cap.firewall_obj.policy

return False

else: # First TCP packet of a flow

if self.SYN:

print self.cap.flow_table

self.cap.flow_table[key] = [1, "in", 0]# First is for # packets, second is for traffic direction and

third to prevent DDOS

print "TCP SYN Packet from inside "

return True

```



```

def track_outside_network(self):
    """
    Checks and installs the flows for outside network traffic
    """
    if self.parser_obj.check_policy(self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'],
self.parser_obj.pkt['dstport']) == True:
        print "Packet matched the role and dropped"
        return True

    key = (self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'], self.parser_obj.pkt['srcport'],
self.parser_obj.pkt['dstport'])
    if key in self.cap.flow_table:
        if self.cap.flow_table[key][:2] == [1, "in"] and self.SYN and self.ACK:
            self.cap.flow_table[key][0] += 1
            self.cap.flow_table[key][1] = "out"
            print "TCP SYN ACK packet from outside"
            return True
        elif self.cap.flow_table[key][:2] == [2, "in"] and self.ACK:
            self.cap.flow_table[key][0] += 1
            self.cap.flow_table[key][1] = "out"
            print "TCP ACK Packet from outside"
            return True
        elif self.cap.flow_table[key][0] == 3 and self.PSH:
            self.cap.flow_table[key][0] += 1
            print "TCP handshacke is done. First packet of connection from outside."
            return True
        elif self.cap.flow_table[key][0] == 4:
            del self.cap.flow_table[key]
            return True
        elif self.cap.flow_table[key][2] < 10:
            self.cap.flow_table[key][2] += 1
            return True

```

```

elif self.cap.flow_table[key][2] > 10:
    del self.cap.flow_table[key]
    print "DDOS attack detected!"

    self.cap.firewall_obj.drop_rule((str(self.parser_obj.pkt['srcip']), str(self.parser_obj.pkt['dstip']),
int(self.parser_obj.pkt['dstport']))) #drop the pkt

    return False

else:
    if self.SYN:

        self.cap.flow_table[key] = [1, "out", 0] # the first parameter is a counter for the number of
packets from that flow, the second is direction, third to prevent from DDOS

        print "Switche Module: TCP SYN Packet from outside "

        return True

```

```

class UDPConnTrack(object):
    """
    Class for analysing UDP connections
    @args:
        parser_obj: PktParserFactory class object for references
    """

```

```

def __init__(self, parser_obj):
    """
    Initializes the UDP class
    """
    self.parser_obj = parser_obj
    self.cap = self.parser_obj.cap_obj

```

```

def track_network(self):
    """
    Tracks inside and outside network traffic
    """

```

```

if self.cap.checkIPinside(self.parser_obj.pkt['srcip']):
    self.track_inside_network()
else:
    self.track_outside_network()

def track_inside_network(self):
    """
    Checks and installs the flows for insider network traffic
    """
    if self.parser_obj.check_policy(self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'],
self.parser_obj.pkt['dstport']) == True:
        print "Packet matched the rule and dropped"
        return True

    key = (self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'], self.parser_obj.pkt['srcport'],
self.parser_obj.pkt['dstport'])

    if key in self.cap.flow_table:
        print self.cap.flow_table
        if self.cap.flow_table[key][:2] == [1, "out"]:
            self.cap.flow_table[key][0] += 1
            self.cap.flow_table[key][1] = "in"
            print "UDP Packet from inside. Counter is 2 now"
            return "Pktout"

        elif self.cap.flow_table[key][:2] == [2,"out"]:
            self.cap.flow_table[key][0] += 1
            self.cap.flow_table[key][1] = "in"
            print "UDP Packet from inside. Counter is 3 now"
            return "Pktout"

        elif self.cap.flow_table[key][:2] == [3, "out"]:
            self.cap.flow_table[key][0] += 1
            self.cap.flow_table[key][1] = "in"
            print "UDP Packet from inside. Counter is 4 now"

```

```

        return "Mod"

    elif self.cap.flow_table[key][:2] == [4, "out"]:
        print "UDP flow is safe. Flow is deleted from flow table"
        del self.flow_table[key]
        return "Mod"

    elif self.cap.flow_table[key][2] < 10:
        self.cap.flow_table[key][2] += 1
        return "Pktout"

    elif self.cap.flow_table[key][2] >= 10:
        del self.cap.flow_table[key]
        print "DDOS attack detected"

        self.cap.firewall_obj.drop_rule((str(self.parser_obj.pkt['srcip']), str(self.parser_obj.pkt['dstip']),
int(self.parser_obj.pkt['dstport'])))

        print self.cap.firewall_obj.policy
        return False

    else:
        self.cap.flow_table[key] = [1, "in", 0]
        print "First UDP Packet from inside."
        return True

def track_outside_network(self):
    """
    Checks and installs the flows for outside network traffic
    """
    if self.parser_obj.check_policy(self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'],
self.parser_obj.pkt['dstport']) == True:
        print "Packet matched the role and dropped"
        return True

    key = (self.parser_obj.pkt['srcip'], self.parser_obj.pkt['dstip'], self.parser_obj.pkt['srcport'],
self.parser_obj.pkt['dstport'])

    if key in self.cap.flow_table:

```

```

if self.cap.flow_table[key][:2] == [1, "in"]:
    self.cap.flow_table[key][0] += 1
    self.cap.flow_table[key][1] = "out"
    print "UDP Packet from outside. Counter is 2 now"
    return "Pktout"

elif self.cap.flow_table[key][:2] == [2, "in"]:
    self.cap.flow_table[key][0] += 1
    self.cap.flow_table[key][1] = "out"
    print "UDP Packet from outside. Counter is 3 now"
    return "Pktout"

elif self.cap.flow_table[key][:2] == [3, "in"]:
    self.cap.flow_table[key][0] += 1
    self.cap.flow_table[key][1] = "out"
    print "UDP Packet from outside. Counter is 4 now"
    return "Mod"

elif self.cap.flow_table[key][0] == [4, "in"]:
    del self.cap.flow_table[key]
    print "UDP flow is safe. Flow is deleted from flow table."
    return "Mod"

elif self.cap.flow_table[key][2] < 10:
    self.cap.flow_table[key][2] += 1
    print "UDP packets that might belong to another type of applications or retransmission."
    return "Pktout"

elif self.cap.flow_table[key][2] > 10:
    del self.cap.flow_table[key]
    print "DDOS attack detected"

    self.cap.firewall_obj.drop_rule((str(self.parser_obj.pkt['srcip']), str(self.parser_obj.pkt['dstip']),
int(self.parser_obj.pkt['dstport'])))

    return False

else:

```

```
if self.SYN:
    self.cap.flow_table[key] = [1, "out", 0]
    print "First UDP Packet from outside."
    return "Pktout"
```

```
def main ():
```

```
    # Initializes the Firewall
```

```
    policy_obj = Firewall()
```

```
    # Initializes the Packet parsing and filtering
```

```
    pkt_cap = PacketCapture(policy_obj)
```

```
    return (policy_obj >> pkt_cap.packet_capture()) + policy_obj >> flood()
```