Early Detection of DDoS Attacks in Software Defined Networks Controller

By

Seyed Mohammad Mousavi

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Applied Science
in
Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

©2014
Seyed Mohammad Mousavi

The undersigned recommend to
the Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

Early Detection of DDoS Attacks in Software Defined Networks Controller

Submitted by

Seyed Mohammad Mousavi

in partial fulfillment of the requirements for the degree of
Master of Applied Science in Electrical and Computer Engineering

---

Chair, Roshdy Hafez, Department of Systems and Computer Engineering

---

Thesis Supervisor, Prof. Marc St-Hilaire

Carleton University
May 2014

Abstract

Software Defined Networks (SDN) is a new network architecture that provides central control over the network. This control works as if it is an operating system that can send instructions and apply changes through its interface. This operating system is called the controller. Although central control is the major advantage of SDN, it is also a single point of failure if it is made unreachable by a Distributed Denial of Service Attack (DDoS).

Two main objectives of this study are utilizing the central control of SDN for attack detection and, proposing a solution that is effective and lightweight in terms of the resources that it uses.

This research shows how DDoS attacks can exhaust controller resources and provides a solution to detect such attacks based on entropy variation of destination IP address. This method is able to detect DDoS within the first five hundred packets of the attack traffic.

Acknowledgements

First I thank GOD the almighty for helping me along the way giving me the strength to work on this research.

I would like to thank Prof. Marc St-Hilaire. He was the first to introduce the topic of this research and he supported me with his advice, supervision and support.

Lastly, I would like to thank my wife for her support and encouragement.

Table of Contents

Abstract ... iii

Acknowledgements ... iv

Table of Contents ... v

List of Tables ... vii

List of Figures ... viii

List of Appendixes ... ix

List of Acronyms ... x

Chapter 1 ... 1

Introduction ... 1

1.1 Problem Statement and Motivation ... 1

1.2 Research Objectives ... 4

1.3 Research Contributions ... 4

1.4 Thesis Organization ... 5

Chapter 2 ... 6

Background and Related Work ... 6

2.1 Software Defined Networks ... 6

2.2 Openflow Protocol ... 7

2.3 Openflow Specifications ... 8

2.3.1 Openflow Switch ... 8

2.3.2 Secure Channel ... 11

2.4 Distributed Denial of Service Attack and its Mitigation ... 12

2.4.1 Types of DDoS Attacks ... 12

2.4.2 Anomaly Detection for DDoS Mitigation ... 15

2.4.3 Types of Anomaly Detection Techniques ... 15

2.5 Effect of DDoS on Openflow Controller ... 17

2.5.1 Openflow Controller Performance ... 18

2.5.2 Openflow Controller for the Cloud ... 19

2.5.3 DDoS Mitigation in Openflow Networks ... 19

2.6 Entropy for DDoS Detection ... 24

2.7 Concluding Remarks ... 25

Chapter 3 ... 26

Early Detection of DDoS Using Entropy ... 26

3.1 Introduction ... 26

3.2 A Measure of Randomness ... 26

3.2.1 Why Entropy? ... 27

3.3 Short-term Statistics for Early Detection ... 28

3.4 Early Detection in Openflow Controller ... 31

3.5 Comparison of Different Detection Methods to SDN Entropy ... 36

3.6 Concluding Remarks 37
Chapter 4 39
Simulation and Results 39
4.1 Controller 39
4.2 Network Emulator 39
4.2 Packet Generation 40
4.3 Network Setup 40
4.4 Choosing a Threshold 42
4.5 Test Cases 43
4.5.1 Attack on One Host 45
4.5.2 Attack on a Subnet 48
4.6 Effects of the Added Functions on Resource Usage 51
4.7 Summary of the Results 53
Chapter 5 55
Conclusion and Future Work 55
5.1 Conclusion 55
5.2 Future work 56
Bibliography 58
Appendix 62

List of Tables

TABLE 2.1 PACKET HEADER MATCH FIELDS....9
TABLE 3.1 ENTROPY OF DIFFERENT WINDOW SIZES [33] 28
TABLE 3.2 WINDOW SIZE COMPARISON....29
TABLE 3.3 COMPARISON OF FIVE WINDOWS 30
TABLE 4.1 THRESHOLD VALUE CALCULATION....43
TABLE 4.2 ATTACK TRAFFIC PROFILE 44
TABLE 4.3 ENTROPIES OF TEST CASES 49
TABLE 4.4 NUMBER OF INCOMING PACKETS PER HOST IN EACH TEST CASE....51

List of Figures

FIGURE 1.1 SAMPLE ATTACK ON THE CONTROLLER 2
FIGURE 2.1 SDN STRUCTURE [1] 7
FIGURE 2.2 A SIMPLE NETWORK WITH AN OPENFLOW SWITCH 7
FIGURE 2.3 FLOW ENTRY PROCESS 10
FIGURE 2.4 DDOS ATTACK COMPONENTS 13
FIGURE 2.5 ATTACK ROUTE IN SOM DDOS DETECTION [25] 20
FIGURE 2.6 USING OPENFLOW FOR MONITORING NETWORK SECURITY [27] 22
FIGURE 2.7 TWO CONTROLLERS FOR RESILIENCY IN OPENFLOW [31] 22
FIGURE 2.8 EVENT BASED INTRUSION DETECTION DESIGN FOR SDN [32] 23
FIGURE 3.1 DDOS DETECTION FLOWCHART 34
FIGURE 3.2 LISTS ADDED TO THE CONTROLLER 35
FIGURE 3.3 FUNCTION TO COLLECT DESTINATION IP ADDRESS STATS 35
FIGURE 3.4 ENTROPY COMPUTATION FUNCTION 36
FIGURE 4.1 EXPERIMENT NETWORK WITH 9 SWITCHES AND 64 HOSTS 41
FIGURE 4.2 SUDDEN INCREASE OF TRAFFIC IN DDOS 45
FIGURE 4.3 25% RATE ATTACK ON ONE HOST 46
FIGURE 4.4 50% RATE ATTACK ON ONE HOST 47
FIGURE 4.5 75% RATE ATTACK ON ONE HOST 47
FIGURE 4.6 50% RATE ATTACK ON FOUR HOSTS 48
FIGURE 4.7 75% RATE ATTACK ON FOUR HOSTS 49
FIGURE 4.8 COMPARISON OF ENTROPY DROP, ONE HOST 50
FIGURE 4.9 COMPARISON OF ENTROPY DROP, FOUR HOSTS 50
FIGURE 4.10 CPU USAGE WITH NO DDOS DETECTION 52
FIGURE 4.11 CPU USAGE WITH DDOS DETECTION 53

List of Appendixes

APPENDIX A: STATISTICS COLLECTION AND ENTROPY COMPUTATION CODE 62
APPENDIX B: NORMAL TRAFFIC GENERATION CODE 64
APPENDIX C: ATTACK TRAFFIC GENERATION CODE 66
APPENDIX D: STARTING MININET..67

List of Acronyms

DDoS	Distributed Denial of Service
DNS	Domain Name System
DoS	Denial of Service
GAU	Gaussian Classifier
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MLP	Multilayer Perception
NIDS	Network Intrusion Detection System
OVS	Open Virtual Switch
SDN	Software Defined Networks
SOM	Self-Organizing Maps
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network

Chapter 1

Introduction

This chapter will first cover the problem statement by looking at the threat of Distributed Denial of Service Attack (DDoS) in SDN architecture. Then, our motivation and the research objectives will be discussed. Finally, the contributions of this research are outlined followed by the thesis organization.

1.1 Problem Statement and Motivation

The idea of Software Defined Network architecture is a new and novel way of network management. In SDN, switches do not process incoming packets. They look for a match of the incoming packet in their forwarding tables and if there is none, it will be sent to the controller for processing. The controller is the operating system of SDN. It processes the packets and decides whether the packet will be forwarded in the switch or will be dropped. By applying this procedure, SDN separates the forwarding and processing planes.

SDN architecture can be a network of several controllers each of which is connected to a network of switches. Each of these networks and its controller can be seen as slice of the network. We are focusing on each of these slices to protect it against DDoS. If the connection between the switches and the controller is lost, the network will lose its processing plane. That means packet processing is no longer done in the controller and by losing the controller, the SDN architecture is lost.

One of the possibilities that can cause the controller to be unreachable is a DDoS attack. In DDoS attacks, a large number of packets are sent to a host or a group of hosts in a network. If the source addresses of the incoming packets are spoofed, which thery usually are, the switch will not find a match and has to forward the packet to the controller. The collection of legitimate and the DDoS spoofed packets can bind the resources of the controller into continuous processing that exhausts them. This will

make the controller unreachable for the newly arrived legitimate packets and may bring the controller down causing the loss of the SDN architecture. Even if there is a backup controller, it has to face the same challenge.

The main goal of this research is detecting a DDoS attack in its early stages. The term early depends on the network itself. Since the controller software can be run on a laptop or a powerful desktop, the term early would depend on the tolerance of the device and traffic properties. However, if the detection happens in the first few hundred packets, the mitigation is applied before the controller is completely swamped with the large number of malicious packets. Figure 1.1 shows a simple DDoS attack on the controller where the normal incoming packet rate is around100 packets per second. When the attack happens, the rate rises sharply to, approximately, 250 packets per second. The simulated DDoS attack was directed to a SDN controller that is connected to a network with 64 hosts and nine switches. The attack lasted for 40 seconds and sent 500 packets with spoofed source addresses all destined for one host. For the purpose of this research, all packets will have spoofed IP addresses. This way, the switches do not have a match and all the packets are sent to the controller. The network operating system and its persistent performance is the center of attention in this work.

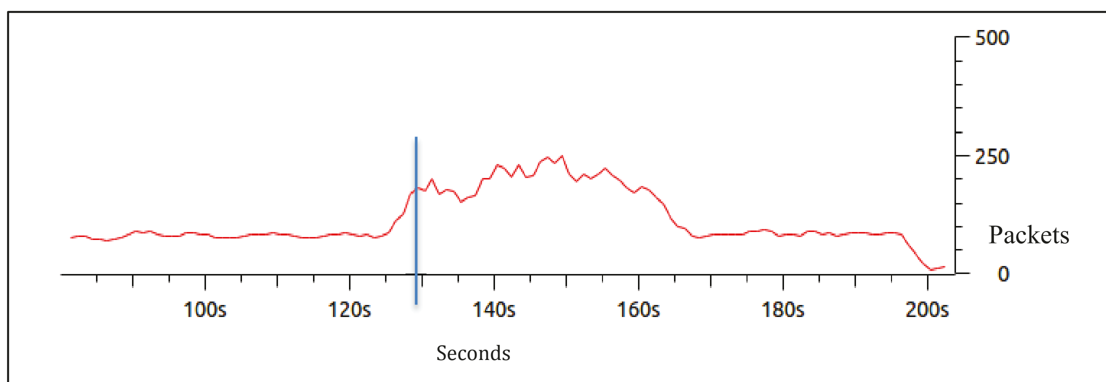

Figure 1.1 Sample attack on the controller

One of the main objectives of this research is detecting the attack at the start of its launch. An ideal place for detection is where the blue line is shown in the Figure 1.1.

If DDoS is detected at this point, any mitigation techniques that follow, have enough time to secure the controller before it is overwhelmed by the attack and becomes unreachable.

To accomplish this goal, a fast and effective method is needed that works within the controller. At the same time, it must be lightweight to avoid excessive processing power usage, specially, at the peak of an attack.

Collecting statistics is one of the functions of the controller. In this study, this attribute is used for adding another set of statistics collection to the controller; destination IP addresses. In this research, we used IP address but SDN allows for any fields of the packet header to be collected.

In our solution, randomness of the incoming packets is measured. A good measure of randomness is entropy. Entropy measures the probability of an event happening with respect to the total number of events. For instance, in a network of 64 hosts, all hosts should have a reasonably close probability of receiving new incoming packets. This will results in, reasonably, high entropy. New packet, in the sense that there is no flow for it in the switch table and it has to be sent to the controller to be validated for a new flow. If one or a number of hosts starts to receive excessive incoming packets, the randomness decreases and entropy drops. This research makes use of this property of entropy to detect an attack at its early stages. Based on the tests that are done in this research, we choose a threshold for entropy and lower values will be considered as attacks. Being programmable is one of the major advantages of SDN. Any time the network configuration changes, the threshold can be adjusted. And, it can be adjusted while the network is running live traffic so there is no restriction. Depending on the network, the entropy can be of the destination IP address, VLAN tag, destination port or any other applicable field. If it is lower than the set threshold, it will be considered an attack.

The solution is, partially, based on a paper on DDoS detection proposed by Oshima et al. [33] where a small window of 50 packets is used to calculate the entropy of incoming packets. The paper is not done for SDN networks but considering the crucial role that controller has in SDN, short-term statistics with smaller windows are

ideal for SDN. This is due to the fact that the controller only processes the new incoming packets not the entire traffic. The method and calculation of the entropy will be further examined in Chapter 3 of the thesis.

1.2 Research Objectives

In this research, we studied SDN to find possible weak points with respect to DDoS attacks. This study led to the controller. We found that the controller is the weak link in a DDoS attack scenario. With protecting the controller in mind, we studied different methods in DDoS detection that could be used in the controller. However, the structure of SDN posed its limitation on the type of the solution and the way it was implemented. These limitations were:

i) Limited resources of the controller.

ii) The need to detect the attack before the controller is out of reach due to the large number of malicious packets.

The main objectives are:

a) To Find the weak link in SDN when DDoS happens.

b) Find a solution to detect DDoS in SDN before it overwhelms the controller.

1.3 Research Contributions

In this research, we were able to find the weak point of the SDN when a DDoS attack happens and, propose a solution that is, specifically, tailored for SDN.

The main contribution of this research is to show how DDoS can bind controller resource into processing malicious packets and add a DDoS detection mechanism to SDN controller. The contributions are as follows:

a) Show how DDoS attack can overwhelm the controller in SDN architecture.

b) Propose a lightweight and simple DDoS detection mechanism based on entropy, in order to protect the controller.

c) Implement the proposed mechanism using Mininet [39].

d) Show the effectiveness of the solution through extensive simulations.

1.4 Thesis Organization

The thesis is structured in the following order:
Chaper 2 covers an overview of SDN and its components followed by DDoS attacks and their detection and mitigation techniques. Later, the effects of DDoS attacks and their effects on the controller will be discussed. At the last parte of the chapter, literature related to DDoS detection and mitigation in SDN will be covered.

Chapter 3 will examine the proposed solution for DDoS detection in the controller and compare it to other methods.

Chapter 4 presents the procedure of setting the experiment and experimental results of the solution.

Chapter 5 is the conclusion of the thesis. It presents an overview of the proposed method and the results obtained from simulation. Limitations of the solution and future work are also outlined.

Chapter 2

Background and Related Work

This chapter will first cover the background of Software Defined Networks, the Openflow protocol and its specification. In addition, it will cover DDoS and its detection techniques in non-SDN networks. Lastly, the SDN security related literature will be reviewed.

2.1 Software Defined Networks

Software Defined Network is a different way of looking at networks. The main purpose is greater control over network assets. In current production networks, both control and forwarding actions are configured in the hardware by vendors, and they are, mostly, proprietary software. The SDN architecture separates control plane and forwarding plane and allows network admins to take over the control plane [1]. This separation is done by restructuring the network so the switch will receive instructions for forwarding instead of using its resources for processing the incoming packets. The switch will contain tables with flows that instruct forwarding. Openflow is the protocol that orchestrates the SDN architecture. This architecture consists of Openflow enabled switches, a controller and a secure channel between the controller and switches. Figure 2.1 shows different layers of SDN structure. The application layer will have a single view of the network through the control layer and the whole system looks like one logical switch. The control layer is where the controller abstracts the network infrastructure from the application layer. By using the control layer, any configurations and modifications can be done in real-time. In the infrastructure layer, there is no need for each device to learn different protocols and the only task left is forwarding. The Open Networking Foundation is the main organization that promotes the adoption of SDN, works with several vendors, and has different groups that are working on Openflow specification [1].

