TAMPERE UNIVERSITY
OF APPLIED SCIENCES

Preventing DDOS Attacks From IOT

Rashid Ali Mirza

Master’s thesis
October 2018
Degree Programme in
Information Technology

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Master’s Degree Programme in Information Technology

ALI MIRZA, RASHID:
Preventing DDOS Attacks From IOT

Master’s thesis 38 pages, appendices 11 pages
October 2018

The IT world has witnessed and been a recipient of Distributed Denial of Service attacks. The attacks have grown over the past years, and it is expected that the carnage caused will get worse.

The purpose of this project was to demonstrate early DDoS attack detection. Though recent attacks have been instigated through IOT devices, the thesis will focus on DDoS detection, as IOT devices have only been a medium of the attack. It is important to counter act the vulnerability and this is the prime objective in this document.

A detection technique has been explored, using a test setup. Detection alone is not the objective, but necessary mitigation process that needs to be triggered has also been looked at.

Key words: ddos, entropy, iot, sdn

CONTENTS

1 INTRODUCTION 6
1.1. Understanding DDOS attacks 7
1.2. Types of DDOS attacks 8
1.3. Malware 8
1.4. Botnets 8
1.5. Purpose of this thesis 9
2 UNDERSTANDING THE ENEMY 10
2.1. The Technique 10
2.2. The Code synopsis 11
2.2.1 The Code structure 12
2.2.2 The Loader directory 12
2.2.3 The Mirai directory 12
3 SOFTWARE DEFINED NETWORKING 14
3.1. Traditional infrastructures 14
3.2. SDN basics 14
3.3. SDN security 15
4 TEST ENVIRONMENT 16
4.1. Intrusion detection 16
4.1.1 Entropy 17
4.1.2 Base entropy 18
4.1.3 Intrusion detection algorithm 18
4.1.4 Choosing the intrusion threshold 19
5 TEST SETUP 20
5.1. Virtual Box 20
5.2. Mininet network emulator 20
5.3. Environment 20
5.3.1 Topology 20
5.3.2 Layer 3 switch 21
5.3.3 POX controller 22
5.3.4 Layer 3 class 22
5.3.5 Switch class object 23
5.3.6 Layer 3 functions 23
5.3.7 ARPTableEntry class object 26
5.3.8 Entropy class object 26
5.3.9 Normal traffic 27
5.3.10 Class details 27

5.3.11 Entropy values 29
5.3.12 Attack behaviour 30
5.3.13 Class details 30
6 TEST RESULTS 34
7 CONCLUSION AND RECOMMENDATIONS 37
REFERENCES 38
APPENDICES 39
Appendix 1. Source Code Listing – traffic_simulator.py 39
Appendix 2. Source Code Listing – attack_simulator.py 40
Appendix 3. Source Code Listing – virtual_switch.py 41
Appendix 4. Source Code Listing – detection.py 48
Appendix 5. Source Code Listing – config.json 49

ABBREVIATIONS AND TERMS

ARP	Address Resolution Protocol
CNC	Command and Control Server
DOS	Denial of Service
DDOS	Distributed Denial of Service
IP	Internet Protocol
IT	Information Technology
IOT	Internet of Things
ISP	Internet Service Provider
LDAP	Lightweight Directory Access Protocol
MAC	Media Access Control
POX	open source development platform for Python-based SDN
SDN	Software Defined Network
TCP	Transmission Control Protocol
Tbps	Terra bits per second
TFTP	Trivial File Transfer Protocol
VM	Virtual Machine
Wireshark	Most popular network protocol analyzer tool

1 INTRODUCTION

This project attempts to highlight the possible ways hackers can use DDOS and use IOT devices as a medium to paralyze the network and cripple critical services. In addition, various methods of preventing such attacks have been stated, but only one will be in focus.

The recent spate of attacks on major and minor infrastructures has prompted the research into the root cause of the issue, and an attempt will be made to identify in detail the recent attacks. Once a common pattern is established in the way the attacks have been transgressing, all possible methods used till date, for mitigating DDOS attempts will be discussed. A solution will be proposed. The feasibility of the solution, the methodology applied, and the result of the solution will be discussed. If a common pattern cannot be established, then that would mean there are other vulnerabilities in the IOT framework, which hackers are exploiting, and therefore a high level of security feature will need to be looked at.

A detailed synopsis of the proposed solution will be presented, with full information on the setup, how the test was conducted, and the results of the tests. Any flaws in the test setup will be highlighted and discussed.

What is of concern is not only how to mitigate present attacks, but also how to mitigate and withstand future attacks. The DDOS attack landscape is growing exponentially and could reach tens of terra bits per second. Questions arise on how an infrastructure deal with that scenario can. What changes need to take place in the infrastructure to combat present and future impeding threats.
According to the security firm Correro, this new attack vector uses the LDAP, which if combined with an IoT botnet, could break records in DDoS power, and at the same time, break the backbone of any IT infrastructure.

1.1. Understanding DDOS attacks

It is vital that there is understanding in the difference between DDOS and DOS attacks. Please note that DDOS attacks are a concentrated effort of multiple devices to one destination attack, while on the contrary, DOS attacks are based on one source, which is generating the attack, and the endpoint of the attack is one destination.
A single IP attack falls into the category of a DoS attack, and a Multiple IP attack falls into the category of a DDoS attack (1).

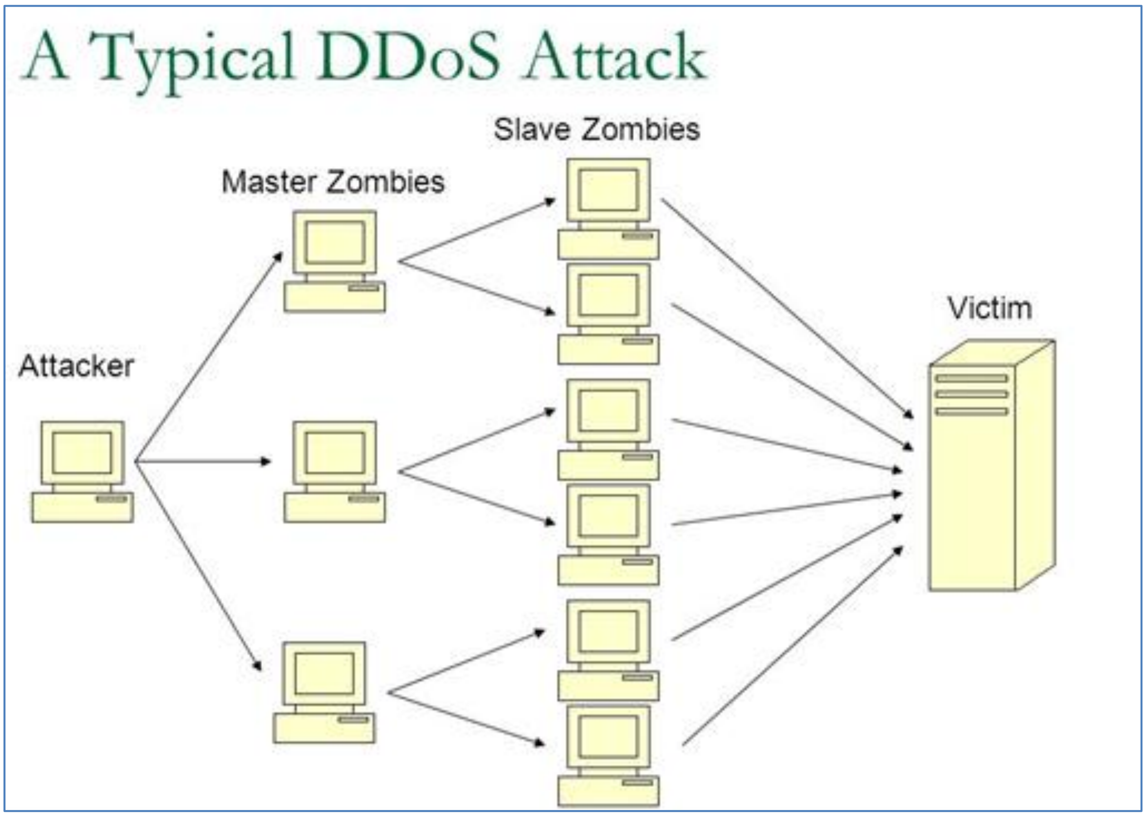


FIGURE 1. Typical DDOS attack

1.2. Types of DDOS attacks

Primarily, there are 3 types of DOS attacks.

- Application layer DDOS attacks: also known as layer 7 DDOS attacks. These types of attacks are generally difficult to detect. These types of attack target Windows, Apache, and openBSD systems, and attempt to crash the server
- Protocol DDOS attacks: attacks at a protocol level. Great examples would be Ping of Death, Synflood etc.
- Volume based DDOS attacks: attacks that include ICMP floods, UDP floods. The detection techniques can be statistical or machine learning. Some of the popular SDN based DDoS attack detection techniques are (2):
    - Entropy
    - Machine learning
    - Traffic pattern analysis
    - Connection rate

1.3. Malware

The word malware, as per definition is intrusive software that gains authorized access to a computer system.

1.4. Botnets

The word botnet has its origins in the words, robot and networks, where the ‘bots’ part comes from the word robot, and the ‘nets’ part comes from network. Many infected systems, that are inter connected over the network, are effectively botnets, where the network is the medium through which the malware can spread.

1.5. Purpose of this thesis

As one goes through the thesis, it will become evident the nature of the thesis, the topic, and its complexity.

I would like to inform that it is important to establish the direction of the attack, and the reader should not confuse this with cyber-attacks against IOT devices. Here we are talking about the combined effort of IOT devices, to establish a botnet attack on an infrastructure. The emphasis of the thesis is on DDOS attack detection, and the 2016 events of the Mirai attack triggered the author to highlight the contribution of IOT devices to the attack.

2 UNDERSTANDING THE ENEMY

21st October 2016 was a dark day for mankind in terms of cyber-attacks. Never was a malware attack of this magnitude experienced by the industry, and its affect was devastating in terms of the downtime for services for major sites, such as Guardian, Netflix, CNN and many more.

The Mirai botnet was deployed to “slaves” who group together and bombard a server with traffic, until the server dies out due to the heavy network load. In this case it was the servers at Dyn that were the target, and as some of you would know, Dyn provides a significant portion of the internet’s DNS feature, as it is an Internet Service Provider (ISP). Going back to the “slaves” these were none other than “Internet of Things” devices. These are devices that we on a day to day basis, interact with, but, put together a group of such devices, they can cause havoc (2).

