

Summer 6-24-2014

Application Layer Firewall Using OpenFlow

Alaauddin Shieha

University of Colorado Boulder, alaauddin.shieha@colorado.edu

Follow this and additional works at: https://scholar.colorado.edu/tlen_gradetds



Part of the [Computer Sciences Commons](#), and the [Engineering Commons](#)

Recommended Citation

Shieha, Alaauddin, "Application Layer Firewall Using OpenFlow" (2014). *Interdisciplinary Telecommunications Graduate Theses & Dissertations*. 1.

https://scholar.colorado.edu/tlen_gradetds/1

This Thesis is brought to you for free and open access by Interdisciplinary Telecommunications at CU Scholar. It has been accepted for inclusion in Interdisciplinary Telecommunications Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.

APPLICATION LAYER FIREWALL USING OPENFLOW

by

ALAAUDDIN SHIEHA

B.E., Telecommunication Engineering

University of Aleppo, 2009

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Master of Science



Department of Interdisciplinary Telecommunication Program

2014

Certificate of Completion

This thesis entitled:
Application Layer Firewall Using OpenFlow
written by Alaaudhin Shieha
has been approved for the Department of
Interdisciplinary Telecommunication Program
University of Colorado at Boulder

Dr. Douglas Sicker (Thesis Chair)

Dr. Jim Lansford (Thesis Committee)

Mark Dehus (Thesis Committee)

Date_____

The final copy of this thesis has been examined by the signatories, and we
Find that both the content and the form meet acceptable presentation standards
Of scholarly work in the above mentioned discipline

ABSTRACT

Shieha, Alaauddin (M.S., Telecommunication Engineering

[Interdisciplinary Telecommunication Program])

Application Layer Firewall Using OpenFlow

Thesis directed by Endowed Full Professor Douglas Sicker

Security is one of the most important aspects in networking. Companies and service providers spend much money on expensive firewalls to enforce security. Software-Defined Networking (SDN) is a new architecture that can save companies and service providers money, reduce provisioning time from weeks to minutes, provide centralized management, promote innovation, and allow programmability. SDN achieves this by decoupling the control plane from the data plane. This paper demonstrates the benefits of implementing an application layer firewall using OpenFlow protocol, which is one way of implementing SDN. The firewall is capable of detecting application layer traffic, such as BitTorrent and YouTube, and of preventing some Denial-of-Service Attacks. This paper discusses the implementation details and provides a performance analysis of the firewall module.

CONTENTS

CHAPTER

1. INTRODUCTION	1
2. BACKGROUND	3
2.1 History of Firewalls.....	3
2.1.1 Why We Need Firewalls	3
2.1.2 Firewall Types in Historical Order	4
2.2 Software Defined Networking (SDN).....	8
2.2.1 The Importance of the Separation.....	11
2.3 OpenFlow Protocol	13
2.3.1 OpenFlow Switch Components	14
2.3.2 OpenFlow Ports	15
2.3.3 Flow Table	15
2.3.4 OpenFlow Message Types.....	16
2.3.5 Connection Setup.....	18
2.3.6 Multiple Controllers.....	19
2.3.7 Flow Match Fields	20
2.3.8 Action Structure	20
2.3.9 OF-Config Versions.....	21
2.3.10 OpenFlow Versions	21
2.4 SDN Controllers.....	22
2.5 Northbound APIs.....	23
2.6 SDN Use Cases	24
3. IMPLEMENTATIONS.....	26
3.1 Setup and POX Controller	26
3.1.1 Test Bed Setup	26
3.1.2 Pox Controller.....	27
3.2 Applications	29
3.2.1 BitTorrent Protocol	29
3.2.2 YouTube	33
3.3 Flow Chart.....	34

4. PERFORMANCE ANALYSIS	40
5. FINDINGS	45
6. CONCLUSION	47
BIBLIOGRAPHY	48
APPENDIX	
A. Firewall Code	52
B. Pica8 Configuration	69

TABLES

Table

1. Required Match Fields	20
2. Action Structure	20
3. Capability progression of OF-Config	21
4. The progression of enhancements to the OpenFlow pipeline from OF v1.1 through OF v1.3	22
5. Comparison among the controllers	23

FIGURES

Figure

1. DEC SEAL - First Commercial Firewall	5
2. Software-Defined Network Architecture	9
3. Control and data plane example implementation	11
4. Relationship between components defined in the specification, the OF-CONFIG protocol and the OpenFlow protocol	14
5. Main components of an OpenFlow switch	14
6. Main components of a flow entry in a flow table	16
7. OpenFlow protocol messages	19
8. Northbound API	24
9. Logical Test Bed	26
10. Tracker HTTP GET Request	30
11. Peer Wire Protocol Handshake over TCP	31
12. Peer Wire Protocol Handshake over UDP (uTP)	32
13. Youtube.com DNS query	33
14. Requesting YouTube through its resolved IP address	34
15. Host Field in HTTP GET Request	34
16. Flowchart 1 of Firewall Module	37
17. Flowchart 2 of Firewall Module	38
18. Flowchart of Switch Module	39
19. Comparison between Firewall Module, Switch Module only, and Traditional Switch in Term of Jitter	41
20. Comparison between Firewall Module and Traditional Switch in Term of Throughput	41
21. Throughput of 20 parallel TCP Steams over Conventional Switch	42
22. Throughput of 20 parallel TCP Steams with Controller Running Firewall Module	43
23. Throughput of 50 parallel TCP Steams over Conventional Switch	43
24. Throughput of 50 parallel TCP Steams with Controller Running Firewall Module	44

CHAPTER 1

INTRODUCTION

Enterprises and service providers started to notice the limitations of conventional network architecture, which is inflexible compared to the server environment [1]. In addition, introducing any change in the conventional network is expensive and time consuming [1]. Software-Defined Networking (SDN) was suggested in 2005 in order to overcome conventional network architecture limitations. It is expected that SDN will enhance programmability and make it easier to configure and manage the network by decoupling the control plane from the data plane.

OpenFlow is a communication protocol between the control plane and the data plane. It is supported by the Open Networking Foundation (ONF) [2]. This protocol can be used along with a piece of software that resides in the controller in order to implement routing, switching, a firewall, or a load balancer. Since some SDN controllers have only a simple built-in layer 2 firewall, enterprises would still have to buy a physical firewall to secure their network. The motivation behind this paper is to prove that SDN can replace expensive equipment such as layer 7 firewalls.

The main contributions of this paper are:

- Implementation of layer 7 firewall capable of blocking BitTorrent and YouTube traffic and preventing some Denial-of-Service attacks.
- Performance analysis and comparison between OpenFlow controller and conventional switch.

The remainder of this paper is organized as follows. In chapter 2, I summarize firewall history and technologies, introduce SDN architecture, explain OpenFlow protocol, compare SDN

controller, and note the importance of northbound APIs and SDN use cases. In chapter 3, I present the implementation details and explain BitTorrent and YouTube applications. In chapter 4, I show the results of the performance analysis. In chapter 5, I list the finds and future work. Finally, I conclude in chapter 6.

CHAPTER 2

BACKGROUND

2.1 History of 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 [3]–[5]. 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 [5]. 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 [5], [6].

2.1.1 Why We Need 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. [5].
- 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) [5].

- To prevent information leaks to the outside.
- To enforce policy on network traffic.
- To provide auditing information for the network administrator.

2.1.2 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 [5], [7]. 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) [5], [6]. A chemical company was the first to have DEC SEAL on June 13, 1991 [5], [6]. The DEC SEAL consisted of three devices, shown in figure 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 [5].

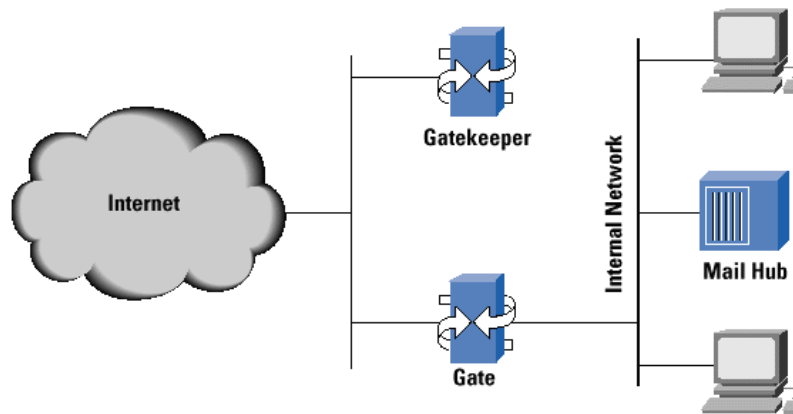


Figure 1: DEC SEAL - First Commercial Firewall [6]

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 has to be decapsulated and encapsulated two times, one at the Gatekeeper and one at the final destination [5].

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 [8]. 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 [5]. 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 [9]. 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 [6]. The TIS firewall became the Network Associates Incorporation's (NAI) Gauntlet Internet firewall after the merger between TIS and NAI in 1998 [5].

Packet filtering firewalls started with Mogul's paper [5]. 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 [5]. 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 [5]. As a result, stateful firewalls were developed to keep track of TCP sessions and to allow packets coming from outside to access the network as long as 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 [5]; however, the first published peer-reviewed paper was by Chow and Julkunen in 1998 [5].

Network Address Translation (NAT) is considered to be a layer of protection similar to 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 [5], [10].

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 [5].

Signature-based firewall, which might work at the user level as a transparent proxy, monitors the payload for known malicious strings in order to prevent an attack from happening. This approach is sometimes called “fingerprint scrubber” or “application scrubbing” [5]. Snort is an intrusion detection system that Hogwash firewall uses to drop packets that match the rules [11].

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 [5]. In addition, a software bug in P2P programs such as Gnutella could be used by attackers to gain access to the victim's host [5].

2.2 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 [12], [13]. 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 [14], [15]. Figure 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 [16]. The location of the control plane and how far away it could be located from the data plane, whether or not 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 [14].

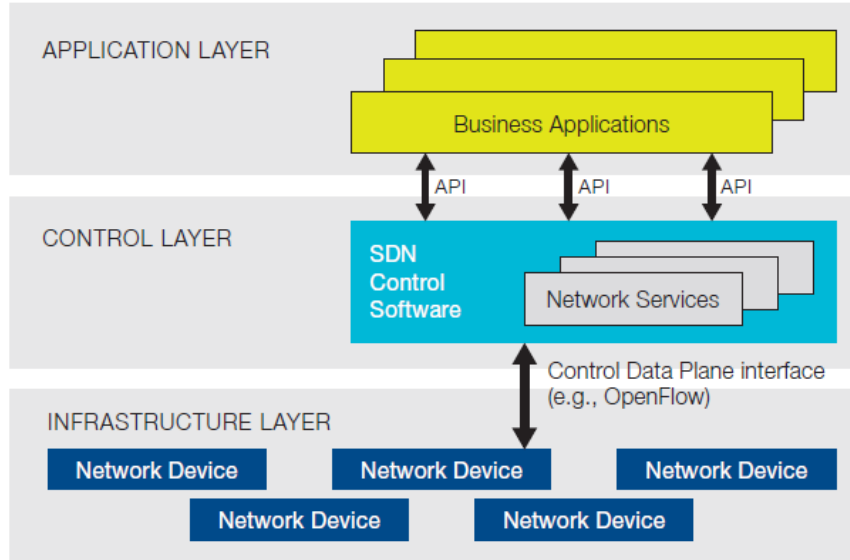


Figure 2: Software-Defined Network Architecture [17]

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 [14]. 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 [14]. 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 is allowed to handle other functions that utilize the underlay network (switching devices) [14]. 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 have to cooperate with each other in order to have a functional network [14].

