CHAPTER ONE

INTRODUCTION

CHAPTER ONE
INTRODUCTION

1.1 Preface

The current network infrastructure known as the 'Internet' has settled for more than a decade and has rooted deeply in our society. However, enterprises and carriers are starting to realize the limitations of current state with the rapidly evolving network technologies and the growing demands of the users [1]. Also, the level of access network differentiation was not foreseen in the development of basic network protocols. It was assumed, that different applications could communicate with each other without any restrictions in a global network. This assumption required an adjustment quite quickly and there were developed specialized devices such as firewalls, Intrusion Prevention Systems (IPS), network anti-viruses and Web Application Firewalls (WAF) [2].

To overcome this limitation and security problems, Software-Defined Network (SDN) was suggested around 2005 as one of the most brilliant, flexible and cost-efficient solutions. The idea of Software Defined Networking (SDN) associated with OpenFlow protocol can provide a strong solution to resolve the problem of network threats from host or end user level, and from network level as well.

This project focused on how to discover and prevent possible vulnerability and recognize abnormal behavior in virtual network devices and hosts that could be easily monitored and investigated without damage of the real production environment. The proposed technique reduces the cost of equipment, intellectual property and data recovering.

The main contribution of this project, is to implement an easy, flexible and cheaper firewall to secure virtual network environment based on an Open Source and SDN technology and make network administration easier and more effective. A basic network identifier that have the ability to recognize the host activities when trying to connect to another network objects in the local or external networks and how will be devolved.

1.2 Problem Statement

There is need for a centralized, flexible, programmable and efficient mechanism to handle the procedures of network access control and malicious attack prevention. As well as, to achieve maximum leverage of network devices such as switches, routers and firewalls and to eliminate usage of expensive physical devices.

1.3 Proposed Solution

An attractive solution for the mentioned problems is provided by Open Source applications with user friendly interface and flexible configuration. A well-known switch supporting OpenFlow protocol, which is managed by SDN Controller with firewall module to control the forwarding behavior of the switch will be implemented.

1.4 Research Aim and Objectives

The aim of this research is to propose a new flexible, robust and cost-efficient method for network threats detection and prevention instead of the existing methods and solutions. As well as, to improve the network performance and its efficiency.

The detailed objectives of this research work are achieved through the following steps:

- To establish a security algorithm running with SDN controller as a virtual firewall for the network devices.
- To manage the forwarding behavior of the OpenFlow switch by SDN controller and the established firewall module.
- To test the network performance and check wither any improvement has been occurred or not.

1.5 Methodology

This research work is based on SDN controller and OpenFlow protocol which is a powerful solution that can be used to detect and control abnormal and suspicious network behavior by using switches that support OpenFlow protocol as network security appliances. The project introduces an implementation of packet filtering and detection mechanism on SDN controller and a comparison between the existing non-SDN controller methods and the proposed solution.

The research methodology contains different stages, these have been illustrated in the following points:

- A virtual network environment of three hosts connected by OpenFlow supporting switch have been implemented to test basic switch functionality (Without SDN Controller).
- The switch has been managed with POX controller, reconfigured the OpenFlow switch to hub module, switch module and vice versa (With SDN controller and Without Firewall module).
- POX controller has configured with the firewall module (security algorithm) which is a Paython_based Code. This algorithm added a firewall functionality by providing traffic filtering and decision-making mechanism.
- Hping3 is a free packet generator and analyzer for the TCP/IP protocol and it is a type of network security tester. This tool has been used to generate SYN flood - which is a kind of DoS attack - against the hosts and find out wither the POX firewall able to detect and terminate this kind of threat or not (First Scenario).
- Iperf tool which is a traffic generator tool has been used to generate streams of TCP and UDP traffic that the POX controller was able to identify them and made the appropriate actions as drop, allow or deny traffic (Second Scenario).
- Traffic Performance tests have been applied on both non-SDN method and the proposed methods and all the results have been written down.

1.6 Thesis Outlines

In Chapter two, general overview of firewalls, SDN controller and OpenFlow protocol technologies have been given. Also, some of the previous studies in the same field have been mentioned to take the maximum benefits.

In chapter three, the methodology of this work has been illustrated in detail with all Operating Systems, software packets and commands that have been used in the project.

In Chapter four, the firewall modules (Paython_based codes) have been used. Then, the traffic performance and penetration tests were applied and the results from each scenario have been presented.

In Chapter five, the conclusion has been obtained from all the results that were drew. Some recommendations for the future work related to this project are also proposed.

CHAPTER TWO

LITERATURE REVIEW

CHAPTER TWO
LITERATURE REVIEW

In this chapter, a detailed background about SDN controller and OpenFlow protocol with all different aspects that are related have been mentioned. Also, some researches outcomes have been written down in this chapter.

2.1 Background

2.1.1 Traditional Firewalls

Firewalls are either software components or hardware devices that enforce security policies in order to restrict unauthorized network access. The security policies filter network traffic based on the information in one or more of the Open Systems Interconnection (OSI) layers [2] [3]. Even though the term firewall is widely used as a technical term, it was originally used by Lightoler in 1764 to describe a wall that confined a potential fire from spreading from one location to another [3]. The term was used also to describe the iron walls behind the engine compartment of steam trains. These iron walls were used to stop fire from spreading to the passenger compartment.

Routers were considered the first network firewalls in the late 1980s because they were used to separate a network into different broadcast domains. This separation limited problems from a domain or local area network (LAN) from affecting the whole network. In addition, routers helped isolating "chatty" protocols, which use broadcasts messages for communication, from affecting the bandwidth of the rest of the network [3], [4].

2.1.1.1 Needs of Firewalls

Firewalls can be used to enforce security policies for the following reasons:

- To secure the underlying operating systems by preventing some types of communication, malware, attacks, etc. [3].
- To limit access to information on the Internet, an example of which is the filtering rules mandated in the United States by the Children's Internet Protection Act (CHIPA) [3].
- Preventing the leakage of information to the outside of the network.
- Enforcing policy rules on network traffic.
- To provide auditing information for the network administrator [3].

2.1.1.2 Firewall Types in Historical Order

In 1989, Jeffery Mogul described a solution that worked at the application layer to decide whether or not to pass packets through a router [3], [5]. His solution was to monitor the source address, destination address, protocol type, and port numbers to make the decision to allow or deny packets. However, Mogul's solution considered neither the state of TCP connections nor the pseudo-state of UDP traffic [5]. The first commercial firewall was developed by Digital Equipment Corporation (DEC) and was based on the technology proposed by Mogul. However, Marcus Ranum at DEC rewrote the rest of the firewall code after inventing security proxies, and the final firewall product was called DEC Secure External Access Link (SEAL) [3], [4]. A chemical company was the first to have DEC SEAL on

June 13, 1991 [3], [4]. The DEC SEAL consisted of three devices, shown in Figure 2-1: an application proxy server called Gatekeeper, a packet filtering gateway called Gate, and an internal mail server called Mailgate.

Traffic from inside to outside should pass through the Gate and then to the Gatekeeper, which decided whether the traffic would be allowed to be sent to the destination. Traffic was not allowed to be sent directly from the source to the destination without passing through the Gatekeeper [3].

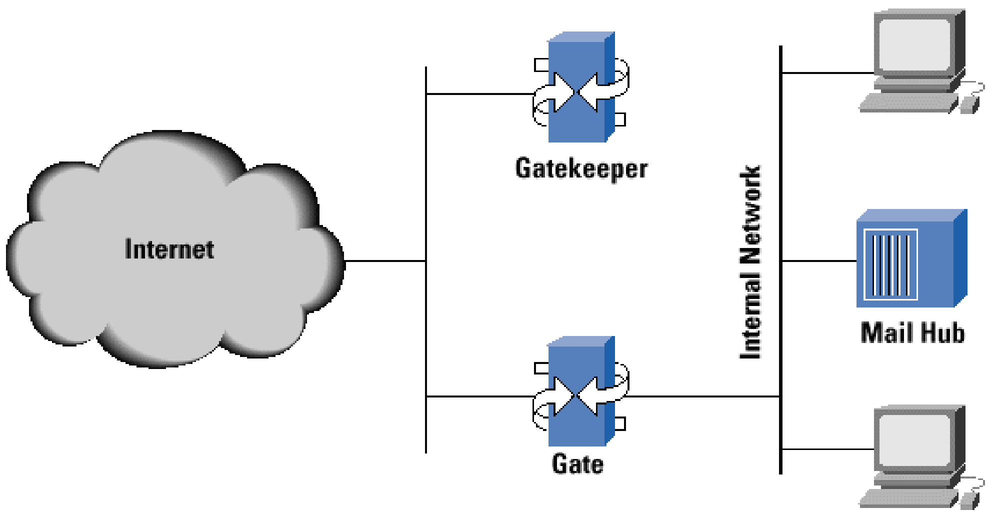


Figure 2-1: DEC SEAL - First Commercial Firewall

Application level proxies such as DEC SEAL provide good security and auditing capabilities because each packet is stopped at the proxy firewall, then examined, and finally recreated if it passes the rules. However, one drawback of this process is that a new application type requires a new application proxy to be developed. Moreover, the client program must be modified to account for the proxy in the network. A final drawback is performance, as each packet must be encapsulated two times, one at the Gatekeeper and one at the destination [3].

During the same time DEC was working on SEAL, Cheswick and Presotto at AT&T Bell Labs were designing a proxy-based firewall that worked at the transport layer, rather than at the application layer proxy as in DEC SEAL [6]. This design had the same issue that DEC SEAL had, which is the lack of performance as each packet was required to cross the network protocol stack two times [3]. Cheswick reported that the file transfer speed without the proxy was around 60-90 Kbps, while it was around 17-44 Kbps with the proxy. Furthermore, client programs need to be modified to account for the proxy-like application layer proxies.

In order to simplify the use of proxies, David Kolbas developed Socket Secure protocol (SOCKS), which routes traffic between a server and a client through a proxy. Some web browsers such as Netscape supported SOCKS. Avolio and Ranum released the source code of the Trusted Information Systems (TIS) Firewall Toolkit (FWTK) on October 1, 1993. This toolkit supported Simple Mail Transfer Protocol (SMTP), Network Transport Protocol (NTP), Telnet, File Transfer Protocol (FTP), and generic circuit-level application proxies [7]. However, FWTK did not support User Datagram Protocol (UDP) services. In 1994, Check Point introduced Firewall-1, which had a user-friendly interface that simplified the installation and administration of the firewall [4]. The TIS firewall became the Network Associates Incorporation's (NAI) Gauntlet Internet firewall after the merger between TIS and NAI in 1998 [3].

Packet filtering firewalls started with Mogul's paper [3]. This type of firewall is much faster than the application and transport layer proxies because it does not require the packet to traverse the OSI network stack twice. In addition, it does not require any changes on the user side. Packet Filtering firewalls filter packets based on one or more of the following parameters: source address, destination address, options in the packet header, options in the segment header (TCP or UDP header), and the physical interface number [3]. Even though packet filtering firewalls are faster compared to proxies, there are disadvantages. First, configuring the filter rules is complex and error-prone. Second, the IP addresses could be spoofed by attackers. Last, the original packet filter firewalls were stateless, i.e. they did not keep track of the state of the connections; therefore, an attacker might bypass the firewall by claiming to be part of an existing TCP connection [3]. As a result, stateful firewalls were developed to keep track of TCP sessions and to allow packets coming from outside to access the network if they belonged to an active session. Darren Reed was the first to implement that concept in his IP Filter (IPF) version 3.0 in 1996 [3]; however, the first published peer-reviewed paper was by Chow and Julkunen in 1998 [3].

Network Address Translation (NAT) is a layer of protection like that provided by proxies since the inside network is isolated from the outside network through the router performing NAT. NAT device replaces the source IP address of the outbound packet with its own IP address, and it might also change the source port number of the packet to a random unused port number above 1024 and map that into a table to keep track of each translation. However, one drawback of NAT is that it might interfere with Internet Protocol Security (IPsec) operation, which uses a set of cryptography algorithms to ensure the integrity of Data [3], [8].

In addition to the previous types, there are some packet filter firewalls and proxies that work on the data link layer but still use the information in layer 2 – 4 to filter the packets. Working on layer 2 makes a firewall / proxy transparent at the network level, meaning that it could be placed anywhere in the network and that it neither requires an IP address to operate (except for management) nor changes to the host operating system; therefore, the installation time could be minimal [3].

Signature-based firewall, which might work at the user level as a transparent proxy, monitors the payload for known malicious strings to prevent an attack from happening. This approach is sometimes called "fingerprint scrubber" or "application scrubbing" [3]. Snort is an intrusion detection system that Hogwash firewall uses to drop packets that match the rules [9].

The emergence of new technologies such as Virtual Private Networks (VPNs) and Peer-to-Peer (P2P) networking raise new challenges for previous firewalls. For example. If the laptop's security of a remote-user who uses VPN to access the internal network of a company is breached, then the entire inside network might be accessible by the attacker [3]. In addition, a software bug in P2P programs such as Gnutella could be used by attackers to gain access to the victim's host [3].

2.1.2 Software Defined Networking (SDN)

SDN is a network architectural paradigm that separates the control plane, which is the logic that controls how traffic is forwarded from the data plane of a networking device, which is the underlying system that forwards traffic, such as a router or a switch [10], [11]. Proponents of SDN believe that this separation provides the network operator with many advantages over the conventional network architecture, such as promoting innovation and features development. In addition, it provides the operator the ability to use less expensive commodity switching hardware under the control of a logically centralized programmatic control plane. This design uses elastic less-expensive computing power instead of over-priced high-end routing and switching products [12], [13]. Figure 2-2 illustrates the logical view of SDN architecture. Even though the separation of the control and data planes is one of the fundamental principles of SDN, it is also the most controversial [14]. The location of the control plane and how far away it could be located from the data plane, whether all the functions in the control plane could be relocated, and the number of instances needed to provide high-availability are all highly debated topics [12].

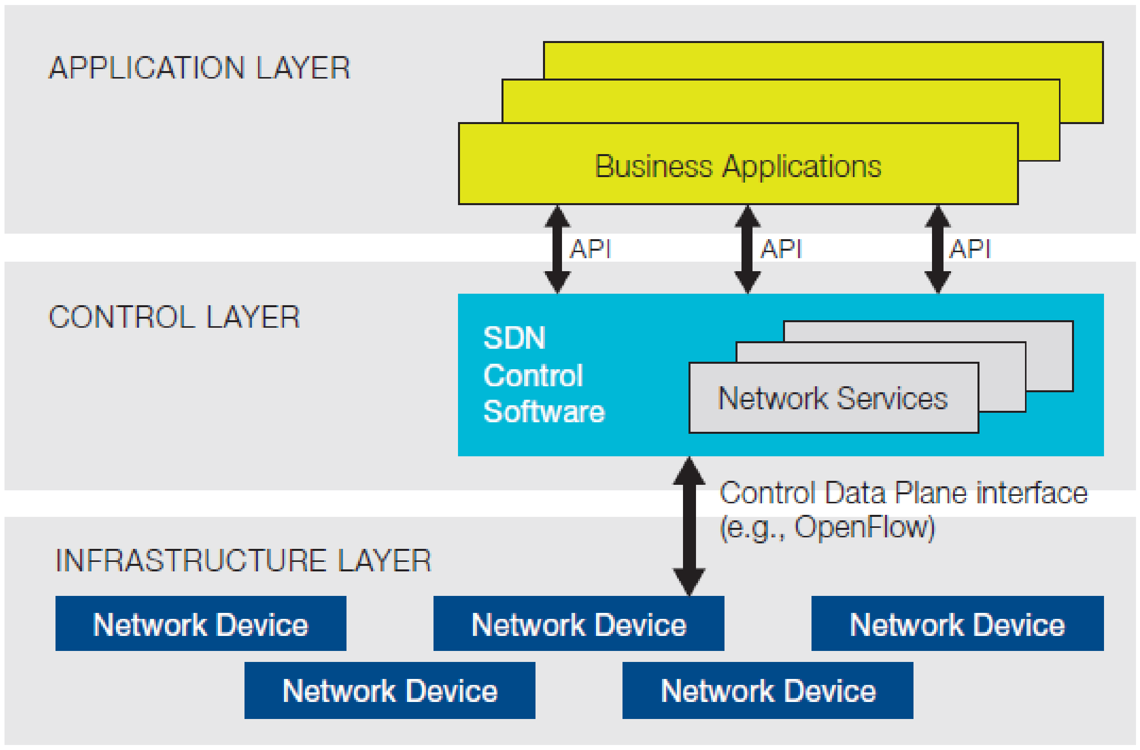


Figure 2-2: Software-Defined Network Architecture

There are three approaches to the distribution of the control and data plane: the strictly centralized, the logically-centralized, and the full distributed control plane [12]. In the first approach, the switching devices are dumb yet fast devices under the control of a centralized controller, which is considered the brain of the network. However, this extreme approach does not scale well, and it introduces a single point of failure in the network [12]. In the second approach, logically-centralized control plane, the switching devices retain some of the control plane functions, such as MAC addresses learning or ARP processing while at the same time a centralized controller can handle other functions that utilize the underlay network (switching devices) [12]. The last approach is the classical distributed control plane that each device has in addition to one or more data plane. These distributed control planes must cooperate with each other to have a functional network [12].

The control plane is the brain of the device. It exchanges protocol updates and system management messages [15]. It also maintains the routing table, which is also called the routing information base (RIB), through exchanging updates between other control plane instances in the network (routing protocol updates) and the forwarding table [12]. The FIB, or forwarding information base, is just a reformatting of the stable RIB table into an ordered list with the most specific route for each IP prefix at the top [16]. The control plane provides the data plane with an accurate up-to-date forwarding table through an internal link [15]. The data plane could use different types of technology to store the FIB tables, such as hardware-accelerated software, application-specific integrated circuits (ASICS), field-programmable gate array (FPGA), or any combination [12], [15]. In addition to forwarding traffic, the data plane implements some advance features such as policers, access control lists, and class of service (COS) [15]. All traffic is compared against the FIB table entries once it enters an ingress port, and it is forwarded out an egress port. However, if there is no entry for a packet's destination address, then the packet must be sent to the control plane for further processing. In addition, the following conditions that might cause the same behavior are [16]:

- Packets that are addressed to the router/switch, such as routing updates, pings, and trace routes.
- Full FIB table.
- IP packets that have the IP options field enabled.
- Packets that require the Internet Control Message Protocol (ICMP) to be generated.
- Packets that require compression or encryption.
- Packets in which the Time-To-Live (TTL) field has expired.
- Packets that require fragmentation due to exceeding the Maximum Transmission Unit (MTU).

2.1.2.1 The Importance of the Separation

The separation of the data plane from the control plane is not a new idea. Network device manufacturers have applied the same concept to the multi-slot routers and switches that they built in the last 10 years [12]. The control plane is implemented on a dedicated card - Cisco usually calls it the supervisor engine. To provide high availability, two supervisor engines are required – and the forwarding plane is implemented on one or more cards (line cards) independently, as shown in Figure 2-3 below [12]. However, the high cost of this design, along with other components, discussed below, is the motivation behind SDN.