2.2 Openflow Protocol

The Openflow protocol can be considered as the workhorse of SDN. It manages the switches in the network and allows an external entity like the controller to manipulate the flow of packets through the network. Openflow was designed as a tool focused on network research [2].

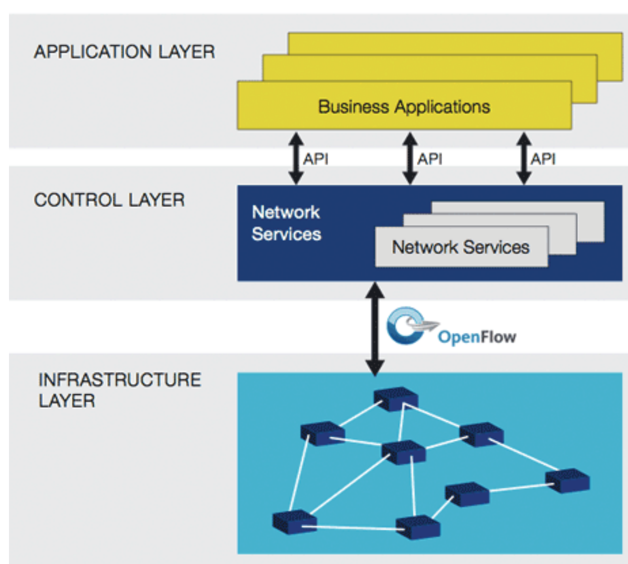

Figure 2.1 SDN structure [1]

However, in recent years, several vendors started to offer their Openflow-enabled switches [3]. All switches have tables showing the ingress and egress paths of a packet for that switch. Openflow makes use of this property and makes these tables accessible by the controller. An Openflow switch will receive its flow table entries and deletion from the controller through a secure channel. A simple network is shown in Figure 2.2.

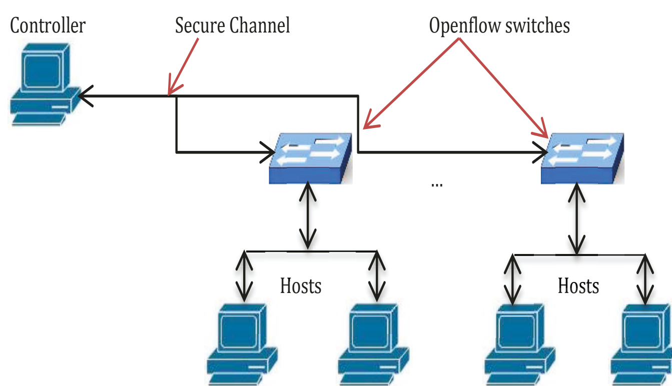

Figure 2.2 A simple network with an Openflow switch

When a new packet arrives to an Openflow switch, it will look into the flow table to find a match. If there is no match in the table, the packet will be sent to the controller. The controller processes the packet and marks the packet with an action like:

- Add a new flow for similar incoming packets
- Drop similar packets
- Tag with a queue ID

2.3 Openflow Specifications

Lookup, matching, forwarding and requesting action from the controller are all done based on the Openflow specification that is published by the Open Networking Foundation. In this work, version 1.0 [1] has been used. Version 1.0 was the first set suitable for production networks. From time to time, version 1.3 specifications will be referenced for the new features. This section will cover the main areas of the specifications related to the topic of this research without dwelling in too many details.

2.3.1 Openflow Switch

An Openflow switch consists of a flow table or a group of them and a secure channel to the controller. Each table has a match field, counters, and a set of instructions for every entry. The matching process in the switch covers different fields of the packets' header. Table 2.1 shows the fields that a switch can use to find a match in its tables. In the table, there is a metadata field that is defined (second row in Table 2.1) as a maskable register to carry information from one table to the other when there is more than one table. It is a mean to carry header information from one table to the other. Often switches have multiple tables that are pipelined. The packet will move from one table to the other for a match and carry the metadata. If a match is found, the metadata tag will be updated accordingly.

Any packets entering the switch will be checked against all existing flows in the tables. If a match is found, the action assigned to that entry is applied and the counter

for the entry will be updated. Counters cover a number of components in the switch like counters per flow entry, per table, per port, per queue and other areas.
For instance, duration refers to the amount of time a flow spends in the table. The counters are all wrap around with no overflow. The controller can and will poll some of these counters for different reasons. It is worth mentioning that not all counters will be used because controllers are developed or configured to match the needs of the vendor. In version 1.3 counters can be disabled.

If a match is not found, the packet will be sent to the controller. In version 1.3 of the specifications, if there is no field that can be matched, the packet will be dropped. Since its header does not have any of the field mentioned in Table 2.1, it will be considered an invalid or illegal packet. Our solution works with IP address which exists in the table.

Table 2.1 Packet header match fields

Header Field
Ingress Port
Metadata
Ether src
Ether dst
Ether type
VLAN id
VLAN priority
MPLS label
MPLS traffic class
IPv4 src
IPv4 dst
IPv4 proto / ARP opcode
IPv4 ToS bits
TCP / UDP/ SCTP src port ICMP Type
TCP / UDP / SCTP dst ICMP code

New packets can be sent as a whole to the controller or the switch can buffer the payload and send only the header. The latter is the default mode.

When a packet is sent to the controller, it will be encapsulated and marked as OFPT_PACKET_IN message. We will refer to it as Packet_In. Considering the number of switches, time of day, length of packet, priority and other factors, the controller has to process these packets and send a response with an action to deal with that packet and the packets coming after from the same source. This is the point where the processing will be completely handled by the controller and the switch will only cover the forwarding.

There is a set of actions that the controller will send to the switch; forward, drop, push in a queue, quality of service and modifying a field, i.e., modifying VLAN tag, MAC address or IP address. The main actions, also called required actions, are forward and drop while queuing and modify fields are optional. The action is set for the packet in the controller and then sent back to the switch through Packet_Out message.

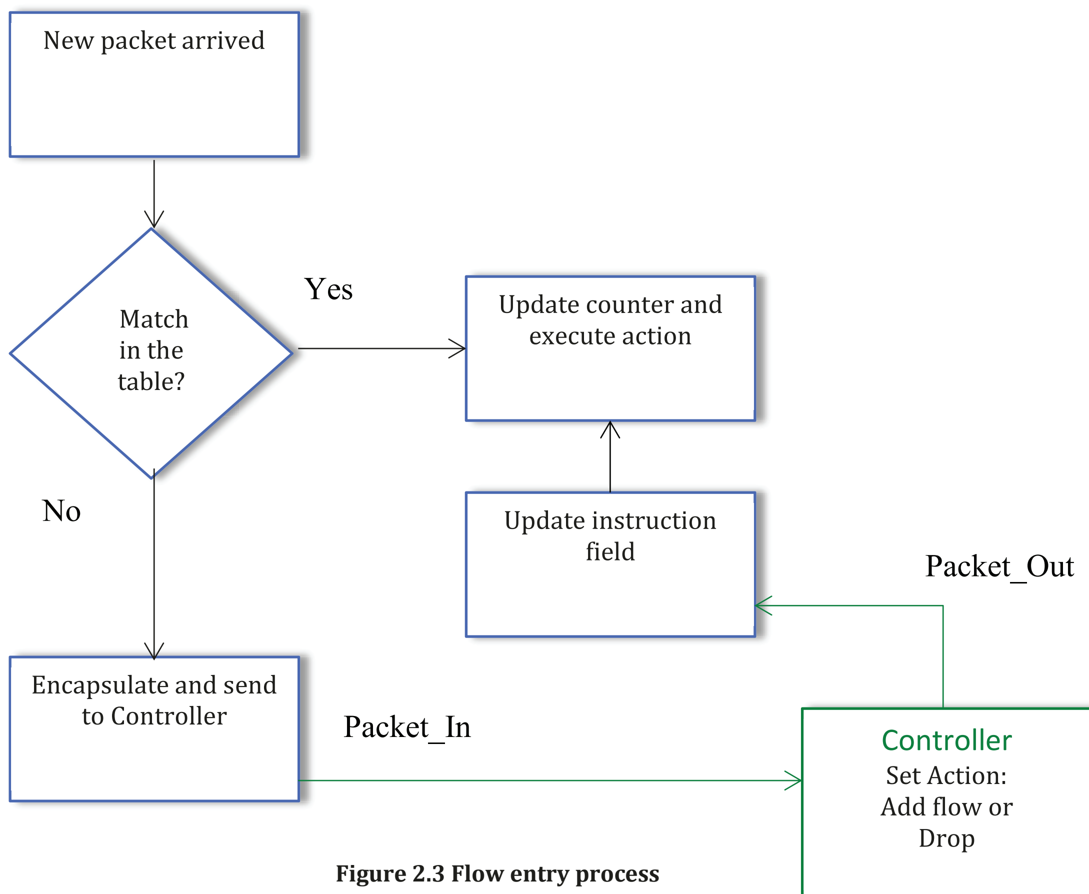

Figure 2.3 Flow entry process

Figure 2.3 shows the process of flow entry. If a packet is marked with drop action, a flow entry will be added. Any packet matching that flow will be dropped. If a flow does not receive packets, the flow entry will time out and it will be removed from the table after some time.

2.3.2 Secure Channel

A brief look at the specification of Openflow shows the single point where the forwarding plane and control plane are connected. It is the secure channel between the controller and the switch. If the connection to the controller is lost, a pure Openflow switch will not be able to deal with unknown incoming packets.

The secure channel is the lifeline of all Openflow switches in SDN. It is a TLS or TCP connection established between the controller and the switch. If the connection is lost, the switch will try to connect to a backup controller if there is one. This is the "fail secure mode" and all packets to the controller will be dropped.

In an Openflow switch, all new packets will be processed in the switch and will not be sent to the controller. If the switch is capable of working both in SDN and in none SDN then, it is called Hybrid switch. In this case, the switch will not follow the Openflow protocol and the network loses its SDN architecture.

The Openflow specification shows that without a controller in the network, we are dealing with a non-SDN network with no central control or separation of forwarding and control plane. This section shows the importance of detecting any threat that can make the controller unreachable.

2.4 Distributed Denial of Service Attack and its Mitigation

The Distributed Denial of Service (DDoS) attack is a well-known malicious attempt to exhaust the resources of a computer or a network of computers by sending heavy traffic to them [4]. The two main goals of the attacker are:

i) Bandwidth depletion.

ii) Resource exhaustion [5].

DDoS attack starts from an attacker planting a code in compromised PCs which are referred to as Botnet. At the time of the attack, these codes are run and a stream of traffic is directed towards the victim. A more sophisticated attack uses a thin layer of compromised PCs called handler to control a larger number of PCs called zombie hosts. The zombie hosts are responsible for generating the attack traffic [5]. Using botnets makes the attack more concentrated and keeps the perpetrator hidden behind the scene.

DDoS is one of the most common methods of overloading and disrupting service in a network. Each day, hackers launch more than 7000 such attacks and, statistics show that in the first quarter of 2013 average attack bandwidth reached 48.25 Gbps which is 718% higher than the last quarter of 2012 [6]. From May 2013 to Sep 2013, United States and China have suffered significant daily attacks. Google’s Digital Attack Map, captures these attacks in its website [7]. Not all attacks can be detected or documented but looking at the numbers, it is an imminent danger for every network. Figure 2.4 shows a simple example of DDoS Attack route.

2.4.1 Types of DDoS Attacks

Launching any attack requires an access to machines in the subnet of the victim to be used as zombies. Hackers use scanning to find vulnerable computers in the network. Scanning can be random, based on a hit list, local subnet scanning or based on an algorithm designed by the hacker [8].

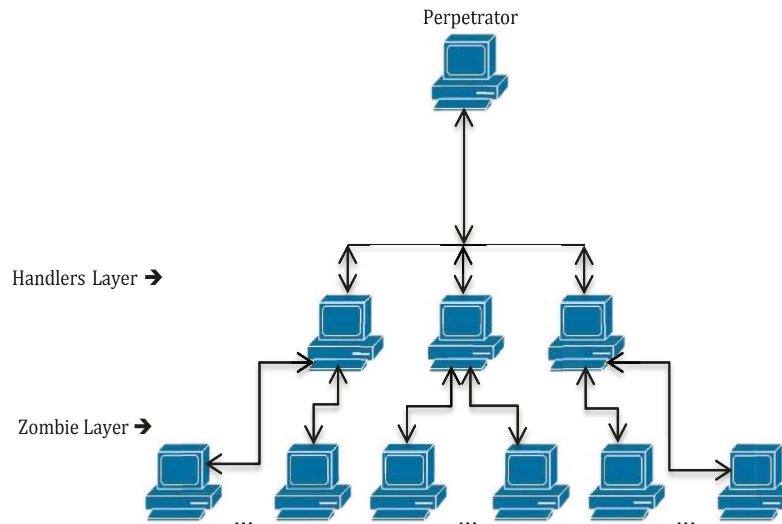

Figure 2.4 DDoS attack components

The attacks can be categorized as application, host, resource, network and infrastructure attacks [8].

1. Application: targeting an application on a host to deny legitimate use.
2. Host: making a host unreachable.
3. Resource: overwhelming a server to keep it bound to the continuous stream of fake requests.
4. Network: sending a large volume of traffic to a network to exhaust the bandwidth.
5. Infrastructure: simultaneously targeting a domain name server in different places.

A few types of DDoS have been very frequent in recent years. The pattern of attacks shows that hacker have been using certain types of methods for launching attacks [6] [7]. Some of these methods will be looked at next.

Goolge and Arbor networks have dedicated a website that follows the attacks and collects statistics from all over the world [7]. The statistics show the type of attacks, duration of attacks, bandwidth used, mostly used source and destination ports, origin of attack and the victim's country. The following paragraphs describe different types of attack.

1. UDP Flood is a type of attack that sends a large number of packets to random ports in the victim's machine causing the machine to look for applications on these ports. The machine has to send Destination Unreachable packets in response for each incoming packet. As the number of incoming packets increases, the delay increases and, eventually, the machine will be inaccessible.

2. SYN Flood is an attack using TCP connection initiation for targeting the victim's machine. Several SYN packets are sent to the victim but no ACK is returned to the victim causing the resources to be used up waiting for acknowledgement from the attacker. TCP Flood was the highest detected attack up to mid 2013 with 38.7% of perpetrated attacks [9].

3. DNS Reflection attack is sending spoofed IPs and asking for a response that is much larger than the request and directing it to the victim. The attacker changes the source IP address to the victim's IP and causes a heavy traffic to be directed to the victim. Spoofing packets is a common practice for DDoS attackers.

4. HTTP Flood consists in sending a huge number of requests to a server and overwhelming it to the point where it cannot respond to legitimate requests. This was the second highest type of attack with 37.2% [9].

5. ICMP Flood is another type of attack that exhausts the resources of the victim by sending a very large number of ICMP pings (echo request), which keeps the server bound in sending responses (echo replies).

The common factor in all of these attacks is pushing heavy traffic into the victim's network and exhausting its resources. The detection and mitigation of these heavy flows in current conventional networks will contribute to a better understanding of their effect on the SDN architecture.

2.4.2 Anomaly Detection for DDoS Mitigation

The common factor in different types of DDoS attack is the abnormal traffic sent to the victim. In normal circumstances, there is a pattern in the network activity and an accepted rate of bandwidth consumption. If there is a sudden increase in traffic, delay, CPU utilization, or sudden drop in performance of any of the network assets, this, often, will be considered abnormal. Any DDoS detection will be looking for such abnormalities in the network. In general, anomalies are related to the nature of data in the network [10]. The attack can be directed to the network layer to cause a bottleneck or an application layer type causing CPU resources exhaustion. Understanding the type of data and its characteristics in the network is the first step to detecting anomalies. These characteristics can be packet header information, delay, packet size, protocol, etc. For instance, in a server that responds to TCP SYN requests, attacks are, most likely, TCP SYN request flooding. And, this is where anomaly detection is likely to occur. In fact, the characteristics of the network dictate the type of intrusion. If a network is susceptible to a certain type of threat, then detection and mitigation of the threat has to be matched to it.