The control plane is the brain of the device. It exchanges protocol updates and system management messages [18]. 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 [14]. 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 [19]. The control plane provides the data plane with an accurate up-to-date forwarding table through an internal link [18]. 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 [14], [18]. In addition to forwarding traffic, the data plane implements some advance features such as policers, access control lists, and class of service (COS) [18]. 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 has to be sent to the control plane for further processing. In addition, the following conditions that might cause the same behavior are [19]:

- 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).

2.2.1 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 [14]. 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 3 below [14]. However, the high cost of this design, along with other components, discussed below, is the motivation behind SDN.

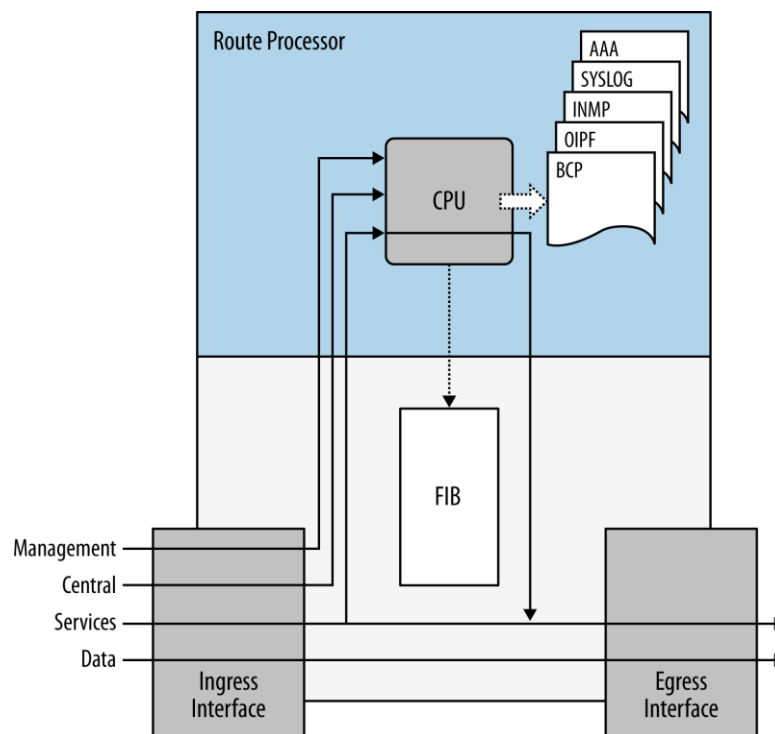


Figure 3: Control and data plane example implementation [14]

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 [14]. 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 [14].

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 [14], [20].

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 [14].

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 [14]. 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 [14].

2.3 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 [21]. 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 [17].

OpenFlow consists of a set of protocols and Application Programming Interface (API). The protocols are divided into two parts, as shown in figure 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 [14], [22].
- 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 particular controller [23].

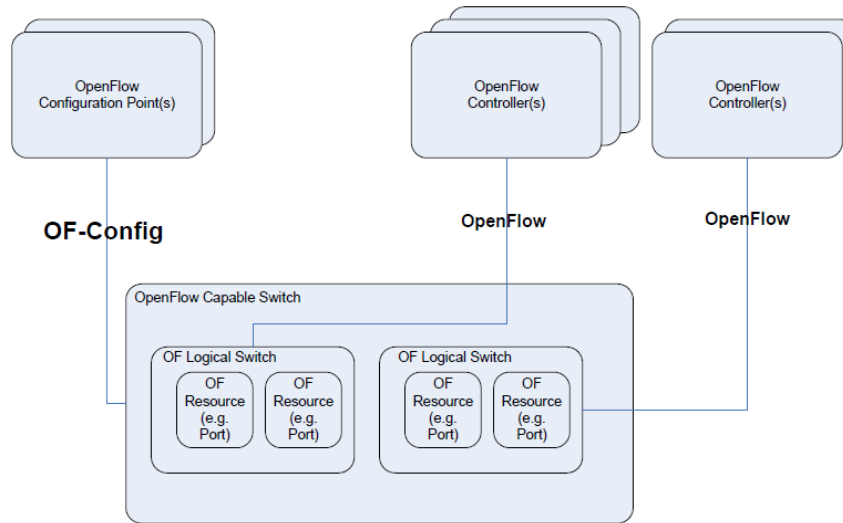


Figure 4: Relationship between components defined in the specification, the OF-CONFIG protocol and the OpenFlow protocol [23]

2.3.1 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 5 below [24].

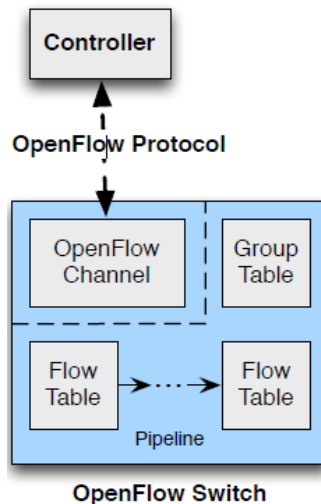


Figure 5: Main components of an OpenFlow switch [25]

2.3.2 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 [22]:
 - 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.

2.3.3 Flow Table

Each entry in the flow table is made of the following fields, as shown in figure 6 [22]:

- 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 `OFPPF_SEND_FLOW_REM` means that a flow removed message will be sent to the controller once this entry is removed.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags
--------------	----------	----------	--------------	----------	--------	-------

Figure 6: Main components of a flow entry in a flow table [22]

2.3.4 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 [22].

The sub-types of each category are as follows [22]:

2.3.4.1 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 request or send 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.

2.3.4.2 Asynchronous 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.

2.3.4.3 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.

2.3.5 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 [22].

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 7 illustrates [22]. The right side of figure 7 shows an example of exchanging different OpenFlow messages.

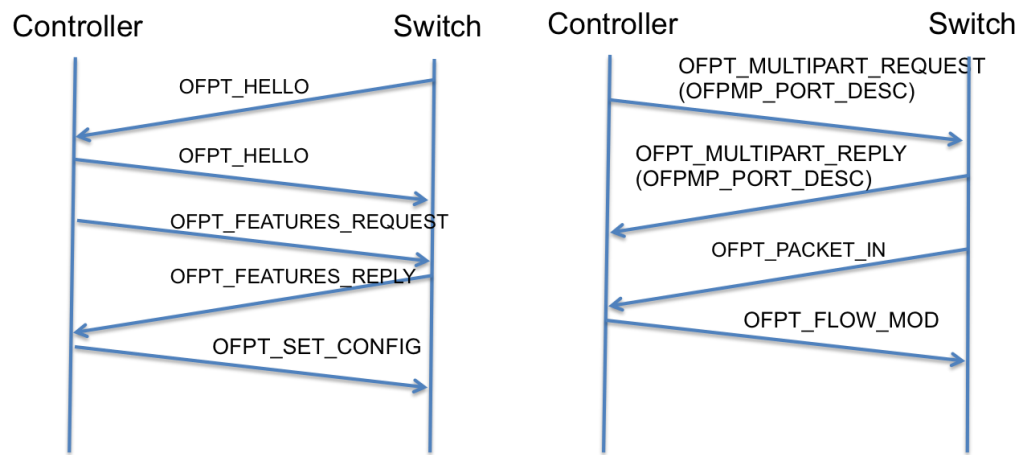


Figure 7: OpenFlow protocol messages [26]

2.3.6 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].

2.3.7 Flow Match Fields

Table 1 below explains the match fields that must be supported by the OpenFlow-enabled switch in its pipeline.

Table 1: Required Match Fields [22]

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

2.3.8 Action Structure

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

Table 2: Action Structure [22]

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	

2.3.9 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 [14], [27]. Table 3 compares OF-config versions [14].

Table 3: Capability progression of OF-Config [14]

OF-Config v1.0	OF-Config v1.1	OF-Config v1.2 (proposed)
Based on OpenFlow v1.2 <ul style="list-style-type: none">• assigning controllers to logical switches• retrieving assignment of resources to logical switches• configuring some properties of ports and queues	Based on OpenFlow v1.3 <ul style="list-style-type: none">• added controller certificates and resource type “table”• retrieving logical switch capabilities signed to controller• configuring of tunnel endpoints	Based on OpenFlow v1.4 (proposed) <ul style="list-style-type: none">• retrieving capable switch capabilities, configuring logical switch capabilities• assigning resources to logical switches• simple topology detection• event notification

2.3.10 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 [14]. Table 4 below compares OpenFlow v1.1 – 1.3.

Table 4: The progression of enhancements to the OpenFlow pipeline from OF v1.1 through OF v1.3 [14]

OF v1.1	OF v1.2	OF v1.3
MPLS <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • Supports multiple levels of VLAN tagging • Actions to set VLAN ID, priority • Actions push, pop VLAN headers Groups <p>Group properties: Group ID, Type (all, select, indirect, fast-fallover), Counters</p>	IPv6 <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • 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

2.4 SDN Controllers

There are several open source 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 [28], [29]. POX is the Python version of NOX, and it supports the same graphical user interface (GUI) as NOX [30]. Table 5 below compares the features of some SDN controllers [12].

Table 5: Comparison among the controllers [12]

	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

2.5 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) [31].

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 [31].

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 8 shows [32]. One of the main advantages of Pyretic over traditional OpenFlow programming is that Pyretic offers parallel and sequential composition of policies in order 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 (\gg) is the input of the policy on the right of the operator, as shown in the Pyretic policy below [32]:

$$\text{match}(\text{dstip}'2.2.2.8') \gg \text{fwd}(1)$$

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 [32]:

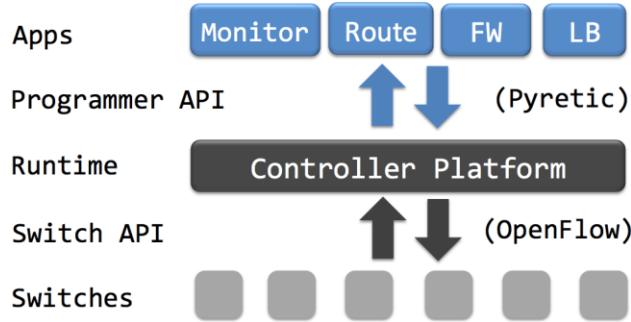
$$R = (\text{match}(\text{dstip}='2.2.2.8') \gg \text{fwd}(1)) + (\text{match}(\text{dstip}='2.2.2.9') \gg \text{fwd}(2))$$


Figure 8: Northbound API [32]

2.6 SDN Use Cases

Since one of the main features of SDN is that it drives innovation [25], it is hard to summarize and imagine all SDN use cases. SDN could be utilized in campus, data center, cloud, and service provider networks [17].

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 [17].

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 [17].

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 [17].

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 [17]. 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 [20], [33]–[35].

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. In this paper, I have implemented Layer 7 firewall and managed to block several applications that are explained in the next chapter.

CHAPTER 3

IMPLEMENTATIONS

3.1 Setup and POX Controller

This chapter of the paper explains how to build a state-full application layer (layer 7) firewall as a module / component on POX controller. It also explains the required changes to POX controller and the limitations of the firewall module. The next section starts with the logical test bed setup.

3.1.1 Test Bed Setup

The test bed consists of Pica8 OpenFlow switch and a server that runs four VMs. One is the POX controller, which runs on Ubuntu server, while the rest are Windows and Ubuntu hosts. They are connected as shown in the logical topology in Figure 9. Pica8 has been configured to run as Open vSwitch (OVS). The configuration is in the appendix for reference. The OF switch is connected to the internet through port “1” and to the controller through the management interface “M”.