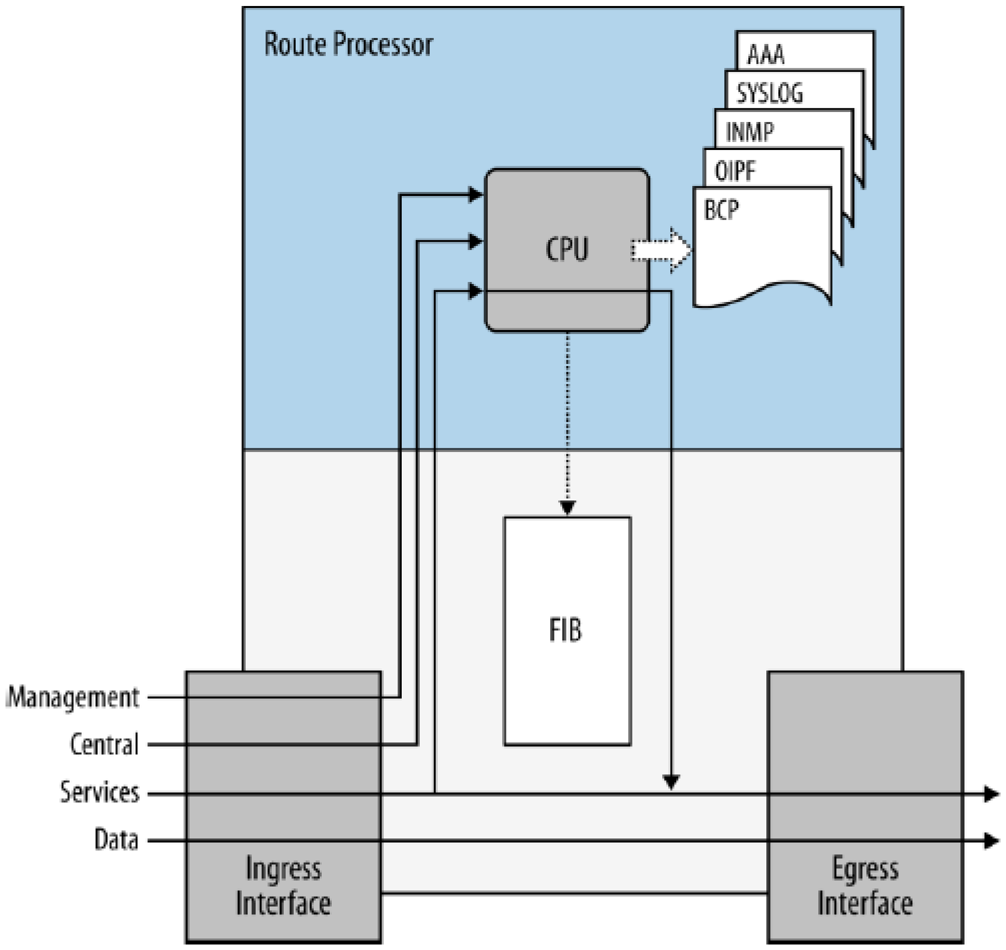


Figure 2-3: Control and data plane example implementation

First, I discuss the scaling issue of the service, forwarding, and control cards. Service cards can support only a limited number of subscribers or services state based on the generation of the embedded CPUs that they have. It takes a great deal of time for equipment vendors to take advantage of a new processor family in their service cards. In addition, the forwarding and control cards suffer from the same issue, which is the limitation of the embedded CPUs as well as the expensive memory that is limited in size [12]. Even though the SDN architecture still needs an upgrade in the control and service plane to accommodate scale, this upgrade could take advantage of the commercial off-the shelf (COTS) computing power that evolved dramatically and was driven by cloud computing [12].

Second, SDN will save large enterprises and service providers money on capital expenditure (CAPEX) since the cost of commodity devices is low in comparison to cost of high-end routes and switches from well-known vendors [12], [17].

Third, the separation of the control plane and data plane enables innovations in both planes since network operators would be able to provide new services by changing the software release independently from the hardware. This will also promote competition between enterprises or service providers to provide new services and features.

Fourth, the separation would make the forwarding elements more stable due to the smaller codebase required to implement the same network functionality in comparison to the conventional way. It is common these days to consider a smaller codebase more stable than a longer one that had many feature upgrades, such as the Multiprotocol Label Switching (MPLS) protocol [12].

Finally, in conventional networks, the greater the number of control planes, the more complex and fragile the system. That is, adding more devices (control planes) will impact the scale of the network, i.e. convergence time [12]. To address this issue, equipment vendors created the concept of system clusters where elements of the cluster are connected (through an external link) to create a single logical system controlled by a single control plane. A distributed control plane in the cluster is also available to provide load balancing. Even though this solution has characteristics of SDN, it does not solve the programmability issues of the control plane. Thus, SDN architecture is more flexible and provides a centralized control plane that reduces the complexity of the system [12].

2.1.3 OpenFlow Protocol

In 2008, a group of engineers from Stanford University developed an open standard protocol called OpenFlow, which enables researchers to evaluate and run experimental protocols in an existing production network without exposing the internal network. To allow that, OpenFlow enabled switch must be able to isolate experimental traffic from production traffic through either applying Virtual Local Area Networks (VLANs) or forwarding production traffic to the normal process of the switch [18]. OpenFlow protocol is a standard communications interface between the controller and the forwarding plane of the underlying network devices that allows network operators to manipulate the forwarding plane of these devices [19].

OpenFlow consists of a set of protocols and Application Programming Interface (API). The protocols are divided into two parts, as shown in Figure 2-4 below:

- The OpenFlow protocol, also called the wire protocol. This defines a message structure that enables the controller to add, update, and delete flow entries in the OpenFlow Logical Switch flow tables as well as to collect statistics [12], [20].
- The OpenFlow management and configuration protocol that defines an OpenFlow enabled switch as an abstraction layer called an OpenFlow

Logical Switch. This enables high availability by allocating physical switch ports to a controller [21].

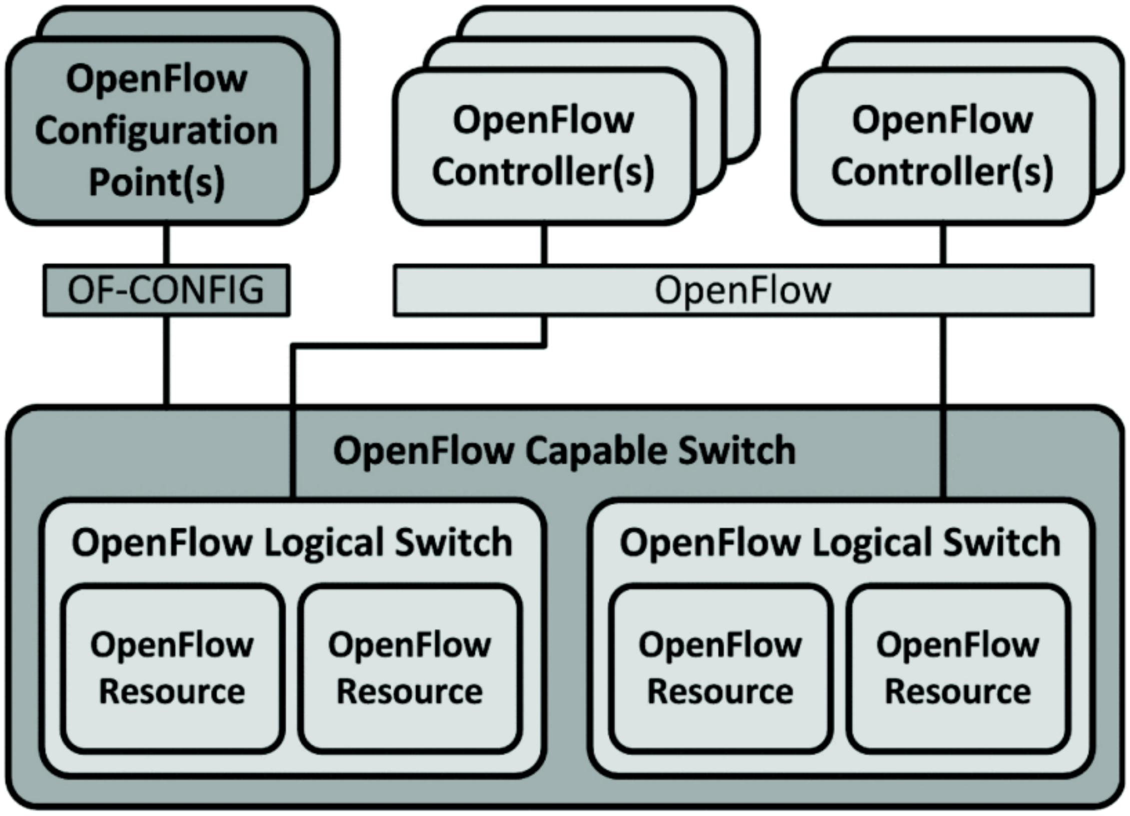


Figure 2-4: Relationship between components defined in the specification, the OF-CONFIG protocol and the OpenFlow protocol

2.1.3.1 OpenFlow Switch Components

An OpenFlow-enabled switch consists of a group table and one or more flow tables, one or more OpenFlow secure channels that connect the switch to an external controller, and an OpenFlow protocol that defines the control messages between the switch and the controller, as shown in Figure 2-5 below [22].

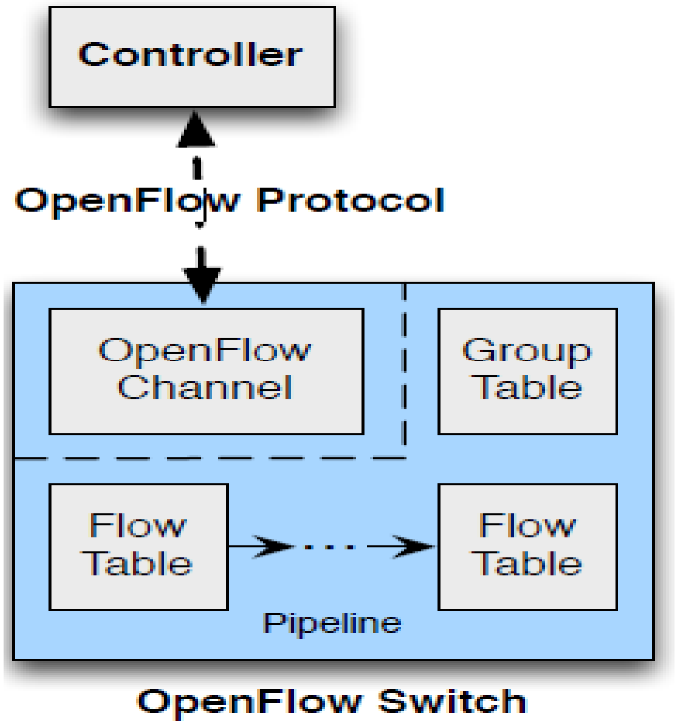


Figure 2-5: Main components of an OpenFlow switch

2.1.3.2 OpenFlow Ports

An OpenFlow protocol has different port types that pass traffic between OpenFlow processing and the rest of the network. The OpenFlow standard ports are as follows:

- Physical ports, which correspond to a hardware interface such as Ethernet switch port.
- Logical ports, which do not correspond to a hardware interface directly, such as tunnels, loopback interfaces, and link aggregation groups.
- Reserved ports, which are defined by the OpenFlow specification. The * means mandatory port. They define forwarding actions as follows [20]:

  - ALL *: Represents all the switch ports except the packet ingress port and ports configured with OFPPC_NO_FWD. It can be used only as an egress port.
  - CONTROLLER *: Represents the port to the OpenFlow controller.
  - TABLE *: Represents the beginning of the OpenFlow pipeline. It used as an output action in the “packet-out” message’s action list.
  - IN_PORT *: Represents the ingress port of the packet.
  - ANY *: Represents a wildcard value.
  - LOCAL: Represents the management stack of the local switch.
  - NORMAL: Represents the traditional layer 2 or layer 3 forwarding.
  - FLOOD: Represents flooding using the traditional pipeline of the switch to all ports except the ingress port and ports with flooding disabled state.

2.1.3.3 Flow Table

Each entry in the flow table is made of the following fields, as shown in Figure 2-6 [20]:

- Match fields: The matching criteria used against packets. They could be based on ingress port, packet headers, and metadata from the previous flow table.
- Priority: Priority of the entry. The higher the number, the higher the priority.
- Counters: This field increases when a packet matches an entry.
- Instructions: Actions applied to matching packets.

- Timeouts: This could be an idle-time or hard-time that specifies the amount of time before an entry expires.
- Cookie: This is not used to process packets, but it might be used by the controller to filter flows based on their types (statistics, modifications, and deletion).
- Flags: This field changes how flow entries are managed; for example, an entry with the flag OFPFF_SEND_FLOW_REM means that a flow removed message will be sent to the comptroller once this entry is removed.

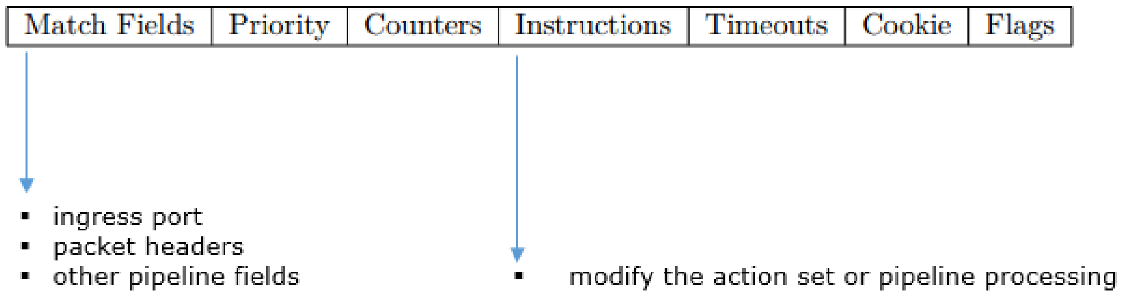


Figure 2-6: Main components of a flow entry in a flow table

2.1.3.4 OpenFlow Message Types

There are three main categories for message types and each category has its own types. The main categories are: controller-to-switch, asynchronous, and symmetric. The controller-to switch messages are originated from the controller, and they might not require the switch to respond to them. The asynchronous messages are originated from the switch to notify the controller of a packet arrival, an error, or switch state change. Last, the symmetric messages are created, without solicitation, by either the controller or the switch [20]. The sub-types of each category are as follows:

❖ **Controller-to-Switch Messages**

- Features: Sent by the controller to request the identity and capabilities of a switch that should reply with a feature reply message containing the requested information.
- Configuration: Allows the controller to quest or sent configuration parameters in the switch.
- Modify-State: Sent by the controller to add, delete, or modify flow entries or a group of entries in the OpenFlow table as well as to set the port properties of the switch.
- Read-State: Uses multipart messages to read the current configuration and collect statistics and capabilities information from the switch.
- Packet-out: Used by the controller to send packets out a specific port. This type of message should have either a full packet as raw data created by the controller or the buffer ID of the packet stored in the switch, which the controller received via Packet-in message. Furthermore, it should have a list of actions. If the list of actions is empty, the switch will drop the packet.
- Barrier: Used by controller to request and reply messages to either receive notification once operations are completed or to confirm that message requirements have been met.
- Role-Request: Used to set the OpenFlow channel's role or query for it. This is helpful when the switch is connected to multiple controllers to provide redundancy [20] [21].

❖ **Switch-to-Controller Messages**

- Packet-in: Generated by the switch and sent to the controller for processing. This can be triggered if there is no entry for the received packet in the flow table, if the output action of the flow entry is to send the packet to the controller, or if the packet needs other processing, such as TTL processing. Switches that support internal buffering will buffer the packet and send a configurable number of bytes, 128 bytes by default, along with a buffer ID to the controller. However, if the switch does not support internal buffering or if the buffer is full, it has to send the full packet to the controller.

- Flow-Removed: Used to notify the controller about the removal of a flow entry if that entry has the OFPFF_SEND_FLOW_REM flag.
- Port-status: Used to notify the controller when a change to the port state or configuration occurs.
- Error: Used to notify the controller of a problem [20].

❖ **Symmetric**

- Hello: Exchanged between the controller and the switch during the connection setup phase.
- Echo request/ reply: Used as keep-alive messages between the controller and the switch. They might also be used to measure the latency and bandwidth between them.
- Experimenter: Used to test new features [22].

2.1.3.5 Connection Setup

After configuring the switch with the IP address of the controller, the switch initiates a standard Transport Layer Security (TLS) or TCP connection to the controller listening on TCP port number 6653 or a user-specified TCP port. Once the TLS connection is established, each participant in the connection must send an OFPT_HELLO message with the highest protocol version supported by the sender in the version field. If the sent and received Hello messages contain OFPHET_VERSIONBITMAP, the negotiated version must be the highest version supported by both. Otherwise, the smallest version should be supported [20].

Upon successfully exchanging OFPT_HELLO messages and negotiating the protocol version number, the connection setup is completed and the OpenFlow messages described in the previous section can be exchanged; for example, the controller should first send an OFPT_FEATURE_REQUEST message to identify the Datapath ID of the switch, as the left side of Figure 2-7 illustrates [20]. The right side of Figure 2-7 shows an example of exchanging different OpenFlow messages.

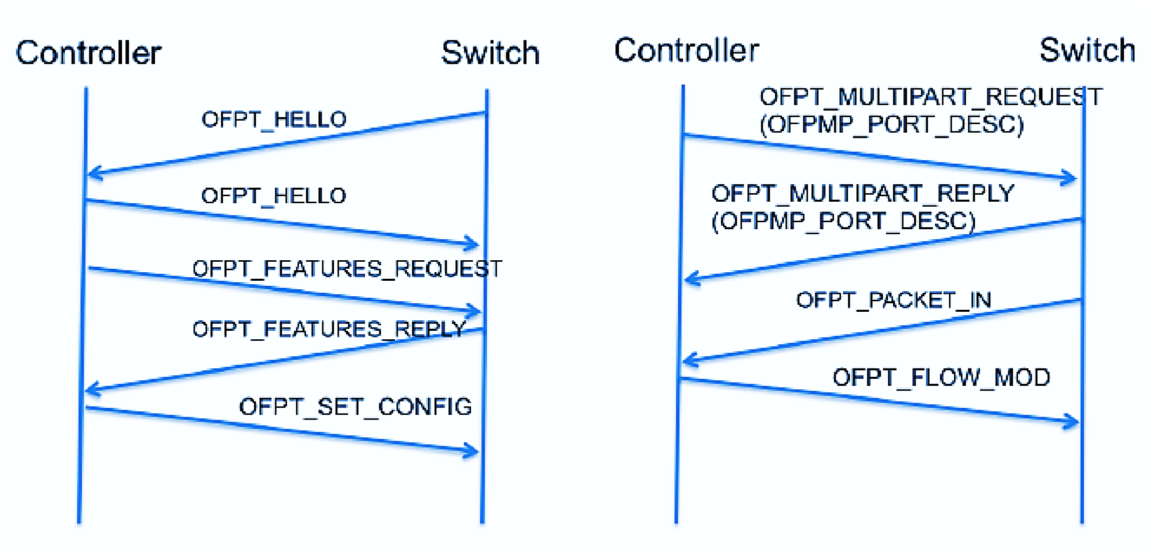


Figure 2-7: OpenFlow protocol messages

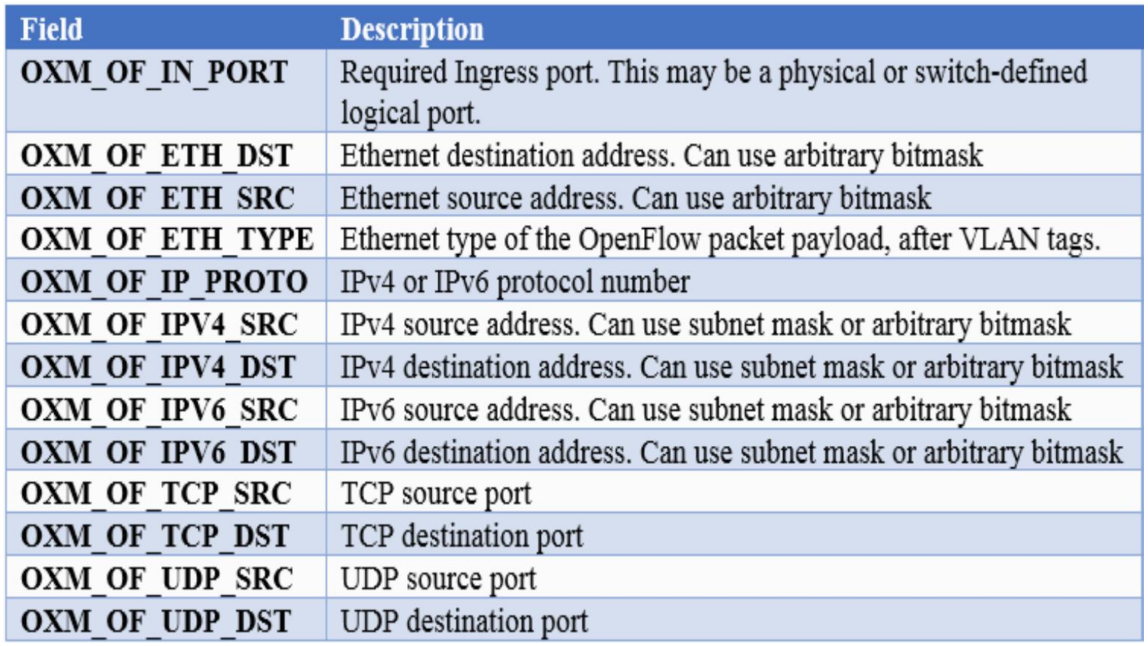
2.1.3.6 Multiple Controllers

To provide redundancy, high-availability, and load balancing, OpenFlow support multiple controllers that allow a switch to establish communication with each one of them. However, the handover process between the controllers is performed by the controllers themselves. The default role of the controller is EQUAL “OFPCR_ROLE_EQUAL”, which gives the controller full access to the switch. However, the role can be changed to SLAVE (read-only) “OFPCR_ROLE_SLAVE” upon the request of the controller [22].