The havoc must translate into a number, and in the case of the Mirai attack, it was of the magnitude of 1.2 Tbps, and this in layman terms means that there was 1.2 Terra bits EVERY second.

To understand the strength of the DDOS attack is not as simple as it seems. When the links get saturated it is difficult to tell if the limits of the network bandwidth are reached or the magnitude of the attack is the cause. For the victim, it is immaterial, since the network connection is saturated, and the IT infrastructure is already compromised.

2.1. The Technique

IOT devices sitting on the internet are like sitting ducks in the wild. The reason for this is that most of us do not change the default password, the one that comes with the device, when the product is released from the factory. Hackers exploit this loop hole, and gain access to the device. Once the device is compromised, it can be told or instructed to do anything, and hence my earlier use of the word “slave”.

As per (3), credentials for 15 percent of the worlds IOT devices are not changed, during the lifespan of the product. The problems don’t end here, because of sites such as shodan, and Zoomeye, as these sites are search engines for IOT devices, and can provide default credential data, that can be used by hackers, and port numbers, services running etc. Therefore, these sites act as dictionaries for hackers who are on the prowl for default credentials for IOT devices.

2.2. The Code synopsis

The source code for the Mirai botnet was made available for the public on GitHub soon after the 21st October 2016 attack. Now the reason for the public release remains unknown, but we can benefit from it by going through it, and possibly understanding of any variants of the Mirai botnet, that could come.

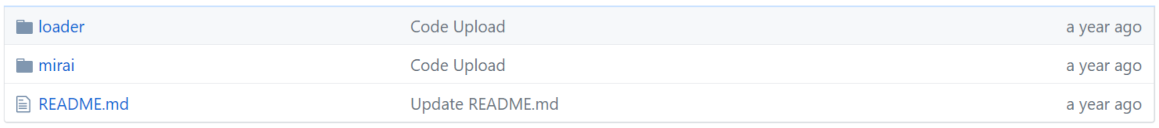
In general, the code is written in c/c++. It works by connecting to IOT devices using default credentials via the telnet protocol. The botnet would remain dormant, until the trigger command would come to it via a CNC application. The CNC code was also put onto GitHub, along with the Mirai source code. The CNC is written in Go programming language.

In brief the code is trying to:

- Scan vulnerable for devices, by using the factory credentials for devices
- For each device, the attacker will attempt to login and start a session to install a malware from a server, onto the device
- The malware is made active, and once that is successfully done, the device is compromised

2.2.1 The Code structure

Github has a copy of the Mirai source code. It is debatable if the code has been uploaded for research purposes or for academic purposes. There are different Github projects who have hosted the source code. Source code on its own can be harmless, but the way it is used, or the intentions of the person who uses it, are to be watched out for.



loader	Code Upload	a year ago
mirai	Code Upload	a year ago
README.md	Update README.md	a year ago

FIGURE 2. Mirai Code on Github

On Github, along with other folders, there is the loader and the Mirai folder.

2.2.2 The Loader directory

The loader folder contains code written in c programming language that allows the servers to be created and monitors the connections between the servers and the clients, that are the IOT devices. It uses TFTP and WGET to retrieve the malware, residing on a server, and push it across to the devices. Primarily, TFTP, also known as trivial FTP, comes as default for windows vista, and below operating systems. WGET on the other hand is a Windows and Linux tool, used for non-interactive downloads.

2.2.3 The Mirai directory

The *main.c* program is the starting point for the executable. The program disables the watchdog timer and prevents it from restarting. After that it makes calls to the fork() command, which creates processes for each module.

Here the crucial connection to the CNC application is established. There are 3 processes running besides the main process, the attack, killer, and scanner (4).

- The attack module parses the data received and launches the DDDOS attack. A defined number of attacks can be implemented.
- The Killer module terminates any processes that use ports meant for ssh, telnet and http, and tries to reserve for itself

- The Scanner module uses port 23 and randomly generated public IP’s to check for other IoT devices. The telnet credentials are obtained from a table containing default username and password pairs.

The CNC application is written in Go programming langauge. It connects to a database and creates two sockets on port 23 and 101.

3 SOFTWARE DEFINED NETWORKING

A network layer is divided into three different planes:

- Data plane: the data plane is where all the data packets are sent by the endpoints. For example, any forwarding packets, fragmentation and reassembly, and any replication for multicasting
- Control plane: this is where all activities that are necessary to perform data plane activities but do not involve the endpoint data packets. For example, routing table preparation, and from a security perspective the data packet handling policies
- Management plane: this is the plane where all activities related to provisioning and monitoring of the network takes place
- Service plane: this is the layer 7 application flow built on the foundation of the other layers (5)

In the simplest form, Software Defined Networking (SDN) is the separation of the network control plane from the forwarding plane and where the control plane controls several devices.

3.1. Traditional infrastructures

For any infrastructure, a firewall is used to secure incoming and outgoing network packets. They can monitor and control the flow of data and work based on predefined rules, also referred to as firewall policies, which can be inbound or outbound.

3.2. SDN basics

Network control is decoupled from forwarding and is directly programmable. Therefore, instead of each networking device forwarding packets to the next hop, the controls are centralized on SDN controllers. In the SDN architecture the control logic is decoupled from the switches and centralized to the network controller. Therefore, the switches are free from routing calculations and can be focused on packet forwarding (4).The defining characteristics of SDN are as follows:

- Central management
- Can be configured programmatically
- Easier and fast upgrades

3.3. SDN security

SDN Security should deliver network security by segregating the security control plan from the security processing and forwarding planes. The security aspect needs to be built into the architecture so that it can provide a service to protect the availability, integrity and privacy of all connected resources and information (5).

4 TEST ENVIRONMENT

To demonstrate the setup, the following have been configured:

- Mininet: in Mininet we can create various topologies of nodes and switches and can test out a virtual network by sending packets to each other. Switches in Mininet are software-based switches like Open vSwitch or the OpenFlow reference switch. Links are virtual Ethernet pairs, which live in the Linux kernel and connect the emulated switches together
- All flowing data can be analysed using a packet capture tool such as Wireshark
- Scapy is a powerful interactive packet manipulation program. It can forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more

4.1. Intrusion detection

As per (6), detection mechanisms are as follows:

- Entropy
- Machine learning
- Traffic Pattern Analysis
- Connection rate
- SNORT and OpenFlow integrated

As per (4) there are 6 traffic flow features that can be used:

- Average of packets per flow
- Average bytes per flow
- Average of duration per flow
- Percentage of pair-flows
- Growth of single flows
- Growth of different ports

Using combination of Entropy with average packet flow should provide a mechanism to detect thresholds.

4.1.1 Entropy

Entropy (9) can be defined as a number that gives you an idea of *how random an outcome will be based on the probability values of each of the possible outcomes in a situation*.
Can also be defined as the uncertainty in a random variable.

$$H(X) = -\sum_{i=0}^{n-1} p_i \, {}_{\log_2 \times} p_i$$

Or

$$H(X) = \sum_{i=0}^{n-1} p_i \, {}_{\log_2 \times} \frac{1}{p_i}$$

Entropy will be maximum when all outcomes are equally likely. Any time we move away from equally likely outcomes then then the entropy must go down. Therefore, the entropy of variable X is the sum over all possible outcomes of i of X, of the product of the probability of outcome i times the log of the inverse of the probability of i. The variable n is the number of symbols that the information source can give us. The units of information entropy are bits.
If the values of X where Bernoulli or binary random variable, then H(X) will be a concave output as shown in figure 3.

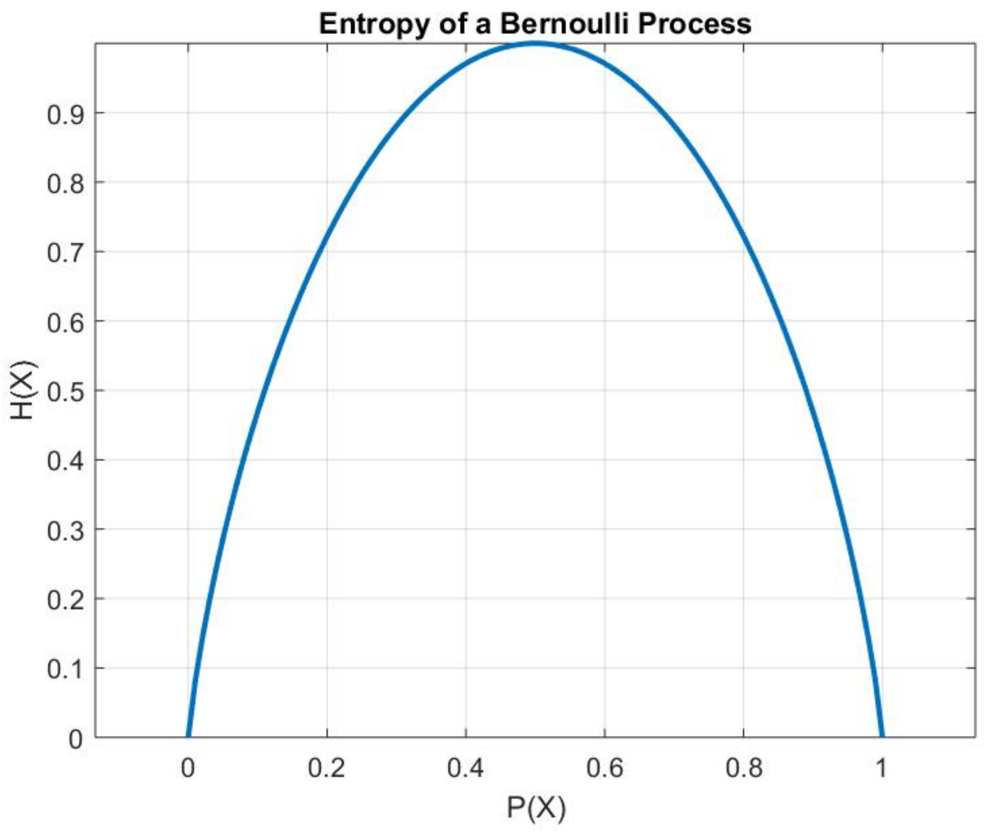


FIGURE 3. Bernoulli Process

4.1.2 Base entropy

Base Entropy is defined as the average entropy for the target IP address when there is no DDOS attack in place. The purpose of this is to define a normal scenario. Establishing the base entropy will define a point of reference.