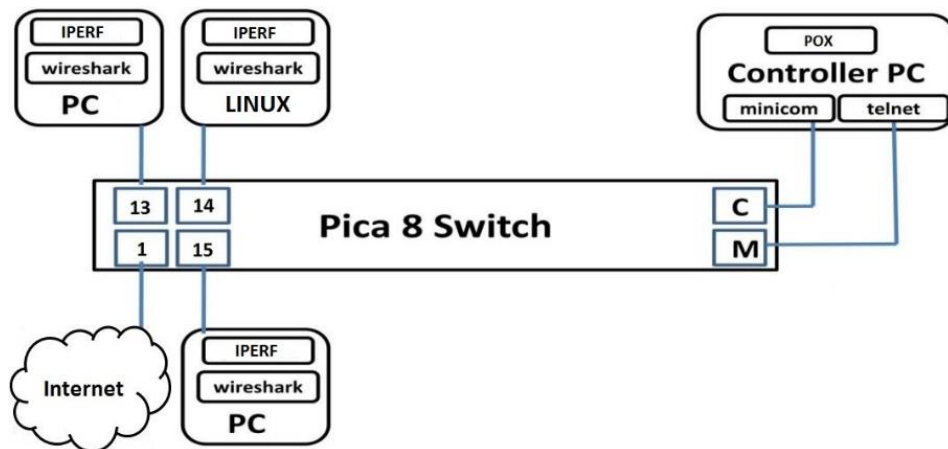


Figure 9: Logical Test Bed

3.1.2 Pox Controller

As mentioned in section 2.4, POX is a Python-based SDN controller. POX can be installed from GitHub quickly through the command:

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

POX is invoked by running “pox.py” followed by the module name in the format “directory.filename.” For example, the following command will invoke POX and run a layer 2 learning switch module located inside the “forwarding” directory:

```
./pox.py forwarding.l2_learning
```

Even though naming the module on the command line after “pox.py” is enough to load it, it is better if each module (component) contains a launch function. POX calls the launch function to initialize the module. It is basically a function that has the name “launch,” and it is also used to pass command line argument to the module.

POX provides an object called “core,” which serves as a rendezvous point between modules and as a central point for POX’s API so that they can interact with each other. Instead of having one module importing another module (through “import” statement), modules will “register” themselves on the “core” object [36]. There are two ways to register a module [36]:

- Using *core.register()*: takes two arguments. The first is the name the user wants to use, and the second is the object.
- Using *core.registerNew()*: takes generally a single argument, which is the class name. It might also take an optional argument and pass it to the `__init__` method of the registered class.

POX uses an event based programming paradigm in which certain objects raise events, while others can listen to these specific events. In other words, when an event occurs, a piece of code usually known as “event handler” or “event listener” is called [36]. There are mainly two ways for an application to listen for events in POX [36]:

1. Register a call back function for specific events fired by either the OpenFlow handler module or other modules through:

core.openflow.addListenerByName("EVENTNAME",CALLBACK_FUNCTION, PRIORITY)

2. Register an object with the OpenFlow handler module or other modules through:

core.openflow.addListeners(self)

Once “core.openflow.addListeners(self)” is called, it looks through the events of “core.openflow,” and if it finds a method on “self” with a name such as “_handle_EventName,” it registers that method as an event listener [36].

In addition to event handling, POX provides a library for parsing and constructing well-known packets [36]. All packet classes can be found in pox/lib/packet directory. Packets can be navigated either by using the “payload” attribute of the packet object or by using the “find()” method [36].

To build a layer 7 firewall module, certain changes to the POX controller must be considered. First, the OpenFlow switch should send the complete packet payload to the controller. By default, it sends 128 bytes of the packet once it misses all the entries in the flow table [36]. This value might be enough to detect some, but not all, applications. Therefore, the miss send length value must be modified to reflect this, which could be done by changing the value of “core.openflow.miss_send_len” to 0x7fff during startup or by running “full_payload” module

along with the firewall module. Second, the firewall module will be responsible for classifying only the packets; thus, it needs to run with the switching module. However, running these two modules at the same time will cause OpenFlow error messages as both will try to access the same buffer ID. In addition, each module might send different rules to the OpenFlow switch, which would cause policy conflicts.

There are two approaches to solve this issue. The first approach is combining the firewall module with the switch module and creating a “Master” class to manage the order of calling the module’s functions. This approach is similar to the concept of northbound APIs explained before, and it is the one implemented in this paper. The second approach is keeping the modules separated while modifying the switch module to listen to events fired by the firewall module instead of the ones fired by OpenFlow protocol (PacketIn events).

3.2 Applications

It is essential to understand how applications work in order to detect them. Therefore, this paper focuses on two types of applications that network administrators in enterprises or service providers might want to block/ throttle in order to save bandwidth or to prevent illegal use.

3.2.1 BitTorrent Protocol

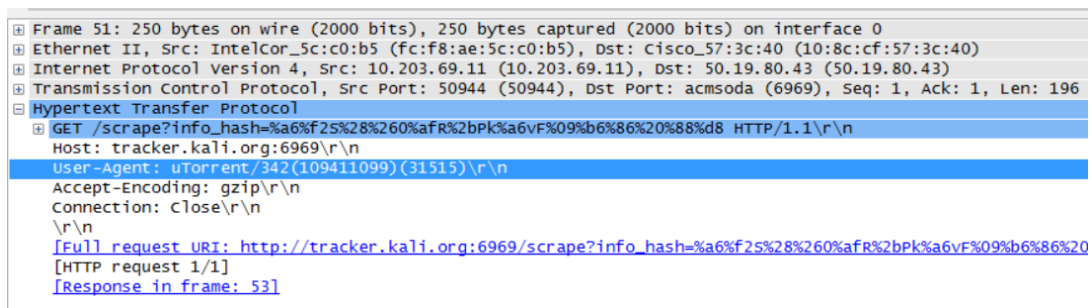
In order to detect BitTorrent traffic, it is important to understand how BitTorrent protocol works first. BitTorrent is a peer-to-peer (P2P) protocol for sharing distributed files. It enables multiple downloads of the same file simultaneously, making it possible for the file source to support a large number of downloaders without a large increase in its load [37].

BitTorrent file distribution consists of the following entities [37]:

- A web server, which hosts the “metainfo” (“.torrent”) file.

- A “metainfo” file, which is an encoded dictionary that contains in the “info” field information, such as the filename, length, and number of bytes in each piece of data and the announce URL of the tracker in the “announce” field.
- A BitTorrent tracker, which is an HTTP/HTTPS service that maintains statistics about the torrent file and coordinates the action between all peers.
- The end user BitTorrent client.

The user downloads the .torrent file from the web server. The BitTorrent client runs the file and reads the info from the file and then sends an HTTP GET request to the tracker, which includes the info field, peer ID, the total amount uploaded/ downloaded, and the User-Agent, as figure 10 illustrates. The tracker responds with a list of peers’ IP addresses and IDs along with other information. The client uses this list to participate in the torrent by initiating connections to the peers [37].



```

Frame 51: 250 bytes on wire (2000 bits), 250 bytes captured (2000 bits) on interface 0
Ethernet II, Src: IntelCor_5c:c0:b5 (fc:f8:ae:5c:c0:b5), Dst: Cisco_57:3c:40 (10:8c:cf:57:3c:40)
Internet Protocol Version 4, Src: 10.203.69.11 (10.203.69.11), Dst: 50.19.80.43 (50.19.80.43)
Transmission Control Protocol, Src Port: 50944 (50944), Dst Port: acmsoda (6969), Seq: 1, Ack: 1, Len: 196
Hypertext Transfer Protocol
  GET /scrape?info_hash=%a6%f25%28%260%afR%2bPk%a6vF%09%b6%86%20%88%d8 HTTP/1.1\r\n
    Host: tracker.kali.org:6969\r\n
    User-Agent: uTorrent/342(109411099)(31515)\r\n
    Accept-Encoding: gzip\r\n
    Connection: close\r\n
  \r\n
  [Full request URI: http://tracker.kali.org:6969/scrape?info_hash=%a6%f25%28%260%afR%2bPk%a6vF%09%b6%86%20%88%d8]
  [HTTP request 1/1]
  [Response in frame: 53]
  
```

Figure 10: Tracker HTTP GET Request

BitTorrent’s peer protocol operates over either TCP or the uTorrent transport protocol (uTP). If TCP is used, the TCP handshake will start first, followed immediately by the required BitTorrent handshake, which is followed by a stream of length-prefixed messages. The BitTorrent handshake starts with character nineteen followed by a “BitTorrent protocol” string, as shown in figure 11

[37]. Eight reserved bytes come after the fixed headers, followed by 20-byte sha1 hash and 20-byte peer id.

No.	Time	Source	Destination	Protocol	Length	Data rate	Info
1309	9.418223000	10.200.151.67	194.226.155.1	UDP	62		Source port: 10463 Destination port: 56999
1310	9.418932000	10.200.151.67	194.226.155.1	TCP	66		56187 > 56999 [SYN] Seq=0 win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
1375	9.626921000	194.226.155.1	10.200.151.67	TCP	66		56999 > 56187 [SYN, ACK] Seq=0 Ack=1 win=16384 Len=0 MSS=1408 SACK_PERM=1 WS=8
1376	9.627222000	10.200.151.67	194.226.155.1	TCP	54		56187 > 56999 [ACK] Seq=1 Ack=1 win=66048 Len=0
1377	9.627457000	10.200.151.67	194.226.155.1	BitTorrent	122		Handshake
1449	9.835636000	194.226.155.1	10.200.151.67	BitTorrent	243		Handshake Extended

Frame 1377: 122 bytes on wire (976 bits), 122 bytes captured (976 bits) on interface 0	
Ethernet II, Src: Intelcor_5c:c0:b5 (fc:f8:ae:5c:c0:b5), Dst: Cisco_57:2f:40 (10:8c:cf:57:2f:40)	
Internet Protocol Version 4, Src: 10.200.151.67 (10.200.151.67), Dst: 194.226.155.1 (194.226.155.1)	
Transmission Control Protocol, Src Port: 56187 (56187), Dst Port: 56999 (56999), Seq: 1, Ack: 1, Len: 68	
BitTorrent	
Protocol Name Length: 19	
Protocol Name: BitTorrent protocol	
Reserved Extension Bytes: 000000000100005	
SHA1 hash of info dictionary: a6f253282630af522b506ba6764609b6862088d8	
Peer ID: 2d5554333432302d1b7b0dfcfc3e774297a7902	

0000	10 8c cf 57 2f 40 fc f8 ae 5c c0 b5 08 00 45 00	...W/0... ..E..
0010	00 6c 0c ae 40 00 80 06 ed ee 0a c8 97 43 c2 e2	..,.,0... ..C..
0020	9b 01 db 7b de a7 5b 7a fe 11 03 2e 06 69 50 18	...[...[ZTP.
0030	01 02 59 cc 00 00 13 1f 07 74 25 8c 8e 22 87 80 05	...V.... 8 (3105420
0040	74 20 70 73 6f 74 6f 63 ef 64 00 00 00 00 00 10	...BitTorrent.....
0050	00 05 a6 f2 53 28 26 30 af 52 2b 50 6b a6 76 46S(60 ..R+PK.vf
0060	09 b6 86 20 88 d8 2d 55 54 33 34 32 30 2d 1b 7b-U T3420-..{
0070	0d ff cc b3 e7 74 29 7a 79 02t}z y.

Figure 11: Peer Wire Protocol Handshake over TCP

Since BitTorrent uses multiple TCP connections, BitTorrent traffic will fill up the upload pipe (send buffer) quickly, which adds delay to all interactive traffic [38]. The reason is that TCP distributes the bandwidth equally between connections [38]. As a result, uTP was developed by Strigeus, Hazel, Shalunov, Norberg, and Cohen to solve this issue. uTP is a transport protocol layer built on top of UDP. It has its own congestion control mechanism that utilized the modem queue size in order to adjust its send rate [38].