2.1.3.7 Flow Match Fields

As explained in previous that one of the main components of flow entry in the flow table is the match fields. Table 2-1 illustrates the match fields that must be supported by the OpenFlow-enabled switch in its pipeline.

Table 2-1: Required Match Fields [20]



Field	Description
OXM_OF_IN_PORT	Required Ingress port. This may be a physical or switch-defined logical port.
OXM_OF_ETH_DST	Ethernet destination address. Can use arbitrary bitmask
OXM_OF_ETH_SRC	Ethernet source address. Can use arbitrary bitmask
OXM_OF_ETH_TYPE	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_IP_PROTO	IPv4 or IPv6 protocol number
OXM_OF_IPV4_SRC	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_SRC	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_TCP_SRC	TCP source port
OXM_OF_TCP_DST	TCP destination port
OXM_OF_UDP_SRC	UDP source port
OXM_OF_UDP_DST	UDP destination port

2.1.3.8 Action Structure

Table 2-2 summarizes the actions that the SDN controller could use with each flow entry, packet, or group.

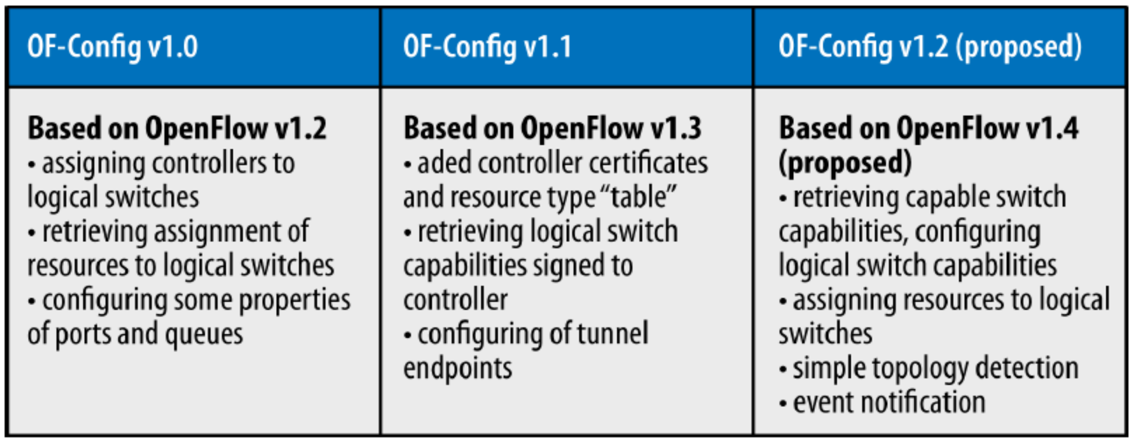
Table 2-2: Action Structure [20]

Action Type	Description
OFPAT_OUTPUT	Output to switch port
OFPAT_COPY_TTL_OUT	Copy TTL "outwards" -- from next-to-outermost to outermost
OFPAT_COPY_TTL_IN	Copy TTL "inwards" -- from outermost to next-to-outermost
OFPAT_SET_MPLS_TTL	MPLS TTL
OFPAT_DEC_MPLS_TTL	Decrement MPLS TTL
OFPAT_PUSH_VLAN	Push a new VLAN tag
OFPAT_POP_VLAN	Pop the outer VLAN tag
OFPAT_PUSH_MPLS	Push a new MPLS tag
OFPAT_POP_MPLS	Pop the outer MPLS tag
OFPAT_SET_QUEUE	Set queue id when outputting to a port
OFPAT_GROUP	Apply group
OFPAT_SET_NW_TTL	IP TTL
OFPAT_DEC_NW_TTL	Decrement IP TTL
OFPAT_SET_FIELD	Set a header field using OXM TLV format
OFPAT_PUSH_PBB	Push a new PBB service tag (I-TAG)
OFPAT_POP_PBB	Pop the outer PBB service tag (I-TAG)
OFPAT_EXPERIMENTER	

2.1.3.9 OF-Config Versions

The OF-config protocol is structured around NETCONF protocol to set information related to OpenFlow on the network elements. With OF-config, the operator does not have to use other tools, such as FlowVisor to provide switch virtualization [12], [23]. Table 2-3 compares OF-config versions [12].

Table 2-3: Capability progression of OF-Config [12]

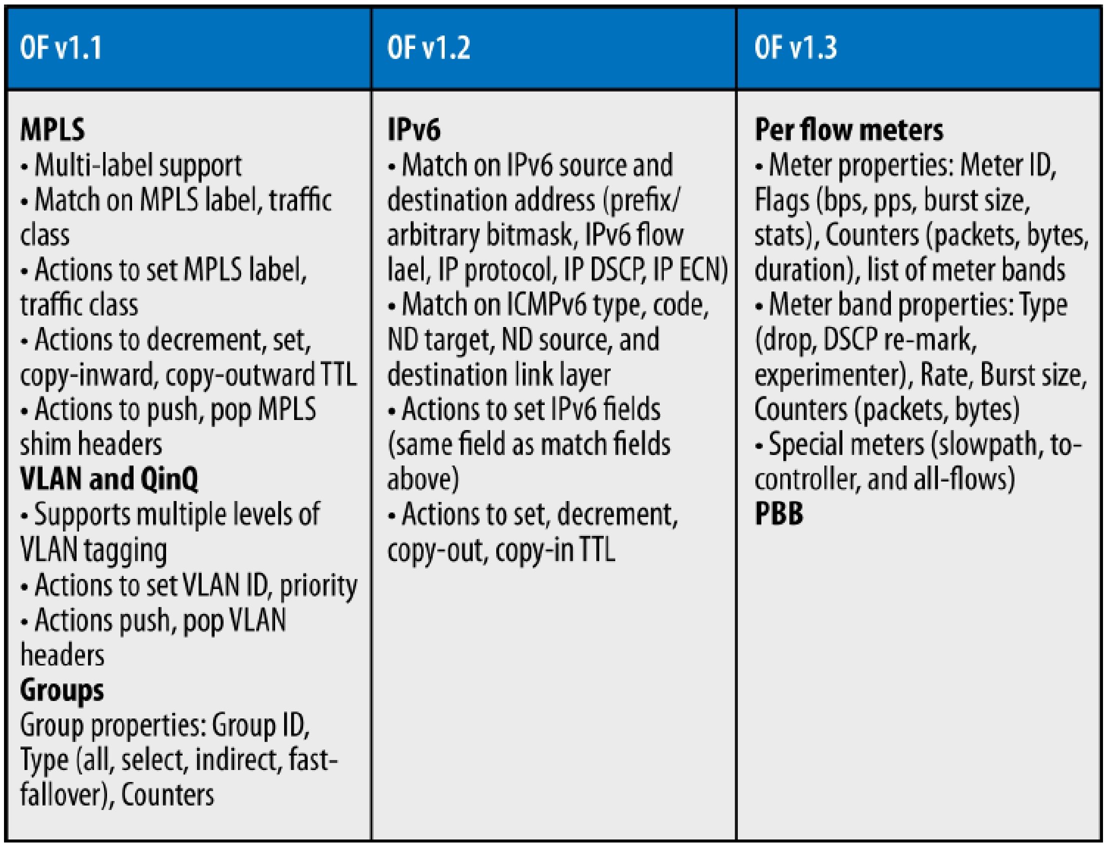


OF-Config v1.0	OF-Config v1.1	OF-Config v1.2 (proposed)
Based on OpenFlow v1.2 • assigning controllers to logical switches • retrieving assignment of resources to logical switches • configuring some properties of ports and queues	**Based on OpenFlow v1.3** • aded controller certificates and resource type “table” • retrieving logical switch capabilities signed to controller • configuring of tunnel endpoints	**Based on OpenFlow v1.4 (proposed)** • retrieving capable switch capabilities, configuring logical switch capabilities • assigning resources to logical switches • simple topology detection • event notification

2.1.3.10 OpenFlow Versions

The Extensibility Working Group added new functionality and features to OpenFlow protocol v1.0. When OpenFlow protocol v1.3 was released in April 2012, the Open Networking Foundation (ONF) decided to slow down releasing new versions to allow for higher adaption rate of OpenFlow v1.3 and to focus on bug-fix releases [12]. Table 2-4 below compares OpenFlow v1.1– 1.3.

Table 2-4: The progression of enhancements to the OpenFlow pipeline from OF v1.1 through OF v1.3 [12]

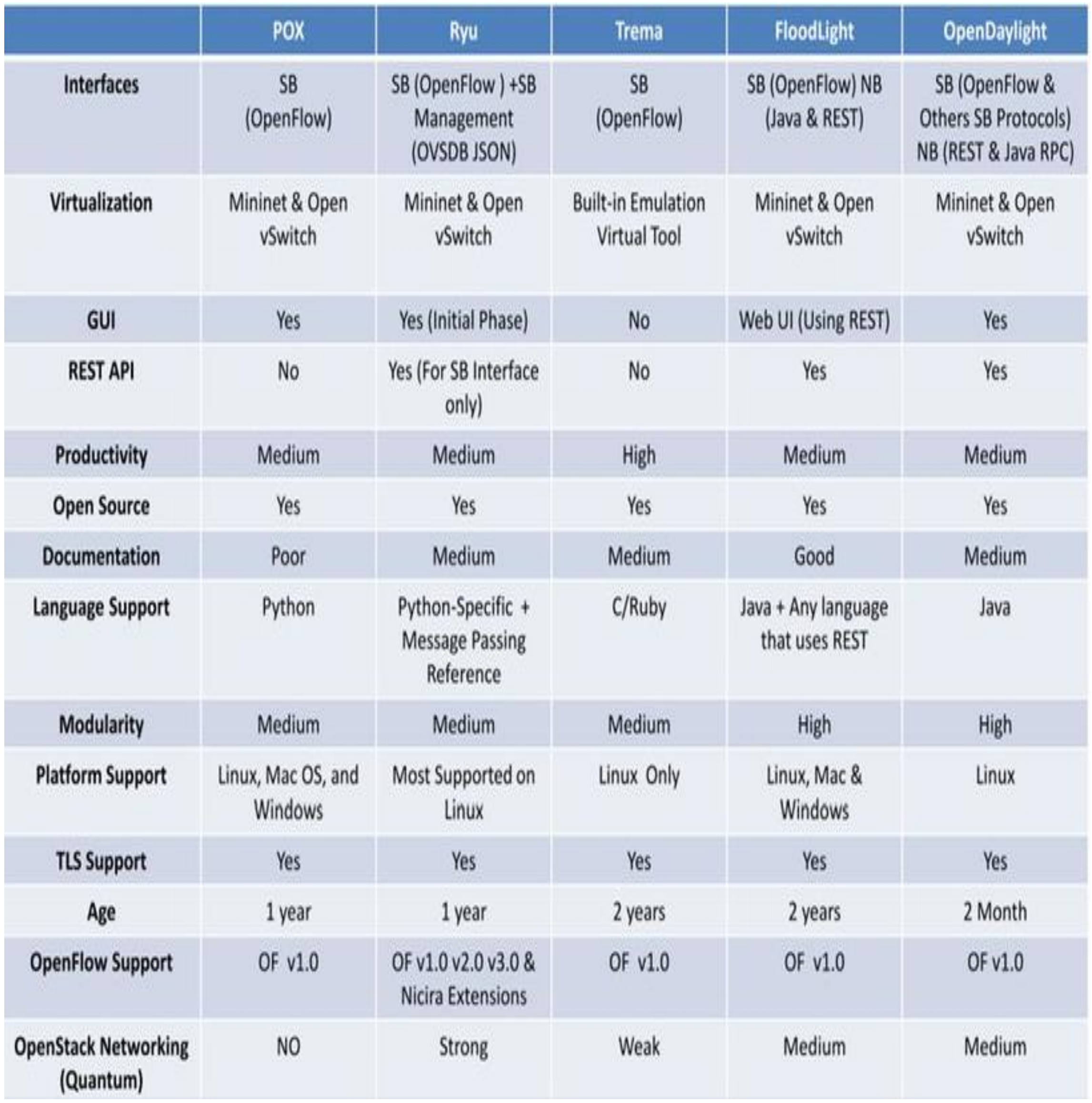


OF v1.1	OF v1.2	OF v1.3
MPLS • Multi-label support • Match on MPLS label, traffic class • Actions to set MPLS label, traffic class • Actions to decrement, set, copy-inward, copy-outward TTL • Actions to push, pop MPLS shim headers **VLAN and QinQ** • Supports multiple levels of VLAN tagging • Actions to set VLAN ID, priority • Actions push, pop VLAN headers **Groups** Group properties: Group ID, Type (all, select, indirect, fast-fallover), Counters	**IPv6** • Match on IPv6 source and destination address (prefix/ arbitrary bitmask, IPv6 flow lael, IP protocol, IP DSCP, IP ECN) • Match on ICMPv6 type, code, ND target, ND source, and destination link layer • Actions to set IPv6 fields (same field as match fields above) • Actions to set, decrement, copy-out, copy-in TTL	**Per flow meters** • Meter properties: Meter ID, Flags (bps, pps, burst size, stats), Counters (packets, bytes, duration), list of meter bands • Meter band properties: Type (drop, DSCP re-mark, experimenter), Rate, Burst size, Counters (packets, bytes) • Special meters (slowpath, to-controller, and all-flows) **PBB**

2.1.4 SDN Controllers

There are several open sources and commercial SDN controllers that have been developed. NOX/ POX, Floodlight, and OpenDaylight are examples of open source SDN controllers. On the other hand, Cisco OnePK controller is a commercial controller that integrates multiple southbound APIs. NOX is considered the first open source controller after being donated by Nicira to the research community in 2008. NOX provides a C++ OF v 1.0 API and an event-based programming model [24], [25]. POX is the Python version of NOX, and it supports the same graphical user interface (GUI) as NOX [30]. Table 2-5 compares the features of some SDN controllers [10].

Table 2-5: Comparison among the controllers [10]



	POX	Ryu	Trema	FloodLight	OpenDaylight
Interfaces	SB (OpenFlow)	SB (OpenFlow) +SB Management (OVSDB JSON)	SB (OpenFlow)	SB (OpenFlow) NB (Java & REST)	SB (OpenFlow & Others SB Protocols) NB (REST & Java RPC)
Virtualization	Mininet & Open vSwitch	Mininet & Open vSwitch	Built-in Emulation Virtual Tool	Mininet & Open vSwitch	Mininet & Open vSwitch
GUI	Yes	Yes (Initial Phase)	No	Web UI (Using REST)	Yes
REST API	No	Yes (For SB Interface only)	No	Yes	Yes
Productivity	Medium	Medium	High	Medium	Medium
Open Source	Yes	Yes	Yes	Yes	Yes
Documentation	Poor	Medium	Medium	Good	Medium
Language Support	Python	Python-Specific + Message Passing Reference	C/Ruby	Java + Any language that uses REST	Java
Modularity	Medium	Medium	Medium	High	High
Platform Support	Linux, Mac OS, and Windows	Most Supported on Linux	Linux Only	Linux, Mac & Windows	Linux
TLS Support	Yes	Yes	Yes	Yes	Yes
Age	1 year	1 year	2 years	2 years	2 Month
OpenFlow Support	OF v1.0	OF v1.0 v2.0 v3.0 & Nicira Extensions	OF v1.0	OF v1.0	OF v1.0
OpenStack Networking (Quantum)	NO	Strong	Weak	Medium	Medium

2.1.5 Northbound APIs

Even though SDN provides a way to program the network, it does not make it easy. SDN controllers such as NOX/ POX and Floodlight support a low-level interface that forces the applications to deal with the state of individual devices. Applications are developed as event handlers that respond to packet arrivals event. Having only a low-level interface, also called southbound interface, makes it extremely difficult to support multiple tasks/ module such as routing, switching, and firewall at the same time because the rules generated by one task/ module might have a conflict with the others (e.g., a rule to allow certain flow and another to deny it) [26].

Frenetic is projected to increase the level of abstraction and make developing applications for SDN much easier. It provides a suite of abstractions for defining rules, querying the state of the network, and updating rules in a consistent way [26].

Pyretic is an SDN programming language embedded in Python, and it is a member of the Frenetic family. It also provides powerful abstractions that enable programmers to develop modular network applications as Figure 2-8 shows [27]. One of the main advantages of Pyretic over traditional OpenFlow programming is that Pyretic offers parallel and sequential composition of policies to perform multiple tasks without worrying about potential policy conflicts. For example, in sequential composition, the output of the policy on the left of the operator (>>) is the input of the policy on the right of the operator, as shown in the Pyretic policy below [27]:

```
match(dstip’2.2.2.8’) >> fwd(1)
```

On the other hand, in parallel composition the operator (+) combines and applies two policies to the same packet, as shown in the routing policy R below, and forwards packets destined to 2.2.2.8 out to port 1 and those destined to 2.2.2.9 out to port 2 [27]:

```
R = (match(dstip='2.2.2.8') >> fwd(1)) + (match
(dstip='2.2.2.9')   >> fwd(2))
```

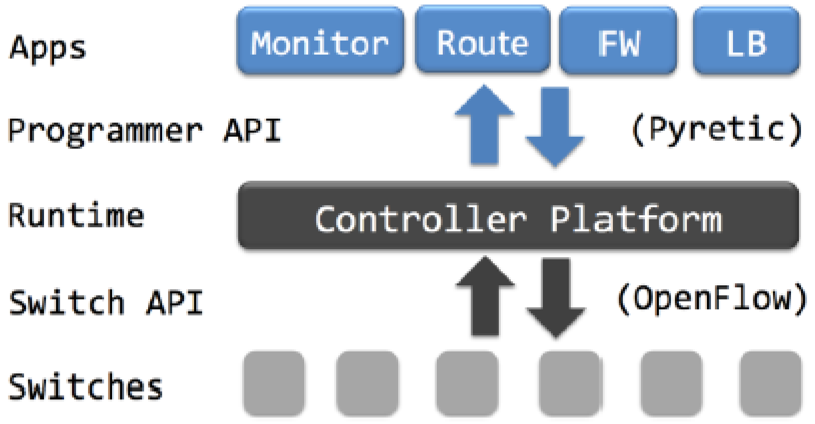


Figure 2-8: Northbound API [27]

2.1.6 SDN Use Cases

Since one of the main features of SDN is that it drives innovation [28], it is hard to summarize and imagine all SDN use cases. SDN could be utilized in campus, data center, cloud, and service provider networks [19].