2.4.3 Types of Anomaly Detection Techniques

In IP networks, there is a certain bandwidth and certain processing power for carrying traffic. When some attributes of the network are subjected to statistical analysis, for each attribute a pattern will appear. The longer the time, the more reliable is the pattern. However, this is only true if the network has a steady traffic all the time. If there are variations that are accepted as normal traffic, in the long run, the statistics will stabilize and cannot be considered completely reliable.

Data collection, filtering and processing for anomaly detection are approached by a variety of techniques. Statistical analyses and machine learning are two of the common methods of anomaly detection.

1. Statistical analyses, like Entropy and Chi-Square techniques, have been suggested for detecting change in network traffic [11].

   Entropy represents packet headers as independent information symbols with unique probability of occurrence. It is a common method for DDoS detection [12] [13] [14]. By selecting a window of some number, 10,000 for instance, and moving the window forward, a pattern will emerge with probabilities for each type of packet header. Drastic changes in the bins of each header that deviates from the average bin limits will alert the system of anomalies. This method will be explored further in the next chapter to examine its potential use in SDN.

   If a certain type of intrusion is expected and the type of packet header is known, then Chi-Square is a better model. For instance, if TCP SYN flood is the expected type of attack, then sampling bin of data and measuring the number of TCP SYN headers will show a pattern of the average number of such headers. Any deviation beyond the recognized limits is assumed abnormal.

   Equation 2.1 shows Chi-Square equation. N_i is the number of packets for one sample and n_i is the expected number of packets in normal circumstance. The value of targeted packets per bin is updated as times passes and more samples are taken.

$$X^2 = \sum_{i=1}^{B} \frac{(N_i - n_i)^2}{n_i} \qquad (2.1)$$

2. Machine learning and cognitive detection is another method used for defending networks against intrusion. Instead of setting up a fixed filter an algorithm is trained to constantly update its filtering criteria based on the events of the network.

An example of such system is neural networks [15]. Neural networks consist of several nodes working in parallel to process data. They work like human brain. When they are trained or given a large amount of information, the collective knowledge of neurons or nodes develop a pattern for the processing of similar data. Three main layers of neural networks are input, output and hidden layers in the middle to process the input data. As time passes and more data is processed, the nodes are learning more and a clearer pattern emerges.

An algorithm is the driving force behind the decision making of these networks. Some common algorithms in network intrusion and anomaly detection are Multilayer Perceptron (MLP), Gaussian Classifier (GAU), K-means Clustering (K-M) and Markov model [16] [17] [18].

In the first chapter, it was mentioned that entropy is the method that is used in this research for implementing a detection method in Openflow controller. Before looking into the entropy literature, the effect of DDoS on SDN and its detection techniques will be reviewed. Later on, entropy detection literature will be discussed.

In the previous sections, the nature of attack, its types and mitigation techniques were briefly covered. A more thorough investigation is beyond the scope of this research as it is more focused on the affect of DDoS on SDN. As a result, the next section will focus on the DDoS in SDN.

Being an ever present danger, DDoS attacks are a real threat to any network, especially, SDN. Being a new structure, and in the process of maturing, there is an opportunity for discovering weak points of the SDN for a better defense against such attacks.

2.5 Effect of DDoS on Openflow Controller

If Openflow is the protocol that is, currently, used for SDN structure, then, the controller is the brain or the operating system of SDN. It can modify network assets and dictate new rules to switches in real-time. However, SDN lifeline to the controller

is the secure channel. Once the secure channel is disconnected, the SDN structure loses its operating system and, in the best case, it has hybrid switches that can fall back to the normal operation mode (i.e. switches doing the processing and forwarding). This single point of failure is the where a DDoS attack can really hurt SDN. The other effect of DDoS attack is the filling of switch flow tables. For every incoming packet, the controller will add a new flow to the table. With a high volume of traffic coming, soon the flow tables will be filled with fake flows.

One serious scenario of DDoS that can directly affect the controller is swamping the controller with Packet_In events. Any new packets that do not have a match in the flow table will be sent to the controller for processing. Most DDoS attacks use spoofed source address, which translates into new incoming packet at the switch. This part is considered one of the advantages of SDN where the control plane is separated and manageable at the controller. It is also the main disadvantage when the number of new incoming packets is greater than the secure channel's bandwidth and the controller's processing power.

2.5.1 Openflow Controller Performance

Jarschel et al. [19] have shown that performance of the Openflow architecture is directly related to the processing power of the controller. As the number of new flows increases, the wait time and overall delay in the system rises. One reason for that is the buffering that happens in both the switch and the controller. In the event of any threats, these buffers are a death trap for the whole system. Packet_In events in large numbers fill up the queues. If the incoming events are spoofed packets, they will paralyze the network by limiting the access to legitimate flows.

Cai et al. [20] propose a solution for better performance by parallelism. This method makes use of the multicore feature of new CPUs. Recent CPUs have two, four or more cores. They are called dual or quad core processors. This method utilizes each core of the CPU for processing new packets. The performance improvement is achieved by using a threading program that they called Maestro. When a packet

arrives for processing, Maestro will pull the state of each core and forwards the packet to the idle core. This will reduce the wait time in the buffer for incoming packets.

2.5.2 Openflow Controller for the Cloud

Openflow has been a research topic for cloud networks and datacenters. Some areas of interest are network virtualization based on Openflow [21], resource control [22] and improving data center scalability using Openflow [23].

Cloud networks often deal with geographically distributed hosts. Applying SDN architecture to this type of networking will require a number of controllers to deal with scalability issue.

Bifulco et al. [24] show an example of multi-controller network where a mobile node travels from one network to another and its local network IP address is translated to a fixed IP in the controller. The local IP is called locator and fixed the IP is called identifier. When a connection to the node is requested, its fixed IP address is sent to the switch connecting to the local network. The Fixed IP address is an identifier of the mobile node and the switch can do the translation through a flow in its table. Controllers share the information and fill up switch tables of the network where the mobile node is connected. In this architecture, the mobile node is always known by its fixed IP address.

2.5.3 DDoS Mitigation in Openflow Networks

Production networks have been studied for a long time and the types of threats to them are mostly known. SDN however, is a new architecture and there are not many papers discussing the topic. Existing papers look at DDoS and apply the existing solutions to SDN. This means treating SDN as a normal production network where the role of the controller is ignored. In SDN, the switches have no control over incoming packets and they do not spend any time processing them. This also means, in an attack, the entity affected first and most is the controller.

Braga et al. [25], use Self-organizing Maps (SOM) [26] machine learning technique for DDoS detection. In this method, SOM is trained by collecting the flow statistics from Openflow switches. The parameters that are checked for training SOM are average packet per flow, average bytes per flow, average of duration per flow, percentage of pair flows (i.e. two flows one in switch *A* and the other in switch *B*, flow *A* destination is input port of Flow *B*), growth of single flow, and growth of single ports. The SOM will improve the statistics of its vectors as time passes and more statistics are gathered.

A look at Figure 2.5 shows the path of a DDoS attack to a host target and nothing goes through the controller. The red line shows the path of attack and NOX [27] is the controller. In a DDoS attack, the packets are always spoofed or coming from several zombie nodes. Hence, the packets have to go through the controller to get access to the target through a flow in the switch table. This case was not covered in this paper.

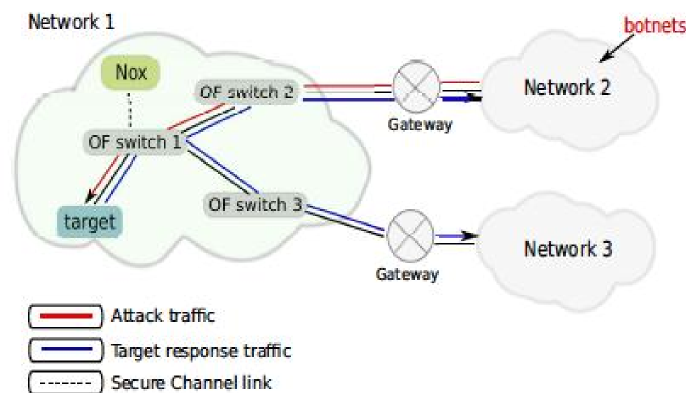

Figure 2.5 Attack route in SOM DDoS detection [25]

Other related work in security has been the use of SDN architecture for intrusion detection in cloud networks. Shin et al. [28] use Openflow as a flow regulation tool to monitor traffic in the cloud. The main idea is reconfiguring the flow of packets by using the Openflow protocol so it takes a path where a Network Intrusion Detection System (NIDS) is installed. By deploying the SDN architecture, there is no need to relocate and reinstall NIDS devices. Openflow will process the packets and add flow rules to routers and switches to go through a monitored path. Few algorithms are offered by the paper to find the shortest path for a secure route and each one is categorized by the time it takes for the controller to find it.

In this paper, the SDN controller computes the shortest path to a monitored link in the network and it is not involved in the detection process. A look at Figure 2.6 shows the structure of proposed detection system. In the figure, device and policy manager module contains a list of devices and the security policy of each device. The routing rule generator module first discovers the topology of the network and then collects status and cost information of the network to send it to the controller, NOX in this figure. The flow rule enforcer module is the controller that adds flows in the routers and switches. Network operating system is Openflow and its controller combined.

Xing et al. [29] propose adding Openflow to SNORT [30], a common intrusion detection tool for cloud, to reconfigure the network when an attack is detected by SNORT.

Although above examples of cloud intrusion detection were not applied to detect DDoS on the SDN architecture itself, they show the potential of SDN for intrusion detection and mitigation in a complex network like a cloud. The main purpose of this research is to find a way to protect the structure of SDN, in particular, the controller.

Fonseca et al. [31] discus the possibility of losing the controller and identify the need for a backup one. The paper proposes a second controller that runs in parallel to the current running controller. If the switches lose connection to the controller, they will look for the second controller, which is added to the configuration of the switch. One of the mentioned scenarios is losing the controller in a DDoS attack. The solution for such situation is to use a machine learning solution like SOM in [25] for attack mitigation while recovery is in process. Figure 2.7 shows running two controllers in parallel. The first controller will continuously send status updates to the backup controller announcing that it is alive. If the first controller goes to an unknown state or becomes unreachable, the second controller will take control and starts running the network normally.

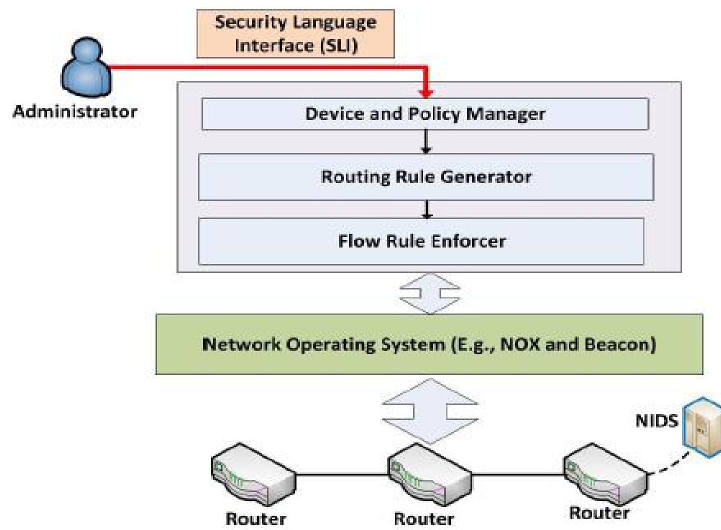

Figure 2.6 Using Openflow for Monitoring network security [27]

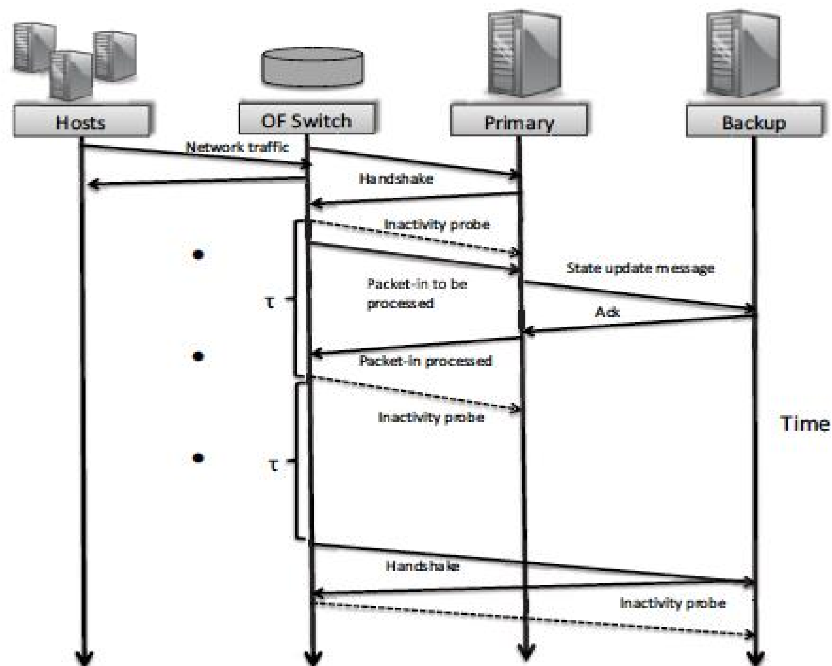

Figure 2.7 Two controllers for resiliency in Openflow [31]

Hu et al. [32] propose an intrusion detection system that works on top of Openflow and SDN. Figure 2.8 shows the structure of such system. The event processing software is a hyper controller that receives events for sub-controllers and processes them for possible attacks. The event bus in the figure is the link between sub-controllers and the event processor. The method is in the research phase and there are no experimental results.

In [28], SDN was used to discover shortest path and there was no mention of the effect of that attack on the controller itself. In [31], if an attack happens, and the primary controller is unreachable, the attack continues and the backup controller has to face the same problem. The event-based system employs a hyper controller to do the packet processing for the sub-controller. It seems that the events are processed twice in the system. Once when they arrive at the sub-controller and then when they are sent to the event processing engine. This overhead of processing can be removed by applying detection technique at the sub-controller level and mitigation at the hyper controller level.

To the best of our knowledge, there are not many papers discussing the security of SDN and Openflow in the event of an attack. This might be due to the fact that the structure is new and not widely used. The papers that are presented here are the most directly related papers. In fact, to the best of our knowledge, there is no paper addressing the issue of losing the controller in SDN and its solution.

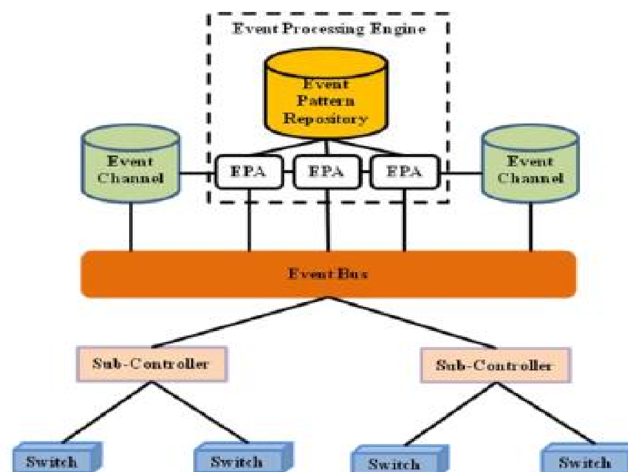

Figure 2.8 Event based intrusion detection design for SDN [32]

2.6 Entropy for DDoS Detection

Entropy is the method used in this research to detect DDoS attacks in SDN. A look at the used methods in non-SDN networks is necessary before introducing it in SDN. Since there is no research in using this method in SDN, we have to rely on what is done in non-SDN research.

There are two essential components to DDoS detection using entropy; window size and a threshold. Window size is either based on a time period or number of packets. Entropy is calculated within this window to measure uncertainty in the coming packets. To detect an attack, a threshold is needed. If the calculated entropy passes a threshold or is below it, depending on the scheme, an attack is detected.