The connection setup is similar to that in TCP. However, BitTorrent handshake over uTP uses only three packets to establish the connection instead of four, as is the case with TCP. It is clear in figure 12 that the “BitTorrent protocol” string is in the third UDP packet of the flow. However, after capturing a lot of BitTorrent traffic, this string was also found in the fifth packet. This also implies that uTP has its own retransmission mechanism.

No.	Time	Source	Destination	Protocol	Length	Data rate	Info
108	9.988832000	10.203.69.11	71.173.202.202	UDP	62		Source port: 10463 Destination port: 51413
131	10.075139000	71.173.202.202	10.203.69.11	UDP	72		Source port: 51413 Destination port: 10463
132	10.075536000	10.203.69.11	71.173.202.202	UDP	130		Source port: 10463 Destination port: 51413
151	10.174049000	71.173.202.202	10.203.69.11	UDP	62		Source port: 51413 Destination port: 10463
152	10.174255000	10.203.69.11	71.173.202.202	UDP	62		Source port: 10463 Destination port: 51413

<

Frame 132: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface 0

Ethernet II, Src: IntelCor_5c:c0:b5 (fc:f8:ae:5c:c0:b5), Dst: Cisco_57:3c:40 (10:8c:cf:57:3c:40)

Internet Protocol Version 4, Src: 10.203.69.11 (10.203.69.11), Dst: 71.173.202.202 (71.173.202.202)

User Datagram Protocol, Src Port: 10463 (10463), Dst Port: 51413 (51413)

Data (88 bytes)

Data: 010041560f06fe75f4b62ced0000c350d1003ab813426974...

[Length: 88]

0000	10 8c cf 57 3c 40 fc f8 ae 5c c0 b5 08 00 45 00	...We@... \....E.
0010	00 74 6a 2e 00 00 80 11 6d fd 0a cb 45 0b 47 ad	.tj....; M...E.G.
0020	ca ca 28 df c8 d5 00 60 43 52 01 00 41 56 0f 06	..(.....CR..AV..
0030	fe 75 f4 b6 2c ed 00 00 c3 50 d1 00 3a b8 13 42	.u.....P....B
0040	69 74 54 6f 72 72 65 6e 74 20 70 72 6f 74 6f 63	itTorren t protoc
0050	6f 6c 00 00 00 00 00 10 00 05 a6 f2 53 28 26 38	ol.....5(&0
0060	af 52 2b 50 6b a6 76 46 09 b6 86 20 88 d8 2d 55	R+Pk-wfu
0070	54 33 34 32 30 2d 1b 7b f6 cd 83 e8 e4 ac 16 89	T3420-..{
0080	de 8c	..

Figure 12: Peer Wire Protocol Handshake over UDP (uTP)

After studying the BitTorrent protocol and its peer protocol, it is clear that there are two ways to detect BitTorrent traffic, each with its own advantages and disadvantages. The first approach is detecting BitTorrent based on the User-Agent header string in the tracker HTTP GET request message, as shown in figure 10. Blocking this packet will prevent the user from receiving the list of peers and, as a result, the user will not have the required IPs to initiate any connection. The second approach is detecting the “BitTorrent protocol” string in the BitTorrent handshake. Blocking this packet will prevent the user from connecting to another peer as this message is required by the protocol to complete the connection.

The advantages of the first approach is that the number of trackers is small compared to the number of peers, i.e. the number of packets that need to be dropped using the first approach is less than those in the second approach. As a result, it requires less flow entries, which means less memory and load on the controller. However, the first approach has to account for all the User-Agents, such as utorrent, Bittorrent, and BTWebClient, which will complicate the firewall code. In addition, a regular update to the code is required to account for new BitTorrent clients.

On the other hand, the firewall code that matches based on the “BitTorrent protocol” string requires an update only when the protocol itself is changed. However, it adds more load on the controller and OpenFlow switch resources, as previously noted.

3.2.2 YouTube

Internet service providers use virtual servers in order to solve the problem of running out of IPv4 addresses. As a result a huge customer such as Google assigns multiple services to the same IP address. For example, a domain name system (DNS) query to different Google services might return the same IP address. Figure 13 illustrates a DNS look up for `www.youtube.com` and its resolved IP address. Using one of these IP addresses in the browser should return YouTube.com; however, it returns Googl.com as shown in figure 14. The solution that Google uses to redirect the user to the right service is to check the Host field in the HTTP GET request message, which has the name of the server. Therefore, trying to access YouTube by the resolved IP address will return the default service, which is Google search engine.

```
C:\Users\Admin>nslookup www.youtube.com
Server:    Unknown
Address:   192.168.64.2

Non-authoritative answer:
Name:      youtube-ui.l.google.com
Addresses: 2607:f8b0:4001:c01::be
           74.125.225.165
           74.125.225.164
           74.125.225.160
           74.125.225.163
           74.125.225.167
           74.125.225.162
           74.125.225.161
           74.125.225.166
           74.125.225.169
           74.125.225.174
           74.125.225.168
Aliases:   www.youtube.com
```

Figure 13: Youtube.com DNS query

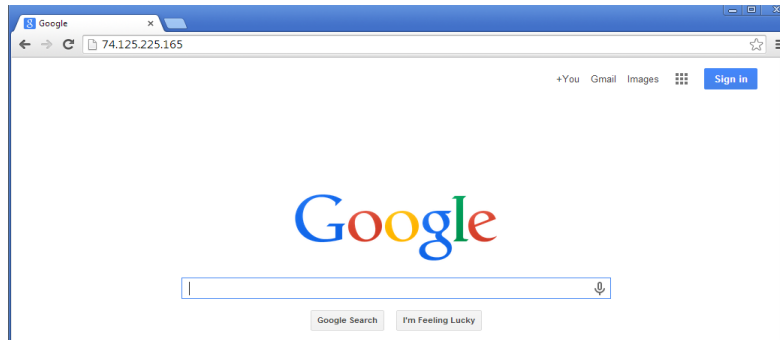


Figure 14: Requesting YouTube through its resolved IP address

Therefore, blocking the HTTP GET message is enough to detect and block access to the YouTube website regardless of the IP address the DNS reply message returns. Figure 15 shows the Host field in different HTTP GET requests. This also works with embedded YouTube videos in websites that do not use SSL to encrypt the HTTP connection (HTTPS).



Figure 15: Host Field in HTTP GET Request

3.3 Flow Chart

The flow chart below explains my implementation of the application layer firewall. It shows what happens to the packet from the moment it arrives at the switch port to the moment it leaves the switch to its final destination as figures 16, 17, and 18 illustrates. The process is as follows. Once the packet reaches the switch, the switch matches the information in the packet headers against its flow table entries. If there is a match, the switch applies the action located in the rule. However, if the packet misses all the entries, the switch forwards the complete packet to the controller for further processing. The firewall module in the controller checks whether the packet is an IP packet or not. If it is not (ARP message, for example), it passes it to the switch module with instructions to install a flow entry through OpenFlow modification message. If it is an IP

packet, it proceeds to check whether or not it is a TCP or UDP packet. If not, it treats the packet in the same way that it treats non-IP packets. It is worth noting that an attacker might use ICMP messages, which are neither TCP nor UDP, to send and receive data [5]. However, this implementation does not take that into account. The next step is to check if the packet is coming from inside or outside as a way to implement a zone-based firewall. After this, the packet will be checked against the firewall table, which is a simple CSV file that can be updated before running the module. If the packet matches a rule/ policy, and the action assigned to that rule is ‘drop,’ the firewall module sends an OpenFlow modification message to instruct the OpenFlow switch to drop the packet for a configurable amount of idle-time/ hard-time. In addition, it will not pass the packet to the switch module. However, if there is no match or the action is ‘permit,’ the packet will be checked against the application table. This is to ensure that an application such as BitTorrent client is not using allowed port numbers to bypass the firewall policies.

Since BitTorrent protocol might operate over TCP or UDP, it is important to consider both cases when implementing the firewall. It is also important to keep the state of each flow until it is completely classified. As a result, at least four packets are required to detect both BitTorrent traffic, which uses TCP as transport protocol, and YouTube. In the case of BitTorrent over UDP (uTP), at least three UDP packets are required for classification; however, this implementation keeps the state of five UDP packets to consider for retransmission of lost packets, as shown in figure 17.

The firewall module keeps the state of each flow inside a dictionary. The key of that dictionary is a unique combination of the source IP, destination IP, source port, and destination port. The value of each key consists of three elements: a counter, a string (signature), and a retransmission counter. The firewall module increments the first counter once a packet from an existing flow reaches the controller. The firewall module uses the signature along with TCP flags as a way to

insure that the packet has the expected direction. This prevents attackers from sending multiple TCP packets from one side in order to bypass the firewall rules. The second counter is used to count the number of retransmitted packets, which is helpful to prevent a denial-of-service attack by setting an upper limit for retransmission.

It is important to mention the following scenarios. First, if the first TCP packet does not have the SYN flag, the firewall module will drop it and will not pass it to the switch module. This prevents attackers from sending TCP packets without a SYN flag or from establishing a TCP connection, and waiting until the idle-time expires to send, for example, a BitTorrent handshake. However, a packet belonging to an inactive session that does not support keepalive messages will be dropped, according to this policy. A solution to this issue is to flag the flow entry with `OFPPF_SEND_FLOW_REM` flag to instruct the switch to send flow removed message to the controller, which responds by adding the key back to the firewall flow table. Second, the UDP implementation of the firewall shown in figure 17 considers only BitTorrent protocol handshake and does not take into consideration other protocols. This will add extra delay and reduce the performance of other applications that use UDP and do not support a TCP-like handshake as the performance results in the next chapter show.

The rest of the process is clearly explained in figure 16, 17, and 18. It is worth noting that the dashed line in figure 16 should be replaced with figure 17, which shows the process of only one direction (packets from inside-zone going to outside-zone) for simplicity. On the other hand, Figure 18 represents the switch module that is instructed by the firewall module to either install a flow entry in the OpenFlow switch or send the packet out without installing an entry. It is a modified version of the layer 2 learning switch that comes with POX controller.