Network administrators in campus networks could use the SDN model to enforce policies across the wireless and wired network consistently. In addition, SDN ensures an optimal user experience by supporting automated management of network resources and provisioning [19].

SDN architecture supports network virtualization that enables automated migration of virtual machine (VM) and hyper-scalability. Furthermore, it saves costs by reducing energy use and provides a better server utilization [19].

SDN also enables cloud service providers to allocate network resources in a very elastic way, which enhances provisioning. Moreover, SDN provides businesses with tools to safely manage their VMs in order to increase adaption of cloud services [19].

Considering the features that SDN brings, it is much easier for service providers to deploy resources optimally, to support multi-tenancy and to reduce both operational expenditure (OPEX) and CAPEX [19]. In addition, cellular service providers could utilize SDN to provide new services, such as base transceiver station (BTS) virtualization, and to reduce handover latency and many more [17], [29] – [30].

Finally, SDN could be used to replace expensive Layer 4-7 firewalls, load-balancers, and IPS/IDS with cost-effective high-performance switches and a logically centralized controller.

2.2 Related Works

In [31], it is introduced the SDN technology and systematically investigated its usage for security. Although many people have interests in this technology, until now, it is not yet well embraced by security researchers. They believe that SDN can, in time, prove to be one of the most impactful technologies to drive a variety of innovations in network security. They hope this study can not only provide a quick introduction and systematic survey but also give significant insights for using SDN for better security applications and stimulate more future research in this important area.

The authors in [32] had a surveyed research on security in SDN, a set of topics for future research have been identified. A strong theme amongst these topics is projection of potential security issues and automated response for quick reaction to network threats. By implementing proven security techniques from their current network deployments, resolving known security issues in SDN, and further exploiting the dynamic, programmable, and open characteristics of SDN, software-defined networks may well be more secure than traditional networks. There is much work to do before this vision is realized.

The authors in [33] had undertaken a comprehensive review of security-oriented research in software defined networks. They have classified current work in two main streams: threat detection, remediation and network correctness which simplify and enhance security of programmable networks, and security as a service, which offers new innovative security functionality to users, such as anonymity and specialized network management. Furthermore, they discussed possible challenges and future directions for security in SDN: these include the critical question of securing SDN itself, of orchestrating security policies across heterogeneous networks, customizing overlay networks to provide secure environments, and extending the OpenFlow paradigm with customized hardware and network functions virtualization and building a richer set of features in the forwarding path.

In [1], it is figured out that the SDN is not only revolutionary in making the control flexible and manageable, but also for firewalls to achieve programmability by separating the firewall hardware and the control software. An OpenFlow-based firewall with a straightforward UI that integrates priority switching can bring another wave of innovation in the Internet world.

In [34], it was argued for the need to consider security and dependability when designing Software Defined Networks. They have presented several threats identified in these networks as strong arguments for this need, together with a brief discussion of the mechanisms they are using in building a secure and dependable SDN control platform. The novel concepts introduced by SDN are enabling a revolution in networking research. The know-how and good practices from several communities (databases, programming languages, systems) are being put together to help solve long-standing networking problems.

The essential idea of the authors in [35] and [36] was to provide a security kernel (e.g., by extending a controller like NOX) capable of ensuring prioritized flow rule installation on switches. Applications were classified in two types, one for security related applications and another for all remaining applications. The first type represents specialized programs used to ensure security control policies in the network, such as to guarantee or restrict specific accesses to the network or take actions to control malicious data traffic. Flow rules generated by security applications have priority over the others. The security kernel is responsible for ensuring this behavior. FRESCO [36] was an extension of that work which made it easy to create and deploy security services in software-defined networks.

In [37], NICE had been built which was a tool for automating the testing of OpenFlow applications that combines model checking and concolic execution in a novel way to quickly explore the state space of unmodified controller programs written for the popular NOX platform. Further, it was devised a number of new, domain-specific techniques for mitigating the state-space explosion that plagues approaches. NICE had been contrasted with an approach that applies off-the-shelf model checkers to the OpenFlow domain, and it was demonstrated that NICE was five times faster even on small examples. NICE had been applied to implementations of three important applications, and found 11 bugs.

The authors in [38] have shown that Software Defined Networks using OpenFlow and NOX allow flexible, highly accurate, line rate detection of anomalies inside Home and SOHO networks. One of the key benefits of this approach is that the standardized programmability of SDN allows these algorithms to exist in the context of a broader framework. They envision a Home Operating System built using SDN, in which their algorithm implementations would coexist alongside other applications for the home network e.g. QoS and Access Control. The standardized interface provided by a SDN would allow their applications to be updated easily as new security threats emerge while maintaining portability across a broad and diverse range of networking hardware.

In [39], a set of attributes of a secure, robust, and resilient SDN controller have been presented. The extent to which current state-of-the-art open-source controllers support these attributes has been discussed. It is promising that all except one of the defined security attributes is supported by one or more controller. The missing feature is the management of multiple application instances for application resilience. This must be a design consideration for future controller developments. With the clear split between high availability controllers and secure/resilient control layers, the next evolution in SDN controller design must be a means to achieve the combined goal of security, robustness, and resilience.

The authors in [40] had presented a lightweight method for DDoS attack detection. They showed that their technique extracts features of interest with a low overhead when compared to approaches based on the KDD-99 dataset. It is also able to monitor more than one observation point. The method is also very efficient at detecting DDoS attacks. It uses Self Organizing Maps, an unsupervised artificial neural network, trained with features of the traffic flow. The detection rate obtained is remarkably good as it is very close to other approaches.

CHAPTER THREE

DETECTION OF NETWORK THREATS USING SDN

CHAPTER THREE

DETECTION OF NETWORK THREATS USING SDN

In this chapter, the methodology of the work has been discussed in detail with all environment software, tools, configuration and technologies that are used.

3.1 Preparation

This project provides a brilliant solution that is based on SDN OpenFlow protocol to monitor, identify, control and detect abnormal network behavior in LAN by using OpenFlow supporting switch and SDN controller (POX controller) as a network security appliance. An implementation of network anomaly detection algorithm (Firewall module) on SDN controller has been provided and a comparison between the existing non-SDN Open Flow methods and the proposed solution has been done.

To implement and test the provided theory, first, Mininet emulator - which works based on Linux operating system - will be used. A simple SDN OpenFlow based configuration which includes one POX controller, one OpenFlow supporting switch and four hosts - as shown in Figure 3-1 - will be built to implement behavioral network security on LAN networks. The idea is that there is no need for the controller to inspect every packet. The SDN OpenFlow controller applies distributed communication with switch to detect performance and security problems in the LAN networks.

Finally, traffic generator tools such as **hping3** and **iperf** will be used. The traffic generator will generate traffic with different protocols and rates, then SDN OpenFlow controller will send specific commands to the switch based on the behavioral algorithm, and the switch will deploy firewall flow policy based on the received commands. A better understanding of how this research works will be obtained from the upcoming explanations.

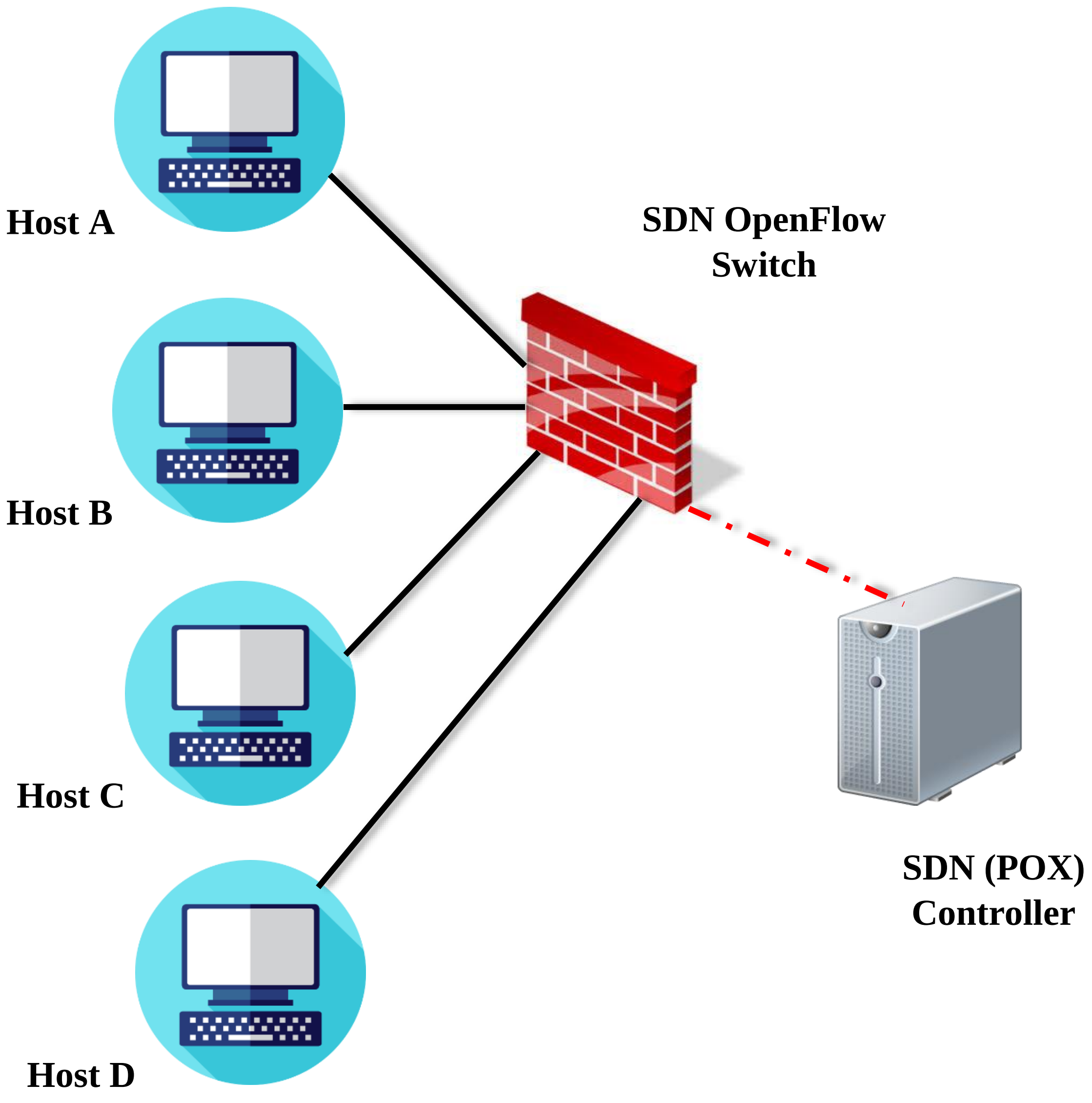


Figure 3-1: Simplified Project Topology

3.2 Research Activities

Figure 3-1 illustrates an overview in general of how this project was convened, in order to bring out the final implementation of a robust firewall using SDN controller and OpenFlow Switch.

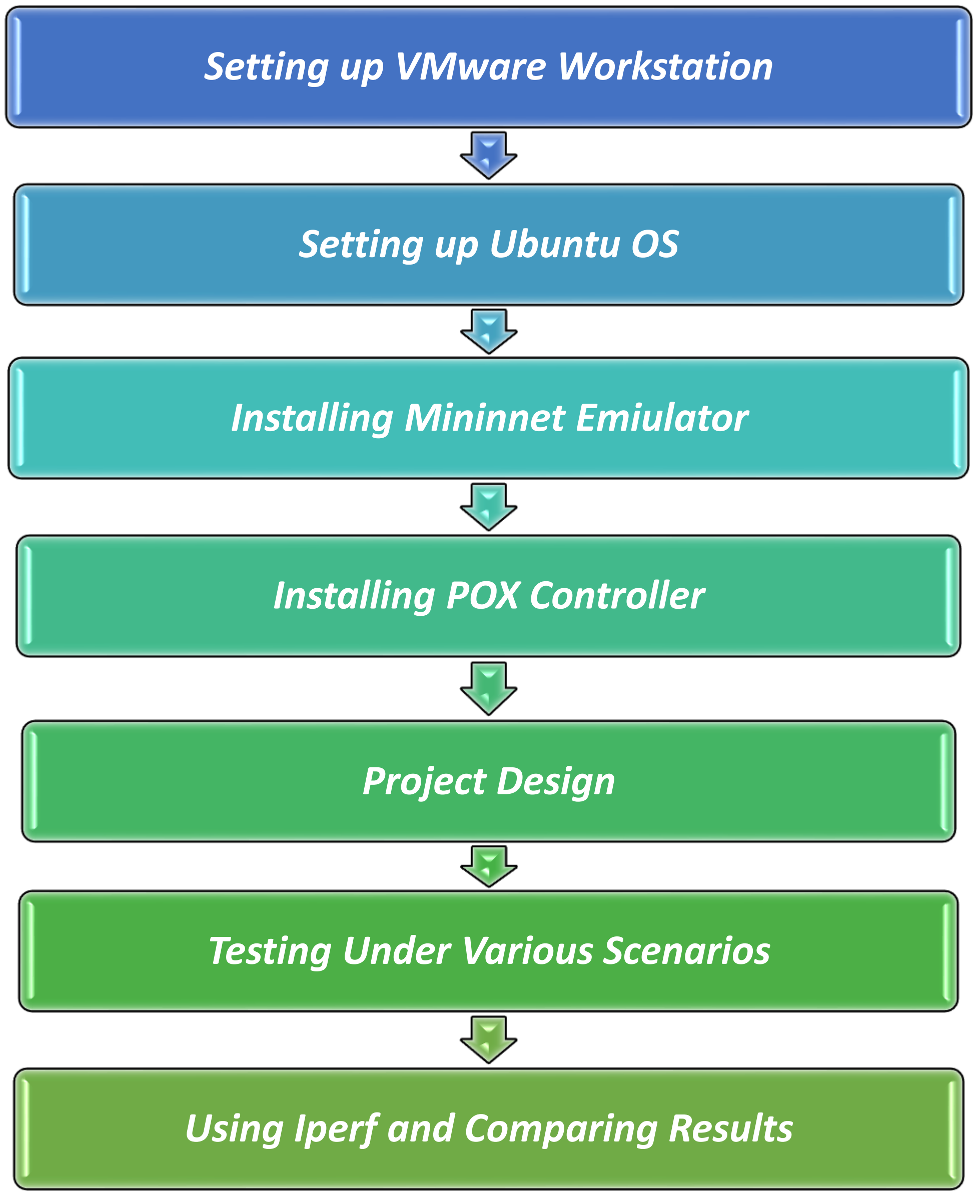

Setting up VMware Workstation
Setting up Ubuntu OS
Installing Mininnet Emiulator
Installing POX Controller
Project Design
Testing Under Various Scenarios
Using Iperf and Comparing Results

Figure 3-2: Flowchart of Research Activities

3.3 Setting up VMware Workstation

VMware Workstation is a hosted hypervisor that runs on x64 versions of Windows and Linux operating systems (an x86 version of earlier releases was available); it enables users to set up virtual machines (VMs) on a single physical machine, and use them simultaneously along with the actual machine. Each virtual machine can execute its own operating system, including versions of Microsoft Windows, Linux, BSD, and MS-DOS [41].

Here in this project, VMware Workstation Pro.lnk was used to run Ubuntu operating system in a physical machine (laptop).

3.4 Setting up Ubuntu OS

Ubuntu is an open source operating system for personal computers and network servers. It is a Linux distribution based on the Debian architecture.

For this project, Ubuntu 14.04 LTS x64 had been chosen to provide an environment for the virtual network to be formed.

3.5 Installing Mininet Emulator

Mininet is a network emulator, or perhaps more precisely a network emulation orchestration system. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code [42].

Mininet supports parametrized topologies. With a few lines of Python code, you can create a flexible topology which can be configured based on the parameters you pass into it, and reused for multiple experiments [42].

After opening the CLI (Command Line Interface), the following commands should be running before starting to install the Mininet.

```
sudo apt-get update
```

This command downloads the package lists from the repositories and updates them to get information on the newest versions of packages and their dependencies. It will do this for all repositories and PPAs. From [43]: Used to re-synchronize the package index files from their sources. The indexes of available packages are fetched from the location(s) specified in /etc/apt/sources.list. An update should always be performed before an upgrade or dist-upgrade [44].

```
sudo apt-get upgrade
```

This Command will fetch new versions of packages existing on the machine if APT knows about these new versions by way of apt-get update.
From [43]: Used to install the newest versions of all packages currently installed on the system from the sources enumerated in /etc/apt/sources.list [44].

```
Sudo apt-get install git
```

Git is a version control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source code management in software development, but it can be used to keep track of changes in any set of files. As a distributed revision control system it is aimed at speed, data integrity, and support for distributed, non-linear workflows [45] [46].

After achieving these prerequisites, the following commands should be done to implement the research test environment:

```
cd /home/

Sudo apt-get install mininet
```

This command used to start installing the Mininet in the home directory.

```
git clone git://github.com/mininet/mininet
```

This command is executed to download a clone of the Mininet files from github.com into the mininet directory that have been created in the previous command.

```
cd /mininet

git tag

git checkout –b cs244-spring-2012-final
```

These commands are used to switch the Mininnet clone to the latest branch (version).

3.6 Installing POX controller

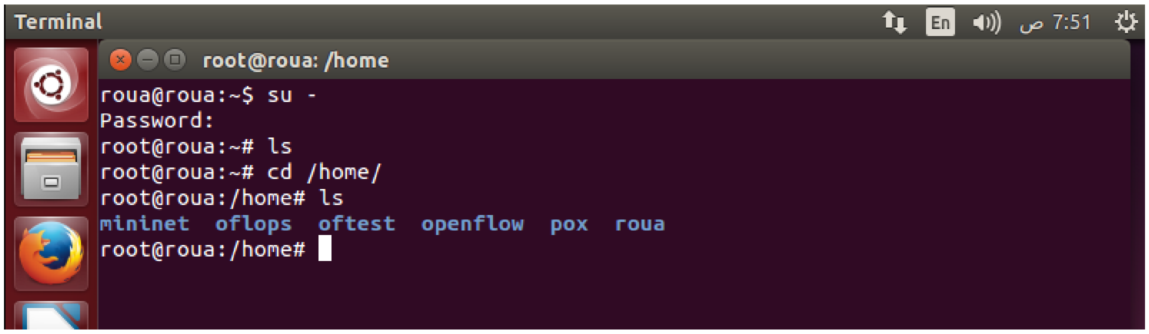
POX controller can be installed by one of the two ways, either cloning it directly from the github.com using the following command:

```
git clone http://github.com/noxrepo/pox
```

Or it can be installed with all components of the Mininet utility that are obtained using the following commands:

```
Cd ..

Mininet/util/install.sh -a
```

Figure 3-3 shows the /home directory with all packages that have been installed.



```
roua@roua:~$ su -
Password:
root@roua:~# ls
root@roua:~# cd /home/
root@roua:/home# ls
mininet  oflops  oftest  openflow  pox  roua
root@roua:/home# 
```

Figure 3-3: Installed Packages

Finally, along with the POX controller, the previous command will install the Wireshark which is a free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education. Or it can be easily installed using the following command:

```
apt-get installed wireshark
```

3.7 Project Design

This project is based on two main parts: the basic configuration and firewall implementation.