Qin et al. [12] propose a method with a window of 0.1 seconds and three levels of threshold. This method is concerned with avoiding false positive and false negatives in the network. However, as the authors themselves mention, the method is time consuming and uses more resources.

Ra et al. [13] propose a faster way of computing entropy by basing the calculation on both packet type and the volume of packets in the network. This method also uses a time period window. For the threshold, the authors ran several datasets to find a suitable threshold and it is a multiple of standard deviation of entropy values. In this method, the false negatives are higher than other methods and false positives are lower. No percentage of accuracy is indicated. There is also no mention of resources used for fast computation.

Entropy has been used in different ways to detect DDoS attacks in the network but, to the best of our knowledge, it has not been used in SDN.

In SDN, when passing packets to the controller, the limitation of available resources and the quick detection of attacks are key features of any detection scheme. In this research, we will apply entropy for DDoS detection with the above limitation of the controller in mind.

2.7 Concluding Remarks

In this chapter, it was shown that the operating system of Openflow networks is the controller and losing it means losing the advantages of broad control with SDN.

The high frequency of DDoS occurrence will pose a challenge to SDN and it can be focused on just the controller bringing down the operating system. By examining DDoS attack and its mitigation, two main elements of detection stand out:

1. Packet header processing
2. Collecting statistics

At the same time, when looking at the tasks that the controller performs, we see two similar but very important elements:

1. Packet header processing and assigning actions
2. Collecting statistics from the switches

In [28] and [29], Openflow abilities were used for detecting DDoS attack on cloud networks. It seems that Openflow has all the elements that are needed for a successful detection. The next chapter will focus on utilizing these elements for an early detection of DDoS attacks within a SDN environment.

Chapter 3

Early Detection of DDoS Using Entropy

3.1 Introduction

In this chapter, the proposed method for early detection will be explained. Due to the limited resources of the controller, an early detection should be within the first few hundred packets of the attack. First, entropy, its formulas and computation will be discussed. Then, early detection will be examined, followed by our proposed detection method. Finally, the solution will be compared to other DDoS detection schemes.

3.2 A Measure of Randomness

In Chapter 2, we showed the use of entropy for DDoS detection and looked at its main components. Here, we will look at the formulation and computation. The main reason for choosing entropy is its ability to measure randomness in a network. The higher the randomness the higher is the entropy and vice versa.

Let *W* be a set of data with *n* elements and *x* is an event in the set. Then, the probability of *x* happening in *W* is shown in Equation 3.2. To measure the entropy, referred to as *H*, we calculate the probability of all elements in the set and sum that as shown in Equation 3.3.

$$W = \{x_1, x_2, x_3, \ldots, x_n\} \qquad (3.1)$$

$$p_i = \frac{x_i}{n} \qquad (3.2)$$

$$H = -\sum_{i=1}^{n} p_i \log p_i \qquad (3.3)$$

The entropy will be at its maximum if all elements have equal probabilities. If an element appears more than others, the entropy will be lower. The size of W is called the window size. If there is a continuous stream of incoming data, in our case the data is packet header, it will be divided into equal sets that are called windows. In the window, each element and its occurrence are counted.

For instance, if the window has 64 elements and, all elements appear only once, the entropy will be 1.80. If one element appears 10 times, the entropy will be 1.64. This property of entropy will be used for calculating the randomness in the SDN controller.

3.2.1 Why Entropy?

When packets arrive at the controller, the source address is always new. This is the reason they come to the controller. There has not been an instance of them in the table of the switch so they are passed on to the controller. It was shown in Chapter 2 that for every new incoming connection, the controller will install a flow in the switch so that the rest of the incoming packets will be directed to the destination without further processing. Hence, any time a packet is seen in the controller, it is new.

The other known fact about the new packets coming to the controller is that the destination host is in the network of the controller. The network consists of the switches and hosts that are connected to it. Knowing the packet is new and the destination is in the network, the level of randomness can be quantified by calculating the entropy based on a window size. The window size is the number of incoming new packets that are used for calculating entropy. In this case, maximum entropy occurs when each packet is destined to exactly one host. Minimum entropy occurs when all the packets in a window are destined for a single host.

Being able to quantify randomness and have minimum and maximum based on entropy makes it a suitable method for DDoS detection is SDN. Using entropy, it is possible to see its value drop when a large number of packets are attacking one host or a subnet of hosts.

In Chapter 2, we saw that machine learning methods needed extensive training to be able to detect anomalies. Also intrusion detection devices for DDoS detection had to be installed on different links in the network. Entropy does not have these limitations.

3.3 Short-term Statistics for Early Detection

Before discussing our solution, we will look at an entropy-based DDoS detection method that is used in a non-SDN network.

Oshima et al. [33] propose a short-term statistics detection method based on entropy computation. "Short-term" here refers to calculating entropy in small size windows. The study proposes a window size of 50 packets for gathering statistics. In this method, different window sizes where tested for optimal entropy measurement. Table 3.1 shows the results of the tests for different window sizes.

Table 3.1 Entropy of different window sizes [33]

W	H_N	H_A	$\lvert\bar{H}_N - \bar{H}_A\rvert$	S_N	S_A	z
5	1.36	1.98	0.62	0.79	0.48	1.29
10	1.89	2.72	0.83	0.98	0.56	1.49
50	3.11	4.22	1.11	1.35	0.65	1.70
100	3.59	4.73	1.15	1.39	0.64	1.80
500	4.54	5.51	0.96	1.05	0.40	2.40
1000	4.88	5.67	0.79	0.78	0.32	2.48
5000	5.50	5.92	0.42	0.31	0.13	3.25

In Table 3.1, W is the window size, H_N is the entropy in normal condition, H_A is entropy during an attack, S_N and S_A are the standard deviation of entropy for normal and attack traffic conditions respectively. z is the test of significance. More precisely, it is a test of validity for the hypothesis between two averages of different populations. When it is higher than 1.64, the hypothesis is valid. In Table 3.1, it can be seen that for a window size of 50, z is 1.7. The value can be computed using Equation 3.4.

$$z = \frac{\mid \bar{H}_N - \bar{H}_A \mid}{\sqrt{\sigma_n^2 / n} + \sqrt{\sigma_r^2 / r}} \quad (3.4)$$

σn and σr are the same as S_N and S_A. n is the population of normal traffic packets (value of n is not given) and r is set to 25. To test the hypothesis, a one-sided test of

significance with 5% confidence interval was used. The formula for the one-sided test is shown in Equation 3.5 where $\bar{x}$ is the mean of the population, μ_0 is the sample mean, σ is the standard deviation and n is the sample count.

$$\frac{\bar{x} - \mu_0}{\sigma / \sqrt{n}} \tag{3.5}$$

Instead of this test, we chose an experimental Threshold. In the previous section, we showed that the maximum and minimum values of entropy can be deterministic in the controller. This will give us the freedom to choose a suitable threshold by simulating the attacks on the controller. We run several attacks to choose a threshold within the difference between H_N and H_A. If the entropy value is below the threshold, an attack has happened and no attack otherwise.

We have chosen the window size to be 50 for this research. The main reason for choosing 50 is the limited number of incoming new connection to each host in the network. In SDN, once a connection is established, the packets will not pass through the controller unless there is a new request. The other reason is the fact that a limited number of switches and hosts can be connected to each controller. The third reason for choosing this size is the computation that is done for each window. A list of 50 values can be computed much faster than 500 and, an attack in a 50-packet window is detected earlier. We also tested the entropy with three other window sizes and measured the CPU and memory usage. Table 3.2 shows that there is no difference in memory usage but CPU usage increases with window size.

Table 3.2 Window size comparison

Window size	CPU1/CPU2	Memory usage
20	60% / 67%	1.2 GB
50	62% / 67%	1.2 GB
100	64% / 68%	1.2 GB
500	65% / 68%	1.2 GB

Table 3.3 shows the difference in entropy and the number of attack packets from each window size. H_N is the normal traffic entropy, H_A is the attack traffic entropy and

H_N - H_A is the difference. Last column shows the number of malicious packets when the attack traffic is 25% of all incoming packets. This is the lowest attack traffic rate that our method can detect with accuracy.
In a window of size 20, the difference of entropies is less than 10% making it difficult to choose a threshold. With only five packets, probabilities of false positives will increases. On the other side, window of 500 does not offer a better difference of entropies and takes a much longer time than a window of 50 to compute entropy. The difference is 0.19 for 500 packet window size which is 11% drop in normal traffic entropy. The difference in the window size of 50 is 0.12 which 10% drop in normal traffic entropy. Difference of 1% does not justify choosing a 10 times bigger window size.

Window sized of 50 and 100 look close. Because the number of hosts in our test network is less than 100, we chose 50. It is very easy to change window size in the controller and this flexibility is the advantage of SDN.

Table 3.3 Comparison of five windows

Window size	H_N	H_A	H_N - H_A	H_A Packets 25%
20	1.22	1.1	0.12	5
50	1.5	1.3	0.2	12
100	1.6	1.4	0.2	25
500	1.66	1.47	0.19	125

Looking at Table 3.1, the entropy of attack is higher than that of normal condition. In [33], the entropy is based on the destination port and source IP address. It is higher because the attack packets have different IP source addresses and they are, most likely, spoofed. In normal conditions, a connection between source and destination is established and the packets in these types of flows have the same IP source address. Since these packets have a higher number in the network, the entropy of normal condition is lower and an attack with several new source IP addresses increases entropy. However, in SDN, the packets are trying to establish a connection to the target by getting a flow rule in the controller. Thus, having different source IP addresses is a known fact that does not help in the detection of DDoS.

In SDN, the number of hosts and switches connected to the controller are known. Knowing the window size, the maximum entropy of the destination IP address is also known. It is happening when each packet is destined to exactly one host. What is left to be calculated is the number of packets that are targeting a specific host or a subnet. Hence, in our method, destination IP address is used for entropy computation and different that [33], where lower entropy indicates, possibly, an attack is in progress.

3.4 Early Detection in Openflow Controller

As it was shown before, one function of the controller is collecting statistics from all Openflow switches to detect inactive flows. These flows will be removed if they do not receive any packets for a period of time. This time period is called time-out in the Openflow specification and it can be set to different values.

For a lightweight solution, we propose adding another set of statistics to the controller. In this work, it is the entropy of the destination IP address in the controller. The function will determine if a higher than normal rate of incoming packets destined to the same destination.

In the previous section, the window size was set to be 50. The assumption is that the network has 50 or more hosts connected to it. One other component of DDoS detection by entropy is selecting an appropriate threshold, which will be discussed in the next chapter.

In the new function, every 50 Packet_In messages will be parsed for their destination IP address and the entropy of the list will be computed. The calculated entropy, then, will be compared to a threshold. If the calculated entropy is less than the threshold and it persists for a minimum of 5 consecutive entropy periods, it will be considered an attack. Detection within 5 entropy periods is 250 packets in the attack, which gives the network and early alert of attack. We tested with values one to five consecutive periods and five has the lowest false negative and positive for early detection. The results of these tests will be shown in Chapter 4.

With a window of 50 packets and a network of 50 hosts or more, maximum entropy, H_{MAX} is when each of the 50 packets is equally distributed among all the hosts. When an attack happens, the number of packets going to the same destination host, or the same subnet, is much higher so it will make the target unreachable to legitimate traffic. This would be the main objective of the attack. The other factor in an attack is its flow to the target. The attack packets will be targeting a single host or a subnet. If the rate of attack to a host is higher than the normal traffic level, which is always the case, the number of packets to that particular host in a window will increase. Because of that, the entropy will fall with a certain percentage. If it falls below the threshold, it is an attack.

One advantage of our method is the liberty of testing the controller with different attack rates to calibrate a threshold. In SDN, the controller can be connected to a simulator and tested then be deployed in the field to accept production network traffic. This property allows for the threshold to be tested before applying it.

Let I be the IP addresses of all hosts connected to the network, see Equation 3.6, and W be the window containing new packets' destination IP address x and their number of occurrence y ,Equation 3.7.

$$I = \{x_1, x_2, x_3, ..., x_N\} \quad (3.6)$$

$$W = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), ...\} \quad (3.7)$$

$$x \subseteq I \quad (3.8)$$

Then the entropy will be at its maximum if each destination IP is unique, see Equation 3.9:

$$\forall x \in W : H_{MAX} \Rightarrow y = 1 \quad (3.9)$$

If the above condition does not hold, then some IP addresses have appeared more than once.

Two conditions are chosen to be the trigger for an attack in our method. One is the threshold and the other is the continuity of the attack. There might be glitches in the network that cause irregularity in normal traffic. If a link to a switch goes down or some hosts become temporarily unavailable, the entropy might fall and trigger a false positive. To avoid false positives of this type, we propose a limit for the number of consecutive low entropy windows. Based on that, the condition for declaring an attack is shown in Equation 3.10 where T is the threshold, S is an array of five windows with lower than T entropy. "$\neg$" is the sign of negation. An attack happened if entropy of attack H_A, is smaller than the threshold and, having five consecutive lower than threshold entropies is true. Otherwise, there is no attack:

$$\begin{cases} attack \Rightarrow H_A < T \wedge S, \\ \neg attack, otherwise. \end{cases} \quad (3.10)$$

Figure 3.1 shows the flowchart of our detection method. Packet_In step shows that a new packet has arrived with new source address. First, we check the destination IP address to see if the destination IP address has an instance in our window. If it does, we increase the count for that IP address. Otherwise, it will be added as a new IP address. In the next step, we check if we have the 50th packet. If we do, the entropy of the window will be computed. Lastly, the entropy is compared to the threshold. If it is higher than the preset threshold, we go back to step one, waiting for new packets. Here, the count for consecutive lower-than-threshold entropies is set to 0. This is done to clear the count if there was lower entropy but had not reached five. If the entropy is lower than the threshold we increase the count for the consecutive lower-than-threshold entropies. If the count is five, an attack is detected.

The algorithm in Figure 3.1 is done with the addition of two function is the controller. The first one is called when a new Packet_In message arrives and it accepts the destination IP address as an argument, Figure 3.2. The second function is used to

computes the entropy, Figure 3.3. This is the only addition to the controller. Just to show the amount of code change in the controller the two functions are shown below.