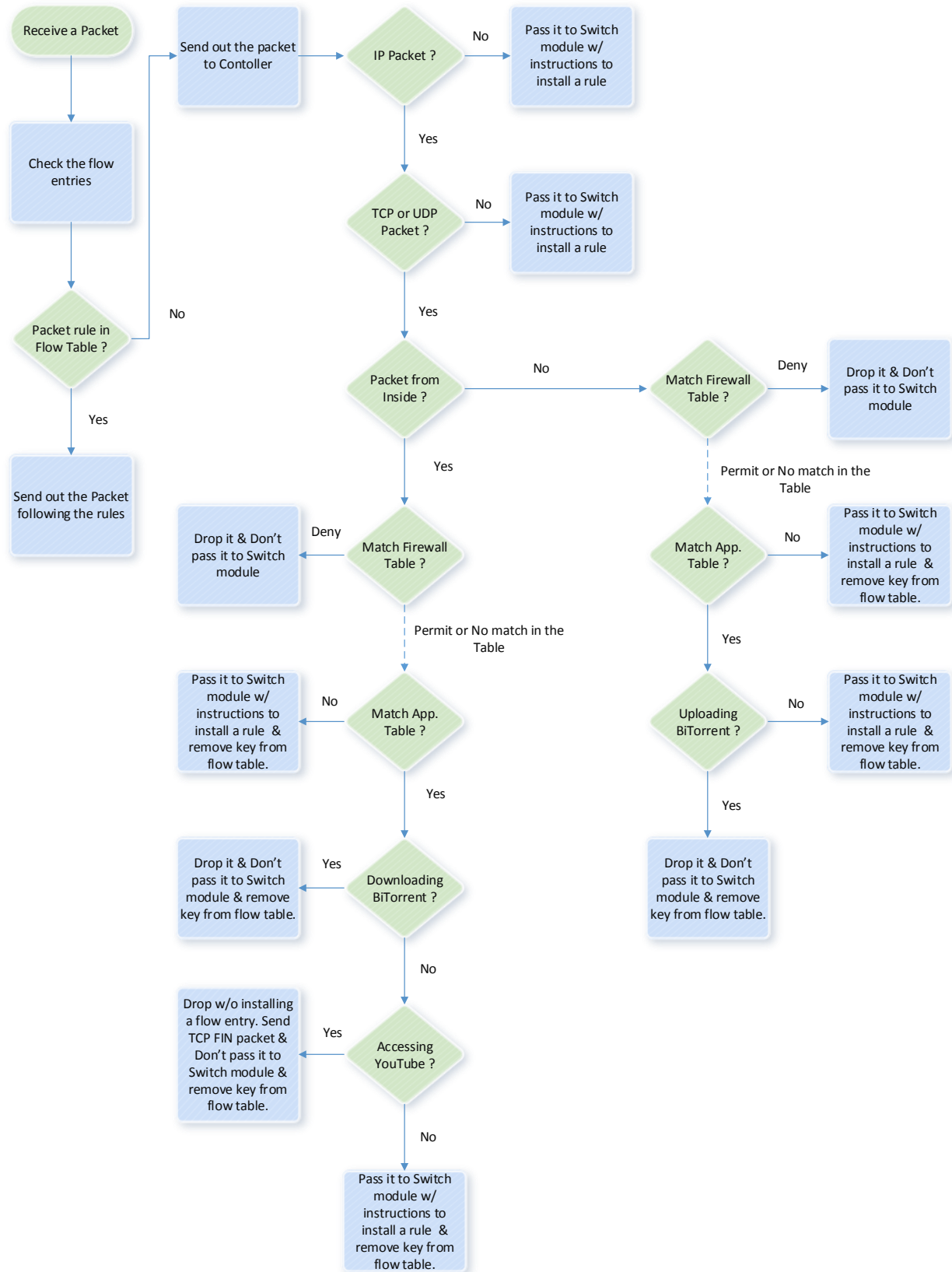


Figure 16: Flowchart 1 of Firewall Module

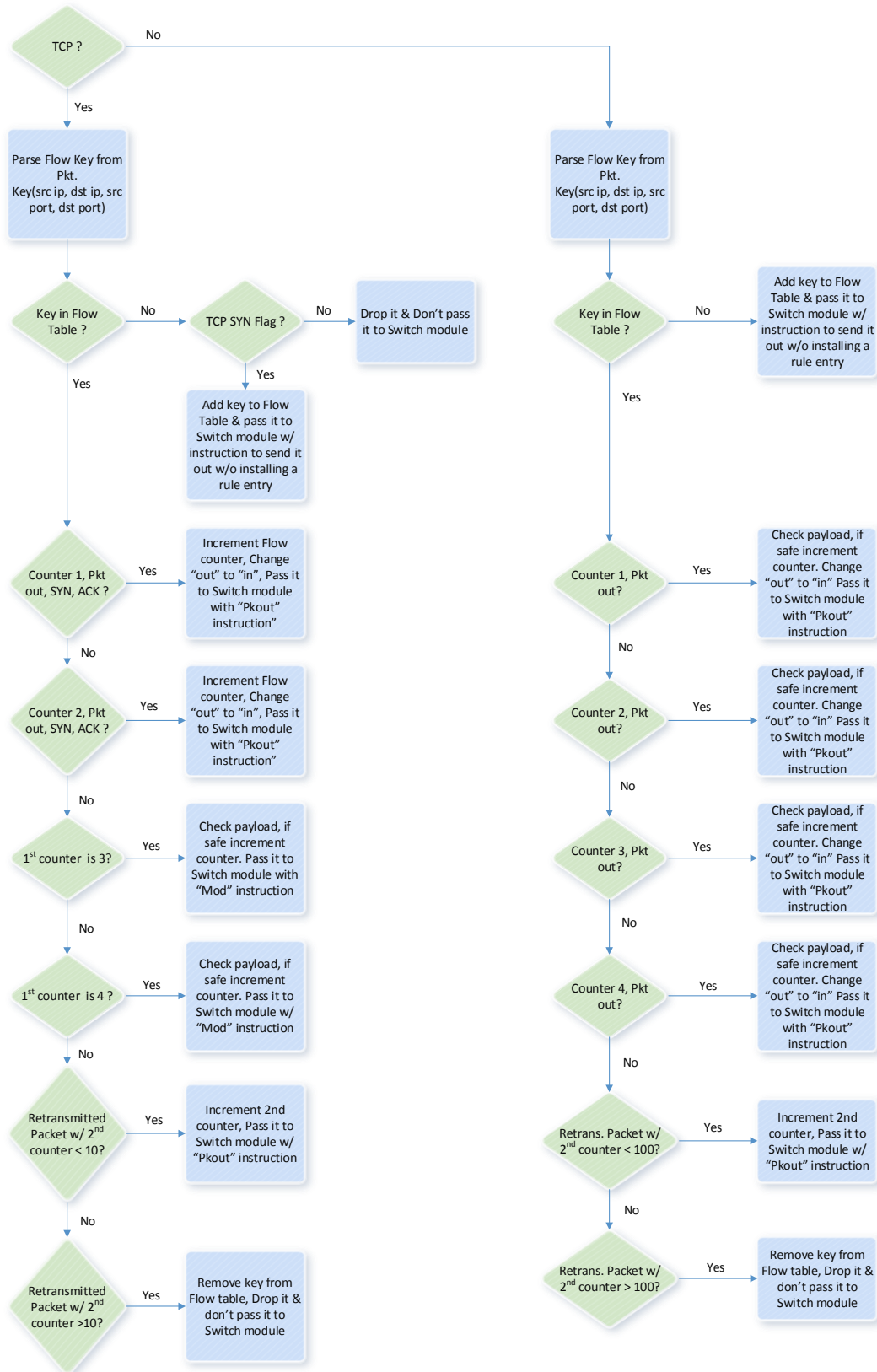


Figure 17: Flowchart 2 of Firewall Module

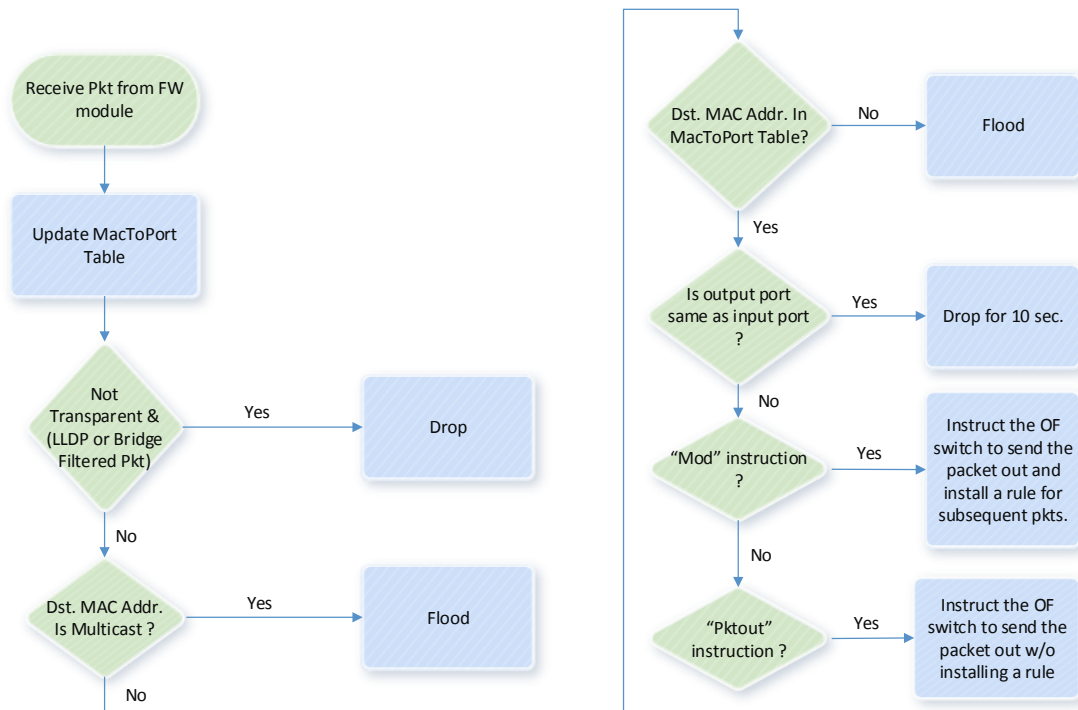


Figure 18: Flowchart of Switch Module

CHAPTER 4

PERFORMANCE ANALYSIS

This chapter evaluates the performance of the implemented firewall. It compares the performance of the firewall module, combined with the switch module, with the performance of a traditional switch and the switch module only.

The same test bed in figure 9 is used to conduct the performance analysis. IPERF [39], which is a tool for network performance measurement, is installed on both host one and host three. Each host has the same amount of memory and processing power. The bandwidth of all the links in this experiment is 1 Gbps. Pica8 switch is used also as a traditional switch since it supports both modes.

In the first test, IPERF is used as a client on one host and as a server on the other host to send parallel UDP streams. The controller process is killed and restarted each time after running IPERF to insure that the OpenFlow switch is ready for the next test and has no flow entries. Figure 19 shows the test results for multiple parallel streams and different modes. It is clear from figure 19 that the firewall module (the blue line) introduces extra delay to the UDP streams compared with the traditional switch and the switch module running in the controller. The reason is that the UDP implementation of the firewall module takes into account only uTP protocol (BitTorrent over UDP). Therefore, it 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 BitTorrent traffic. In the end, it is a trade-off between performance and security.

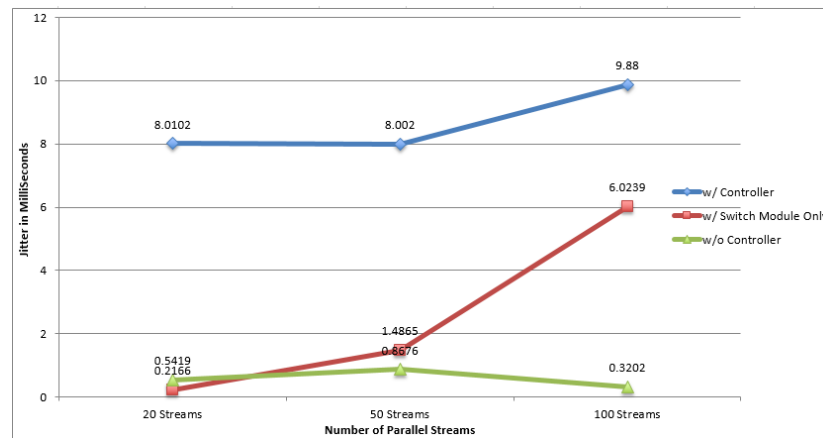


Figure 19: Comparison between Firewall Module, Switch Module only, and Traditional Switch in Term of Jitter

Even though the firewall module suffers from a poor performance for UDP packets due to its security policy, the second test proves that it has a good TCP performance compared to a traditional switch. In this test, IPERF is used to send parallel TCP streams. The average throughput of 20, 50, and 100 streams is calculated for both traditional switch and the firewall module, as shown in figure 20.