3.7.1 Basic Configuration

First stage, a test environment of virtual network of four hosts connected by a switch (OVSK) - as shown in Figure 3-4 - had been built to test basic OpenFlow functionality. To verify the lab functionality, the connectivity between hosts through the virtual switch has been tested using Ping utility. The following commands were used.

```
sodu mn -- topo single,4 -- mac  -- switch ovsk

mininet> pingall
```

In the second stage, the OpenFlow switch was managed with POX controller. Running POX controller with switch module to hub module and vice versa. This test provides information about how to manage OpenFlow switch, control traffic routing and deploy flows to the switch. The following commands were used.

```
./pox.py  log.level  -- DEBUG  forwarding.hub
```

```
./pox.py  log.level  -- DEBUG  forwarding.l2_learning

sodu mn -- topo single,4 -- mac  -- switch ovsk -- controller
remote
```

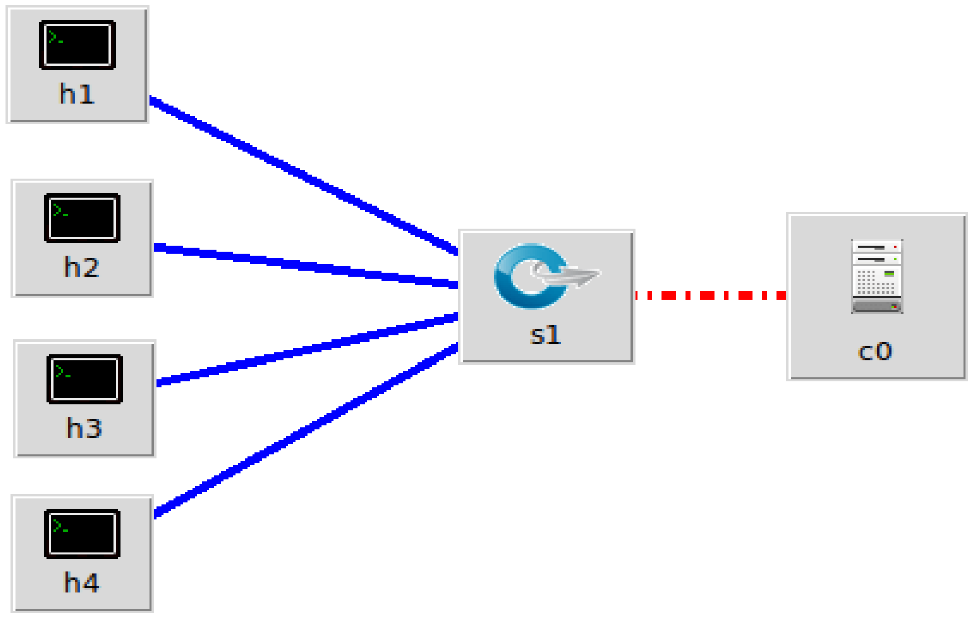


Figure 3-4: Project Topology in Mininet

3.7.2 Firewall Implementation

To implement firewall functionality, threat detection, traffic filtering and decision-making mechanisms have been added to POX controller. POX is a Python-based SDN controller platform geared towards research and education and allows flexibility in development. To identify malicious activities, a POX based firewall has been developed that will notice any abnormal network behavior in the virtual network.

With a listing of different security concerns in Software Defined Networks, one of the main security threats were concentrated upon in this research work is on Denial-Of-Service. When a large number of packets are forwarded to a network device with an intention to either stop the service or affect the performance then such attacks are termed as Denial-of-service attacks. This kind of attacks can be detected at an early stage by monitoring few of packets based on the entropy changes. By applying entropy as a detection method, it could be able to detect attacks on one host or a subnet of hosts in a network and prevents the controller going down.

Entropy is the randomness collected by an operating system or application for use in cryptography or other uses that require random data. The main reason for choosing entropy is its ability to measure randomness in a network.

If assumed that **W** is a set of data with **n** elements, and **X** is an event in the set, then, the probability **Pi** of **X** can be calculated using the following equation:

$$Pi = \frac{Xi}{n} \qquad 3.1$$

Where **Xi** is one of the elements in **W** that is represented by equation 3.2**:**

$$W = \{X1, X2, X3, \dots\dots\dots\dots\dots, Xn\} \qquad 3.2$$

The size of **W** is called the **window size**.

To measure the entropy, referred to as **H**, the probability of all elements in the set were calculated and gathered as shown in equation 3.3:

$$H = \sum_{i=1}^{n} Pi \log Pi \qquad 3.3$$

The entropy will be at its maximum if all elements have equal probabilities. If an element appears more than others, the entropy will be lower.

So, after creating this firewall module, it can be executed using the following commands.

```
./pox.py  log.level  -- DEBUG  forwarding.POXFW1

sodu mn -- topo single,4 -- mac  -- switch ovsk -- controller
remote
```

To show the flexibility of developing a POX controller, second module has been built as a firewall which can identify ICMP traffic to the specific destination IP addresses 10.0.0.2 and 10.0.0.3. POX controller sends specific rules to the OpenFlow switch to deny the specific flows from the predefined hosts (h2 and h3) and allow them for other hosts.

Also, this module can filter all TCP packets and identify those with destination IP address 10.0.0.4 and deny them while allowing all other TCP packets to other hosts.

Finally, all UDP packets with destination IP address 10.0.0.1 will be dropped while other UPD packets will be switched to their destination hosts. After implementing this module, it can be executed using the following commands.

```
./pox.py  log.level  -- DEBUG  forwarding.POXFW2

sodu mn -- topo single,4 -- mac  -- switch ovsk -- controller
remote
```

CHAPTER FOUR

RESULTS AND DISCUSSIONS

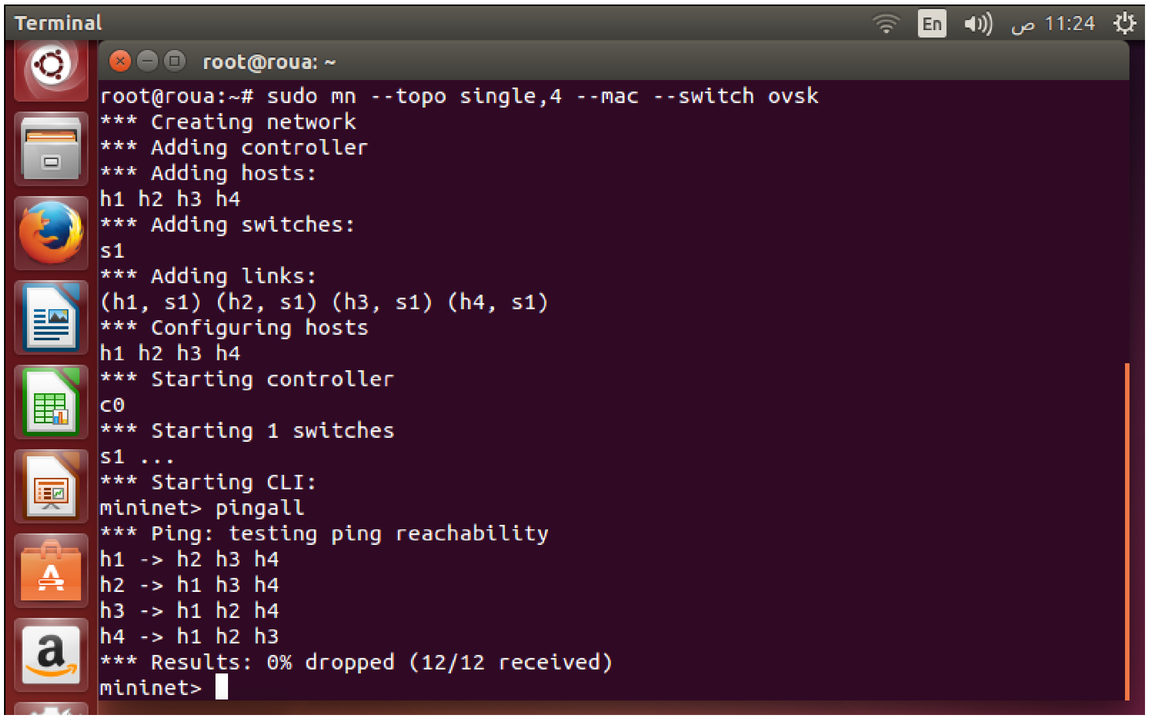
CHAPTER FOUR

RESULTS AND DISCUSSIONS

In this chapter, the POX controller will be tested under various scenarios and all the results will be captured and discussed. Also, different performance tests will be done.

4.1 Mininet without POX Controller

Figure 4-1 shows the Mininet devices that have been created and used in this project and the result of the connectivity test which made using Ping utility. This Mininet has been created without managing the OpenFlow switch with a POX controller.



```
root@roua:~# sudo mn --topo single,4 --mac --switch ovsk
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet>
```

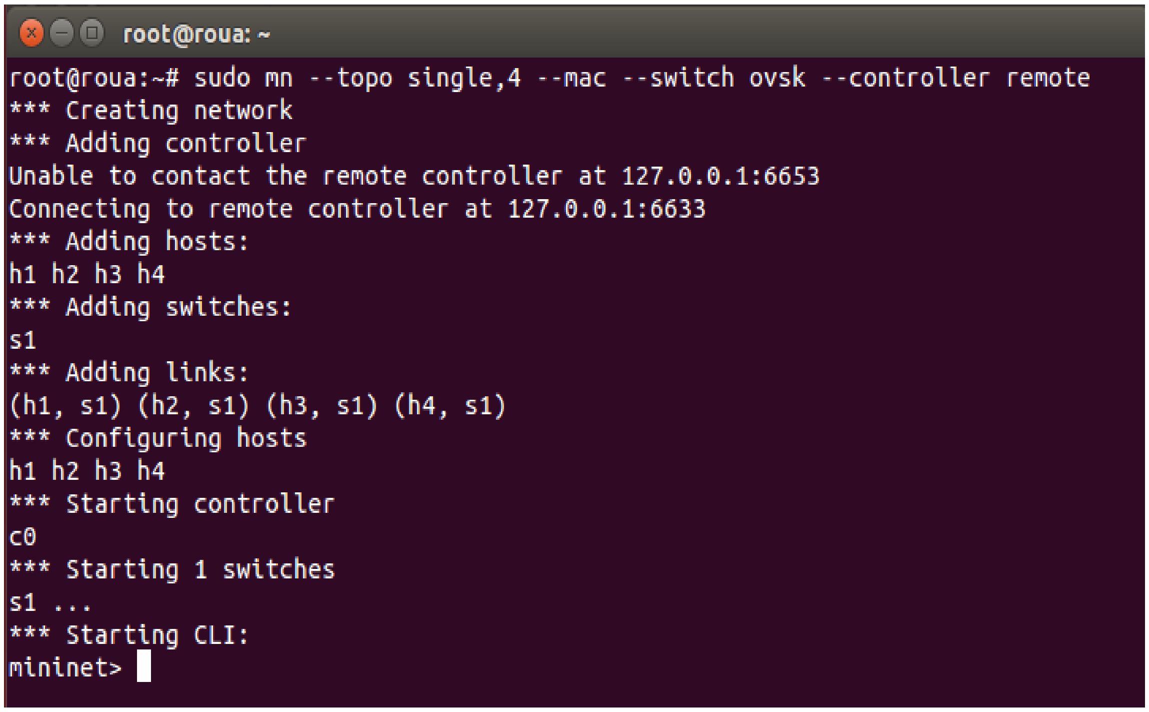
Figure 4-1: Mininet Devices and Connectivity Test Result

4.2 Mininet with POX Controller

POX controller could be used with different modules to perform several types of network devices tasks. In this research work some modules will be undertaken.

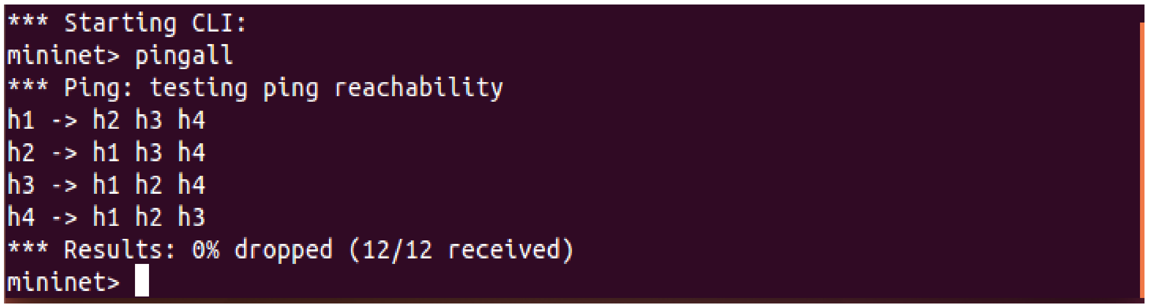
4.2.1 POX controller Running Hub Module / Switch Module

Figure 4-2 shows the Mininet topology that is configured to be managed by a POX controller which will be operated with a hub module and a switch module respectively. Also in both cases the connectivity is up between all hosts and all of them can ping each other without any losses as illustrated in Figure 4-3. If the Mininet would be managed with a POX controller and no one of the connecting modules would be operated, the connectivity between all hosts will be lost as show in Figure 4-4.



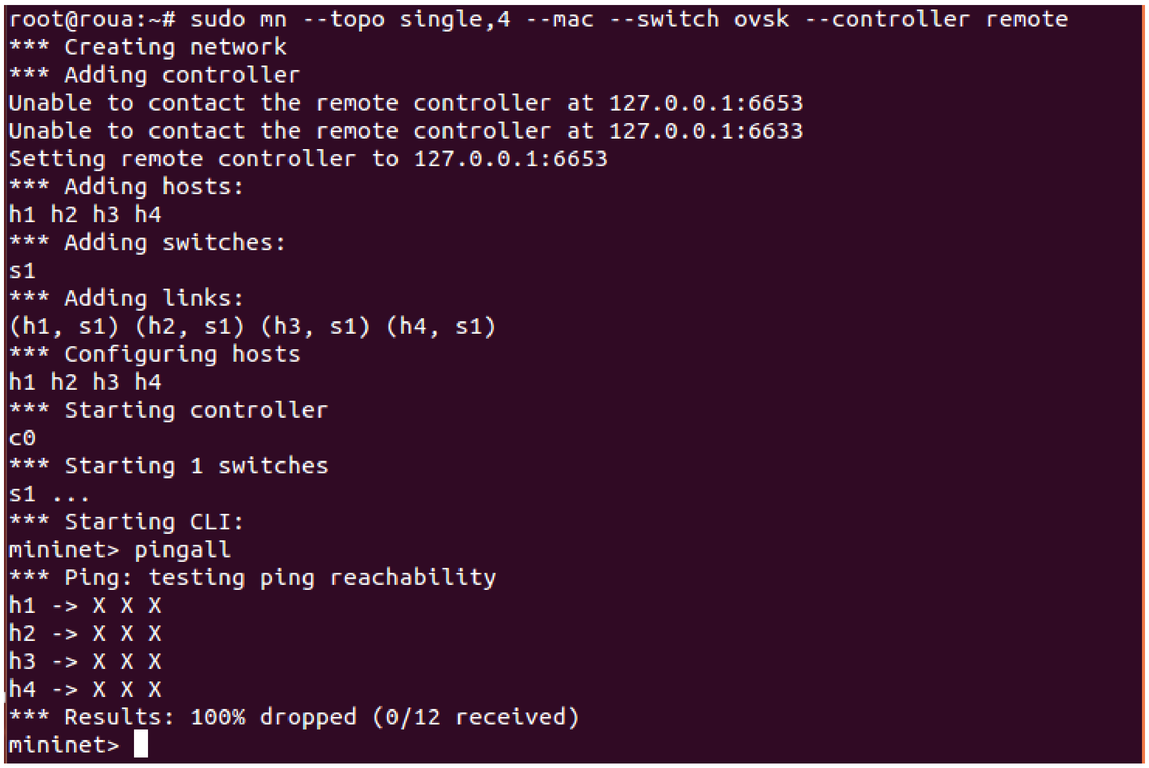
```
root@roua:~# sudo mn --topo single,4 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Figure 4-2: Mininet Devices Managed by POX controller



```
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet>
```

Figure 4-3: Connectivity Test Using Ping tool (With connecting Module)



```
root@roua:~# sudo mn --topo single,4 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X X
h4 -> X X X
*** Results: 100% dropped (0/12 received)
mininet>
```

Figure 4-4: Connectivity Test Using Ping tool (Without connecting Module)

4.2.2 POX controller Running Firewall Module

In this research work, two emulation scenarios have been conducted. These scenarios are explained in detail - including their working algorithms and test results - in the upcoming sections.

4.2.2.1 First Emulation Scenario

A POX based Firewall has been implemented which is able to detect any DoS attack and take an action against this serious situation in the virtual network based on the entropy value. Entropy can be calculated by certain equations that are configured in the Firewall script (Python_based code). The algorithm of this code is illustrated in Figure 4-5.

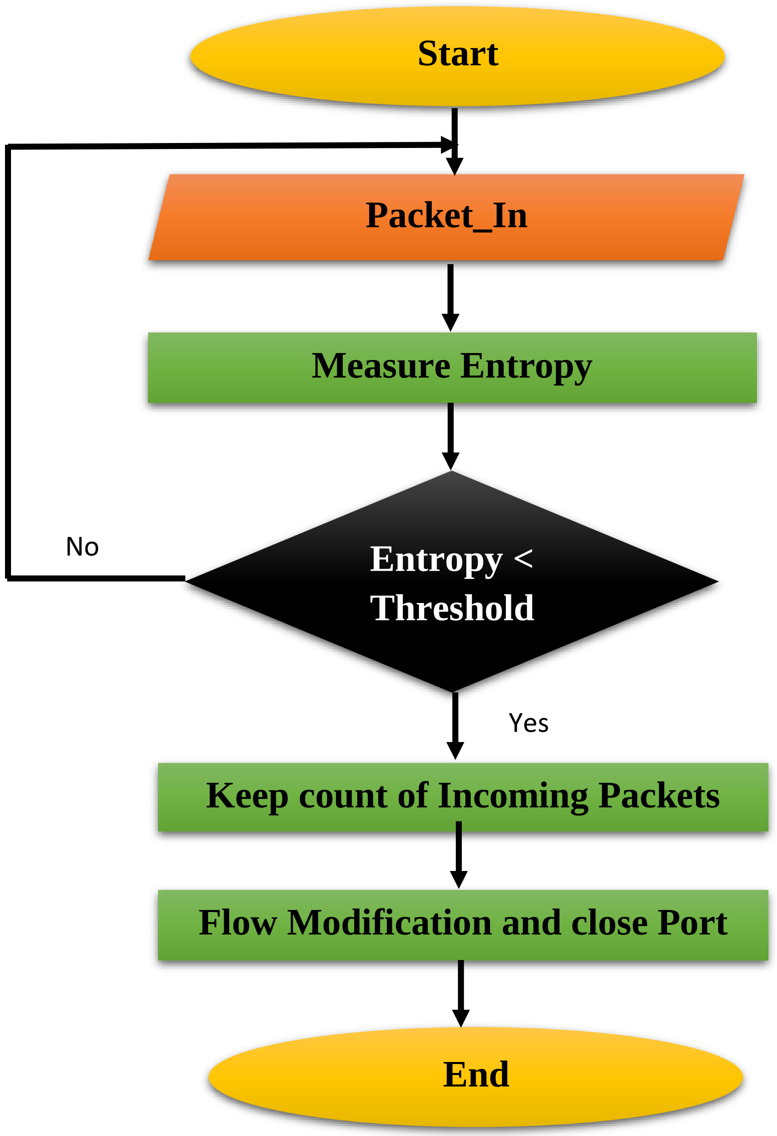
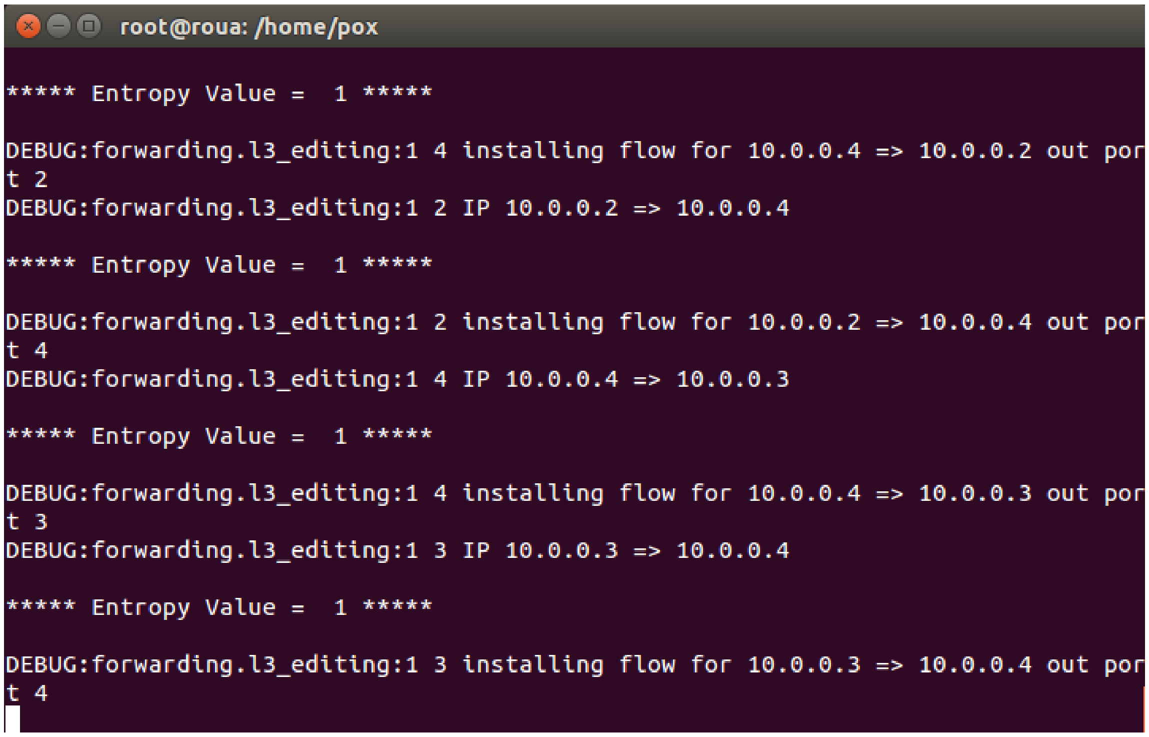

Start
Packet_In
Measure Entropy
No
Entropy < Threshold
Yes
Keep count of Incoming Packets
Flow Modification and close Port
End

Figure 4-5: First POXFW Algorithm

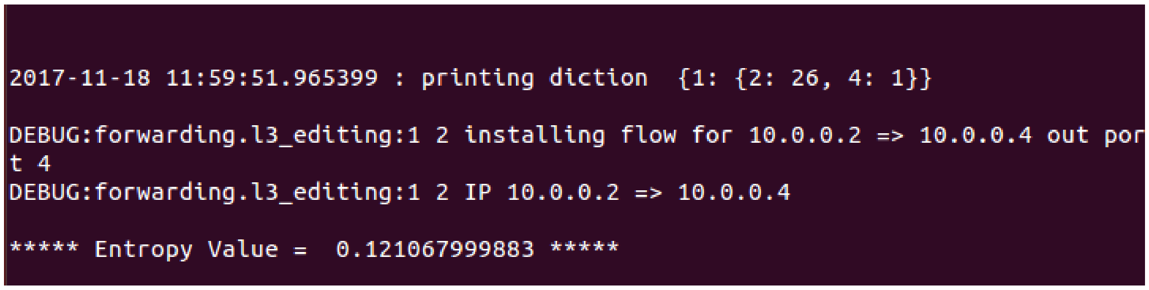
When POX controller running with this module and using **pingall** to make sure that the connectivity is up between hosts, it can be seen that the Entropy value equal one which is - normal flood of traffic - as appears in Figure 4-6.