4.1.3 Intrusion detection algorithm

Detecting DDoS attacks on a server in an SDN network can be done by running a light weight application on the SDN controller. The application counts the number of packet flooding into the SDN controller and obtains the packet rate and the occurrences of data packets according the destination IP address, TCP port and source IP addresses and calculates the entropy of the destination IP address, TCP port and source IP address (6).

4.1.4 Choosing the intrusion threshold

The threshold is dependent on the rate of normal traffic, or traffic that is not classified as being as attack traffic. To answer questions on how strong the attack is, a simple ratio of attack Packets P_a to the number of normal traffic packets P_n is equal to the rate R:

$$R = \frac{P_a}{P_n} \times 100\%$$

5 TEST SETUP

The below are the pre-requisites for the environment setup

5.1. Virtual Box

The latest version of the Virtual Box was downloaded for the windows environment.

5.2. Mininet network emulator

After a suitable version of Virtual box was installed, an appropriate Mininet VM image was downloaded. Using Mininet it is possible to create realistic virtual networks. Mininet allows experimenting with OpenFlow and Software Defined Networks.

5.3. Environment

A virtual network topology has been implemented through Mininet with the purpose of simulating a real environment.

5.3.1 Topology

Mininet allows custom topologies to be created. Due to the power of virtualization, a single system can be made to look like a network. It is required to setup a network for the test, and below id the command used to create the network, from inside the Mininet environment.

```
# sudo mn --switch ovsk --topo tree,depth=2,fanout=8 --controller=remote,IP=127.0.0.1,port=6633
```

This command creates a network and adds controllers, hosts, switches and links between them. 127.0.0.1 is the loopback address.

A remote POX controller will be connected to the open flow switch. The command will create a topology of 64 hosts connected to 9 switches. Each of the 9 switches are created on port 6633 and have an IP address that matches the loopback IP address specified in the statement just entered.

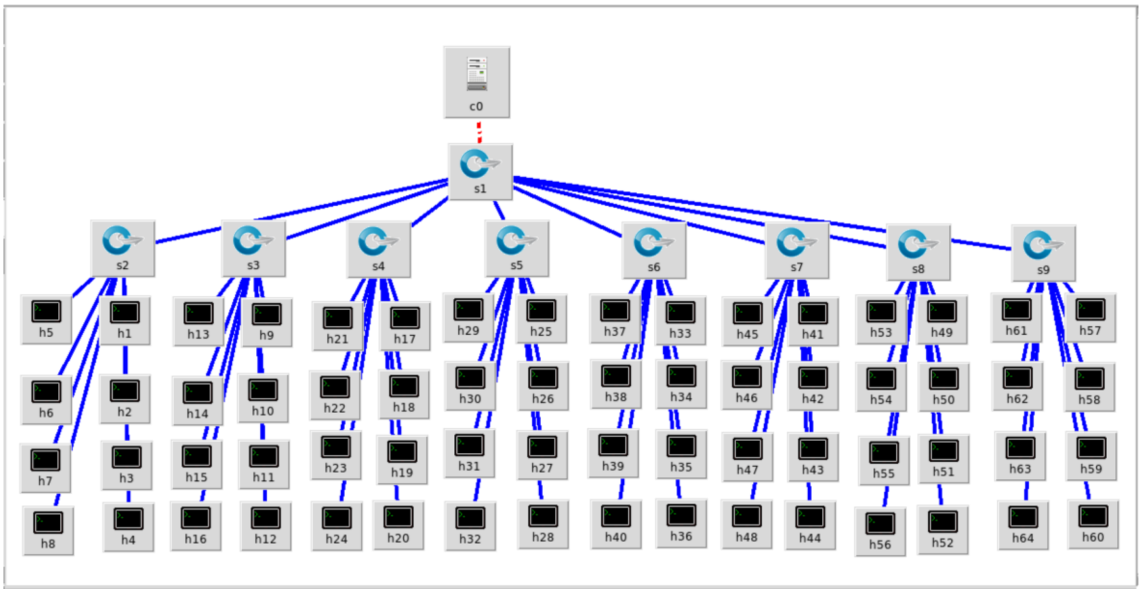


FIGURE 4. Experiment Network with 9 switches and 64 hosts

5.3.2 Layer 3 switch

In an SDN architecture, a switch performs the following functions:

- The switch encapsulates and forwards the first packet of a flow to an SDN controller, enabling the controller to decide whether the flow should be added to the switch flow table.
- The switch forwards incoming packets out the appropriate port based on the flow table. The flow table may include priority information dictated by the controller.
- The switch can drop packets on a flow, temporarily or permanently, as dictated by the controller. Packet dropping can be used for security purposes, curbing DDoS attacks

The switch can examine each packet and learn the source port mapping. Thereafter the source MAC address will be associated with the port. If the destination of the packet is already associated with some port, the packet will be sent to the given port else it will be flooded on all ports of the switch. Each open flow switch is identified by data path id. The Datapath is a 64-bit field that can be thought of as being analogous to an Ethernet Switch bridge MAC. It is a unique identifier for the specific packet processing pipe line being managed. Once a packet arrives to switch it, switch can send packets to single or multiple ports, drop the packet or modify the packet header.

5.3.3 POX controller

POX provides a framework to communicate with SDN switches. Here the OpenFlow protocol has been used. POX needs to be started from the command line, and a layer 3 switch was implemented in an SDN network.

5.3.4 Layer 3 class

The layer 3 switch application was created which uses two classes:

- *ARPTableEntry* class
- *Switch* class

Certain declarations have been done, and they are as follows:

- *MAX_BUFFER_TIME*, this flag has been set to 5, and indicates for how long each packet will be retained for in the lost buffer. Lost buffer is a container which contains packets for which the switch does not know where to send this packet. In simple words the ARP table does not have mapping of destination IP with MAC address and port, so in that case the switch will store the packet in the buffer. In addition, the *MAX_BUFFER_TIME* will ensure that every packet in buffer should retain in the buffer memory for 5 minutes only. After the buffer time has elapsed for any packet, it should be deleted from buffer
- *MAX_BUFFERED_PER_IP*, this flag has been set to 5, and indicate how many packets per IP will be saved in the buffer. Let us suppose one packet arrives for which we don’t know its destination since it’s ARP Entry is empty. Then that packet will be saved to the buffer with its IP information. Let us then suppose 4 more packets comes for this IP address, 4 more packets will be saved for this IP in buffer memory. As soon as another packet come for this IP, switch will not store it, because the switch already added 5 packets for this IP address. So, the value of *MAX_BUFFERED_PER_IP* indicates number of packets to be stored for this IP.

5.3.5 Switch class object

The layer 3 switch class has the following main class variables:

- *arp_for_unknowns*
- *outstanding_arps*
- *lost_buffers*
- *arpTable*
    - It is used as lookup table of MAC Address of host's and ports of the switch to which each host is connected. It is a dictionary which records in [DpID][SourceIPAddress] = (Port, Host MAC Address)
- *expire_timer*:

The layer 3 switch maintains look up table for MAC address and port. Once the switch receives packets it checks its lookup table for the MAC address and obtains the port number so that it can send packets to the destination.

5.3.6 Layer 3 functions

_handle_expiration:

- This is function is a delegate, which is called by the timer every 5 minutes. This function keeps a check on the lost_buffers member variable. If any item expires, it removes item from lost_buffers.

_send_lost_buffers:

- This function receives 'dpid', 'ipaddr', 'macaddr', 'port' as parameters
- It searches each item in lost buffer with key (dpid, ipaddr)
- If it found the packet in the buffer it re-sends it to the port and MAC address of the incoming packet
- let us suppose the switch receives packets and it does not know the destination port, because it searches for destination MAC and port in arpTable using dpid. Then in that case it stores the packet in lost buffer for some time. When the destination host sends packet to the switch to send to another host and while sending that packet, the switch looks up the lost buffer, which holds the old packet for which the destination was unknown. Now in this second cycle it is possible

we already learnt the address of that earlier unknown destination. In this case the switch will pick that packet from the buffer, delete that buffer, and send it to the source again, so the source will receive the earlier buffered packet which was not delivered to it earlier, due to unknown address.

_handle_incoming_packets:

- This function is the most important function which performs all the switching functionality. It processes in coming packets
- Whenever this function fires up, it receives dpid, in port, and packet
- It searches dpid in lookup arpTable. If it is not able to find dpid it creates a new empty dictionary for that data path id. It initializes dpid entry with dummy values
- It checks packet type. If the packet type is IPv4, it proceeds with the following. It uses the Entropy class object to know if source is sending packets for the destination repeatedly. It will calculate entropy value and if the value is below certain threshold then this indicates that the source is attacking host / device on the network. In that case switch port will be blocked
- It then calls send lost buffer to send packets from the buffer
- It will learn and update entry in lookup table arpTable, in case the source MAC address is not found it will add new entry of source. The entry will be for the port from where packets comes from with the source MAC Address. Now the switch learns new entry and knows that in case again the packet comes for this source as destination then it can look up destination MAC address from packet header and will search that MAC address in arpTable and send the packet to corresponding port
- After receiving and adding source IP address, source MAC address, port information to arpTable, the switch will try to obtain the destination address. It will get the destination IP Address and do a lookup operation on arpTable to get MAC address and port. In case it finds the destination address it simply sends the packet to that destination port
- In case switch is unable to find destination, MAC address and port based on IP Address from packet header then it will check class member *arp_for_unknowns* if it is true. Then it will proceed as following
    - In case switch unable to determine packet destination then it will first save the packet to its lost buffer, then broadcast packet to all of its ports.

The host which matches IP with destination IP will pick that packet and remaining hosts will discard the packet. The receiver again re-sends the packet back to switch, in this way the switch will learn the new entry of the source host

blockPort:

- It is used to block port, due to excess send packets requests to this port
- When the entropy value is below threshold (in our case 0.5) , then this function is called by *_handle_incoming_packets*
- It sets *set_Timer* class variable to true
- Once *set_Timer* is true *_port_blocker_callback* handler will be able to block that port

This function only gets invoked once switch detects some unusual activity (it keeps calculating entropy of network for every packet that arrives). Once switch detects there is a malicious user attacking, then it calls *block_port* function, which start to count number of packets coming in from switch ports. Let us say malicious user is targeting specific device from other random ports/devices then *block_port* function keep counts of every in-network packet, and it makes timer function start flag true so that now switch will start timer and start to observe ports counts. As soon as port count reaches 50 then it determines that this incoming port is suspicious, and it will do appropriate action on the port.