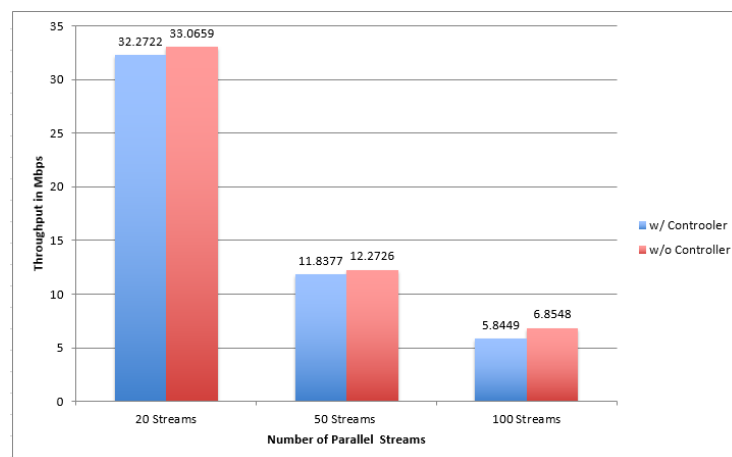


Figure 20: Comparison between Firewall Module and Traditional Switch in Term of Throughput

Figure 21 shows the effective throughput for 20 parallel TCP streams over conventional switch over 10 seconds. It is clear that the throughput in figure 21 is more stable than the throughput in figure 22, which is also for 20 parallel TCP streams but over an OpenFlow switch connected to POX controller running the firewall module. The reason is that the conventional switch uses the layer 2 forwarding table only to forward packets from host one to host three. The layer 2 forwarding table of a conventional switch is usually stored in a Content-Addressable Memory (CAM) and maps the MAC address to the egress port. It is worth noting that the switch uses the same entry in the MAC address table to forward all 20 parallel TCP streams. However, the OpenFlow switch uses TCAM to store the flow table that contains match fields up to layer 4. As a result, there is a unique entry for each TCP stream. Therefore, the difference in throughput between figure 21 and 22 is due to the difference in processing time. 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.

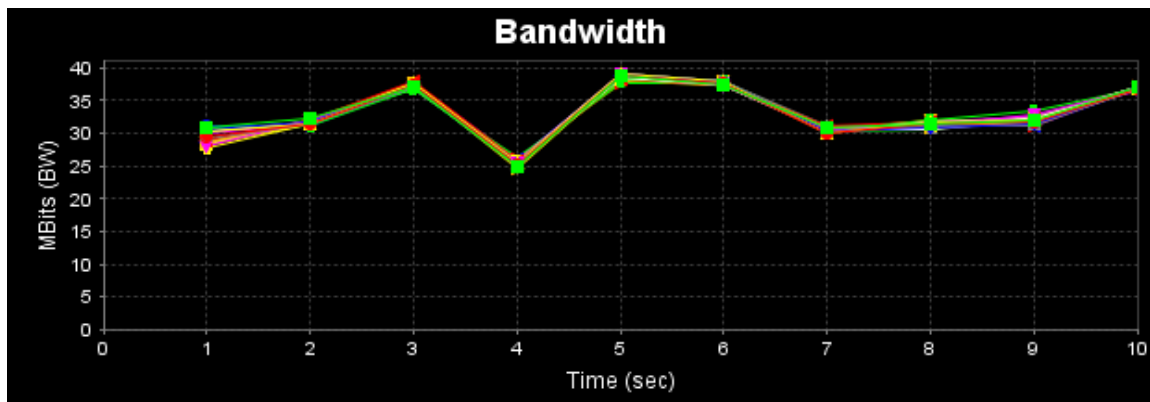


Figure 21: Throughput of 20 parallel TCP Steams over Conventional Switch

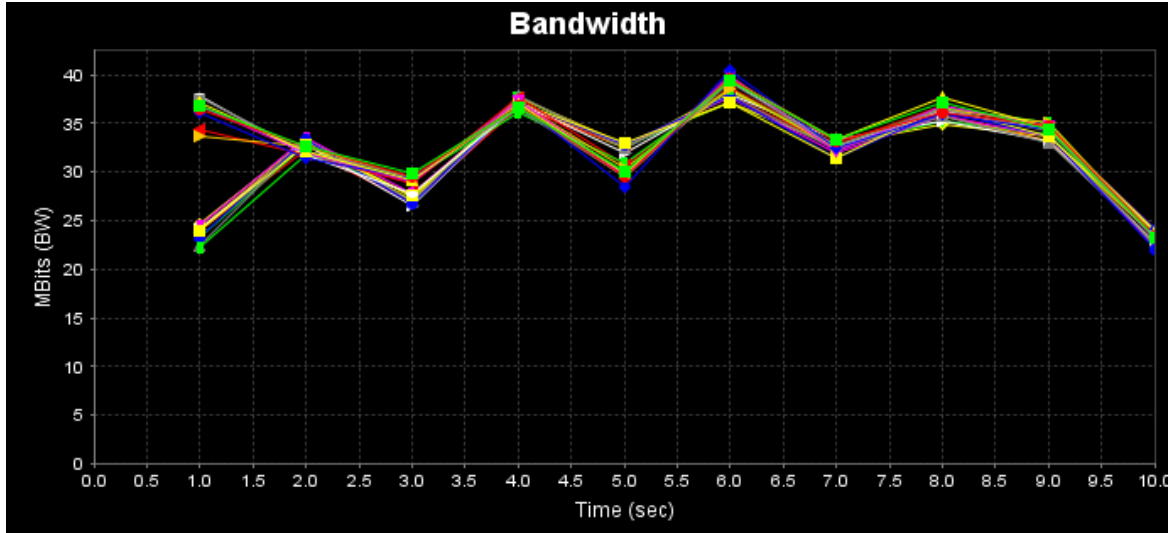


Figure 22: Throughput of 20 parallel TCP Steams with Controller Running Firewall Module

Since the OpenFlow switch is connected to the controller through only one 1 Gbps link, the parallel TCP streams will not reach the controller at the same time. Therefore, some TCP streams will arrive later than others, and this will affect their throughput, as shown by both figure 22 and figure 24 in second 1. Figures 23 and figure 24 both show throughput over 10 seconds after increasing the number of streams from 20 to 50 streams.

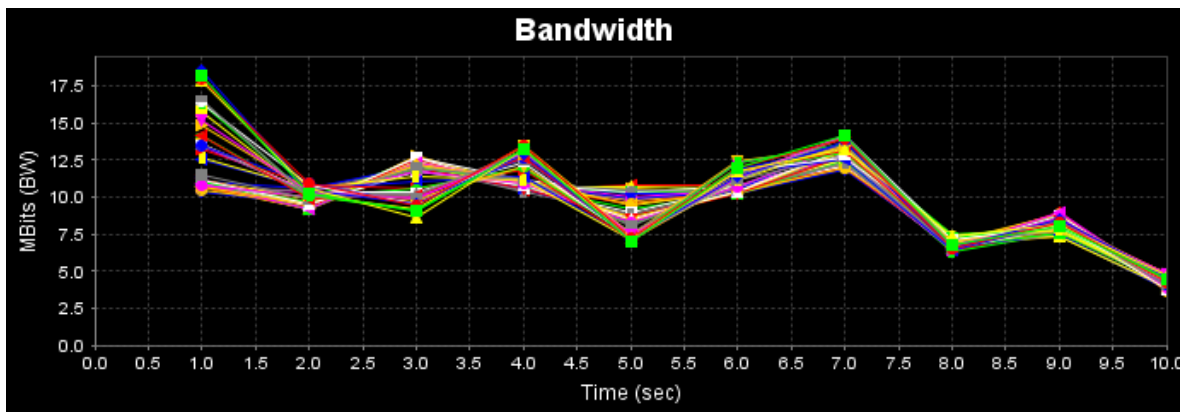


Figure 23: Throughput of 50 parallel TCP Steams over Conventional Switch

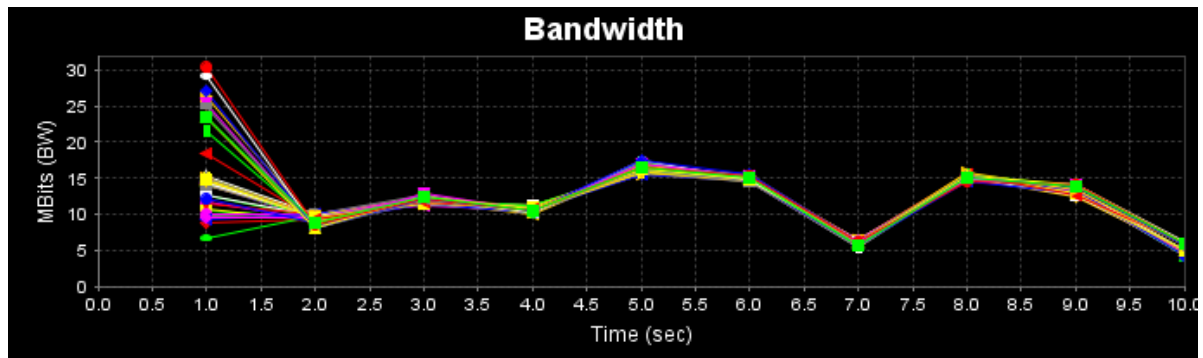


Figure 24: Throughput of 50 parallel TCP Streams with Controller Running Firewall Module

CHAPTER 5

FINDINGS

This paper demonstrates that it is possible to replace expensive specialized network hardware such as application layer firewalls with commodity devices and a logically centralized SDN controller. Even though this implementation used POX, a Python-based controller considered slower than Java or C++ controllers, the performance analysis proved that the firewall module performed well in handling TCP traffic compared with a traditional switch. Despite the successful implementation in terms of delivery of an application layer firewall capable of blocking BitTorrent traffic and restricting YouTube, some limitations must be acknowledged.

First, it is more difficult to consider all applications that use UDP as a transport protocol since UDP does not support flags like TCP.

Second, this implementation cannot detect encrypted traffic. Therefore, a user can easily bypass the firewall policies by running a VPN client. Future work will focus on this area as there are ways to analyze encrypted traffic such as packet size, direction, and timing [40].

Third, this implementation is not meant for production use; therefore, it does not support some features such as updating CSV files, which contain the firewall policies while the controller is running. Moreover, the implementation supports only one OpenFlow switch at this time. Future work will add more features and support for multiple OpenFlow switches.

Even though OpenFlow protocol makes it possible to program the network, it does not make it easy. Therefore, more advanced northbound APIs are required to provide an abstraction layer that enables the programmer to run parallel and subsequent modules/ applications without worrying about the buffer ID issue and policy conflicts.

The current OpenFlow switches allow only a fixed set of “Match-Action” fields. In addition, OpenFlow specifications define a limited set of action fields. Therefore, the data plane needs to be changed without modifying the hardware in order to support new protocols and higher layers [41]. Papers [41] and [42] propose methods to solve this issue. Future work will focus on this area as well.

CHAPTER 6

CONCLUSION

Firewall products are essential in the network to enforce security. Even though there are outstanding firewalls on the market, these firewalls are expensive, their provisioning time is high, and they do not provide programmability to network administrators. On the other hand, SDN allows the use of commodity hardware, reduces provisioning time, and provides a programmable control plane, as explained in chapter two. This paper implemented an OpenFlow-based application layer firewall capable of detecting applications such as BitTorrent and YouTube as well as preventing a DOS attack, as presented in chapter three. The fourth chapter of this paper highlighted the results of the performance analysis and compared the results with those of a traditional switch. The trade-off between the security and performance is inevitable in any kind of network architecture. Finally, SDN architecture is promising, and it provides the network administrators with flexibility and a room for innovation. However, OpenFlow protocol has some limitations. It is hoped that the SDN community will collaborate to enhance it further.