```
***** Entropy Value =  1 *****

DEBUG:forwarding.l3_editing:1 4 installing flow for 10.0.0.4 => 10.0.0.2 out por
t 2
DEBUG:forwarding.l3_editing:1 2 IP 10.0.0.2 => 10.0.0.4

***** Entropy Value =  1 *****

DEBUG:forwarding.l3_editing:1 2 installing flow for 10.0.0.2 => 10.0.0.4 out por
t 4
DEBUG:forwarding.l3_editing:1 4 IP 10.0.0.4 => 10.0.0.3

***** Entropy Value =  1 *****

DEBUG:forwarding.l3_editing:1 4 installing flow for 10.0.0.4 => 10.0.0.3 out por
t 3
DEBUG:forwarding.l3_editing:1 3 IP 10.0.0.3 => 10.0.0.4

***** Entropy Value =  1 *****

DEBUG:forwarding.l3_editing:1 3 installing flow for 10.0.0.3 => 10.0.0.4 out por
t 4
```

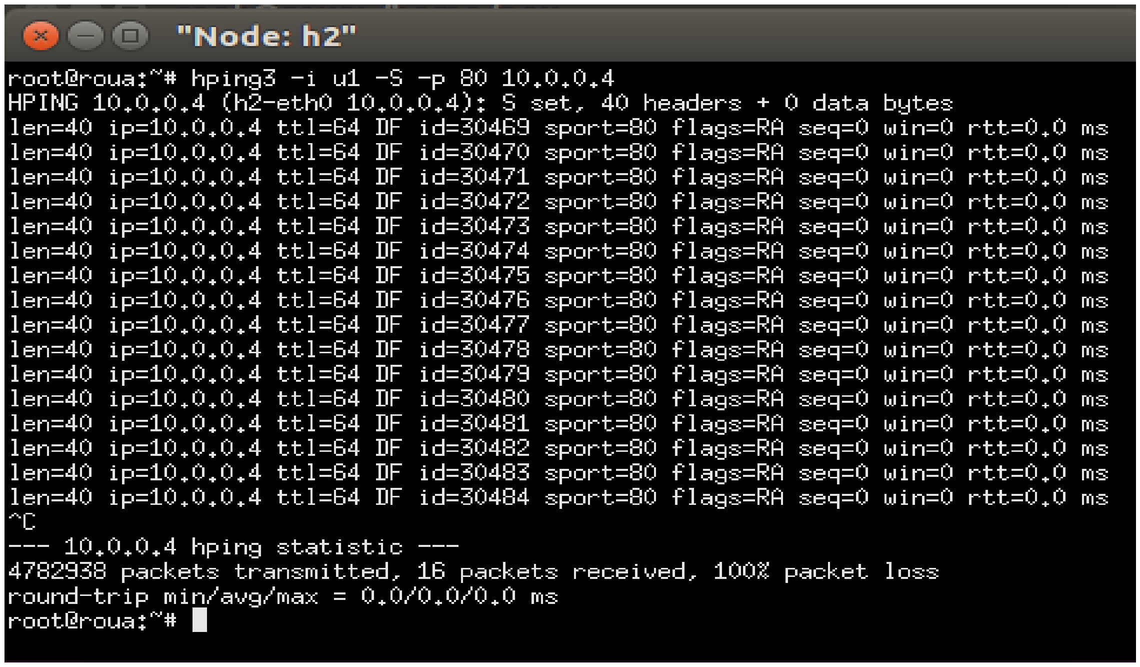
Figure 4-6: Entropy Value before DoS Attack

To test the efficiency of the developed firewall against DoS attack, **hping3** tool has been used to generate a SYN flood from h2 (10.0.0.2) on port 80 of h4 (10.0.0.4) and it is clear from Figure 4-7 that that the Entropy value has been decreased and thus led to stopping of this malicious activity after 16 packets only from 4782938 packets as shown in Figure 4-8.



```
2017-11-18 11:59:51.965399 : printing diction  {1: {2: 26, 4: 1}}

DEBUG:forwarding.l3_editing:1 2 installing flow for 10.0.0.2 => 10.0.0.4 out por
t 4
DEBUG:forwarding.l3_editing:1 2 IP 10.0.0.2 => 10.0.0.4

***** Entropy Value =  0.121067999883 *****
```

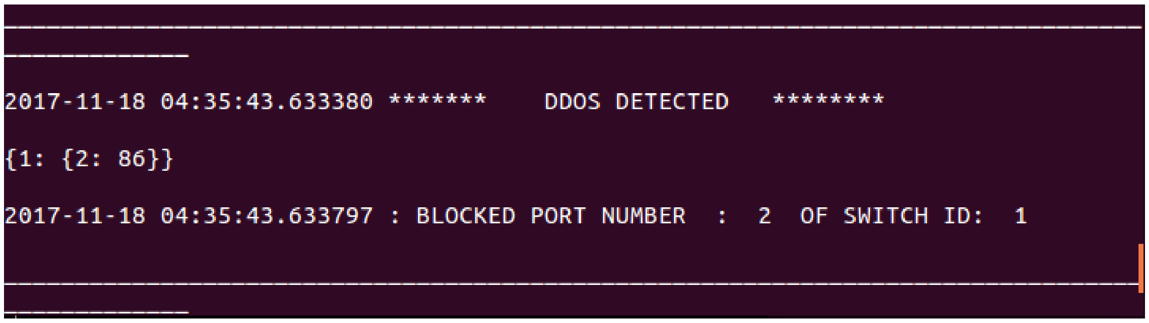
Figure 4-7: Entropy Value after DoS Attack



```
root@roua:~# hping3 -i u1 -S -p 80 10.0.0.4
HPING 10.0.0.4 (h2-eth0 10.0.0.4): S set, 40 headers + 0 data bytes
len=40 ip=10.0.0.4 ttl=64 DF id=30469 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30470 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30471 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30472 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30473 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30474 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30475 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30476 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30477 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30478 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30479 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30480 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30481 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30482 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30483 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
len=40 ip=10.0.0.4 ttl=64 DF id=30484 sport=80 flags=RA seq=0 win=0 rtt=0.0 ms
^C
--- 10.0.0.4 hping statistic ---
4782938 packets transmitted, 16 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@roua:~#
```

Figure 4-8: POXFW Stopping DoS Attack

Also, the POX sends command to the OpenFlow switch to block the malicious port (port of h2) until a further notice as illustrated in Figure 4-9



```
--------------------------------------------------------------------------------
-------------

2017-11-18 04:35:43.633380 *******    DDOS DETECTED    ********

{1: {2: 86}}

2017-11-18 04:35:43.633797 : BLOCKED PORT NUMBER  :  2  OF SWITCH ID:  1

--------------------------------------------------------------------------------
-------------
```

Figure 4-9: POXFW Blocked Port 2

4.2.2.2 Second Emulation Scenario

To identify malicious activities, a POX based Firewall has been devolved that will notice any abnormal network behavior in the virtual network based on the predefined rules and policies that have been configured in the Firewall script (Python_based code). The algorithm of this code is illustrated in Figure 4-10.

When testing the connectivity between all hosts in the Mininet using **pingall** command, the results shows that all ICMP packets sent to h2 10.0.0.2 and h3 10.0.0.3 have been lost due to deny policy which configured in the POX Firewall code and applied by the OpenFlow switch. Figure 4-11 shows the losses that have been occured after applying the Firewall module.

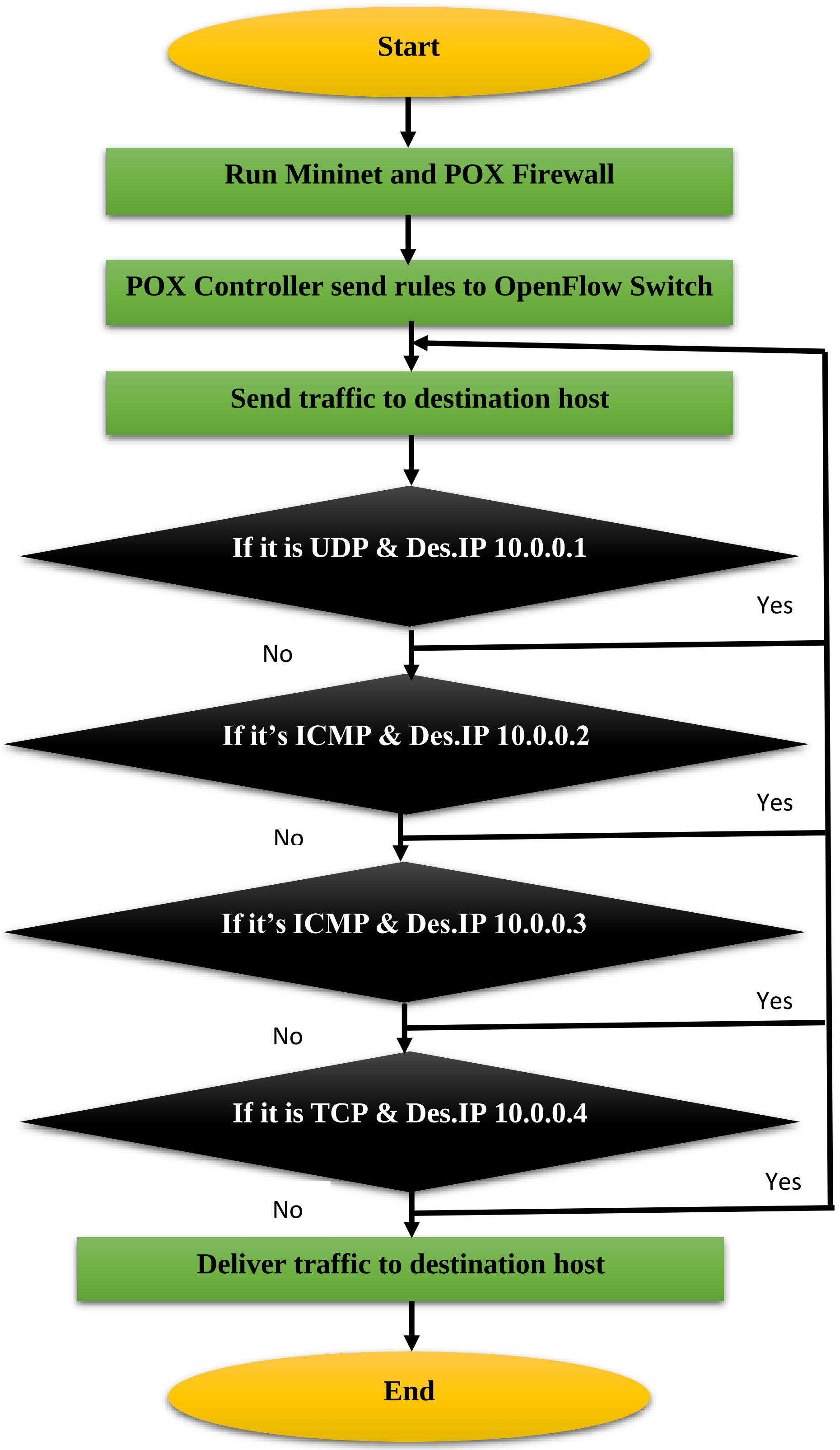
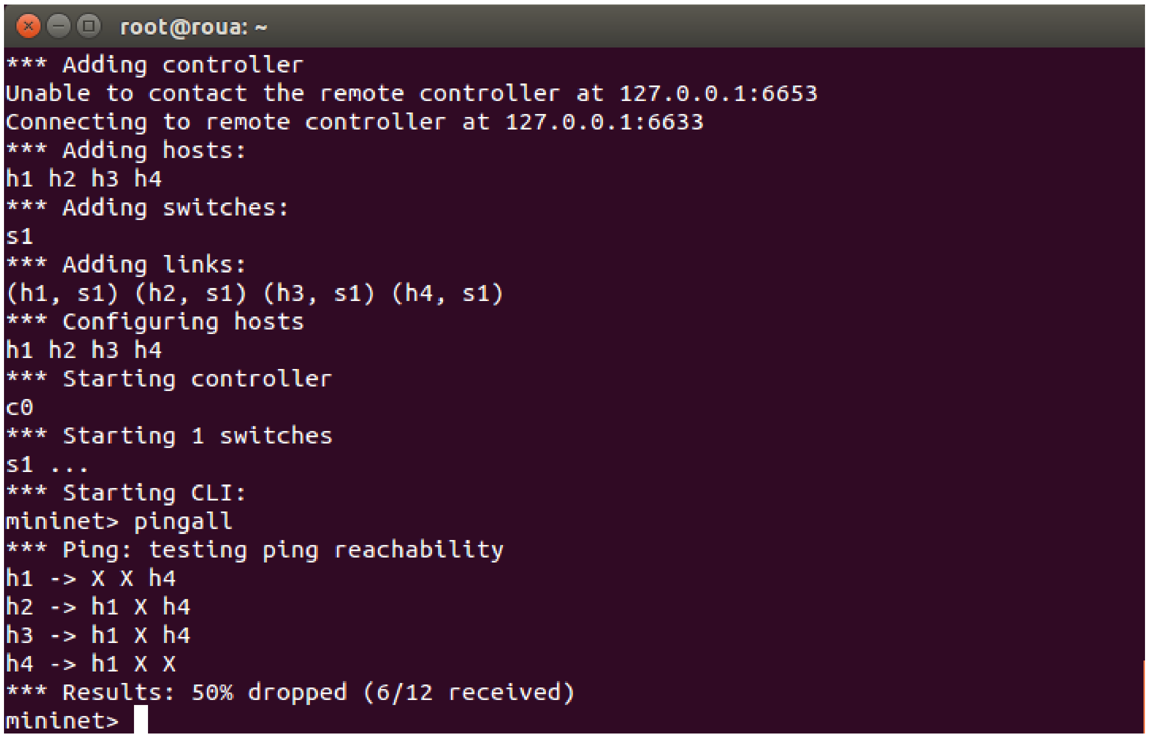

Start
Run Mininet and POX Firewall
POX Controller send rules to OpenFlow Switch
Send traffic to destination host
If it is UDP & Des.IP 10.0.0.1
Yes
No
If it’s ICMP & Des.IP 10.0.0.2
Yes
No
If it’s ICMP & Des.IP 10.0.0.3
Yes
No
If it is TCP & Des.IP 10.0.0.4
Yes
No
Deliver traffic to destination host
End

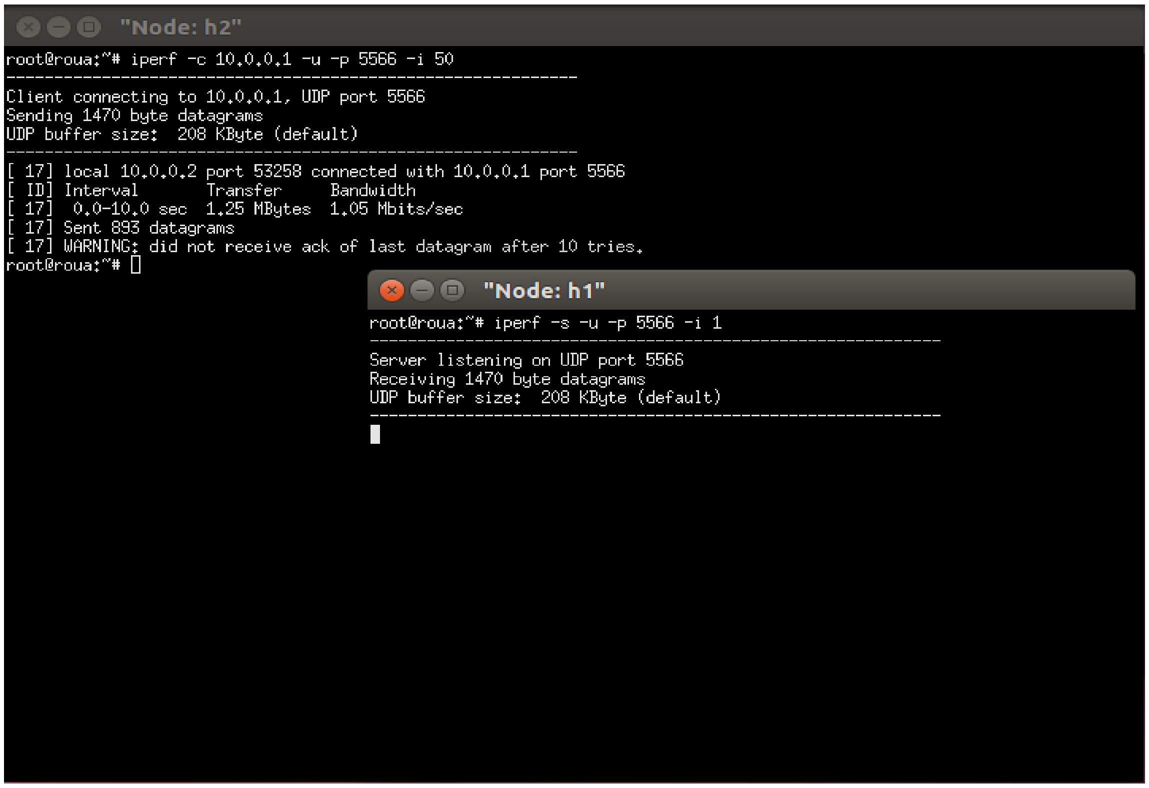
Figure 4-10: Second POXFW Algorithm