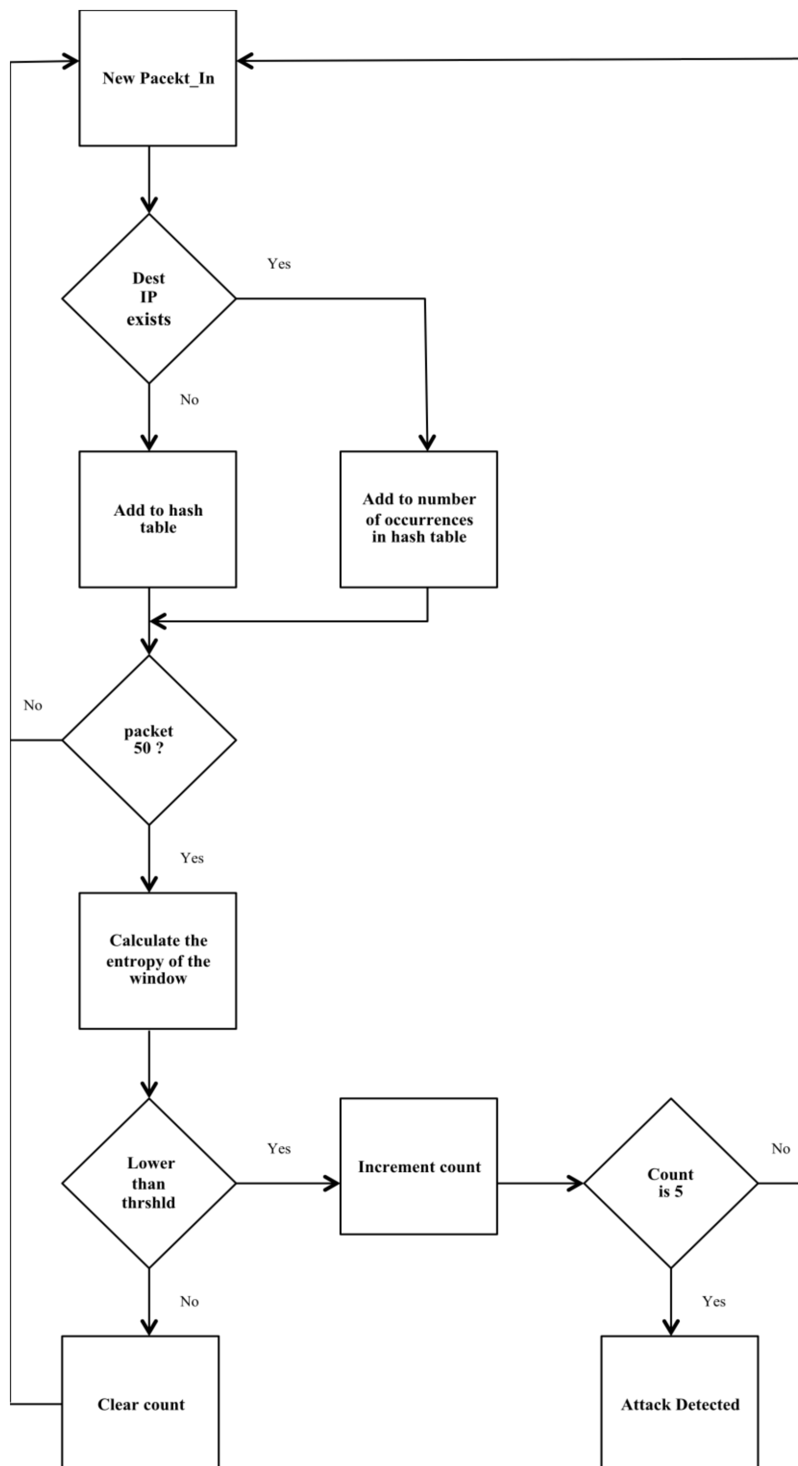

Figure 3.1 DDoS detection flowchart

```
# Statistics Lists
count = 0      # Counting the number of incoming packets
entDic = {}    # Hash table for the IP address and its occurrence
ipList = []    # List of IP addresses
dstEnt = []    # List of entropies
```

Figure 3.2 Lists added to the controller

```
def statcolect(self, element):

    # This function collects IP statistics

    l = 0

    # Increments until we reach 50

    self.count +=1

    self.ipList.append(element)

 # If the number of packets is 50, create a hash table of IP

 # addresses and number times they show up

    if self.count == 50:

        for i in self.ipList:

            l +=1

            if i not in self.entDic:

                self.entDic[i] =0

            self.entDic[i] +=1


        # call entropy

        self.entropy(self.entDic)

        # print the hash table and clear all

        print self.entDic

        self.entDic = {}

        self.ipList = []

        l = 0

        self.count = 0
```

Figure 3.3 Function to collect destination IP address stats

```
def entropy (self, lists):
        # this function computes entropy
        l = 50
        elist = []
        for p in lists.values():
            c = p/l
            elist.append(-c * math.log10(c))

        print 'Entropy = ', sum(elist)
        self.dstEnt.append(sum(elist))
        # collect 80 windows and print the entropy for a graph
        if (len(self.dstEnt)) == 80:
            print self.dstEnt
            self.dstEnt = {}
```

Figure 3.4 Entropy computation function

3.5 Comparison of Different Detection Methods to SDN Entropy

In [25], a machine learning method was used to learn the behavior of the network and based on that, decide whether an attack is in progress or not. This method is, mostly, used in non-SDN networks. When used in SDN networks it follows the same procedure and does not take into account the effect of DDoS on the controller. The solution has to run alongside SDN and has to be trained for a few hours before it can be used in the network. One other issue is the fact that SDN may reconfigure the network frequently. This means the Self-Organizing Maps solution has to be trained again for better protection. Finally, as the network expands, the neurons of the Self-Organizing Maps have to increase resulting in expansive neurons in the network and a lightweight solution turns into a heavy drag for the network. Our proposed solution runs within the controller and can be changed to fit the requirements as needed.

Shin et al. [28], propose Openflow for finding the shortest path to Network Intrusion Detection System devices. The solution requires the addition of NIDS devices along

the links of the network to monitor traffic for suspicious activity. In our solution, we propose using the controller itself for the detection of any attack. Although our solution is not tested on the cloud, the principle remains the same. The only difference is that the hosts are virtual hosts and the statistics can be changed from IP address to VLAN tag.

Like [28], Xing et al. [29] propose a DDoS detection method alongside SDN. SNORT is a DDoS detection tool itself and its combination with SDN is, again, using a non-SDN tool for SDN. The main purpose of our solution is making the detection process transparent by embedding it in the controller.

The closest solution to the one proposed in this thesis is [32]. This method adds an event processing module on top of the existing controllers and renames them to sub-controllers. The event processing module is considered a hyper controller receiving events from the sub-controllers and processes them for possible attacks. It seems that the hyper controller is supposed to have the bigger picture of the network for a better view of the attack. This, however, is the speculation of the writer. No algorithm is mentioned for the hyper controller attack detection and the solution has not been tested. The entropy method uses a different approach by assigning the detection task to each controller thus reducing the complexity.

In none of above solutions the controller has been the center of attention. Being the driving force of SDN and the operating system, it is the most essential component in the structure. This work is mainly concerned with detecting DDoS threats that are endangering the controller which is, in part, also protecting the hosts.

3.6 Concluding Remarks

In this chapter, it was shown how an attack can alter the entropy of normal traffic and how examining the entropy can be used to set a threshold. This principle is used as a way of detecting an attack in the controller.

We believe this is a lightweight and effective solution for SDN architecture with one controller. There are four major advantages that are worth mentioning:

1. Transparency: the statistics collection does not interfere with other function of the controller.

2. Limited resource usage: the collection of the statistics and its calculation is not heavy and does not use a considerable amount CPU power and memory.

3. Fully customizable: every parameter in the solution from window size to threshold and statistics fields can be quickly changed to comply with new requirements if needed.

4. Minimal changes to the controller: the code that is added to the controller is minimal and has two functions.

Controllers are not ready-made unchangeable equipment. They are software running on a machine that can be a laptop. Vendors can build the desired controller to fit their needs. With this solution, any controller can be easily fitted with a lightweight DDoS detection functionality without additional equipment like NIDS, DDoS detection software like SNORT or machine learning techniques like SOM.

Most importantly, this solution shows the flexibility of SDN for accepting important functionalities like DDoS detection with small changes.

In the next chapter, results of the proposed DDoS detection method with entropy will be examined.

Chapter 4

Simulation and Results

In this chapter, an Openflow controller will be connected to a network to form a SDN structure. Then, the entropy of the traffic to the controller is examined under normal and attack conditions.

4.1 Controller

The first part of our experiment is choosing a controller. There are few famous controllers available. The one that is used in this experiment is POX [34]. 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. It is an improved version of its predecessor NOX [35], and both are running on Python. POX works on Linux, Mac OS and windows, and it has topology discovery. For completeness, three other controllers should be mentioned. Floodlight [36] is another widely used controller that is open-source and written in Java. One advantage of Floodlight is facilitating application interface to the controller so they can run alongside it. Beacon [37] is another Java-based controller that is open-source and has high throughput and low latency. OpenDaylight [38] controller is the most recent addition to Openflow controllers. It meant to be a common platform for all SDN users. Recently, OpenDaylight announced its first release: Hydrogen. All the SDN papers that are mentioned in this paper are using NOX. NOX is no longer in development [34], which led to the use of POX in this paper.

4.2 Network Emulator

Mininet [39] is the network emulator that is used for this experiment. It is the standard network emulation tool that can be used for SDN. Mininet can prototype a network on a laptop or PC by using kernel namespace feature. Network namespace provides individual processes with their own network interfaces, ARP tables and routing tables. Mininet makes use of this feature of the kernel. It uses process-based virtualization to

run switches and hosts on the kernel. Large networks[1] with different topologies can be emulated and tested. In fact, the code developed in Mininet emulation can be moved to a real production network.

Creating a network in Mininet is as easy of entering the command *mn* to have a network with one switch, two hosts and a NOX controller. The command is shown in Appendix D. NOX is the default controller of Mininet.

4.2 Packet Generation

Packet generation is done by Scapy [40]. It is a very powerful tool for packet generating, scanning, sniffing, attacking and packet forging. Scapy is used here to generate UDP packets and spoof the source IP address of the packets.

Python programming language is used in POX. The code for generating random source IP addresses and host IP addresses is in Python. The function “randrange” is used which is inheriting the function “random”. This function produces a uniform random float in the range [0.0, 1.0). The generated float has 53-bit precision and has a period of $2^{19937-1}$ [41]. This number shows a long period of random number generation which will result in generating random numbers with uniform distribution. These numbers are joined together to form spoofed source IP addresses. Two other parameters that we set in Scapy are: type of packets and interval of packet generation. UDP packets are used for both attack and normal traffic. The interval was set to suit the test case. For instance, for an attack with 25% rate, normal traffic interval is 0.1 seconds and attack traffic is 0.025. This gave us windows with 25% of packets destined to one host. The code for generating normal and attack is shown in Appendix B and C respectively.

4.3 Network Setup

The experiment was done on a Lenovo laptop with a dual core processor, 2.7 GHz of power and up to 3.2 GHz in turbo mode, 4GB of ram, and 10/100/100/1000Mbitps

---

[1] Mininet claims running up to 4096 hosts on a single OS.

network interface. The operating system is Linux Ubuntu 13.04 and Mininet version 2.0.0 was run native on Linux. Mininet 2.0.0 supports Openflow version 1.0.

Using Mininet, a tree-type network of depth two with nine switches and 64 hosts was created. Figure 4.1 shows the network. Open Virtual Switch (OVS) [42] was used for network switches. OVS is a software switch that runs both on hardware and software. For this work, there is no difference between Openflow and OVS switch. Both do the same job and both are supported in Mininet. In Figure 4.1, all switches refers to Openflow enabled switches. The *L3_learning* module of POX was used for the controller.

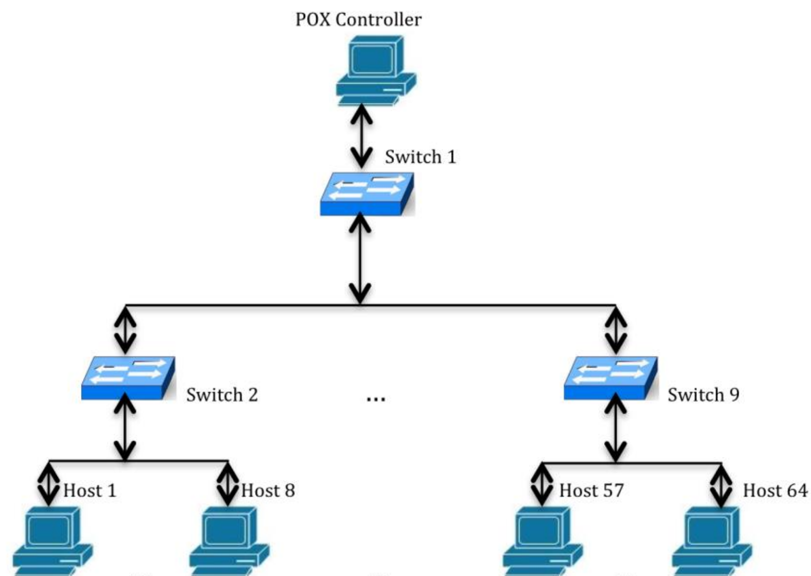

Figure 4.1 Experiment Network with 9 switches and 64 hosts

Next, the simulations and calculations that are run for setting a threshold will be discussed.

4.4 Choosing a Threshold

After setting all the parameters for a full network, a threshold is needed to detect DDoS attacks. The detection mechanism in our solution dictates that if the entropy is lower than the threshold, and it persists for five windows in a row, an attack is in progress.

To find the range for an optimal threshold, we ran a series of experiments to see the effect of an attack on the entropy. The experiments cover an attack to one host and a subnet of four hosts. To compare different rates of incoming packets, we controlled the rate of normal and attack traffic to increase and decrease the intensity of DDoS on the controller. Equation 4.1 is used for showing the rate *R* of incoming attack packets to normal traffic attacks. Where P_a and P_n are the number of attack packets and normal traffic packets respectively.

$$R = \frac{P_a}{P_n} \times 100\% \qquad (4.1)$$

We ran a 25% rate attack on one host for 25 times to find a suitable threshold. This threshold is the highest entropy of all cases so it will enable the controller to detect any attack with packets occupying 25% of the incoming traffic or more. We call it 25% rate attack. Table 4.1 shows the threshold and compares it to normal traffic values. The threshold is set to 1.31. To get this value the following was done:

a) Calculated the lowest value that normal traffic entropy can reach. This is equal to normal traffic mean entropy minus confidence interval, 1.4665.
b) Calculate the highest value that attack traffic entropy can reach. This is equal to attack traffic mean entropy plus confidence interval, 1.3047.
c) Find the difference of the two, 0.1618. We have a drop of 11%.

Even though the above calculation shows 13.047 could be the threshold, after 25 times running the simulation, we found that 1.31 has much less false negatives. Hence, a threshold of 1.31 will give us a clear cut for detecting any DDoS attack that will occupy 25% or more of the incoming traffic. It should be noted that the

confidence intervals are very small, 0.0035 and 0.0047, and as a result, they do not appear on the graphs.

Table 4.1 Threshold value calculation

	Normal Traffic	25% Rate Attack
Mean	1.47	1.3
Standard Deviation	0.009	0.012
Confidence interval	±0.0035	±0.0047
Confidence intvl Max	1.4735	1.3047
Confidence intvl Min	1.4665	1.2953
Difference of Normal traffic min and Attack traffic max.	0.1618	
Threshold	1.31	

In the previous chapter, we chose five consecutive periods of lower than threshold detection for declaring an attack. There are two reasons for choosing this number:

i) If a switch goes down temporarily or a link is broken, the network admin will have time to notice that reducing he false positive.

ii) Probability of having five consecutive windows lower than threshold during normal traffic is very low compared to lower numbers.

In the end, all of these values can be changed to fit the requirements of the network design. Even when the network is running live, these values can be modified and this is one of the advantages of central control in SDN. Next we will look at different test cases that were performed for evaluating our detection method.

4.5 Test Cases

The experiment covers five cases of attacks and a normal traffic run. Three different intensity attacks are run on one host and two different intensity attacks on four hosts belonging to the same switch and subnet. Normal traffic is run on all switches with randomly generated packets going to all hosts. Attack traffic is run from one host. Attacks were run manullay (i.e. a script was run after one third of the length of our simulation.

In Mininet, IP addresses for all hosts are assigned incrementally from 10.0.0.1 onward. For one host attack, we randomly chose a host in a switch to send attack packets to another host while all other hosts and switches are running normal traffic. In the case of four hosts being attacked, a random host will send attack packets while the rest of the network is running normal traffic. Table 4.2 shows the attack traffic profile. All the traffic packets will be UDP, destination port is 80 and type of attack is DDoS. In Openflow, by default, only the packet header is sent to the controller so no payload was added to the generated packets.

Table 4.2 Attack traffic profile

Protocol	Port	Payload	Type of Attack
UDP (Protocol 17)	80	None	DDoS

The three test cases on a single host have rates of 25%, 50% and 75%. In all tests, two Scapy programs are running. One is generating normal traffic and the other generating the attack that sends packets faster than normal traffic. This scheme resulted in having from 9 to 14 packets out of 50 for 25% rate attack with the same destination IP address. This was not by design. Often, there were small glitches in Mininet. For the other two cases, the rates are changed to increase the number of attack packets. The 50% rate reached up 26 out of 50 and 75% up to 39 packets out of 50.

Subnet attack covered rates 50% and 75%. The 25% is not used in subnet attack. Since 25% of attack packets for four hosts will be average of 12 packets. When divided among four hosts, each host gets three packets. This rate is considered normal and does not pose a real threat to the controller.