_port_blocker_callback function:

- It is timer handler call back function.
- Once it is detected that destination have entropy less than 0.5 then it is subscribed to fire up after every 2 minutes
- For every port which received 50 packets will be block in this function

A conditional check has been setup to indicate if the network entropy drops below a certain threshold. If it is below then then packet handler will trigger a monitor to start counting packets going to the out port. As soon as the port count hits 50 or more, it will send packet to inform the port to drop all incoming packets.

5.3.7 ARPTableEntry class object

This class holds three properties:

- Port number, for sending incoming packet. Port number can be any one of the ports in the switch
- MAC Address, is a MAC address of the source host
- Time out

The switch class keeps the mapping table ARP, which contains collection of these 3 objects

5.3.8 Entropy class object

This class is used to keep the list of all destination IP addresses for which the switch sends packets. Each destination IP will be added to this IP address list.
The class is used to monitor every packet and extract destination IP addresses.
It tracks the destination IP for which packets are being sent out. For each packet it calculates an entropy value for requested destination IP addresses.
We set the threshold for the entropy value, in case the calculated entropy value is below the threshold, it will block the port which corresponds to the destination IP address.

In the entropy class the total number of packets travel through switch is equal to 50, then this entropy class will search the current packet destination IP in list of destination IP addresses maintain in Entropy, after calculating the count of IP in the from the list, it then calculates entropy for that destination IP address. In case that entropy value is lower than 0.5 threshold value, then it indicates that sender keeps on sending packets to this IP and due to security risk, it blocks that port.

As traffic increases and more packet flows through network (ideally once number of travelled packets through this switch reach up to 50 and more) this class tries to calculate entropy for each incoming packet.

A conditional check has been done to determine if 80 distinct IP’s exist for the packets outbound from the switch. In case we have a network of 10 hosts, then we have a maximum of 10 destinations, in that case entropy class will never calculates entropy of in-

coming packets. If one host is victim of malicious user sending large packets, then entropy value is always = 1 which will never trigger blocking of port.
To summarize, this class is used to block some specific port if large number of packets are intended for. This observation of destination IP is activated as soon as 50 packets have flown through this switch.

5.3.9 Normal traffic

The traffic simulator module simulates traffic flow over the network. It is executed in the following syntax:

```
# python ./traffic_simulator.py
```

The destination IP addresses must be defined in the config.json file. The destinationLastOctetRange parameter in the config.json file define the start and end for the last octet.
The '2' and '65' are the arguments passed to the generate_destination_ip function that generates the destination IP addresses.
For the destination IP the user will have to specify the range of the IP addresses. For example, if the user were to send the range from 1 to 256, then IP addresses will be generated in the format 10.0.0.D, where D would lie in between the start and end specified as parameters. The source IP address will be generated randomly by a function. Once the source and destination IP are defined, the packets will be sent over the network using the Scapy python library.

5.3.10 Class details

The class contains the following functions:

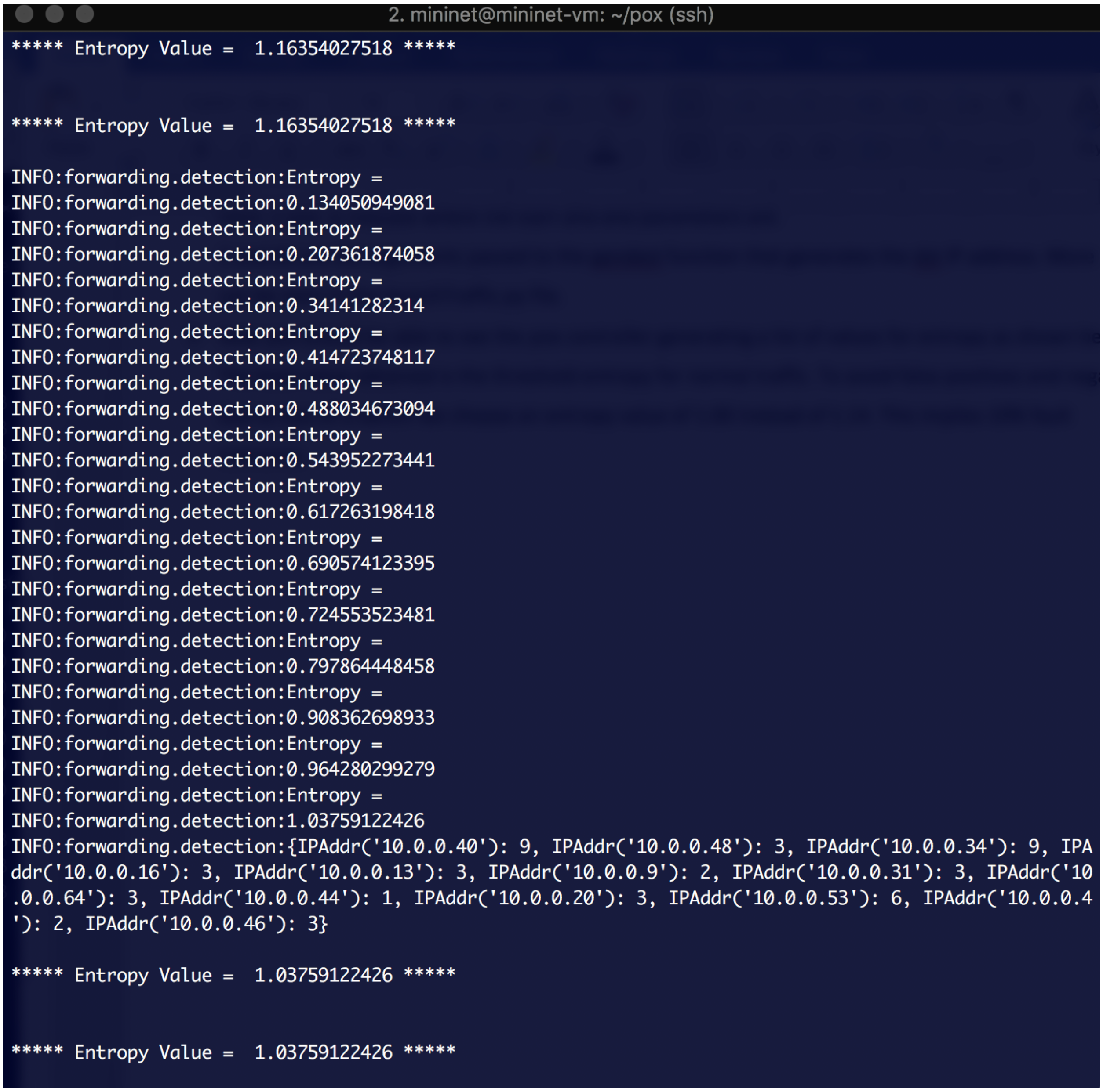
- *generate_source_ip*:
  It generates source IP addresses. The IP address first segment range from 1 to 256 where first segment should be a number except:
    - 10
    - 127
    - 254
    - 255
    - 1
    - 2

    - 169
    - 172
    - 192

  The remaining 3 octets will be from 1 to 256.

- *generate_destination_ip*:
  This function generates destination IP address using range values passed through the *config.json* file
- *simulate_network_traffic*:
  It creates 1000 random packets so that they can be sent to the destination IP address. It calls above methods to generate source IP address and destination IP address where destination IP address will be created using a range

Once the script is executed, we should be able to see the pox controller generating a list of values for entropy as shown below. The least value obtained is the threshold entropy for normal traffic. To avoid false positives and negatives due to loss of a switch we choose an entropy value of 1.00 instead of 1.14. This implies 10% fault tolerance. The threshold entropy can also be referred as the base entropy, as discussed in section 4.1.1.1. A threshold for entropy was chosen and lower values will be considered as attacks.



```
2. mininet@mininet-vm: ~/pox (ssh)
***** Entropy Value =  1.16354027518 *****


***** Entropy Value =  1.16354027518 *****

INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.134050949081
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.207361874058
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.34141282314
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.414723748117
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.488034673094
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.543952273441
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.617263198418
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.690574123395
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.724553523481
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.797864448458
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.908362698933
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.964280299279
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:1.03759122426
INFO:forwarding.detection:{IPAddr('10.0.0.40'): 9, IPAddr('10.0.0.48'): 3, IPAddr('10.0.0.34'): 9, IPA
ddr('10.0.0.16'): 3, IPAddr('10.0.0.13'): 3, IPAddr('10.0.0.9'): 2, IPAddr('10.0.0.31'): 3, IPAddr('10
.0.0.64'): 3, IPAddr('10.0.0.44'): 1, IPAddr('10.0.0.20'): 3, IPAddr('10.0.0.53'): 6, IPAddr('10.0.0.4
'): 2, IPAddr('10.0.0.46'): 3}

***** Entropy Value =  1.03759122426 *****


***** Entropy Value =  1.03759122426 *****
```

FIGURE 5. Normal traffic entropy

5.3.11 Entropy values

Entropy is calculated based on a list of IP addresses that packets are being sent to. All the destination IP addresses have the same first three octets of 10.0.0. The fourth octet is randomized in the traffic_simulator.py file. These IP addresses correspond to the 64 hosts in our Mininet topology. For every 50 packets sent, a dictionary is maintained. The dictionary comprises the destination IP addresses and the number of times a packet has been sent to that address. Finally, entropy is calculated as follows:

Entropy = - (count/50) * log10(count/50)

Where 50 represents the number of packets counted in the dictionary.

5.3.12 Attack behaviour

The Network Attack Simulator sends bulk packets to devices on the network. It is executed using the following syntax:

```
# python ./attack_simulator.py 10.0.0.64
```

Typically, we can configure the destination IP through command line arguments. The main purpose of this module is to simulate how a malicious user tries to attack a resource over the network.

The class creates packet using Scapy. It creates 2500 packets with random source IP's and single destination to which it can send packets.