```
root@roua: ~
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X h4
h2 -> h1 X h4
h3 -> h1 X h4
h4 -> h1 X X
*** Results: 50% dropped (6/12 received)
mininet>
```

Figure 4-11: POXFW Blocking ICMP Packets From h1 and h2

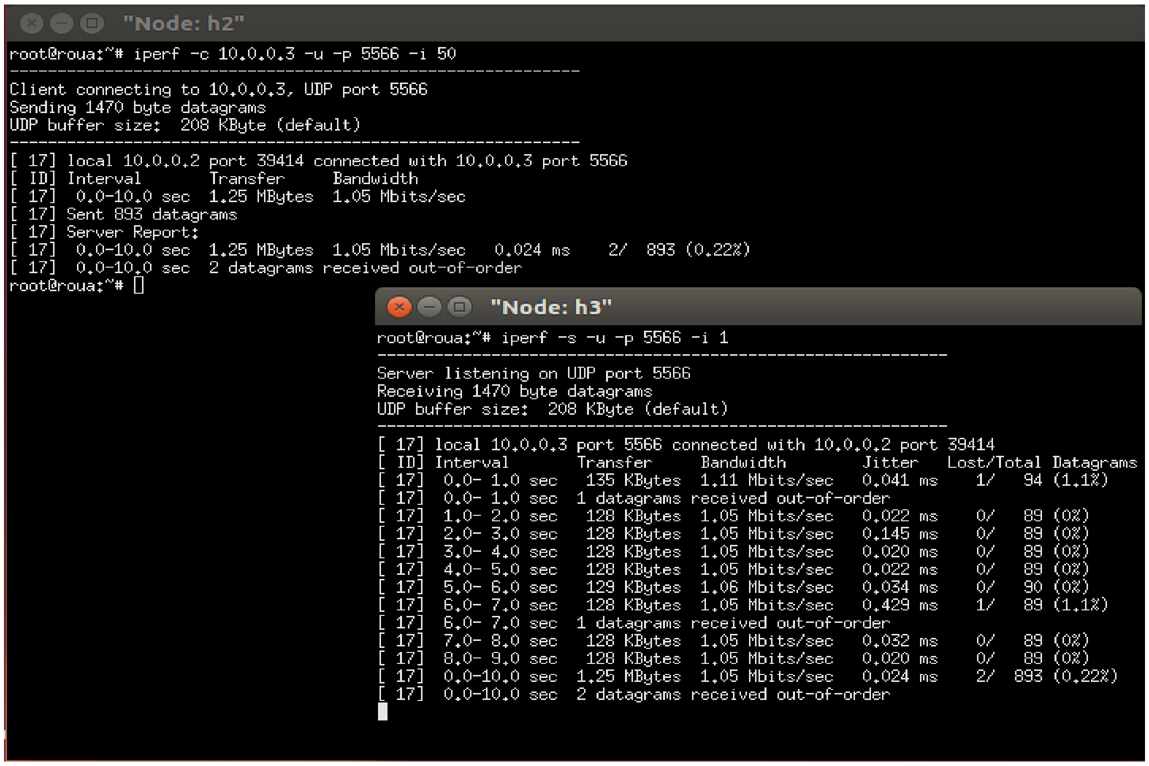
Figure 4-12 and Figure 4-13 illustrate the OpenFlow switch behaviore when UDP traffic flows in the network. The OpenFlow switch blocking any UDP packet that has been sent to h1 10.0.0.1 and switch it to any other host in the Mininet. To generate a stream of UDP packets, **iperf** tool has been installed and used. It is used as a client on one host and as a server on the other host to send parallel UDP streams.



```
root@roua:~# iperf -c 10.0.0.1 -u -p 5566 -i 50
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5566
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 17] local 10.0.0.2 port 53258 connected with 10.0.0.1 port 5566
[ ID] Interval       Transfer     Bandwidth
[ 17]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 17] Sent 893 datagrams
[ 17] WARNING: did not receive ack of last datagram after 10 tries.
root@roua:~# []
```

```
root@roua:~# iperf -s -u -p 5566 -i 1
------------------------------------------------------------
Server listening on UDP port 5566
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
```

Figure 4-12: POXFW Blocking UDP Packets to h1

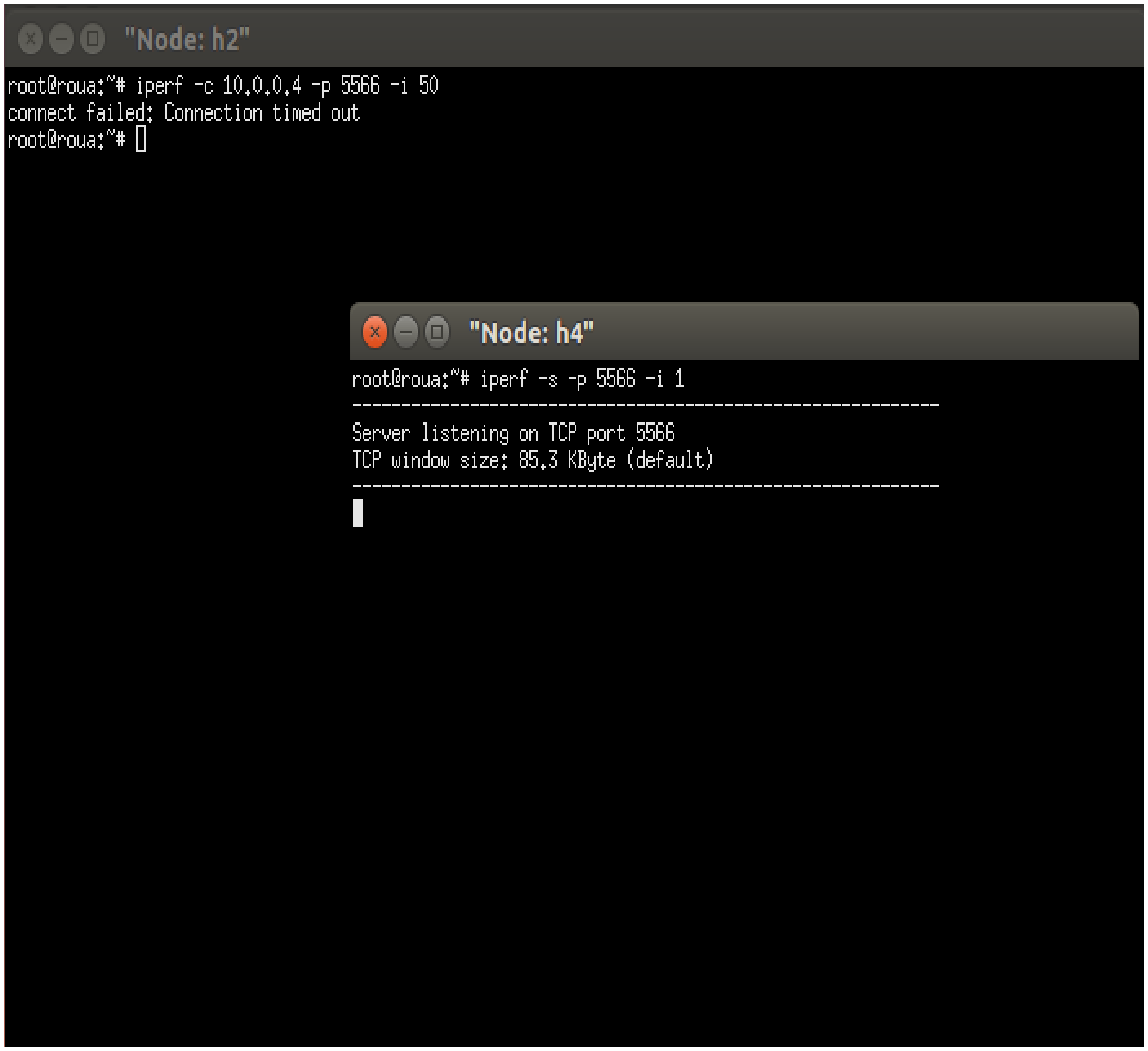


```
root@roua:~# iperf -c 10.0.0.3 -u -p 5566 -i 50
------------------------------------------------------------
Client connecting to 10.0.0.3, UDP port 5566
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 17] local 10.0.0.2 port 39414 connected with 10.0.0.3 port 5566
[ ID] Interval       Transfer     Bandwidth
[ 17]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 17] Sent 893 datagrams
[ 17] Server Report:
[ 17]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.024 ms    2/  893 (0.22%)
[ 17]  0.0-10.0 sec  2 datagrams received out-of-order
root@roua:~# []
```

```
root@roua:~# iperf -s -u -p 5566 -i 1
------------------------------------------------------------
Server listening on UDP port 5566
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 17] local 10.0.0.3 port 5566 connected with 10.0.0.2 port 39414
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[ 17]  0.0- 1.0 sec   135 KBytes  1.11 Mbits/sec   0.041 ms    1/   94 (1.1%)
[ 17]  0.0- 1.0 sec  1 datagrams received out-of-order
[ 17]  1.0- 2.0 sec   128 KBytes  1.05 Mbits/sec   0.022 ms    0/   89 (0%)
[ 17]  2.0- 3.0 sec   128 KBytes  1.05 Mbits/sec   0.145 ms    0/   89 (0%)
[ 17]  3.0- 4.0 sec   128 KBytes  1.05 Mbits/sec   0.020 ms    0/   89 (0%)
[ 17]  4.0- 5.0 sec   128 KBytes  1.05 Mbits/sec   0.022 ms    0/   89 (0%)
[ 17]  5.0- 6.0 sec   129 KBytes  1.06 Mbits/sec   0.034 ms    0/   90 (0%)
[ 17]  6.0- 7.0 sec   128 KBytes  1.05 Mbits/sec   0.429 ms    1/   89 (1.1%)
[ 17]  6.0- 7.0 sec  1 datagrams received out-of-order
[ 17]  7.0- 8.0 sec   128 KBytes  1.05 Mbits/sec   0.032 ms    0/   89 (0%)
[ 17]  8.0- 9.0 sec   128 KBytes  1.05 Mbits/sec   0.020 ms    0/   89 (0%)
[ 17]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.024 ms    2/  893 (0.22%)
[ 17]  0.0-10.0 sec  2 datagrams received out-of-order
```

Figure 4-13: POXFW Allowing UDP Packets to Other hosts

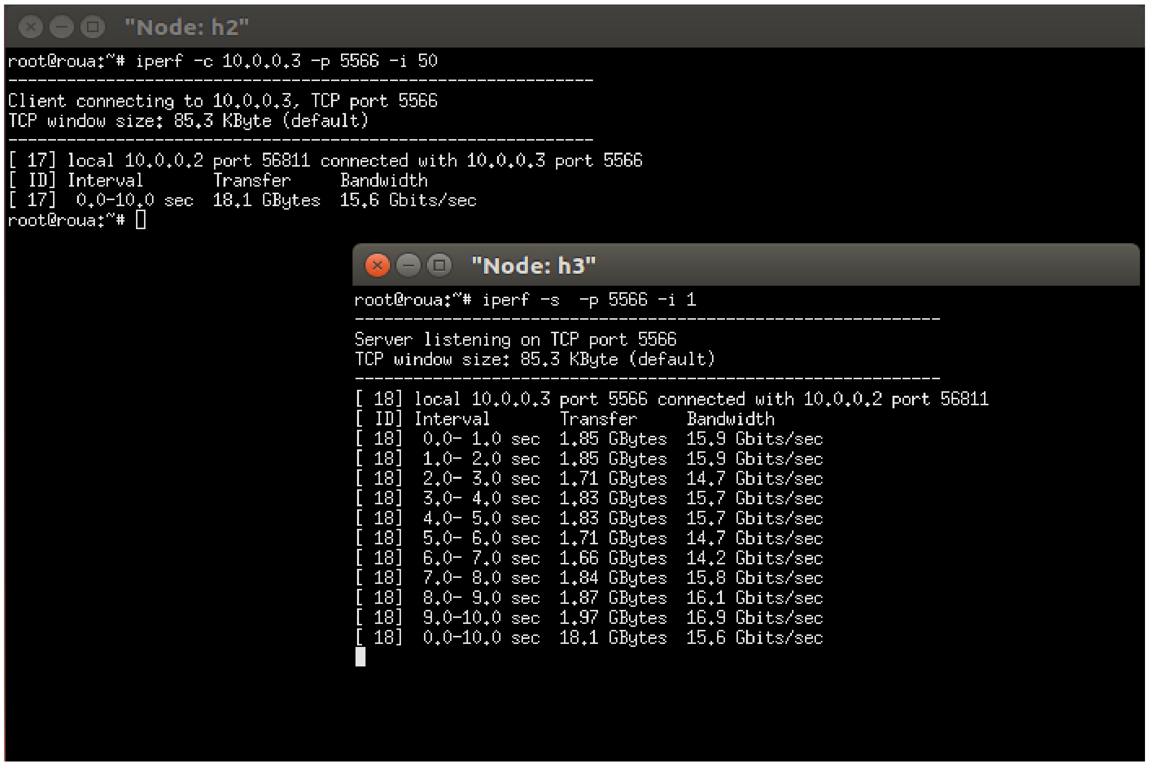
Figure 4-14 shows that any TCP packets sent to h4 10.0.0.4 have been blocked by the OpenFlow switch after applying the Firewall module while other TCP packets that are sending to any other hosts in the Mininet will be delivered to their destination as appears in Figure 4-15. Also in this test, **iperf** tool has been used to generate a stream of TCP packets.



```
root@roua:~# iperf -c 10.0.0.4 -p 5566 -i 50
connect failed: Connection timed out
root@roua:~#
```

```
root@roua:~# iperf -s -p 5566 -i 1
------------------------------------------------------------
Server listening on TCP port 5566
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

Figure 4-14: POXFW Blocking TCP Packets to h4



```
root@roua:~# iperf -c 10.0.0.3 -p 5566 -i 50
------------------------------------------------------------
Client connecting to 10.0.0.3, TCP port 5566
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 17] local 10.0.0.2 port 56811 connected with 10.0.0.3 port 5566
[ ID] Interval       Transfer     Bandwidth
[ 17]  0.0-10.0 sec  18.1 GBytes  15.6 Gbits/sec
root@roua:~# 
```

```
root@roua:~# iperf -s  -p 5566 -i 1
------------------------------------------------------------
Server listening on TCP port 5566
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 18] local 10.0.0.3 port 5566 connected with 10.0.0.2 port 56811
[ ID] Interval       Transfer     Bandwidth
[ 18]  0.0- 1.0 sec  1.85 GBytes  15.9 Gbits/sec
[ 18]  1.0- 2.0 sec  1.85 GBytes  15.9 Gbits/sec
[ 18]  2.0- 3.0 sec  1.71 GBytes  14.7 Gbits/sec
[ 18]  3.0- 4.0 sec  1.83 GBytes  15.7 Gbits/sec
[ 18]  4.0- 5.0 sec  1.83 GBytes  15.7 Gbits/sec
[ 18]  5.0- 6.0 sec  1.71 GBytes  14.7 Gbits/sec
[ 18]  6.0- 7.0 sec  1.66 GBytes  14.2 Gbits/sec
[ 18]  7.0- 8.0 sec  1.84 GBytes  15.8 Gbits/sec
[ 18]  8.0- 9.0 sec  1.87 GBytes  16.1 Gbits/sec
[ 18]  9.0-10.0 sec  1.97 GBytes  16.9 Gbits/sec
[ 18]  0.0-10.0 sec  18.1 GBytes  15.6 Gbits/sec
```

Figure 4-15: POXFW Allowing TCP Packets to Other hosts

4.3 Performance Results

In this section, after firewall implementation and testing, a performance evaluation has been done. It compares the performance of the Firewall module with the performance of switch module and the traditional switch (without POX controller). **IPERF** which is a tool for network performance measurement, is used to analyze the performance.

In the first test, **IPERF** tool is used as a client on one host and as a server on the other host to generate a stream of UDP packets. Each time after running **IPERF** tool, the controller process is killed and restarted to make sure that the OpenFlow switch is ready for the next test and has no flow entries. Figure 4-16 shows the test results of multiple parallel UDP streams and different modes. It is obvious from Figure 4-16 that the POX controller running with a Firewall module (the yellow line) introduces extra delay to the UDP streams compared with the traditional switch (without POX) and the case when POX controller running with switch module (the blue line).

The reason is that the UDP implementation of the firewall module expects a TCP-like handshake over UDP, which is not the case in IPERF, which sends UDP packets in just one direction (client to server). As a result, all the UDP packets will be forwarded to the controller, which has an upper limit for packets coming from one direction to prevent a DOS attack against itself. In addition, sending a flow entry to the switch to allow such behavior will create a security hole. For example, an attacker might send multiple UDP packets from one side in order to bypass the controller and then start sending malicious traffic. In the end, it is a trade-off between performance and security. So, it can be said that the firewall module suffers from a poor performance for UDP packets due to its security policy.

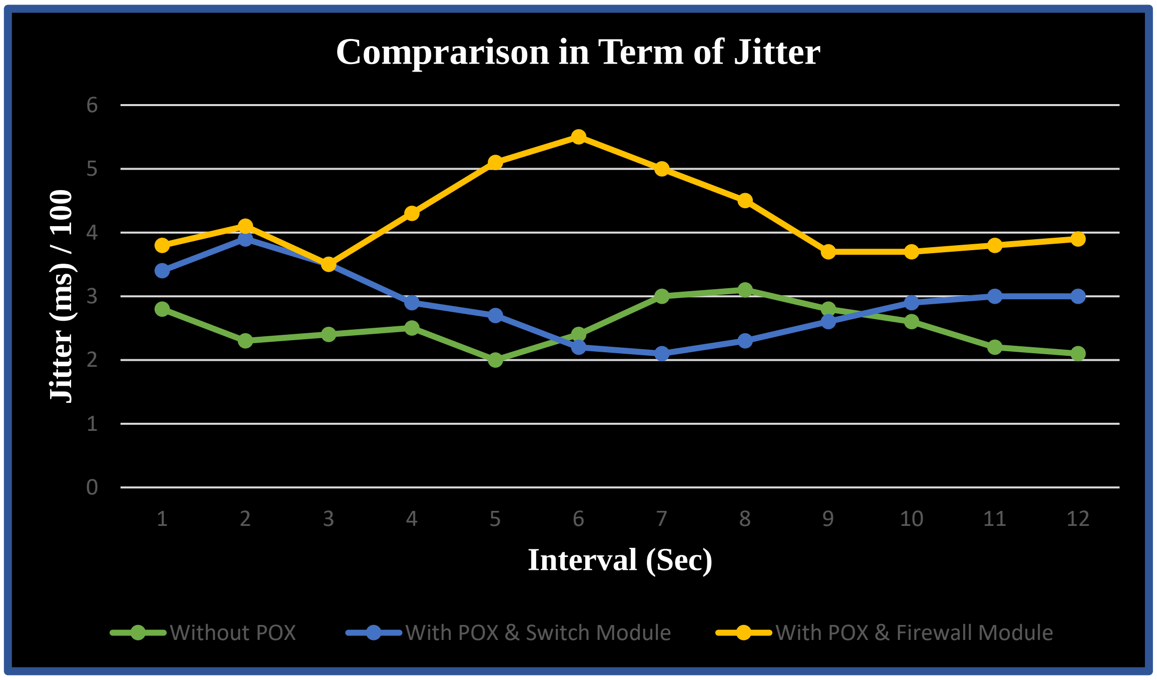


Figure 4-16 : Comparison in Term of Jitter

In the second test, **IPERF** tool is used to send parallel TCP streams. The throughput is calculated for traditional switch (the green line), switch module (the blue line) and the firewall module (the yellow line), as shown in Figure 4-17.This test proves that the switch module that is running with the POX controller has a good TCP performance compared to the traditional switch and that is due to the separation of data plane from control plane which enhancing the device ability to perform its job without caring about decision making which is the responsibility of the POX controller.