We used a subnet of four hosts to test an attack on a group of hosts. In 50% rate attack, each host received 5 to 7 packets and in 75% rate the range was 9 to 10 packets per host. Although this variation in attack packets was unintended, it widened the percentage of attack traffic in the test. For instance, the 25% rate attack on a single host, the attack packets are 9 to 14, which translates to 18% to 28%.

In reality, DDoS attacks reach a much higher intensity. Attacks, often, they generate a traffic that is few times higher than the normal traffic. Figure 4.2 shows an attack that reaches 250 packets per second while normal traffic is only 50 packets per second. If this attack continues without mitigation in a controller, all its resources will be bound in processing attack packets.

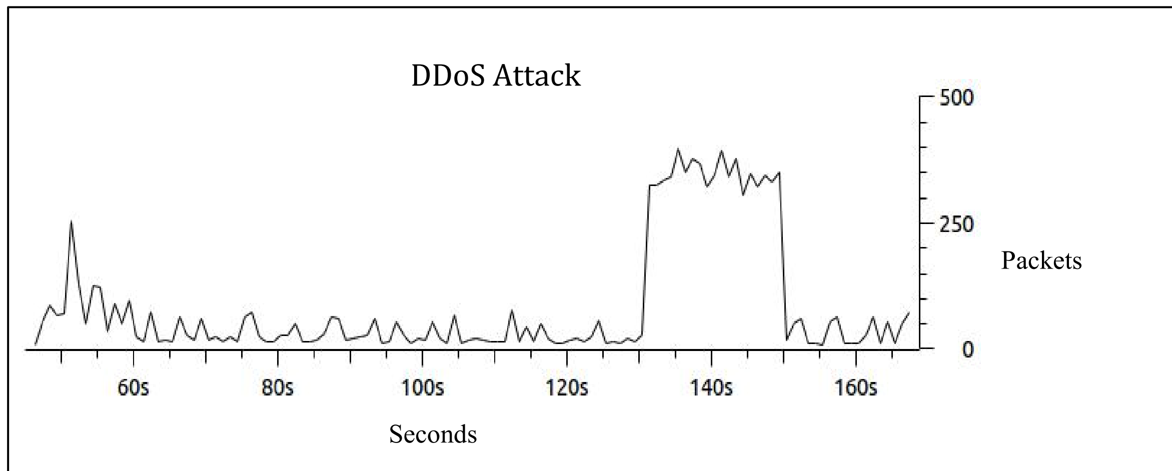

Figure 4.2 Sudden increase of traffic in DDoS

4.5.1 Attack on One Host

In this section, we examine the results of an attack against one host. Each graph is a result of 15 runs with 4000 packets per test. Each point on the horizontal axis shows a window of 50 packets and the vertical axis shows the entropy for that window. The graph's data are the mean values over 15 runs.

Figure 4.3 shows change of entropy in 25% rate attack. In all the graphs, the blue line is the normal traffic. The red line shows the transition from normal traffic to attack and back to normal traffic. In Figure 4.3, a distinct difference between the normal and attack traffic entropy can be seen. In the graph, the first six entropies are well below our threshold of 1.31. In table 4.1, the lowest point of confidence interval for normal traffic was 1.4665 and the highest point of attack entropy was 13.047. The difference of these two values, 0.1618, shows 11% drop in entropy which is 34 bigger than the 25% attack traffic confidence interval, 0.0047. This result shows that this method can, easily, detect any attack occupying 25% or more of the incoming traffic when it is destined for one host. Since 25% rate was run 25 times, we computed the success rate of this method based on this 25 runs of the simulation. Within these runs, only one

false negative was detected where an attack was in progress but was not detected by the controller. This shows a success rate of 96% in detecting DDoS attacks within the first 250 incoming packets. For all other cases, the success rate is 100% and no attack passed unnoticed.

To look at more concentrated attacks, two higher rate tests were launched on one host. Figure 4.4 shows an attack with 50% rate and 4.5 shows a 75% rate attack on a single host. Both simulations are compared to normal traffic average to show the difference of entropy in both conditions.

In 50% and 75% rate attacks, the window of attack is deeper and narrower showing a bigger drop in entropy. In one host attack, 500 packets are sent as attack traffic. When the rate of attack increases and the number of generated attack packets is fixed, the percentage of attack packets in the window increases. This will results in a deeper and narrower graph of attack. In the 75% case, the drop was too narrow so packets were increased to 1000 packets to have a better view of the entropy drop.

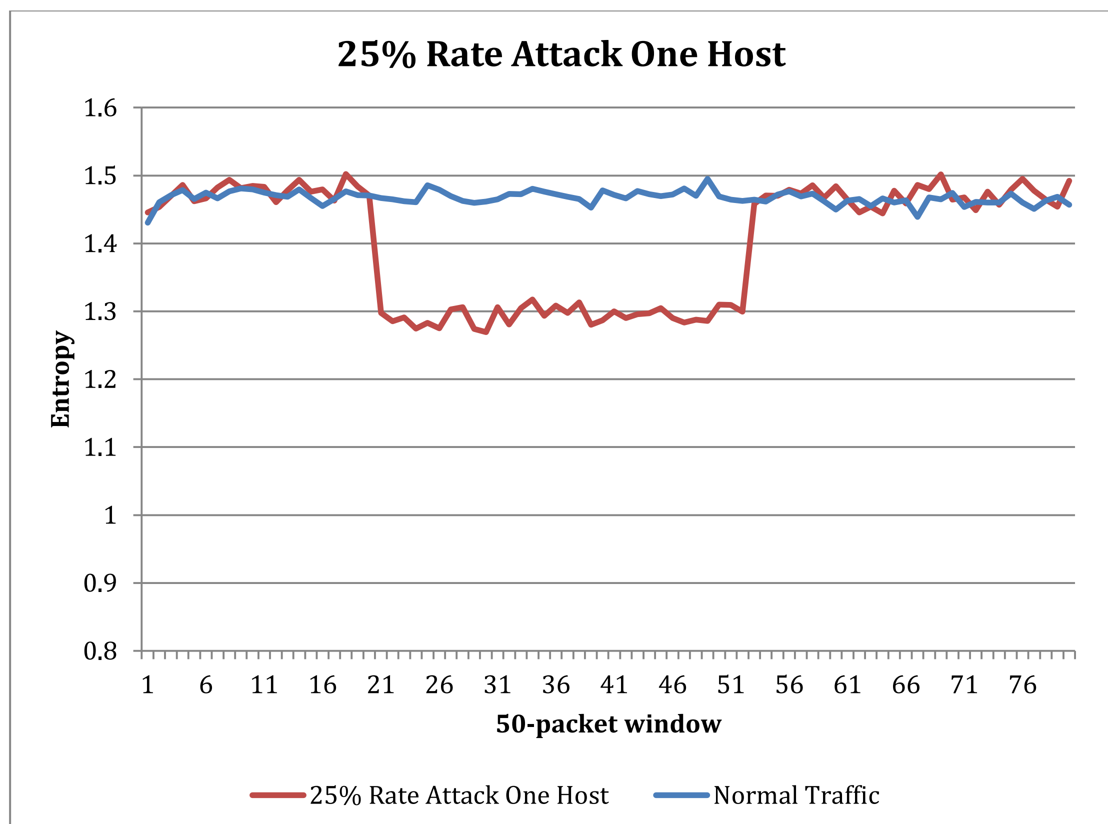

Figure 4.3 25% rate attack on one host

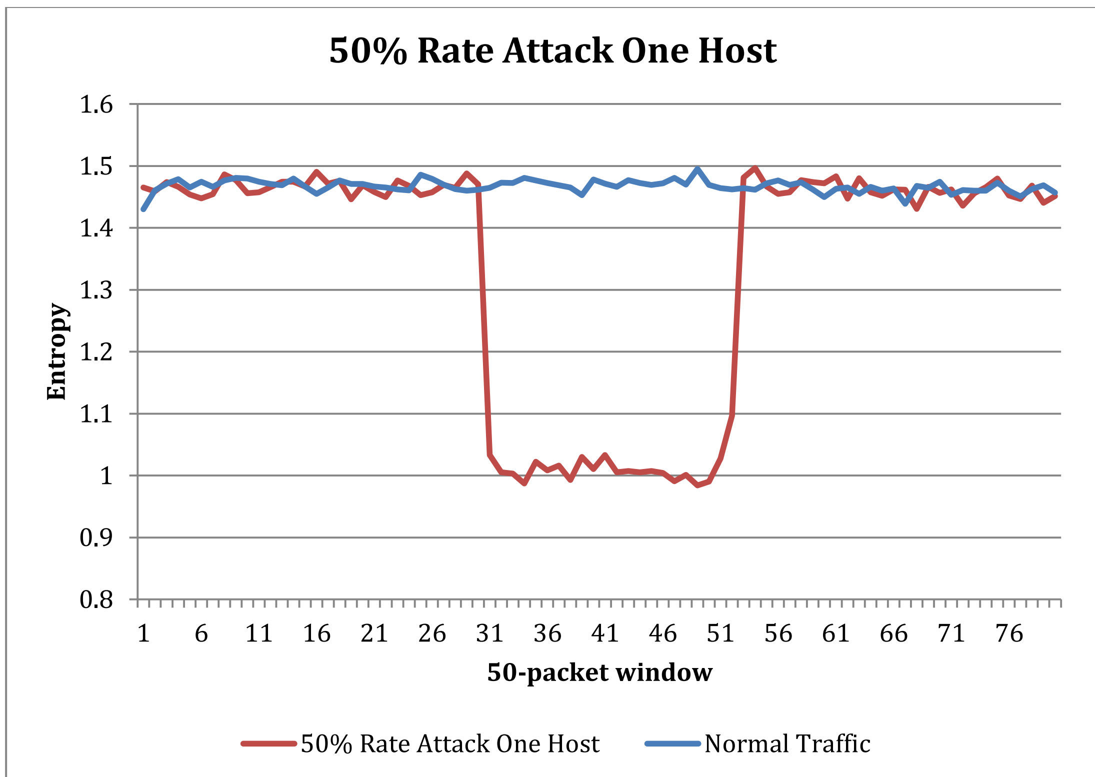

Figure 4.4 50% rate attack on one host

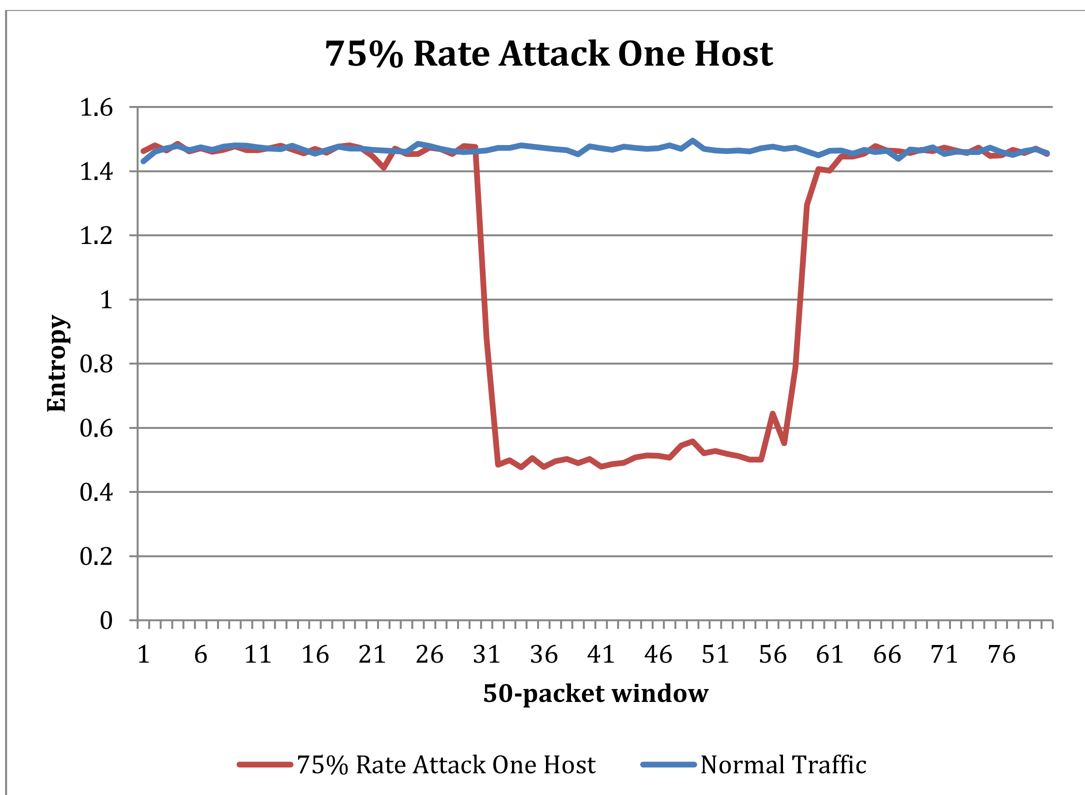

Figure 4.5 75% rate attack on one host

In Chapter 2, we saw that, instead of a single host, a subnet mask might be infiltrated and attacked. In the next section, we will look at a situation where a subnet of hosts is attacked.

4.5.2 Attack on a Subnet

In this section, four hosts of the same subnet are attacked to examine the effectiveness of entropy in detecting such attacks on the controller. Since the baseline for detection was 25% rate on one host, the threshold was kept the same. In addition, 15 runs of attack on a subnet with 50% rate proved to have entropy lower than the threshold. Figure 4.6 has a drop in entropy that is well below the 1.31 threshold of the 25% but higher than 50% rate on one host. Table 4.3 shows the difference of entropy between the attack and the threshold. The confidence interval of 25% rate was 0.0047 and the closest entropy to threshold is the 25% rate with a difference of 0.01. This value is twice that of confidence interval. The end results will show the accuracy of our choice.