BIBLIOGRAPHY

- [1] M. Suh, S. H. Park, B. Lee, and 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] "Home - Open Networking Foundation." [Online]. Available: <https://www.opennetworking.org/>. [Accessed: 07-Jun-2014].
- [3] 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.
- [4] 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.
- [5] K. Ingham and S. Forrest, "A history and survey of network firewalls," *Univ. N. M. Tech Rep*, 2002.
- [6] "Firewalls and Internet Security - The Internet Protocol Journal - Volume 2, No. 2," *Cisco*. [Online]. Available: http://www.cisco.com/web/about/ac123/ac147/ac174/ac200/about_cisco_ipj_archive_article09186a00800c85ae.html. [Accessed: 12-May-2014].
- [7] J. Mogul, "Using screend to implement IP/TCP security policies," DTIC Document, 1991.
- [8] B. Cheswick, "The Design of a Secure Internet Gateway," in *in Proc. Summer USENIX Conference*, 1990, pp. 233–237.
- [9] 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.
- [10] W. Odom, *CCNA Routing and Switching 200-120 Official Cert Guide Library*, 1 edition. Cisco Press, 2013.

- [11] M. Roesch, “Snort: Lightweight Intrusion Detection for Networks.,” in *LISA*, 1999, vol. 99, pp. 229–238.
- [12] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, “Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers.”
- [13] 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. 61–66.
- [14] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks*, 1 edition. O’Reilly Media, 2013.
- [15] S. Azodolmolky, *Software Defined Networking with OpenFlow*. Packt Publishing, 2013.
- [16] E. Dimitriadou, K. Hornik, F. Leisch, D. Meyer, and A. Weingessel, *SDN: Software Defined Networks*. 2008.
- [17] “Software-Defined Networking: The New Norm for Networks.” ONF, 13-Apr-2012.
- [18] P. Southwick, D. Marschke, and H. Reynolds, *Junos Enterprise Routing: A Practical Guide to Junos Routing and Certification*, Second Edition edition. Beijing: O’Reilly Media, 2011.
- [19] D. Hucaby, *CCNP SWITCH 642-813 Official Certification Guide*, 1 edition. Indianapolis, Ind: Cisco Press, 2010.
- [20] 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.
- [21] 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. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.
- [22] “OpenFlow Switch Specification Version 1.3.3 (Protocol version 0x04).” ONF, 27-Sep-2013.
- [23] “OpenFlow Management and Configuration Protocol - OF-CONFIG 1.2.” ONF, 2014.

- [24] “OpenFlow Switch Specification Version 1.4.0 (Wire Protocol 0x05).” ONF, 14-Oct-2013.
- [25] A. Lara, A. Kolasani, and B. Ramamurthy, “Network Innovation using OpenFlow: A Survey,” *IEEE Commun. Surv. Tutor.*, vol. 16, no. 1, pp. 493–512, First 2014.
- [26] “OpenFlow version 1.3 tutorial | SDN Hub.”
- [27] 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.
- [28] 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. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.
- [29] “About NOX | NOXRepo.”
- [30] “About POX | NOXRepo.”
- [31] 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.
- [32] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular SDN Programming with Pyretic.”
- [33] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, “SoftCell: Taking control of cellular core networks,” *ArXiv Prepr. ArXiv13053568*, 2013.
- [34] P. Gurusanthosh, A. Rostami, and R. Manivasakan, “SDMA: A semi-distributed mobility anchoring in LTE networks,” in *Mobile and Wireless Networking (MoWNeT), 2013 International Conference on Selected Topics in*, 2013, pp. 133–139.
- [35] 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.
- [36] “POX Wiki - Open Networking Lab - Confluence.” [Online]. Available: <https://openflow.stanford.edu/display/ONL/POX+Wiki>. [Accessed: 07-Jun-2014].

- [37] B. Cohen, “The BitTorrent Protocol Specification.” [Online]. Available: http://bittorrent.org/beps/bep_0003.html. [Accessed: 08-Jun-2014].
- [38] A. Norberg, “uTorrent Transport Protocol.” [Online]. Available: http://bittorrent.org/beps/bep_0029.html. [Accessed: 08-Jun-2014].
- [39] “Iperf - The TCP/UDP Bandwidth Measurement Tool.” [Online]. Available: <http://iperf.fr/>. [Accessed: 08-Jun-2014].
- [40] C. V. Wright, F. Monroe, and G. M. Masson, “On inferring application protocol behaviors in encrypted network traffic,” *J. Mach. Learn. Res.*, vol. 7, pp. 2745–2769, 2006.
- [41] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN,” in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, 2013, pp. 99–110.
- [42] R. Ozdag, “Intel® Ethernet Switch FM6000 Series-Software Defined Networking,” *Intel Corp.*, p. 8, 2012.

APPENDIX

A. Firewall Code

'''

APPLICATION LAYER FIREWALL USING OPENFLOW

By

Alaauddin Shieha

This project is part of a thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Master of Science

This project intends to use the POX SDN controller and Pica8 OpenFlow enabled Switch to implement an application layer (layer 7) Firewall capable of detecting and blocking BitTorrent and YouTube traffic as well as preventing some kinds of DOS attacks.

The firewall table matching rules will be based on source and destination IP addresses and port numbers.

The application table supports two applications: BitTorrent and YouTube. It blocks these applications if the source IP address of the packet matches the App_table.

The POX build-in L2 learning switch module is combined in this implementation to provide forwarding service.

The firewall-policies.csv file stores the rules that the firewall will be using to build the firewall table and then create flow entries to drop matched packets.

'''

```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.util import dpid_to_str
from pox.lib.util import str_to_bool
from pox.lib.util import dpidToStr
from pox.lib.addresses import EthAddr
from pox.lib.addresses import IPAddr
from pox.lib.packet import *
from pox.lib.packet.packet_base import packet_base
from types import *
import os
import csv
import time
```

```
log = core.getLogger()
```

```
# We don't want to flood immediately when a switch connects.
# Can be overridden on command line.
```

```

_flood_delay = 0
PolicyFile = "%s/pox/pox/misc/firewall-policies.csv" % os.environ[ 'HOME' ] # The file that
contains the rules
ApplicationFile = "%s/pox/pox/misc/Applications.csv" % os.environ[ 'HOME' ] # The file that
contains the rules

```

class Master (EventMixin):

```

'''
Master class that listens to multiple events and decides which module should start first.
Inherit from EventMixin. More information on EventMixin could be found in the link below:
https://github.com/noxrepo/pox/blob/carp/pox/lib/revent/revent.py
'''

def __init__ (self, transparent):
'''
Especial method that gets invoked when the Master object gets created.
It listens to events on core.openflow (source) and creates an empty firewall_table (dic),
flow_table, MacToPort table, and application table
For more info: https://github.com/noxrepo/pox/blob/carp/pox/lib/revent/revent.py
'''

self.listenTo(core.openflow)
self.transparent = transparent
core.openflow.addListener(self, "all")
log.debug("Enabling Firewall Module")
self.firewall_table = {}
self.flow_table = {}
self.macToPort = {}
self.App_table = {}
self.hold_down_expired = _flood_delay == 0
self.insideNetwork = ["172.20.74.0/24"]

def _handle_ConnectionUp (self, event):
'''
This method has a special meaning to listenTo() and/or addListeners()
The method is automatically registered as an event handler, which will be called whenever
core.openflow triggers a ConnectionUp event.
It builds the firewall table and the application table when the switch connects to the
controller.
For more info: https://github.com/noxrepo/pox/blob/carp/pox/lib/revent/revent.py
'''

with open(PolicyFile, 'rb') as f:
    reader = csv.reader(f)
    for row in reader:
        if row[1] != "srcip":

```

```

        self.AddPolicy(dpidToStr(event.dpid), IPAddr(row[1]), IPAddr(row[2]),
int(row[3]), self.Str_Bool(row[4]))

    with open(ApplicationFile, 'rb') as A:
        Apps = csv.reader(A)
        for row in Apps:
            if row[1] != "Application":
                self.Add_App(str(row[1]), IPAddr(row[2]), self.Str_Bool(row[3]))

def Add_App(self, app="", host="", value=False):
    """
    This method adds the applications in the CSV file to the application table (App_table)
    """
    self.App_table[(app, host)] = value
    log.debug("Building Application Table by adding App policy: For: %s, From: %s, Action:
%s", app, host, str(value))

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

def AddPolicy(self, dpidstr, srcip="", dstip="", port=0, value=False):
    """
    This method adds the policies to the dictionary and sets the value to False if it was not
    specified in the csv file.
    """
    self.firewall_table[(dpidstr, srcip, dstip, port)] = value
    self.firewall_table[(dpidstr, dstip, srcip, port)] = value
    log.debug("Building Firewall Table by adding firewall policy: From: %s, To: %s, Action:
%s", srcip, dstip, str(value))

def checkIPinside(self, ip, event):
    """
    Check if the IP is from inside the network or not. Check IN-Port to make sure that the IP is
    not spoofed to look like it's coming
    from outside.
    """
    for network in self.insideNetwork:
        if ip.inNetwork(network) or event.port != 1:

```

```
    return True
return False
```

```
def _handle_all_PacketIn (self, event):
    """
    This method has a special meaning to addListeners()
    It will be called whenever core.openflow triggers a PacketIn event.
    It calls Firewall method first and then based on the value returned by the firewall method it
    might call the LearningSwitch method
    with instruction passed as an argument.
    """
    SendToSwitch = self.Firewall(event)
    if SendToSwitch == "Mod":
        self.LearningSwitch(event, "Mod")
    if SendToSwitch == "Pktout":
        self.LearningSwitch(event, "Pktout")
```

```
def Firewall(self, event):
    """
    This method parses the packet and check it against the firewall table if it is either TCP or
    UDP, and against the application table
    if the source IP address is in the application table. It returns "Mod" or "Pktout" to instruct
    the switch method or it drops the packet
    and return False, which means does not pass this packet to the switch method.
    """
```

```
def Packet_out_Send_FIN(event, segmant, ip, packet):
    """
    This method is part of the firewall method and it is used only when the packet is an
    HTTP GET message destined for YouTube website.
    It drops the packet without installing a flow entry, and builds a TCP FIN packet and
    sends it back to the host in order to close its connection.
    """
```

```
    msg = of.ofp_packet_out()
    msg.data = event.ofp
    event.connection.send(msg)

    tcp_packet = tcp()
    tcp_packet.srcport = segmant.dstport
    tcp_packet.dstport = segmant.srcport
    tcp_packet.len = tcp.MIN_LEN+20
    tcp_packet.win = 1
    tcp_packet._setflag(tcp_packet.FIN_flag,1)
    tcp_packet._setflag(tcp_packet.ACK_flag,1)
    tcp_packet.seq = segmant.ack
```

```

tcp_packet.ack = segmant.seq + segmant.payload_len
ipv4_packet = ipv4()
ipv4_packet.iplen = ipv4.MIN_LEN + tcp_packet.len
ipv4_packet.protocol = ipv4.TCP_PROTOCOL
ipv4_packet.dstip = ip.dstip
ipv4_packet.srcip = ip.srcip
ipv4_packet.set_payload(tcp_packet)
eth_packet = ethernet()
eth_packet.set_payload(ipv4_packet)
eth_packet.src = packet.dst #MAC Controller
eth_packet.dst = packet.src
eth_packet.type = ethernet.IP_TYPE
msg = of.ofp_packet_out(data = eth_packet)
msg.actions.append(of.ofp_action_output(port = event.port))
event.connection.send(msg)

```

```

def Applications(event, segmant, ip, packet, app, direction):

```

```

    """
    This method checks the packet payload for application signatures. If a signature is found,
    it will drop the packet or call Packet_out_Send_FIN
    and delete the key from the flow table.
    """

```

```

    if app == "BitTorrent":
        signature = "BitTorrent protocol"
    elif app == "Youtube":
        signature = "Host: www.youtube.com"

```

```

    if isinstance(segmant.payload, packet_base) or isinstance(segmant.payload, NoneType):
        return True