5.3.13 Class details

It contains the following functions

- *generate_random_ip*:

  It generates IP address of source. The IP address first segment range from 1 to 256 where first segment should be number except

  - 10
  - 127
  - 254
  - 255
  - 1
  - 2
  - 169
  - 172
  - 192

The 4th octet will be from 1 to 256

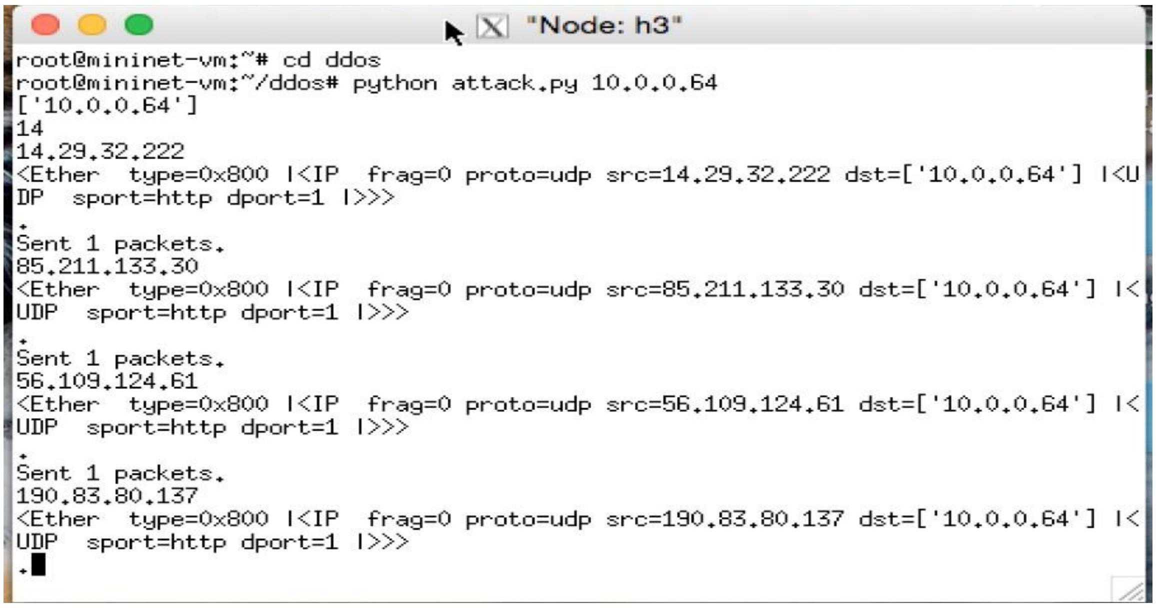
- *send_packets*:

It creates 500 packets and sent to specified IP address. It again uses the Scapy library

- *main*:

  This method gets called on class start-up. It loads the configurations from the json file. It simply loops five time and calls the *send_packets* function

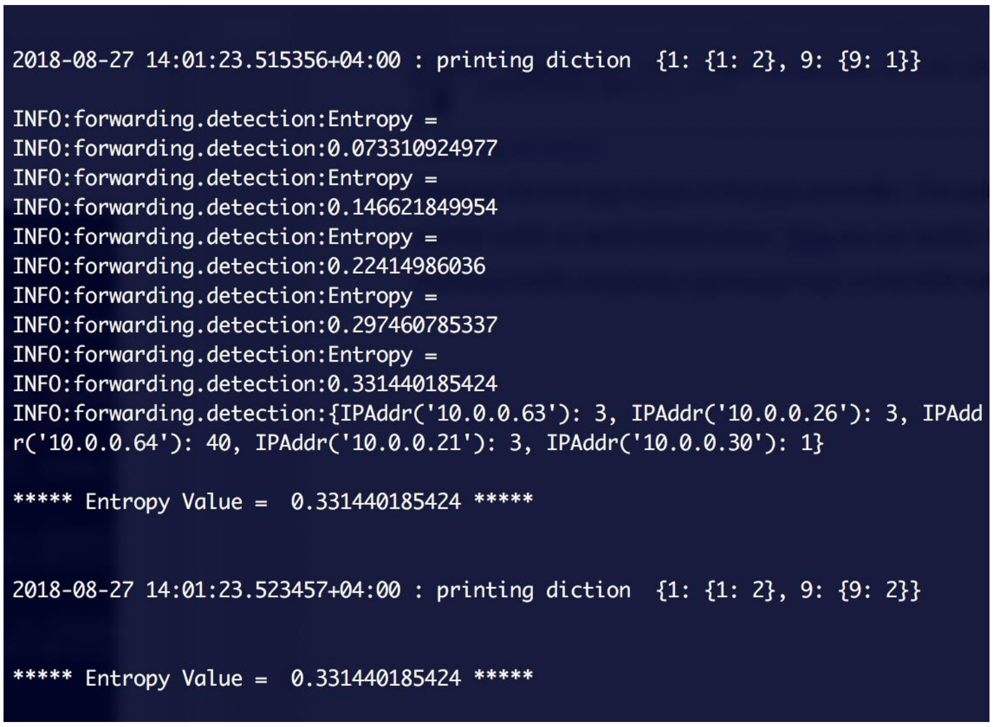
The parameter targetIP being passed through the json configuration file to attack_simulator.py matches the IP address of the 64th host. Attack_simulator.py sends network packets from random source IP addresses to a single host. This host is determined by the argument passed. Thus, attack packets are sent specifically to host 64.



```
root@mininet-vm:~# cd ddos
root@mininet-vm:~/ddos# python attack.py 10.0.0.64
['10.0.0.64']
14
14.29.32.222
<Ether  type=0x800 |<IP  frag=0 proto=udp src=14.29.32.222 dst=['10.0.0.64'] |<U
DP  sport=http dport=1 |>>>
.
Sent 1 packets.
85.211.133.30
<Ether  type=0x800 |<IP  frag=0 proto=udp src=85.211.133.30 dst=['10.0.0.64'] |<
UDP  sport=http dport=1 |>>>
.
Sent 1 packets.
56.109.124.61
<Ether  type=0x800 |<IP  frag=0 proto=udp src=56.109.124.61 dst=['10.0.0.64'] |<
UDP  sport=http dport=1 |>>>
.
Sent 1 packets.
190.83.80.137
<Ether  type=0x800 |<IP  frag=0 proto=udp src=190.83.80.137 dst=['10.0.0.64'] |<
UDP  sport=http dport=1 |>>>
.
```

FIGURE 6. Attack packets sent to the 64th host

When attack_simulator.py is run, we see that the IP list dictionary generated has dramatic changes in its count values. Host 10.0.0.64 now has extremely high values of count as attack packets are being sent to it. Thus, every 50 packets, host 64 has received several packets whereas the other hosts have received far fewer.



```
2018-08-27 14:01:23.515356+04:00 : printing diction  {1: {1: 2}, 9: {9: 1}}

INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.073310924977
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.146621849954
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.22414986036
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.297460785337
INFO:forwarding.detection:Entropy =
INFO:forwarding.detection:0.331440185424
INFO:forwarding.detection:{IPAddr('10.0.0.63'): 3, IPAddr('10.0.0.26'): 3, IPAdd
r('10.0.0.64'): 40, IPAddr('10.0.0.21'): 3, IPAddr('10.0.0.30'): 1}

***** Entropy Value =  0.331440185424 *****


2018-08-27 14:01:23.523457+04:00 : printing diction  {1: {1: 2}, 9: {9: 2}}


***** Entropy Value =  0.331440185424 *****
```

FIGURE 7. Count for Host 10.0.0.64

Host 10.0.0.64 has a count of 40 whereas the other hosts have far fewer.

When one host receives many more packets than the other host, the value of entropy tends to decrease. Thus, we see the entropy fall below the earlier threshold of 1.

The decrease in entropy value serves as a signal that network traffic is irregular.

In Figure 7 above another dictionary is being displayed. This dictionary is only created when the entropy values decreases below 0.5.

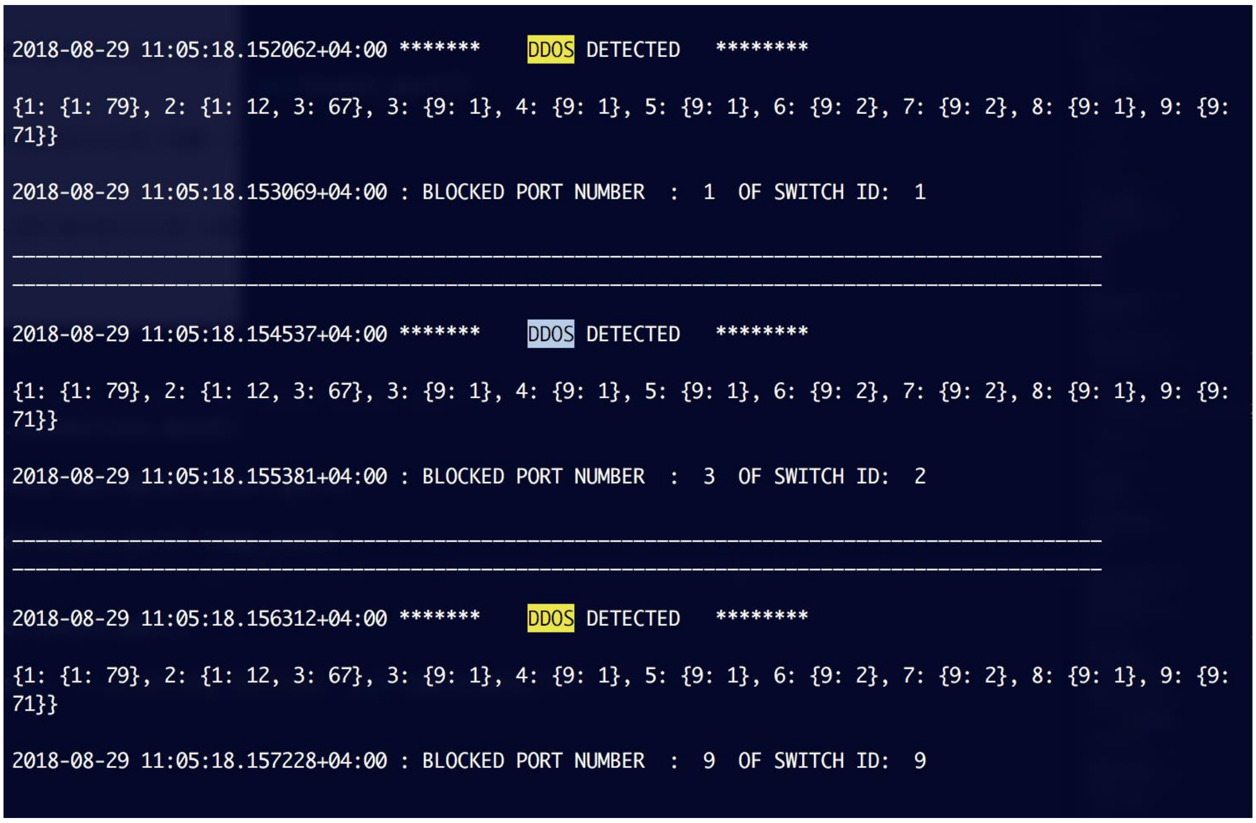
This is a dictionary that contains dictionary values within it.