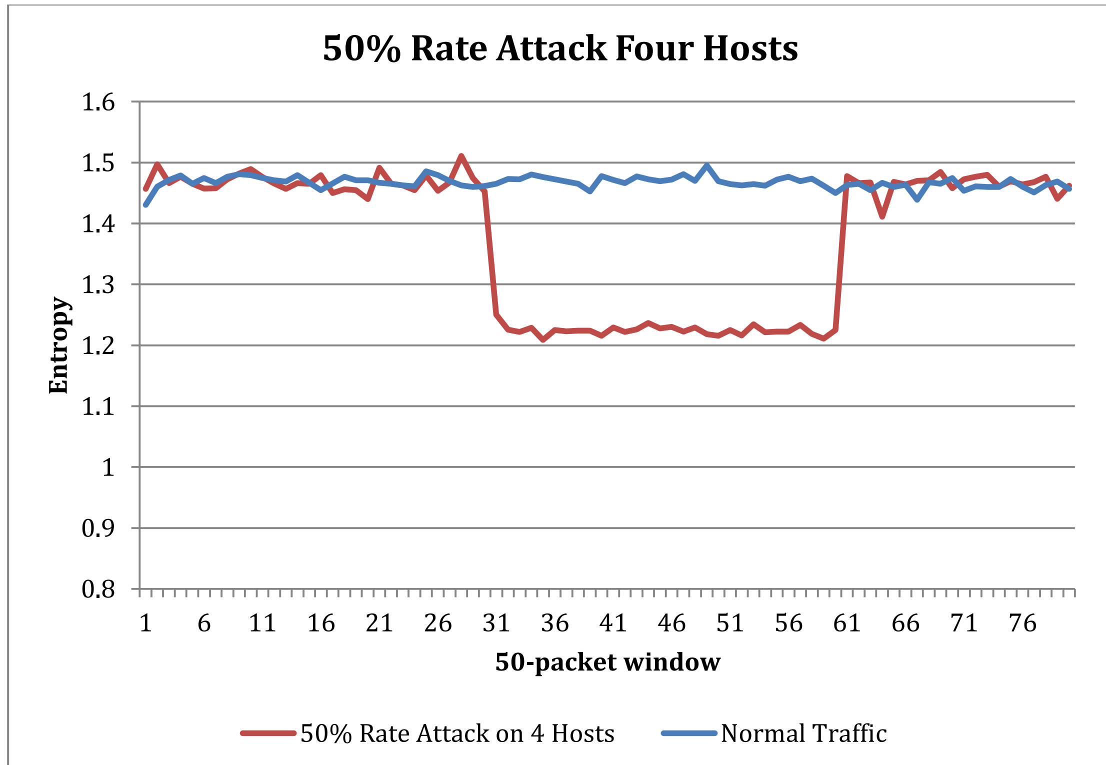

Figure 4.6 50% rate attack on four hosts

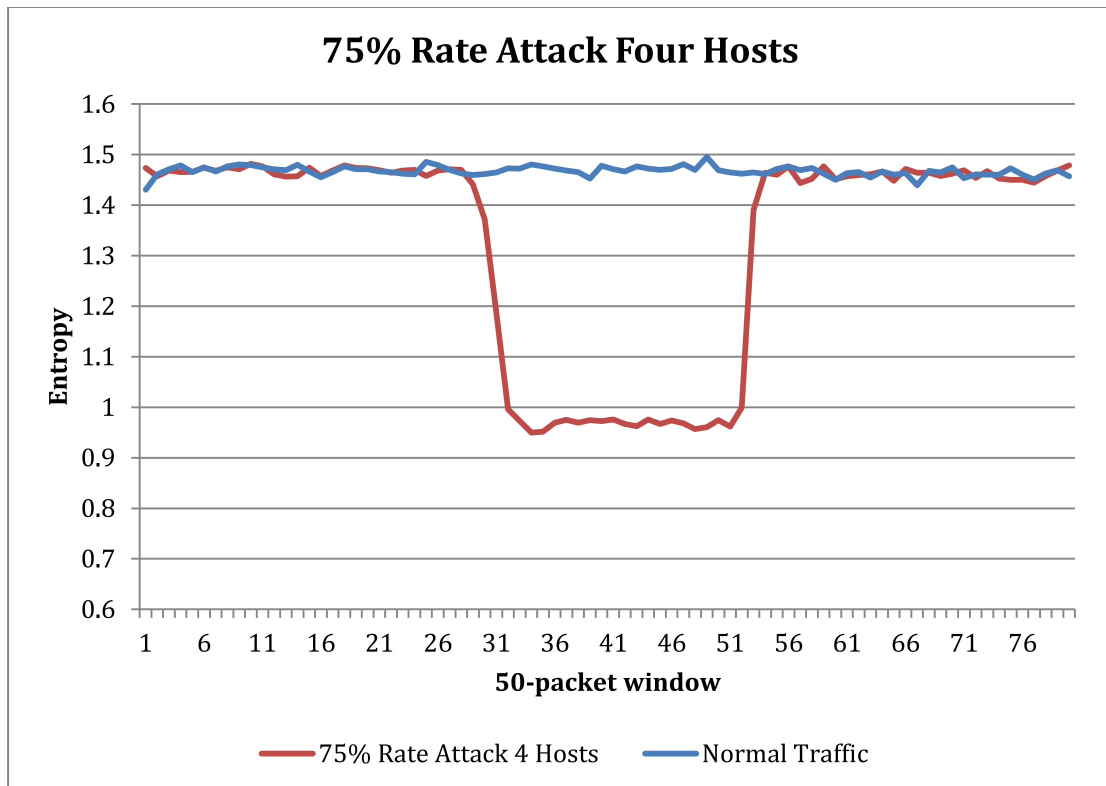

Figure 4.7 75% rate attack on four hosts

Figure 4.7 shows 75% rate attack on four hosts. We can see a sharp drop in entropy when a large number of packets are flowing to the same subnet. The confidence interval of 75% rate attack on a single host and 75% rate on a subnet show the highest confidence interval. If we look at Figure 4.8 and 4.9, we can see how 75% rate attacks have a wedge-shaped line when the attack starts and when it reaches the end. This is the result of gradual increase and decrease of the attack packets' share in the window.

Table 4.3 Entropies of test cases

Traffic Type	Mean Entropy	Threshold – attack entropy	Confidence interval
25% rate on host	1.3	0.01	±0.0035
50% rate on host	1.06	0.24	±0.0099
75% rate on host	0.56	0.74	±0.03
50% rate on 4 hosts	1.2	0.1	±0.0028
75% rate on 4 hosts	0.98	0.32	±0.02

When the attack traffic reaches 100%, the controller will be fully bound in processing malicious packets that are the results of a DDoS attack (due to hardware limitation, we did not test the point of failure of the controller).

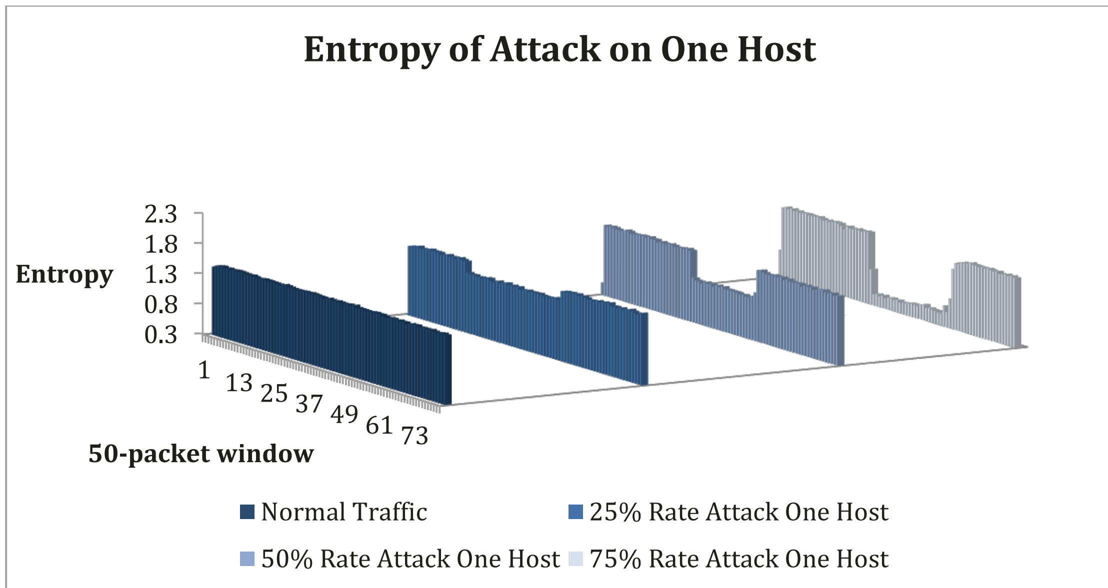

Figure 4.8 Comparison of entropy drop, one host

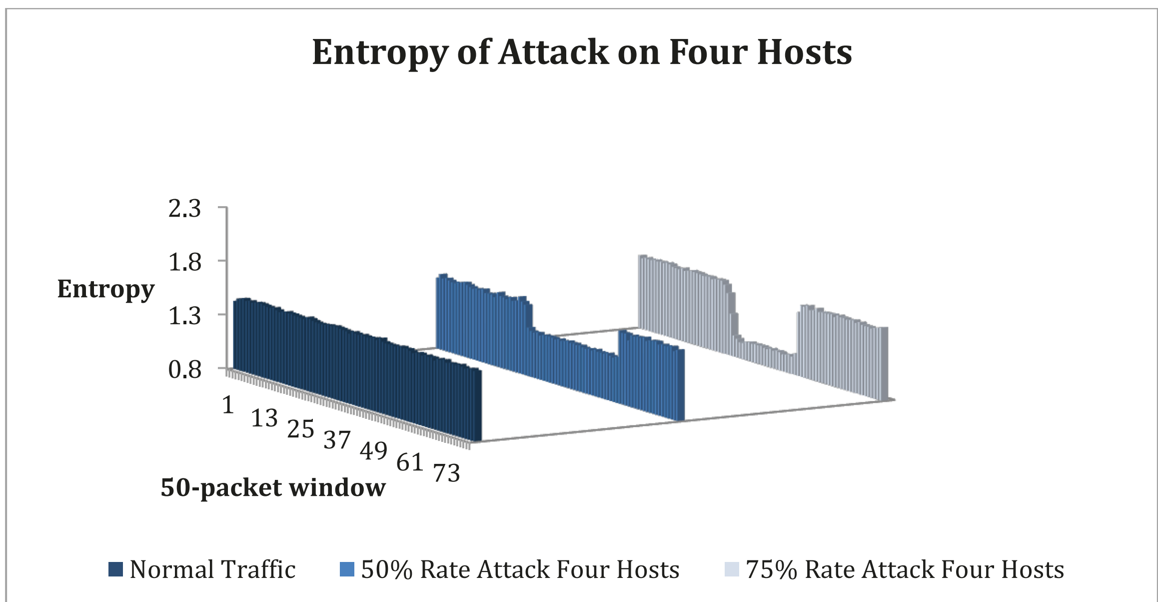

Figure 4.9 Comparison of entropy drop, four hosts

When we look at the number of packets that are sent in an attack, they are varying from seven to 39 packets in a window. Although seven packets in a window of 50 seem small, if the values in Table 4.4 are compared to the values of average incoming packet per host, we see the number is in fact high. If we take the example of 50% rate attack on four hosts, each host will receive an average of (6.25) packet per window in

an attack situation. The normal rate for a single host is (1.28) packet per window. This shows an increase of, nearly, 5 times in the rate of incoming packets. If we take the two examples of 75% rate attack on four hosts and 75% rate attack on one host, the rate of attack packets is (7.3) times and (30) times respectively. Considering the fact that in the four hosts' case each host is receiving (7.3) times the average traffic, the collective rate is a considerable increase. This shows that the methodology of our test is correct and consistent with real-world attack situation where attack traffic has much higher rates than normal incoming traffic.

In this section, we proved the effectiveness of this method in detecting DDoS attacks with an accuracy of 96%. Next, we will examine the effect of our method on the resources of the controller.

Table 4.4 Number of incoming packets per host in each test case

Test case	Average incoming packets per host
Normal traffic	1.28
25% Rate one host	12.5
50% Rate one host	25
75% Rate one host	37.5
50% Rate four hosts	6.25
75% Rate four hosts	9.375

4.6 Effects of the Added Functions on Resource Usage

In chapter 3, we showed that two functions were added to the controller. This was done with the intention of designing a solution that has minimal effect on the controller in terms of code addition and, in part, resource usage.

To examine the effect of our solution on the controllers CPU, we killed all the unnecessary process on our Linux laptop (running Ubuntu 13.04) except for the simulation. We also disabled networking and ran the simulation on Linux's loopback.

We ran two simulations:

i) A 25% rate attack on POX controller without our solution.

ii) A 25% rate attack on POX controller with our solution.

In both cases we left system monitoring application of Linux running. For both simulations, the attack lasted 40 seconds and normal traffic ran for 240 seconds. We captured the screen for both cases. The screen resolution is not high and the captured screen is not better so two black lines were added showing the interval of 60% to 80% on the left in Figure 4.10 and 4.11. In both figures, there is no visible difference between the two graphs in normal traffic except for few small impulses at both ends. In Figure 4.10, during the attack period, we can see a sharper increase and drop compared to Figure 4.11. The blue lines on both sides of the attack period show the difference. However, during the attack, both graphs reach a maximum of 78%. This is a very minor difference and can be considered a minimal use of resources. This shows that the solution is, in fact, transparent and lightweight.

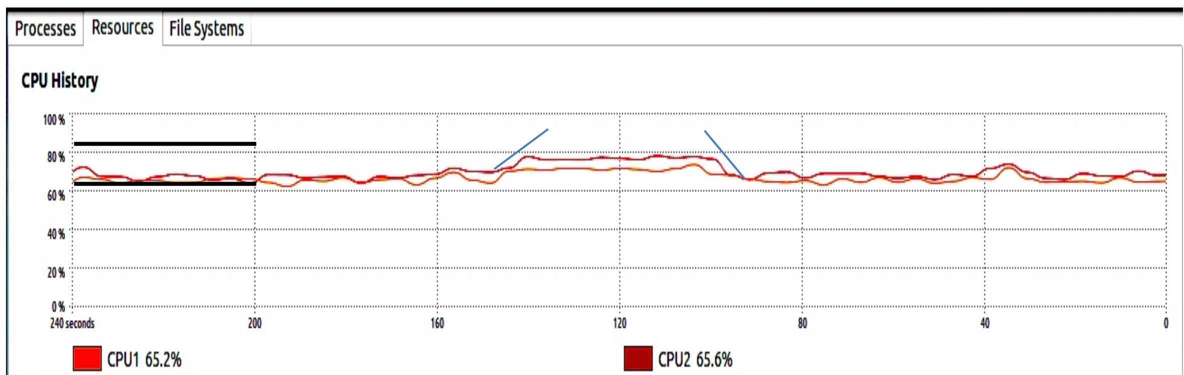

Figure 4.10 CPU usage with no DDoS detection

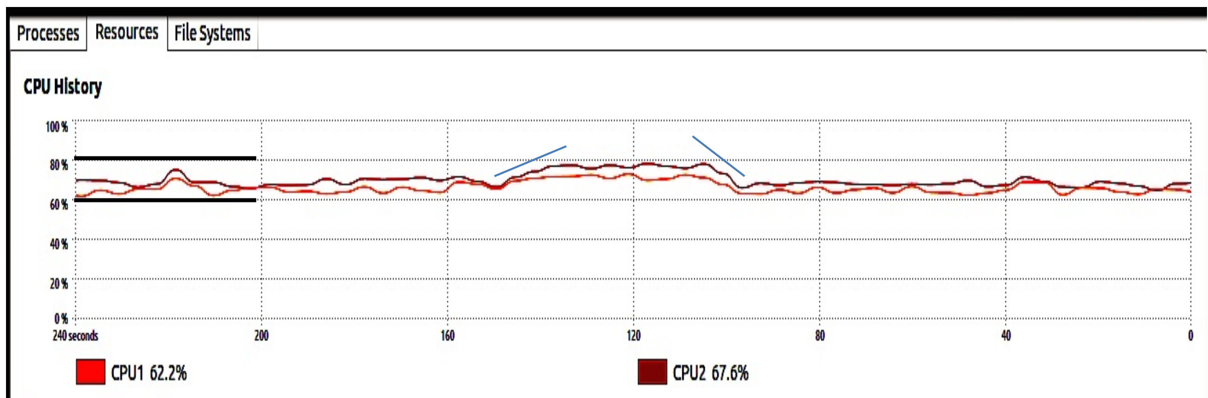

Figure 4.11 CPU usage with DDoS detection

4.7 Summary of the Results

In 25 runs of the 25% rate, we had only one false negative. In that case, first, eight then, nine consecutive windows passed with higher than threshold entropy while attack was happening. This shows a 96% accuracy detecting DDoS in SDN controller. If the threshold is set at a higher rate, 35% rate attack for instance, the accuracy is 100%.

In section 3.5, we compared some methods of DDoS detection in SDN. In [28], Network Intrusion Detection Systems(NIDS) are used in SDN architecture and the controller was used to find the shortest path for sending traffic into NIDS. In [29], SNORT, which is common software for DDoS detection in the cloud, is used alongside the controller to detect DDoS. These two methods do not report a success rate in detection. It seems that the viability of the method is being shown in these two papers. In [32], an event processing module was used which examines all incoming traffic to controllers to detect DDoS. This method was not tested. To validate our results, they should be compared to a solution that is implemented in SDN. The solution in [25] uses Self Organizing Maps (SOM) machine learning method to detect attacks in SDN. This method's results show 98.6% and 99.11% success in detecting DDoS with Slef Organizing Maps. It applies the same principles that are used for non-SDN networks, which is detecting an attack against the network or a host in the network without any specific measures to protect the controller. It runs a software in the network that uses several calculations with large matrices for learning the

behavior of the network. The entropy detection that is used in this research does all the functionality of the above method without any of the complex measures used in it. Perhaps, the biggest difference between the two is the fact that SOM applies non-SDN solution to SDN network while our method takes a non-SDN solution and tailors to fit SDN networks requirements.