Also, it can be noticed that switch module throughput is much better than firewall module and this difference in throughput is due to the difference in processing time and security policies that are applied by the OpenFlow switch. For example, it takes time for the Openflow switch to process a new flow before sending the packet out to the controller (for the first four TCP packets only from each stream), and it takes time for the controller to process the packet and send commands back to the OpenFlow switch.

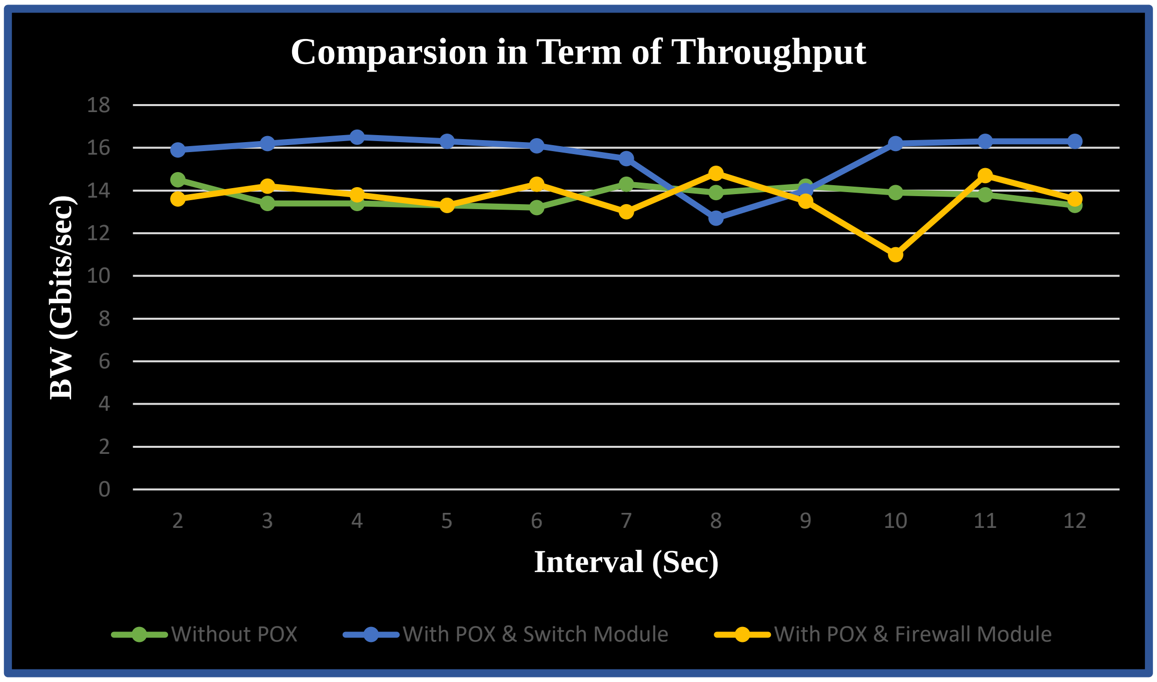


Figure 4-17 : Comparison in Term of Throughput

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

In this chapter, the conclusion of this research work has been mentioned with some recommendations for the future work.

5.1 Conclusion

Firewall devices are fundamental in any network to apply security. Despite of the remarkable firewalls that are available in the market, these firewalls are expensive, their provisioning time is high, and most of them do not have user friendly interface or provide programmability to the network administrators. On the other hand, SDN technology allows the use of commodity hardware, reduces provisioning time, and provides a huge flexibility in programming the control plane.

This research implemented an OpenFlow-based firewall module running with POX controller and capable of detecting and preventing DOS attack and any parallel streams of traffic such as TCP and UDP that could affect or disrupt the performance of network devices.

The performance analysis tests that have been done proved that the firewall module performed well in handling TCP traffic compared with a traditional switch while switch module provides much better throughput than traditional switch and firewall module. On the other hand, some limitations related to firewall module appears in terms of jitter delay and suffering from poor performance of handling UDP packets due to security policy comparing to traditional switch and switch module. So, the trade-off between the security and performance is inevitable in any kind of network architecture.

5.2 Recommendations

After the research work has been finished, some recommendations and research issues can be provided for who wants to carry on from this point. Future work on this topic can include:

- One of the further development of the provided idea could be include the implementation of the self-study firewall that could dynamically identify abnormal network activities and conventional traffic.
- Another important part, is the development of Graphical User Interface (GUI) for POX controller and for modules such as switch module and Firewall Module.
- This research implementation cannot detect encrypted traffic. So, the future work could focus on this area as there are ways to analyze encrypted traffic such as packet size, direction, and timing.
- Currently, this project design only looking at the header fields to identify the action. For future work, developers can further improve this logic by incorporating SDN capacities to improve network security by observing the entire network flow and efficiently block the network attacks in the early stage without having to perform deep packet inspection.

- The firewall implementation in this research supports only one OpenFlow switch. Therefore, future works could add more features and support for multiple OpenFlow switches.
- Although OpenFlow protocol makes the network programmable, but it does not make it easy. Therefore, more advanced northbound APIs are highly required to produce an abstraction layer that makes the programmer be able to run parallel and subsequent modules/ applications without caring about the buffer ID issue and policy conflicts.
- Current OpenFlow switches allow only a fixed set of “Match-Action” fields and their specifications define a limited set of action fields. So, there is a need to support new protocols and higher layers. The future work could focus on this area as well.

References

[1] M. Suh, S. Park, B. Lee, S. Yang, “Building firewall over the software-defined network controller,” in Advanced Communication Technology (ICACT), 2014 16th International Conference on, 2014, pp. 744–748.

[2] S. Khummanee, A. Khumseela, and S. Puangpronpitag, “Towards a new design of firewall: Anomaly elimination and fast verifying of firewall rules,” in Computer Science and Software Engineering (JCSSE), 2013 10th International Joint Conference on, 2013, pp. 93–98.

[3] K. Ingham and S. Forrest, “A history and survey of network firewalls,” Univ. N. M. Tech Rep, 2002.

[4] F. Avolio, ‘Firewalls and Internet Security, the Second Hundred (Internet) Years’. [Online]. Available: http://www.cisco.com/web/about/ac123/ac147/ac174/ac200/about_cisco_ipj_archive_article09186a00800c85ae.html. [Accessed: 12 - May - 2017, 07:30 PM].

[5] J. Mogul, “Using screend to implement IP/TCP security policies,” DTIC Document, 1991.

[6] B. Cheswick, “The Design of a Secure Internet Gateway,” in in Proc. Summer USENIX Conference, 1990, pp. 233–237.

[7] F. M. Avolio, M. J. Ranum, and M. D. Glenwood, “A network perimeter with secure external access,” in Proceedings of the Internet Society Symposium on Network and Distributed System Security, Glenwood, Maryland, 1994.

[8] W. Odom, CCNA Routing and Switching 200-120 Official Cert Guide Library, first edition. Cisco Press, 2013.

[9] M. Roesch, Snort: Lightweight Intrusion Detection for Networks, in LISA, 1999, vol. 99, pp. 229–238.

[10] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, “Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers”, 2012.

[11] R. Bifulco, R. Canonico, M. Brunner, P. Hasselmeyer, and F. Mir, “A Practical Experience in Designing an OpenFlow Controller”, in 2012 European Workshop on Software Defined Networking (EWSDN), 2012, pp. 7–12.

[12] T. D. Nadeau and K. Gray, SDN: Software Defined Networks, first edition. O’Reilly Media, 2013.

[13] S. Azodolmolky, Software Defined Networking with OpenFlow, Publishing, 2013.

[14] E. Dimitriadou, K. Hornik, F. Leisch, D. Meyer, and A. Weingessel, “SDN: Software Defined Networks”, 2008.

[15] P. Southwick, D. Marschke, and H. Reynolds, A Practical Guide to Junos Routing and Certification, Second edition. Beijing: O’Reilly Media, 2011.

[16] D. Hucaby, CCNP SWITCH 642-813 Official Certification Guide, first edition. Indianapolis, Ind: Cisco Press, 2010.

[17] L. E. Li, Z. M. Mao, and J. Rexford, “Toward Software-Defined Cellular Networks,” in 2012 European Workshop on Software Defined Networking (EWSDN), 2012, pp. 7–12.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” ACM SIGCOMM Comput. Commun, pp. 69–74, 2008.

[19] “Software-Defined Networking: The New Norm for Networks.” ONF, 13-Apr-2012.

[20] “OpenFlow Switch Specification Version 1.3.3 (Protocol version 0x04).” ONF, 27-Sep- 2013.

[21] “OpenFlow Management and Configuration Protocol - OF-CONFIG 1.2.” ONF, 2014.

[22] “OpenFlow Switch Specification Version 1.4.0 (Wire Protocol 0x05).” ONF, 14-Oct-2013.

[23] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, “Flowvisor: A network virtualization layer,” OpenFlow Switch Consort. Tech Rep, 2009.

[24] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “NOX: towards an operating system for networks,” ACM SIGCOMM Comput. Commun, pp. 105–110, 2008.

[25] U. Mustafa, M. M. Masud, Z. Trabelsi, T. Wood, and Z. Al Harthi, “Firewall performance optimization using data mining techniques,” in

Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International, 2013, pp. 934–940.

[26] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, and C. Schlesinger, “Languages for software-defined networks,” Commun. Mag. IEEE, vol. 51, no. 2, pp. 128–134, 2013.

[27] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular SDN Programming with Pyretic.”, USENIX; login, vol. 38, no. 5, pp. 128–134, Oct. 2013.

[28] A. Lara, A. Kolasani, and B. Ramamurthy, “Network Innovation using OpenFlow: ASurvey,” IEEE Commun. Surv. Tutor., vol. 16, no. 1, pp. 493–512, First 2014.

[29] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, “SoftCell: Taking control of cellular core networks,” ArXiv Prepr. ArXiv13053568, 2013.

[30] G. Hampel, M. Steiner, and T. Bu, “Applying Software-Defined Networking to the telecom domain,” in 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2013, pp. 133–138.

[31] S. Shin, Lei Xu, S. Hong, G. Gu, "Enhancing Network Security through Software Defined Networking (SDN)", IEEE International Conference on Computer Communication and Networks, Waikoloa, HI, USA, Aug., 2016.

[32] Scott-Hayward, S., Natarajan, S., & Sezer, S, "A Survey of Security in Software Defined Networks", IEEE Communications Surveys & Tutorials, vol. 18, no. 1, pp. 623–654, 2016.

[33] Syed Taha Ali, Vijay Sivaraman, Adam Radford and Sanjay Jha, "A Survey of Securing Networks using Software Defined Networking", IEEE 2014.

[34] D. Kreutz, F. Ramos, and P. Verissimo, “Towards secure and dependable software-defined networks,” in Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013, pp. 55–60.

[35] P. Porras et al. “A security enforcement kernel for OpenFlow networks”. In: HotSDN. ACM, 2012.

[36] S. Shin et al. “FRESCO: Modular Composable Secu- rity Services for Software-Defined Networks”. In: In- ternet Society NDSS. 2013.

[37] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, “A NICE way to test OpenFlow applications,” in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2012.

[38] S. A. Mehdi, J. Khalid, and S. A. Khayam, “Revisiting traffic anomaly detection using software defined networking,” in Recent Advances in Intrusion Detection. Springer, 2011, pp. 161–180.

[39] S. Scott-Hayward, “Design and deployment of secure, robust and resilient SDN controllers,” in Proceedings of the 2015 IEEE Conference on Network Softwarization (NetSoft). IEEE, 2015, pp. 1–5.

[40] R. Braga, E. Mota, and A. Passito, “Lightweight DDoS flooding attack detection using NOX/OpenFlow,” in IEEE 35th Conference on Local Computer Networks (LCN). IEEE, 2010, pp. 408–415.

[41] "Processor Requirements for Host Systems". VMware Workstation 9 Documentation Center. VMware. Retrieved 12 December 2012.

[42] B. Lantz, N. Handigol, B. Heller, and V. Jeyakuma, ‘Introduction to Mininet’, [Online]. Available: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet [Accessed: 10 - October - 2017, 04:00 PM].

[43] http://linux.die.net/man/8/apt-get, [Accessed: 22 - October - 2017, 01:30 PM].

[44]https://askubuntu.com/questions/94102/what-is-the-difference-between-apt-get-update-and-upgrade, [Accessed: 12 – November - 2017, 03:30 PM].

[45] Scopatz, Anthony, Huff Kathryn D., “Effective Computation in Physics”, O'Reilly Media, Retrieved 20 April 2016.

[46] Torvalds, Linus, " linux-kernel (Mailing list)”, [Accessed: 15 - November - 2017, 10:15 PM].

Appendix

Appendix A POXFW1

At this appendix, the python code of the POX Firewall that detects and prevents DoS attack has been illustrated.

```
# Copyright 2012-2013 James McCauley
#
# 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.

"""
A stupid L3 switch

For each switch:
1) Keep a table that maps IP addresses to MAC addresses and switch
ports.
   Stock this table using information from ARP and IP packets.
2) When you see an ARP query, try to answer it using information in
the table
   from step 1.  If the info in the table is old, just flood the
query.
3) Flood all other ARPs.
4) When you see an IP packet, if you know the destination port
(because it's
   in the table from step 1), install a flow for it.
"""
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 Entry (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.
  """
  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):
    # 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 = {}

    # (dpid,IP) -> [(expire_time,buffer_id,in_port), ...]
    # These are buffers we've gotten at this datapath for this IP
which
    # 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 dropDDOS ():
    # 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:
      # Yup!
      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
      #Timer(1, _timer_func(), recurring=True)


      #print"\n\n*********new packetIN************"
      if len(diction) == 0:
        print("Enpty 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
        #print "ERROR"
      else:
        if event.connection.dpid in diction:
      # temp = diction[event.connection.dpid]
      #print(temp)
      #print "error check " ,
str(diction[event.connection.dpid][event.port])
          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 "printting dpid port number and its packet
count: ",  str(event.connection.dpid),
str(diction[event.connection.dpid]),
str(diction[event.connection.dpid][event.port])
          else:
            diction[event.connection.dpid][event.port] = 1

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


    def _timer_func ():
      global diction
      global set_Timer
      if set_Timer==True:
        #print datetime.datetime.now(),": calling timer fucntion
now!!!!!"
        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________________________________________________________________
_____________________________"

              #self.dropDDOS ()
              dpid = k
              msg = of.ofp_packet_out(in_port=i)
              #msg.priority=42
              #msg.in_port = event.port
              #po = of.ofp_packet_out(buffer_id = buffer_id,
in_port = in_port)
              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)] = Entry(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.statcolect(event.parsed.next.dstip)#editing
      print "\n***** Entropy Value = ",str(ent_obj.value),"*****\n"
      if ent_obj.value <0.5:
        preventing()
        if timerSet is not True:
          Timer(2, _timer_func, 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] = Entry(inport,
packet.src)
      #nandan: getting source ip address from the packetIn
      #myPacketInSrcIP= packet.next.srcip
      #myPacketInSrcEth= packet.src
      #myPacketInDstIP= packet.next.dstip
      #myPacketInDstEth= packet.dst

      #print "switcID: "+str(dpid)+" ,Port: "+str(event.port)+"
,MAC address: "+str(myPacketInSrcEth)+" ,SrcIP: "+
str(myPacketInSrcIP)+", Dst Mac: "+str(myPacketInDstEth)+", Dst IP:
"+str(myPacketInDstEth)
      # 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] = Entry(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

      # Didn't know how to answer or otherwise handle this ARP, so
just flood it
      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)
```

Appendix B POXFW2

This appendix illustrates the python code of the POX Firewall which filters any packet based on its protocol number (protocol type) and destination IP address.

```
# Copyright 2012 James McCauley
#
# 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.

"""
A super simple OpenFlow learning switch that installs rules for
each pair of L2 addresses.
"""

# These next two imports are common POX convention
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.packet.ethernet import ethernet

# Even a simple usage of the logger is much nicer than print!
log = core.getLogger()

# This table maps (switch,MAC-addr) pairs to the port on 'switch'
at
# which we last saw a packet *from* 'MAC-addr'.
# (In this case, we use a Connection object for the switch.)
table = {}

# To send out all ports, we can use either of the special ports
# OFPP_FLOOD or OFPP_ALL.  We'd like to just use OFPP_FLOOD,
# but it's not clear if all switches support this, so we make
# it selectable.
all_ports = of.OFPP_FLOOD
```

```
# Handle messages the switch has sent us because it has no
# matching rule.
def _handle_PacketIn(event):
    packet = event.parsed
    if packet.type == ethernet.IP_TYPE:
        if packet.payload.protocol == 17 and packet.next.dstip ==
'10.0.0.1':
            return

    if packet.type == ethernet.IP_TYPE:
        if packet.payload.protocol == 1 and packet.next.dstip ==
'10.0.0.2':
            return

    if packet.type == ethernet.IP_TYPE:
        if packet.payload.protocol == 1 and packet.next.dstip ==
'10.0.0.3':
            return

    if packet.type == ethernet.IP_TYPE:
        if packet.payload.protocol == 6 and packet.next.dstip ==
'10.0.0.4':
            return
    # Learn the source
    table[(event.connection, packet.src)] = event.port

    dst_port = table.get((event.connection, packet.dst))

    if dst_port is None:
        # We don't know where the destination is yet.  So, we'll
just
        # send the packet out all ports (except the one it came in
on!)
        # and hope the destination is out there somewhere. :)
        msg = of.ofp_packet_out(data=event.ofp)
        msg.actions.append(of.ofp_action_output(port=all_ports))
        event.connection.send(msg)
    else:
        # Since we know the switch ports for both the source and
dest
        # MACs, we can install rules for both directions.
        msg = of.ofp_flow_mod()
        msg.match.dl_dst = packet.src
        msg.match.dl_src = packet.dst
        msg.match.dl_type = packet.type
        msg.idle_timeout = 2
        msg.hard_timeout = 0
        msg.actions.append(of.ofp_action_output(port=event.port))
        event.connection.send(msg)

        # This is the packet that just came in -- we want to
        # install the rule and also resend the packet.
        msg = of.ofp_flow_mod()
        msg.data = event.ofp  # Forward the incoming packet
        msg.match.dl_src = packet.src
        msg.match.dl_dst = packet.dst
        msg.idle_timeout = 2
        msg.hard_timeout = 0
        msg.match.dl_type = packet.type
        msg.actions.append(of.ofp_action_output(port=dst_port))
        event.connection.send(msg)

        log.debug("Installing %s <-> %s" % (packet.src,
packet.dst))


def launch(disable_flood=False):
    global all_ports
    if disable_flood:
        all_ports = of.OFPP_ALL

    core.openflow.addListenerByName("PacketIn", _handle_PacketIn)

    log.info("Pair-Learning switch running.")
```