The key values correspond to the switches in our Mininet topology.

The value dictionaries have keys that correspond to the port in the relevant switch. The final value shows us the number of times those ports have been activated.

This serves as another good measure for irregular network traffic and helps us detect a DDOS attack.

When the count for any port of any switch exceeds 50, a warning is displayed in the pox controller.



```
2018-08-29 11:05:18.152062+04:00 *******    DDOS DETECTED    ********

{1: {1: 79}, 2: {1: 12, 3: 67}, 3: {9: 1}, 4: {9: 1}, 5: {9: 1}, 6: {9: 2}, 7: {9: 2}, 8: {9: 1}, 9: {9:
71}}

2018-08-29 11:05:18.153069+04:00 : BLOCKED PORT NUMBER  :  1  OF SWITCH ID:  1

------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------

2018-08-29 11:05:18.154537+04:00 *******    DDOS DETECTED    ********

{1: {1: 79}, 2: {1: 12, 3: 67}, 3: {9: 1}, 4: {9: 1}, 5: {9: 1}, 6: {9: 2}, 7: {9: 2}, 8: {9: 1}, 9: {9:
71}}

2018-08-29 11:05:18.155381+04:00 : BLOCKED PORT NUMBER  :  3  OF SWITCH ID:  2

------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------

2018-08-29 11:05:18.156312+04:00 *******    DDOS DETECTED    ********

{1: {1: 79}, 2: {1: 12, 3: 67}, 3: {9: 1}, 4: {9: 1}, 5: {9: 1}, 6: {9: 2}, 7: {9: 2}, 8: {9: 1}, 9: {9:
71}}

2018-08-29 11:05:18.157228+04:00 : BLOCKED PORT NUMBER  :  9  OF SWITCH ID:  9
```

FIGURE 8. DDOS attack detected

Look at figure 8. We see that port number 1 of switch 1 has been blocked to prevent the attack from causing more damage. If you look at the dictionary you will notice that port 1 of switch 1 has been activated 79 times.

Similar port 3 of switch 2 has been activated 67 times while port 9 of switch 9 has been activated 71 times.

These are direct indicators of an attack taking place and the program blocks these ports once it realizes that the network is under attack.

6 TEST RESULTS

To visualize the packets, it was decided to use Wireshark.

Prior to any traffic on the loopback, using Wireshark, the loop back interface was monitored to obtain the as-is status.

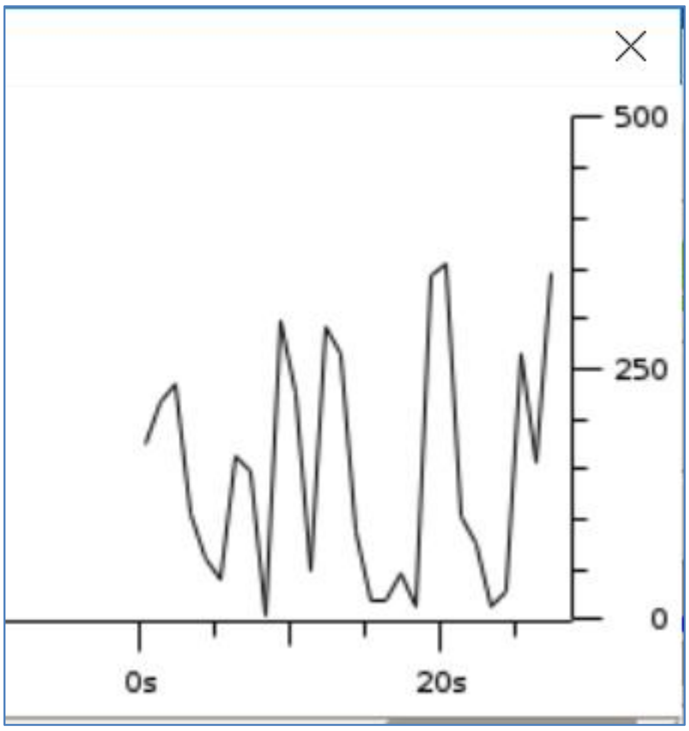


FIGURE 10. No network traffic scenario

Then, from one of the host windows, the traffic was generated, and again the data was captured using Wireshark.

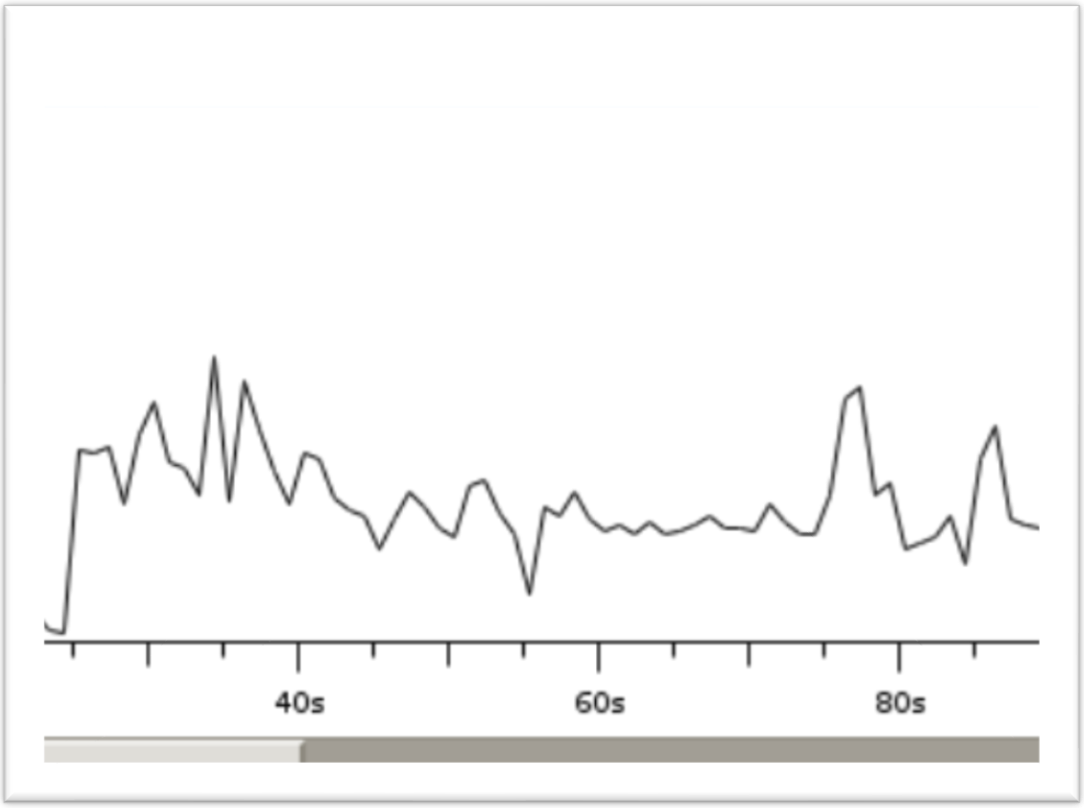


FIGURE 10. Normal traffic scenario

The attack traffic was generated, and in a similar manner to before the traffic was captured.

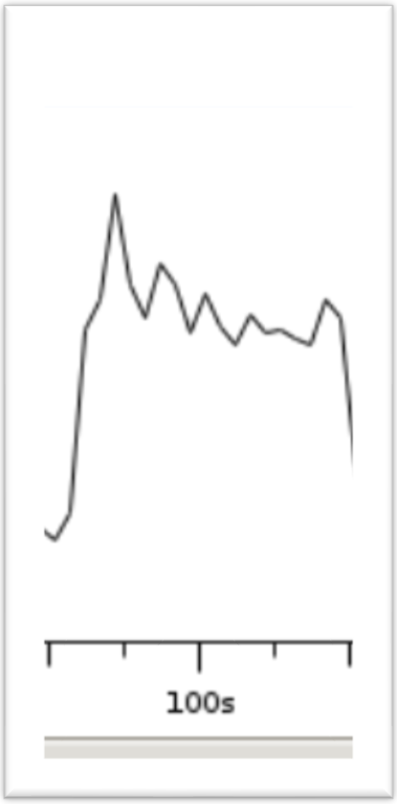


FIGURE 11. Attack traffic scenario

Putting all of it together, below is a how the network traffic looked over a period. Over here only one host was used to generate the attack traffic. Had more than one host been used the amplitude on the Y-axis would have been higher:

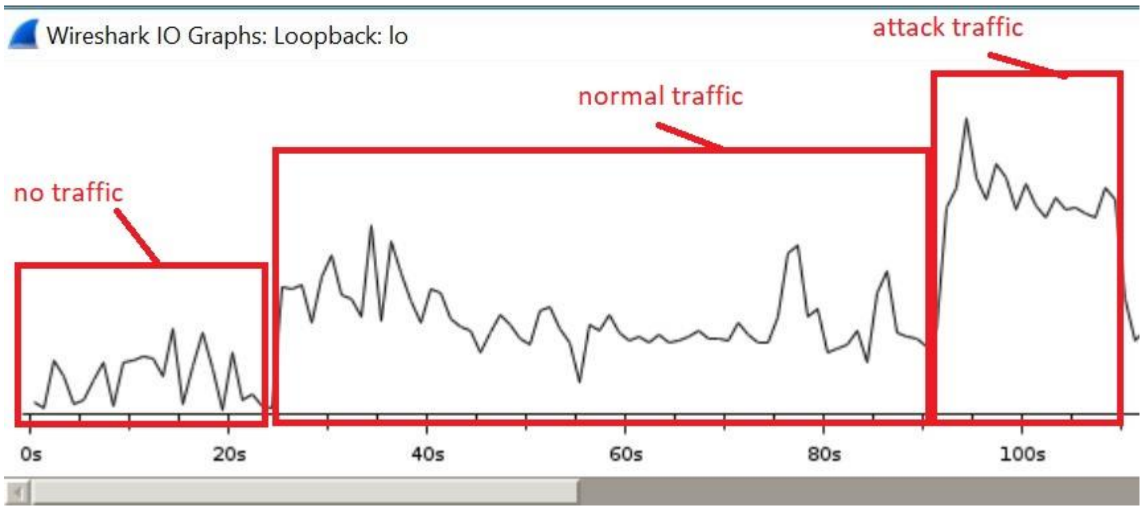


FIGURE 12. Traffic pattern combined

Where the x axis is the tick interval in seconds, and the Y-axis are the packets/Tick.