There are major advantages to our method that do not exist in SOM solution;

a. Directly protects the controller against DDoS
b. Specifically designed for SDN
c. High accuracy of 96%
d. Low complexity with two functions added
e. Minimal resource use

These can be compared to a method that has complex software, must be trained for hours, runs complex matrix computations for detection and, inevitably, uses a lot of resources in the network.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Protecting the operating system of SDN (i.e. the controller) by detecting DDoS attacks was the center of this research. The challenge in detecting any threats to the controller is early detection. Although the term "early" can be used loosely in detecting an attack, we quantified the early detection to the first 250 packets of traffic as minimum and 500 packets maximum.

This solution is not only efficient in detection, it has minimal code addition to the controller program and does not increase CPU load in either normal or attack condition.

There are different methods for detecting attacks and each method is used differently. In this research, we focused on a solution that works particularly well for SDN, based on its specifications, points of strength and limitations. We made use of the fact that SDN specification dictates the forwarding of new packets to the controller. We took into account the abilities of the controller and its broad view of the whole network and used that for adding entropy statistics collection. Finally, understanding the importance of keeping the controller connected to the network at all times, we came up with a solution to detect any threat at its very beginning. Entropy has been used in DDoS detection in non-SDN network but, to the best of our knowledge, it has not been used in SDN and this is the first solution of its kind in SDN.

By applying entropy as a detection method, we were able to detect attacks on one host or a subnet of hosts in a network. In the case of one host, our detection method proved to be able to, successfully, catch drops in entropy when attack packets are as low as 25% of the incoming traffic to the controller. For the subnet attack, our method, successfully, detected the attack when its packets were as low as 50% of coming traffic to the controller. The success rate this method is 96% when using a threshold of 25% rate attack packets per total traffic. The closest method in non-SDN

networks can detect an attack when 75% to 100% of traffic is DDoS. We believe that this is an effective method in addressing the detection of DDoS in SDN with accuracy and efficiency.

One contribution of this research is covering controller security in SDN. There seems to be a lack of research focusing on this specific part of SDN. The topic of SDN is new and the reason might be limited deployment of this structure as a production network. The contributions of this research are:

- Showed how DDoS attack can overwhelm the controller in SDN architecture.
- Proposed a lightweight and fast DDoS detection mechanism based on entropy, to protect the controller.
- Implement the proposed mechanism using Mininet and POX controller.
- Showed the effectiveness of the solution through extensive simulations.

5.2 Future work

One limitation that our method has is the detection of attacks when the entire network is being targeted by DDoS. When malicious packets are targeting every host, entropy might not change by a large margin. Detecting such attacks will be an addition to this research.

Having addressed the detection in one controller network, two more tasks to be done are:

i) Detection of attack in a multi-controller SDN structure
ii) Mitigation of the attack.

In SDN, networks are connected to controllers and, several controllers might be connected to each other. Detecting an attack in one of them could show the source of the attack and make discovery of the source much easier. This method requires an inter-controller communication that sends the threat alert to all the controllers. Adding this communication process to SDN will be an extension to the current work and a topic for future work.

Mitigation of DDoS in SDN is the next future work for this research. The first step of mitigation will be detection of the source or sources of the attack. Adding more statistics collection to the controller will enable it to monitor the flow rate at the switch level where attack flows are directed to the controller. Then, more elaborate techniques can be used to pinpoint the malicious hosts. This is a very interesting future work that can be a baseline for any detection scheme in SDN structure.

Bibliography

[1] Open Networking Foundation. (2014, Jan.) ONF. [Online]. https://www.opennetworking.org/

[2] T. Anderson, H. Balakrishnan, G. Parulkar, J. Rexford, S. Shenker, J. Turner N. McKeown, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM*, vol. 38, no. 2, pp. 69-74, April 2008.

[3] SDN Central. (2013, Oct.) sdncentral. [Online]. http://www.sdncentral.com/announced-sdn-products/

[4] M. Masikos, O. Zouraraki C. Patrikakis. (2004, December) CISCO. [Online]. http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_7-4/dos_attacks.html

[5] A. Mitrocotsa C. Douligeris, "DDoS Attack and Defence Mechanism: A Classification," in *Singnal processing and information tecnology in 3rd IEEE International Symposium*, Apr 2003, pp. 190-193.

[6] Prolexic. (2013, December) DoS and DDoS attack reports, trends and statistics. [Online]. http://www.prolexic.com/knowledge-center-dos-and-ddos-attack-reports.html

[8] P. Reiher J. Mirkovic, "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms ," *SIGCOM*, vol. 34, no. 2, pp. 39-53, April 2004.

[7] Google, Arbor. (2013, Oct) Digital Attack Map. [Online]. http://www.digitalattackmap.com/#anim=1&color=0&country=ALL&time=16003&view=map

[9] NSFOCUS. (2013, Nov) NSFOCUS. [Online]. http://en.nsfocus.com/SecurityReport/2013%20NSFOCUS%20Mid-Year%20DDoS%20Threat%20Report.pdf

[10] C. Ji M. Thottan, "Anomaly Detection in IP Networks," *IEEE Transaction on Signal Processing*, vol. 51, no. 8, pp. 2291-2204, Aug 2003.

[11] D. Schnackenberg, R. Balupari, D, Kindred L. Feinstein, "Statistical Approaches to DDoS Attack Detection and Response," in *DARPA Information Survivability Conference and Expedition*, vol. 2003, Apr.

[12] Z. Qin, L. Ou, J. Liu, A. X. Liu J. Zhang, "An Advanced Entropy-Based DDoS Detection Scheme," in *International Conference on Information, Networking and Automation*, 2010, pp. 67-71.

[13] I. Ra G. No, "An efficient and reliable DDoS attack detection using fast entropy computation method," in *International Symposium on Communication and Information technology*, 2009, pp. 1223-1228.

[14] Y. Chen X. Ma, "DDoS Detection Method Based on Chaos Analysis of Network Traffic Entropy," *IEEE Communications Letters*, vol. PP, no. 99, pp. 1-4, 2013.

[15] F. M. Ham, *Principles of neurocomputing for Science and Engineering*.: McGraw Hill, 1991.

[16] D. O,Brien S. Seufert, "Machine Learning for Automatic Defence against Distributed Denial of Service Attack," in *ICC*, 2007, pp. 1217-1222.

[18] IBM. (2014, Feb) IBM SPSS Modeler. [Online]. http://pic.dhe.ibm.com/infocenter/spssmodl/v15r0m0/index.jsp?topic=%2Fcom.ibm.spss.modeler.help%2Fidh_neuralnet_network.htm

[17] G. Serpen M. Sabhnani. (2014, Jan) BSTU Laboratory of Artificial Neural Networks. [Online]. http://neuro.bstu.by/ai/To-dom/My_research/Papers-0/For-research/D-mining/Anomaly-D/KDD-cup-99/mlmta03.pdf

[19] S. Oechsner, D. Schlosser, R. Pries, S. Goll, P. Tran-Gia M. Jarschel, "Modeling and Performance Evaluation of an OpenFlow Architecture," in *23rd ITC* , 2011.

[20] A. L. Cox, T. S. E. Ng Z. Cai. CAI, Z., "*Maestro: A system for scalable OpenFlow control*", Tech. Rep. TR10-11, Rice University- Department of Computer Science, December 2010.

[21] E. Jacob, D. Sanchez, Y. Demchenko J. Matias, "An Openflow Based Network Virtualization Framework for the Cloud," in *IEEE Third International Conference on Cloud Computing Technology* , 2011, pp. 672-678.

[22] B. Martini, M. Gharbaoui, P. Castoldi D. Adami, "Effective Resource Control Strategies using Openflow in Cloud Data center," in *International Symposium on Integrated Network Management*, 2013, pp. 568-574.

[23] A. Anjum, R. Hill, N. Bessis, S.L. Kiani C. Baker, "Improving Cloud Datacenter Scalability, Agility and Performance using Openflow," in *4th International Conference on Intelligent Networking and Collaborative Systems*, 2012, pp. 20-27.

[24] R. Canonico, M. Brunner, P. Hasselmeyer, F. Mir R. Bifulco, "A paractical experience in designing an OpenFlow controller," *European Workshop on SDN* , pp. 61-66, Oct 2012.

[25] E. Mota, A. Passito R. Braga, "Lightwight DDoS flooding attack detection using

NOX/Openflow," in *IEEE 35th conference on Local Computer Networks*, 2010, pp. 408-415.

[26] S. Ostermann, B. Tjaden M. Ramadas, "Detecting anamalous network traffic with self-organizing maps," in *Recent Advances in Intrusion Detection*, 2003, pp. 36-54.

[28] G. Gu S. Shin, "CloudWatcher: Network Security Monitoring Using OpenFlow in Dynamic Cloud Networks (or: How to provide security monitoring as a service in clouds?)," in *20th IEEE International conference on Network Protocols* , 2012, pp. 1-6.

[27] T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeon, S. Shenker N. Gude. (2014, Jan) acm sigcomm. [Online]. http://www.sigcomm.org/node/2699

[29] D. Huang, L. Xu, C. Chung T. Xing, "SnortFlow: A openflow-based Intrusion Prevention System in Cloud Environment," *Second GENI Research nad Educational Experiment Workshop* , pp. 89-92, 2013.

[30] (2014, Jan) SourceFire Inc. [Online]. http://www.snort.org

[31] R. Bennesby, E. Mota, A. Passito P. Fonseca, "A Replication Component for Resilient Openflow-based Networking," in *Network Operations and Management Symposium*, 2012, pp. 933-939.

[32] W. Su, L. Wu, Y. Huang, S. Kuo Y. Hu, "Design of Event-Based Intrusion Detection System on OpenFlow Network," in *IEEE International Conference on Dependable Systems and Networks (SDN)*, 2013, pp. 1-2.

[33] T. Nakashima, T. Sueyoshi S. Oshima, "Early DoS/DDoS Detection Method using Short-term Statistics," in *International Conference on Complex, Intelligent and Software Intensive Systems*, 2010, pp. 168-173.

[34] M. McCauley. (2013, Nov) NOXREPO. [Online]. http://www.noxrepo.org/

[35] T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker N. Gude, "NOX: Towards an Operating System for Networks," *Computer Communication Review*, vol. 38, no. 3, pp. 105-110, Jul 2008.

[36] Big Switch Networks. (2014, Mar) Project Floodlight. [Online]. http://www.projectfloodlight.org/floodlight/

[38] Linux Foundation. (2014, Jan) Open Daylight. [Online]. http://www.opendaylight.org/

[37] D. Erickson. (2014, Jan) Openflow. [Online]. https://openflow.stanford.edu/display/Beacon/Home

[39] (2014, Feb) Mininet. [Online]. http://mininet.org/

[40] (2014, Feb) Scapy. [Online]. http://www.secdev.org/projects/scapy/

[41] Python Standar Library. (2014, Jan) Python Douments. [Online]. https://docs.python.org/2/library/random.html

[42] (2014, Jan) Open Vswitch. [Online]. http://openvswitch.org/

Appendix

Appendix A: Statistics collection and entropy computation code

```
count = 0         # keeping count for a window size of 50
  entDic = {}      # hash table for (IP, number of time it appeared in a window)
  ipList = []      # list of IP addresses
  dstEnt = []      # list of entropies


  def statcolect(self, element):
    # collecting stats (IP)
    l = 0
    self.count +=1
    self.ipList.append(element)

    if self.count == 50:

      # we reached 50 fill hash table
      for i in self.ipList:
        l +=1
        if i not in self.entDic:
          self.entDic[i] =0
        self.entDic[i] +=1

      # call entropy with the table, then clear all
      self.entropy(self.entDic)
      print self.entDic  # printing to see the values
      self.entDic = {}
      self.ipList = []
      l = 0
      self.count = 0

  def entropy (self, lists):
    # this function computes entropy
    l = 50
    elist = []
    for p in lists.values():

      # calculating probability of each IP
      c = p/l

      # create a list of entropies
      elist.append(-c * math.log10(c))

    print 'Entropy = ', sum(elist) # printing to see the entropy
    self.dstEnt.append(sum(elist))
```

```
# have we reached 4000 packets…
if (len(self.dstEnt)) == 80:

   print self.dstEnt # printing to see the full run
   self.dstEnt = {}
```

Appendix B: Normal traffic generation code

```
#!/usr/bin/env python

import sys
import getopt
import time
from os import popen
from scapy.all import sendp, IP, UDP, Ether, TCP
from random import randrange

def sourceIPgen():
    """
    this function generates  random IP addresses

    """
   # these values are not valid for first octet of IP address
   not_valid = [10,127,254,255,1,2,169,172,192]

   first = randrange(1,256)

   while first in not_valid:
      first = randrange(1,256)

   ip = ".".join([str(first),str(randrange(1,256)),
                 str(randrange(1,256)),str(randrange(1,256))])

   return ip

# host IPs start with 10.0.0.
# the last value entered by user
def gendest(start, end):
    """
    this function randomly generates IP
    addresses of the hosts based on entered
    start and end values

    """

   first = 10
   second = 0; third = 0;
   ip = ".".join([str(first),str(second),
                 str(third),str(randrange(start,end))])
   return ip
```

```
#send the generated IPs
def main():
"""
main method
receives the last number of host IP addresses
i.e. if packets are going to 10.0.0.1 to 10.0.0.8,
user should enter 1 and 8
then sends packets to those IP address
"""

  try:
    opts, args = getopt.getopt(sys.argv[1:],'s:e:',['start=','end='])
  except getopt.GetoptError:
    sys.exit(2)
  for opt, arg in opts:
    if opt =='-s':
      start = int(arg)
    elif opt =='-e':
      end = int(arg)
  if start == '':
    sys.exit()
  if end == '':
    sys.exit()

  # open interface eth0 to send packets
  interface = popen('ifconfig | awk \'/eth0/ {print $1}\'').read()

  # send normal traffic to the destination hosts
  for i in xrange(1000):

    # form the packet
    packets = Ether()/IP(dst=gendest(start, end),src=sourceIPgen())/UDP(dport=80,sport=2)
    print(repr(packets))

    # send packet with the defined interval  (seconds)
    sendp( packets,iface=interface.rstrip(),inter=0.1)


#main
if __name__=="__main__":
  main()
```

Appendix C: Attack traffic generation code

```
#!/usr/bin/env python
import sys
import time
from os import popen
from scapy.all import sendp, IP, UDP, Ether, TCP
from random import randrange

def sourceIPgen():
  """
     this function generates  random IP addresses

     """
  # these values are not valid for first octet of IP address
  not_valid = [10,127,254,255,1,2,169,172,192]

  first = randrange(1,256)

  while first in not_valid:
    first = randrange(1,256)
    print first
  ip = ".".join([str(first),str(randrange(1,256)),
          str(randrange(1,256)),str(randrange(1,256))])
  print ip
  return ip

#send the generated IPs
def main():

  #getting the ip address to send attack packets
  dstIP = sys.argv[1:]
  print dstIP
  src_port = 80
  dst_port = 1

  # open interface eth0 to send packets
  interface = popen('ifconfig | awk \'/eth0/ {print $1}\'').read()

  for i in xrange(0,500):
    # form the packet
    packets = Ether()/IP(dst=dstIP,src=sourceIPgen())/UDP(dport=dst_port,sport=src_port)
    print(repr(packets))

   # send packet with the defined interval (seconds)
    sendp( packets,iface=interface.rstrip(),inter=0.025)

#main
if __name__=="__main__":
  main()
```

Appendix D: Starting Mininet

#Starting Mininet
sduo mn --switch ovsk --topo tree,depth=2,fanout=8 --controller=remote, ip=127.0.0.1,port=6633

sudo mn: strting mininet

--switch ovsk: starting open virtual switch

--topo tree,depth=2,fanout=8: creating a tree network with depth of 2 and eight hosts from each switch

--controller remote, ip=127.0.0.1,port=6633: telling the network to look for a controller with the given IP address and port (running on loopback)