7 CONCLUSION AND RECOMMENDATIONS

Ensuring the network architecture is secure from DDoS attack is essential. We do not want experiences such as the Mirai attack to make us learn the lessons the hard way.

Using SDN it has been demonstrated that it is possible to create an early detection mechanism. It was demonstrated that within the first 50 packets of traffic, a malicious attack can be detected. By using entropy as a detection method, it is possible to detect attacks regardless of the landscape of the hosts.

The drawback of this method is that it could not separate malicious from legitimate packets.

The centralized control in SDN provides a single point of failure in a network and if this is overlooked, it can be detrimental to the setup. One solution would be to have distributed SDN architectures.

Having visualization tools to monitor the flow rate at the switch layer will be helpful.

It is suggested to for future work to explore DDOS detection using machine learning algorithms. A neural network can learn patterns from sequences of network traffic, accordingly 'react' to malicious traffic and take necessary action.

It is also recommended to use GNS tool. Using this tool, complex network topologies can be constructed. Simulation of various platforms can be done with ease. The simulated network can be connected to the real world.

REFERENCES

1. **[Online] https://www.packtpub.com/books/content/pentesting-using-python.**

2. [Online] https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

3. [Online] https://www.bleepingcomputer.com/news/security/15-percent-of-all-iot-device-owners-dont-change-default-passwords/.

4. Hamdija Sianovic, Sasa Mrdovic. *Analysis of Mirai Malicious Software.*

5. *Defending Against New-Flow Attack in SDN-Based Internet of Things.* s.l. : TONG XU, DEYUN GAO, PING DONG, HONGKE ZHANG, CHUAN HENG FOH, HAN-CHIEH CHAO.

6. [Online] https://opensourceforu.com/2016/07/implementing-a-software-defined-network-sdn-based-firewall/.

7. Narmeen Zakaria Bawany, Jawwad A. Shamsi, Khaled Salah. *DDoS Attack Detection and Mitigation Using SDN: Methods, Practices, and Solutions.*

8. *Detection of DDoS Attacks Against Wireless SDN Controllers Based on the Fuzzy Synthetic Evaluation Decision-making Model.* QIAO YAN, QINGXIANG GONG and FANG-AN DENG.

9. [Online] https://www.sdxcentral.com/security/definitions/security-challenges-sdn-software-defined-networks/.

APPENDICES

Appendix 1. Source Code Listing – traffic_simulator.py

```python
import sys
import getopt
import time
from os import popen
import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import sendp, IP, UDP, Ether, TCP
from random import randrange
import json

config = None

def generate_source_ip():
    not_valid = config["trafficConfig"]["notValidFirstOctet"]
    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

def generate_destination_ip():
    first = 10
    second =0; third =0;

    range_str = config["trafficConfig"]["destinationLastOctetRange"]
    range_str = str(range_str)
    start , end = range_str.split(':')
    start = int(start)
    end = int(end)

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

def simulate_network_traffic():
    iterations = config["trafficConfig"]["networkTrafficSize"]
    interface = popen('ifconfig | awk \'/eth0/ {print $1}\'').read()
    for i in range(iterations):
        packets = Ether()/IP(dst=generate_destination_ip(),src=generate_source_ip())/UDP(dport=80,sport=2)
        print(repr(packets))

        if interface != '':
            sendp(packets,iface=interface.rstrip(),inter=0.1)
        else:
            sendp(packets,inter=0.1)

def load_configurations():
    with open('config.json') as j_file:
        config = json.load(j_file)
        return config


if __name__ == '__main__':
  config = load_configurations()
  simulate_network_traffic()
```

Appendix 2. Source Code Listing – attack_simulator.py

```python
import sys
import time
from os import popen
import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import sendp, IP, UDP, Ether, TCP
from random import randrange
import time
import json

config = None

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

def trigger_ddos():
  iterations = config["attackConfig"]["ddosIteration"]
  for i in range (1,iterations):
    send_packets()
    time.sleep (10)

def send_packets():
  dstIP = config["attackConfig"]["targetIP"]
  print(dstIP)
  src_port = config["attackConfig"]["srcPort"]
  dst_port = config["attackConfig"]["targetPort"]
  interface = popen('ifconfig | awk \'/eth0/ {print $1}\'').read()

  attack_size = config["attackConfig"]["ddosAttackMagnitude"]

  for i in range(0,attack_size):
    packets = Ether()/IP(dst=dstIP,src=generate_random_ip())/UDP(dport=dst_port,sport=src_port)
    print(repr(packets))

    if interface != '':
      sendp( packets,iface=interface.rstrip(),inter=0.025)
    else:
      sendp(packets,inter=0.025)

def load_configurations():
      with open('config.json') as json_file:
        json_config = json.load(json_file)
        return json_config


if __name__=="__main__":
  config = load_configurations()
  trigger_ddos()
```

Appendix 3. Source Code Listing – virtual_switch.py

```
# Copyright 2017-2018 Rashid Ali Mirza
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import datetime
from pox.core import core
import pox

from pox.lib.packet.ethernet import ethernet, ETHER_BROADCAST
from pox.lib.packet.ipv4 import ipv4
from pox.lib.packet.arp import arp
from pox.lib.addresses import IPAddr, EthAddr
from pox.lib.util import str_to_bool, dpid_to_str
from pox.lib.recoco import Timer

import pox.openflow.libopenflow_01 as of

from pox.lib.revent import *
import itertools
import time
#editing

from detection import Entropy
diction = {}
ent_obj = Entropy()
set_Timer = False
defendDDOS=False
#blockPort=""
#end of editing
log = core.getLogger()
# Timeout for flows
FLOW_IDLE_TIMEOUT = 10

# Timeout for ARP entries
ARP_TIMEOUT = 60 * 2

# Maximum number of packet to buffer on a switch for an unknown IP
MAX_BUFFERED_PER_IP = 5

# Maximum time to hang on to a buffer for an unknown IP in seconds
MAX_BUFFER_TIME = 5


class ArpTableEntry (object):
  """
  Not strictly an ARP entry.
  We use the port to determine which port to forward traffic out of.
  We use the MAC to answer ARP replies.
  We use the timeout so that if an entry is older than ARP_TIMEOUT, we
   flood the ARP request rather than try to answer it ourselves.
  """
```

```python
 def __init__ (self, port, mac):
  self.timeout = time.time() + ARP_TIMEOUT
  self.port = port
  self.mac = mac

 def __eq__ (self, other):
  if type(other) == tuple:
   return (self.port,self.mac)==other
  else:
   return (self.port,self.mac)==(other.port,other.mac)
 def __ne__ (self, other):
  return not self.__eq__(other)

 def isExpired (self):
  if self.port == of.OFPP_NONE: return False
  return time.time() > self.timeout


def dpid_to_mac (dpid):
 return EthAddr("%012x" % (dpid & 0xffFFffFFffFF,))


class Switch (EventMixin):
 def __init__ (self, fakeways = [], arp_for_unknowns = False, wide = False):

  # These are "fake gateways" -- we'll answer ARPs for them with MAC
  # of the switch they're connected to.
  self.fakeways = set(fakeways)

  # If True, we create "wide" matches.  Otherwise, we create "narrow"
  # (exact) matches.
  self.wide = wide

  # If this is true and we see a packet for an unknown
  # host, we'll ARP for it.
  self.arp_for_unknowns = arp_for_unknowns

  # (dpid,IP) -> expire_time
    # We use this to keep from spamming ARPs
    self.outstanding_arps = {}

    # we can't deliver because we don't know where they go.
    self.lost_buffers = {}

    # For each switch, we map IP addresses to Entries
    self.arpTable = {}

    # This timer handles expiring stuff
    self._expire_timer = Timer(5, self._handle_expiration, recurring=True)

    core.listen_to_dependencies(self)

   def _handle_expiration (self):
    # Called by a timer so that we can remove old items.
    empty = []
    for k,v in self.lost_buffers.iteritems():
     dpid,ip = k

     for item in list(v):
      expires_at,buffer_id,in_port = item
      if expires_at < time.time():
       # This packet is old.  Tell this switch to drop it.
       v.remove(item)
       po = of.ofp_packet_out(buffer_id = buffer_id, in_port = in_port)
       core.openflow.sendToDPID(dpid, po)
     if len(v) == 0: empty.append(k)

    # Remove empty buffer bins
    for k in empty:
      del self.lost_buffers[k]

  def _send_lost_buffers (self, dpid, ipaddr, macaddr, port):
    """
    We may have "lost" buffers -- packets we got but didn't know
    where to send at the time.  We may know now.  Try and see.
    """
    if (dpid,ipaddr) in self.lost_buffers:

      bucket = self.lost_buffers[(dpid,ipaddr)]
      del self.lost_buffers[(dpid,ipaddr)]
      log.debug("Sending %i buffered packets to %s from %s"
                % (len(bucket),ipaddr,dpid_to_str(dpid)))
      for _,buffer_id,in_port in bucket:
        po = of.ofp_packet_out(buffer_id=buffer_id,in_port=in_port)
        po.actions.append(of.ofp_action_dl_addr.set_dst(macaddr))
        po.actions.append(of.ofp_action_output(port = port))
        core.openflow.sendToDPID(dpid, po)

  def _handle_incoming_packets (self, event):
    dpid = event.connection.dpid
    inport = event.port
    packet = event.parsed
    global set_Timer
    global defendDDOS
    global blockPort
    timerSet =False
    global diction

    def block_port():
      global diction
      global set_Timer
      if not set_Timer:
        set_Timer =True

      if len(diction) == 0:
        print("Empty diction ",str(event.connection.dpid), str(event.port))
        diction[event.connection.dpid] = {}
        diction[event.connection.dpid][event.port] = 1
      elif event.connection.dpid not in diction:
        diction[event.connection.dpid] = {}
        diction[event.connection.dpid][event.port] = 1
      else:
        if event.connection.dpid in diction:

          if event.port in diction[event.connection.dpid]:
            temp_count=0
            temp_count =diction[event.connection.dpid][event.port]
            temp_count = temp_count+1
            diction[event.connection.dpid][event.port]=temp_count
          else:
            diction[event.connection.dpid][event.port] = 1

      print("\n",datetime.datetime.now(), ": printing diction ",str(diction),"\n")


    def _port_blocker_callback ():
      global diction
      global set_Timer
      if set_Timer==True:

        for k,v in diction.iteritems():
          for i,j in v.iteritems():
            if j >=50:
              print( "_____________________________________________________________________________________________")
              print("\n",datetime.datetime.now(),"*******    DDOS DETECTED   ********")
              print("\n",str(diction))
              print("\n",datetime.datetime.now(),": BLOCKED PORT NUMBER  : ", str(i), " OF SWITCH ID: ", str(k))
              print("\n_____________________________________________________________________________________________")

              dpid = k
              msg = of.ofp_packet_out(in_port=i)
              core.openflow.sendToDPID(dpid,msg)

      diction={}

    if not packet.parsed:
      log.warning("%i %i ignoring unparsed packet", dpid, inport)
      return

    if dpid not in self.arpTable:
      # New switch -- create an empty table
      self.arpTable[dpid] = {}
      for fake in self.fakeways:
        self.arpTable[dpid][IPAddr(fake)] = ArpTableEntry(of.OFPP_NONE,

         dpid_to_mac(dpid))

    if packet.type == ethernet.LLDP_TYPE:
      # Ignore LLDP packets
      return

    if isinstance(packet.next, ipv4):
      log.debug("%i %i IP %s => %s", dpid,inport,
                packet.next.srcip,packet.next.dstip)
      ent_obj.destination_ip_observer(event.parsed.next.dstip)#editing
      print("\n***** Entropy Value = ",str(ent_obj.value),"*****\n")
      if ent_obj.value <0.5:
        block_port()
        if timerSet is not True:
          Timer(2, _port_blocker_callback, recurring=True)
          timerSet=False
      else:
        timerSet=False


      # Send any waiting packets...
      self._send_lost_buffers(dpid, packet.next.srcip, packet.src, inport)

      # Learn or update port/MAC info
      if packet.next.srcip in self.arpTable[dpid]:
        if self.arpTable[dpid][packet.next.srcip] != (inport, packet.src):
          log.info("%i %i RE-learned %s", dpid,inport,packet.next.srcip)
          if self.wide:
            # Make sure we don't have any entries with the old info...
            msg = of.ofp_flow_mod(command=of.OFPFC_DELETE)
            msg.match.nw_dst = packet.next.srcip
            msg.match.dl_type = ethernet.IP_TYPE
            event.connection.send(msg)
      else:
        log.debug("%i %i learned %s", dpid,inport,packet.next.srcip)
      self.arpTable[dpid][packet.next.srcip] = ArpTableEntry(inport, packet.src)

      # Try to forward
      dstaddr = packet.next.dstip
      if dstaddr in self.arpTable[dpid]:
       # We have info about what port to send it out on...

       prt = self.arpTable[dpid][dstaddr].port
       mac = self.arpTable[dpid][dstaddr].mac
       if prt == inport:
        log.warning("%i %i not sending packet for %s back out of the "
              "input port" % (dpid, inport, dstaddr))
       else:
        log.debug("%i %i installing flow for %s => %s out port %i"
             % (dpid, inport, packet.next.srcip, dstaddr, prt))

        actions = []
        actions.append(of.ofp_action_dl_addr.set_dst(mac))
        actions.append(of.ofp_action_output(port = prt))
        if self.wide:
         match = of.ofp_match(dl_type = packet.type, nw_dst = dstaddr)
        else:
         match = of.ofp_match.from_packet(packet, inport)

        msg = of.ofp_flow_mod(command=of.OFPFC_ADD,
                   idle_timeout=FLOW_IDLE_TIMEOUT,
                   hard_timeout=of.OFP_FLOW_PERMANENT,
                   buffer_id=event.ofp.buffer_id,
                   actions=actions,
                   match=match)
        event.connection.send(msg.pack())
      elif self.arp_for_unknowns:
       # We don't know this destination.
       # First, we track this buffer so that we can try to resend it later
       # if we learn the destination, second we ARP for the destination,
       # which should ultimately result in it responding and us learning
       # where it is

       # Add to tracked buffers
       if (dpid,dstaddr) not in self.lost_buffers:
        self.lost_buffers[(dpid,dstaddr)] = []
       bucket = self.lost_buffers[(dpid,dstaddr)]
       entry = (time.time() + MAX_BUFFER_TIME,event.ofp.buffer_id,inport)
       bucket.append(entry)
       while len(bucket) > MAX_BUFFERED_PER_IP: del bucket[0]

       # Expire things from our outstanding ARP list...
       self.outstanding_arps = {k:v for k,v in
        self.outstanding_arps.iteritems() if v > time.time()}

       # Check if we've already ARPed recently
       if (dpid,dstaddr) in self.outstanding_arps:
        # Oop, we've already done this one recently.
        return

       # And ARP...
       self.outstanding_arps[(dpid,dstaddr)] = time.time() + 4

       r = arp()
       r.hwtype = r.HW_TYPE_ETHERNET
       r.prototype = r.PROTO_TYPE_IP
       r.hwlen = 6
       r.protolen = r.protolen
       r.opcode = r.REQUEST
       r.hwdst = ETHER_BROADCAST
       r.protodst = dstaddr
       r.hwsrc = packet.src
```

```
              r.protosrc = packet.next.srcip
              e = ethernet(type=ethernet.ARP_TYPE, src=packet.src,
                           dst=ETHER_BROADCAST)
              e.set_payload(r)
              log.debug("%i %i ARPing for %s on behalf of %s" % (dpid, inport,
               r.protodst, r.protosrc))
              msg = of.ofp_packet_out()
              msg.data = e.pack()
              msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
              msg.in_port = inport
              event.connection.send(msg)

          elif isinstance(packet.next, arp):
            a = packet.next
            log.debug("%i %i ARP %s %s => %s", dpid, inport,
             {arp.REQUEST:"request",arp.REPLY:"reply"}.get(a.opcode,
             'op:%i' % (a.opcode,)), a.protosrc, a.protodst)

            if a.prototype == arp.PROTO_TYPE_IP:
              if a.hwtype == arp.HW_TYPE_ETHERNET:
                if a.protosrc != 0:

                  # Learn or update port/MAC info
                  if a.protosrc in self.arpTable[dpid]:
                    if self.arpTable[dpid][a.protosrc] != (inport, packet.src):
                      log.info("%i %i RE-learned %s", dpid,inport,a.protosrc)
                      if self.wide:
                        # Make sure we don't have any entries with the old info...
                        msg = of.ofp_flow_mod(command=of.OFPFC_DELETE)
                        msg.match.dl_type = ethernet.IP_TYPE
                        msg.match.nw_dst = a.protosrc
                        event.connection.send(msg)
                  else:
                    log.debug("%i %i learned %s", dpid,inport,a.protosrc)
                  self.arpTable[dpid][a.protosrc] = ArpTableEntry(inport, packet.src)

                  # Send any waiting packets...
                  self._send_lost_buffers(dpid, a.protosrc, packet.src, inport)

                  if a.opcode == arp.REQUEST:
                    # Maybe we can answer

                    if a.protodst in self.arpTable[dpid]:
                      # We have an answer...

                      if not self.arpTable[dpid][a.protodst].isExpired():
                        # .. and it's relatively current, so we'll reply ourselves

                        r = arp()
                        r.hwtype = a.hwtype
                        r.prototype = a.prototype
                        r.hwlen = a.hwlen
                        r.protolen = a.protolen
                        r.opcode = arp.REPLY
                        r.hwdst = a.hwsrc
                        r.protodst = a.protosrc
                        r.protosrc = a.protodst
                        r.hwsrc = self.arpTable[dpid][a.protodst].mac
                        e = ethernet(type=packet.type, src=dpid_to_mac(dpid),
                                     dst=a.hwsrc)
                        e.set_payload(r)
                  log.debug("%i %i answering ARP for %s" % (dpid, inport,
                   r.protosrc))
                  msg = of.ofp_packet_out()
                  msg.data = e.pack()
                  msg.actions.append(of.ofp_action_output(port =
                                                          of.OFPP_IN_PORT))
                  msg.in_port = inport
                  event.connection.send(msg)
                  return

      log.debug("%i %i flooding ARP %s %s => %s" % (dpid, inport,
       {arp.REQUEST:"request",arp.REPLY:"reply"}.get(a.opcode,
       'op:%i' % (a.opcode,)), a.protosrc, a.protodst))

      msg = of.ofp_packet_out(in_port = inport, data = event.ofp,
          action = of.ofp_action_output(port = of.OFPP_FLOOD))
      event.connection.send(msg)




def launch (fakeways="", arp_for_unknowns=None, wide=False):
  fakeways = fakeways.replace(","," ").split()
  fakeways = [IPAddr(x) for x in fakeways]
  if arp_for_unknowns is None:
    arp_for_unknowns = len(fakeways) > 0
  else:
    arp_for_unknowns = str_to_bool(arp_for_unknowns)
  core.registerNew(Switch, fakeways, arp_for_unknowns, wide)
```

Appendix 4. Source Code Listing – detection.py

```python
import math
from pox.core import core
log = core.getLogger()

class Entropy(object):
    count = 0
    entDic = {}
    ipList = []
    dstEnt = []
    value = 1

    def destination_ip_observer(self, element):
        l = 0
        self.count +=1
        self.ipList.append(element)
        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
            self.calculate_entropy(self.entDic)
            log.info(self.entDic)
            self.entDic = {}
            self.ipList = []
            l = 0
            self.count = 0

    def calculate_entropy (self, lists):
        l = 50
        elist = []
        for k,p in lists.items():
            c = p/float(l)
            c = abs(c)
            elist.append(-c * math.log(c, 10))
            log.info('Entropy = ')
            log.info(sum(elist))
            self.dstEnt.append(sum(elist))
        if(len(self.dstEnt)) == 80:
            print(self.dstEnt)
            self.dstEnt = []
            self.value = sum(elist)

    def __init__(self):
        pass
```

Appendix 5. Source Code Listing – config.json

```json
{
    "attackConfig":
  {
    "notValidFirstOctet":[10,127,254,255,1,2,169,172,192],
    "srcPort":80,
    "targetPort":1,
    "targetIP":"10.0.0.64",
    "ddosIteration":5,
    "ddosAttackMagnitude":500
  },

  "trafficConfig":
  {
      "notValidFirstOctet":[10,127,254,1,2,169,172,192],
      "srcPort":2,
      "destinationPort":80,
      "destinationLastOctetRange":"2:65",
      "networkTrafficSize":1000

  }


}
```