```

```

    elif signature in segmant.payload:
        if signature == "BitTorrent protocol":
            self.Drop(event, ip.srcip, ip.dstip, segmant.dstport, "dst")
            log.debug("This is BitTorrent Handshake and drop message has been sent")
        elif signature == "Host: www.youtube.com" and direction == "inside":
            Packet_out_Send_FIN(event, segmant, ip, packet)
            log.debug("This is HTTP GET for Youtube.com. The packet has been dropped and
TCP FIN Packet to close the connection")

```

```

    if direction == "inside":
        key = (ip.srcip, ip.dstip, segmant.srcport, segmant.dstport)
    else:
        key = (ip.dstip, ip.srcip, segmant.dstport, segmant.srcport)
    try:
        del self.flow_table[key]

```



```

    except KeyError:
        log.debug("Another BitTorrent Handshake coming right after deleting the flow from
the other side of the connection")
        return False

    dpidstr = dpidToStr(event.dpid) #to Convert a DPID in the canonical string form into a long
int for more info: https://github.com/noxrepo/pox/blob/carp/pox/lib/util.py
    packet = event.parsed
    ip = packet.find('ipv4')# to check whether it is an IP packet or not
    if ip is None:
        log.debug("This packet isn't IP!")
        return "Mod"# if it is not IP, do not do anything (let the switch method handles it).

    if packet.find('tcp') is None and packet.find('udp') is None:
        return "Mod"# if it is neither TCP nor UDP send it to switch method with "Mod"
instruction.
    else:
        if packet.find('tcp') is not None:
            segmant = packet.find('tcp')### TCP Packet #####
            if self.checkIPinside(ip.srcip, event):

                if self.CheckPolicy(dpidstr, ip.srcip, ip.dstip, segmant.dstport) == False:
                    self.Drop(event, ip.srcip, ip.dstip, segmant.dstport, "dst")
                    log.debug("Packet matched the role and dropped")
                    return False

            key = (ip.srcip, ip.dstip, segmant.srcport, segmant.dstport)# Build the flow table
            if key in self.flow_table:
                if self.flow_table[key][:2] == [1, "out"] and segmant.SYN and segmant.ACK:
                    self.flow_table[key][0] += 1
                    self.flow_table[key][1] = "in"
                    log.debug("TCP SYN ACK packet from inside")
                    return "Pktout"

                elif self.flow_table[key][:2] == [2, "out"] and segmant.ACK:
                    self.flow_table[key][0] += 1
                    self.flow_table[key][1] = "in"
                    log.debug("TCP ACK Packet from inside ")
                    return "Pktout"

                elif self.flow_table[key][0] == 3 and segmant.PSH:
                    for App in self.App_table:
                        if App[1] == ip.srcip:
                            to_switch_module = Applications(event, segmant, ip, packet, App[0],
"inside")

```

```

        log.debug("Packet from inside will be sent to switch ? %s",
str(to_switch_module))
        if to_switch_module is False:
            return False

        self.flow_table[key][0] += 1
        log.debug("TCP handshake is done. First packet of connection from inside.
MOD is done ")
        return "Mod"

    elif self.flow_table[key][0] == 4:
        for App in self.App_table:
            if App[1] == ip.srcip:
                to_switch_module = Applications(event, segmant, ip, packet, App[0],
"inside")
                log.debug("Packet from inside will be sent to switch ? %s",
str(to_switch_module))
                if to_switch_module is False:
                    return False

                log.debug("TCP handshake is done. Second packet of connection from inside.
MOD is done ")
                del self.flow_table[key] # To remove that element (key) from the dictionary in
order to avoid having a huge flow-table
                return "Mod"

            elif self.flow_table[key][2] < 10:# To take into account retransmission. It has an
upper limit to prevent DOS
                self.flow_table[key][2] += 1
                return "Pktout"

            elif self.flow_table[key][2] > 10:
                del self.flow_table[key]
                log.debug("DOS attack detected")
                self.Drop(event, ip.srcip, ip.dstip, segmant.dstport, "dst")
                return False

        else:## # First Packet from the flow #####
            if segmant.SYN:#a good place to prevent illegal flags combinations
                self.flow_table[key] = [1, "in", 0]# the first parameter is a counter for the
number of packets from that flow, the second is direction, third to prevent from DOS
                log.debug("TCP SYN Packet from inside ")
                return "Pktout"
            else: ##### First packet from flow is not SYN. To prevent installing a flow
entry if user tries to send first packet without flags in order to pass fw

```

```

        log.debug("First or middle of session TCP packet but doesn't have a SYN flag
(inside)")

        return False

    else:

        if self.CheckPolicy(dpidstr, ip.srcip, ip.dstip, segmant.srcport) == False:
            self.Drop(event, ip.srcip, ip.dstip, segmant.srcport, "src")
            return False

        key = (ip.dstip, ip.srcip, segmant.dstport, segmant.srcport)
        if key in self.flow_table:
            if self.flow_table[key][:2] == [1, "in"] and segmant.SYN and segmant.ACK:
                self.flow_table[key][0] += 1
                self.flow_table[key][1] = "out"
                log.debug("TCP SYN ACK packet from outside")
                return "Pktout"

            elif self.flow_table[key][:2] == [2, "in"] and segmant.ACK:
                self.flow_table[key][0] += 1
                self.flow_table[key][1] = "out"
                log.debug("TCP ACK Packet from outside")
                return "Pktout"

            elif self.flow_table[key][0] == 3 and segmant.PSH:
                for App in self.App_table:
                    if App[1] == ip.dstip:
                        to_switch = Applications(event, segmant, ip, packet, App[0], "outside")
                        log.debug("Packet from outside will be sent to switch ? %s",
str(to_switch))

                        if to_switch is False:
                            return False

                self.flow_table[key][0] += 1
                log.debug("TCP handshacke is done. First packet of connection from outside.
MOD is done ")
                return "Mod"

            elif self.flow_table[key][0] == 4:
                for App in self.App_table:
                    if App[1] == ip.dstip:
                        to_switch = Applications(event, segmant, ip, packet, App[0], "outside")
                        log.debug("Packet from outside will be sent to switch ? %s",
str(to_switch))

                        if to_switch is False:

```

```

        return False

    del self.flow_table[key]# To remove that element (key) from the dictionary in
order to avoid having a huge flow-table
    return "Mod"

    elif self.flow_table[key][2] < 10:# To account for retransmission
        self.flow_table[key][2] += 1
        return "Pktout"

    elif self.flow_table[key][2] > 10:
        del self.flow_table[key]
        log.debug("DOS attack detected")
        self.Drop(event, ip.srcip, ip.dstip, segmant.dstport, "dst")
        return False

    else:
        if segmant.SYN:
            self.flow_table[key] = [1, "out", 0]# the first parameter is a counter for the
number of packets from that flow, the second is direction, third to prevent from DOS
            log.debug("Switche Module: TCP SYN Packet from outside ")
            return "Pktout"
        else:
            log.debug("Switche Module: first or middle of session TCP packet but doesn't
have a flag (outside)")
            return False

    else:
        segmant = packet.find('udp') ##### UDP Packet #####

    if self.checkIPinside(ip.srcip, event):

        if self.CheckPolicy(dpidstr, ip.srcip, ip.dstip, segmant.dstport) == False:
            self.Drop(event, ip.srcip, ip.dstip, segmant.dstport, "dst")
            log.debug("Packet matched the role and dropped")
            return False

        key = (ip.srcip, ip.dstip, segmant.srcport, segmant.dstport)
        if key in self.flow_table:
            if self.flow_table[key][:2] == [1, "out"]:
                for App in self.App_table:
                    if App[1] == ip.srcip:
                        to_switch_module = Applications(event, segmant, ip, packet, App[0],
"inside")

```

```

        log.debug("Packet from inside will be sent to switch ? %s",
str(to_switch_module))
        if to_switch_module is False:
            return False

        self.flow_table[key][0] += 1
        self.flow_table[key][1] = "in"
        log.debug("UDP Packet from inside. Counter is 2 now")
        return "Pktout"

    elif self.flow_table[key][:2] == [2, "out"]:
        for App in self.App_table:
            if App[1] == ip.srcip:
                to_switch_module = Applications(event, segmant, ip, packet, App[0],
"inside")
                log.debug("Packet from inside will be sent to switch ? %s",
str(to_switch_module))
                if to_switch_module is False:
                    return False

                self.flow_table[key][0] += 1
                self.flow_table[key][1] = "in"
                log.debug("UDP Packet from inside. Counter is 3 now")
                return "Pktout"

    elif self.flow_table[key][:2] == [3, "out"]:
        for App in self.App_table:
            if App[1] == ip.srcip:
                to_switch_module = Applications(event, segmant, ip, packet, App[0],
"inside")
                log.debug("Packet from inside will be sent to switch ? %s",
str(to_switch_module))
                if to_switch_module is False:
                    return False

                self.flow_table[key][0] += 1
                self.flow_table[key][1] = "in"
                log.debug("UDP Packet from inside. Counter is 4 now")
                return "Pktout"

    elif self.flow_table[key][:2] == [4, "out"]:
        for App in self.App_table:
            if App[1] == ip.srcip:
                to_switch_module = Applications(event, segmant, ip, packet, App[0],
"inside")

```

```

        log.debug("Packet from inside will be sent to switch ? %s",
str(to_switch_module))
        if to_switch_module is False:
            return False

        del self.flow_table[key]
        log.debug("UDP flow is safe. Flow is deleted from flow table")
        return "Mod"

    elif self.flow_table[key][2] < 100:
        self.flow_table[key][2] += 1
        log.debug("UDP packets that might belong to another type of applications or
retransmission")
        return "Pktout"

    elif self.flow_table[key][2] > 100:
        del self.flow_table[key]
        log.debug("A possible DOS attack")
        self.Drop(event, ip.srcip, ip.dstip, segment.dstport, "dst")
        return False

    else:
        self.flow_table[key] = [1, "in", 0]
        log.debug("First UDP Packet from inside.")
        return "Pktout"

    else:

        if self.CheckPolicy(dpidstr, ip.srcip, ip.dstip, segment.srcport) == False:
            self.Drop(event, ip.srcip, ip.dstip, segment.srcport, "src")
            return False

        key = (ip.dstip, ip.srcip, segment.dstport, segment.srcport)
        if key in self.flow_table:
            if self.flow_table[key][:2] == [1, "in"]:
                for App in self.App_table:
                    if App[1] == ip.dstip:
                        to_switch = Applications(event, segment, ip, packet, App[0], "outside")
                        log.debug("Packet from outside will be sent to switch ? %s",
str(to_switch))

                        if to_switch is False:
                            return False

                self.flow_table[key][0] += 1
                self.flow_table[key][1] = "out"
                log.debug("UDP Packet from inside. Counter is 2 now")

```

```

        return "Pktout"

    if self.flow_table[key][:2] == [2, "in"]:
        for App in self.App_table:
            if App[1] == ip.dstip:
                to_switch = Applications(event, segmant, ip, packet, App[0], "outside")
                log.debug("Packet from outside will be sent to switch ? %s",
str(to_switch))
                if to_switch is False:
                    return False

                self.flow_table[key][0] += 1
                self.flow_table[key][1] = "out"
                log.debug("UDP Packet from inside. Counter is 3 now")
                return "Pktout"

    if self.flow_table[key][:2] == [3, "in"]:
        for App in self.App_table:
            if App[1] == ip.dstip:
                to_switch = Applications(event, segmant, ip, packet, App[0], "outside")
                log.debug("Packet from outside will be sent to switch ? %s",
str(to_switch))
                if to_switch is False:
                    return False

                self.flow_table[key][0] += 1
                self.flow_table[key][1] = "out"
                log.debug("UDP Packet from inside. Counter is 4 now")
                return "Pktout"

    elif self.flow_table[key][:2] == [4, "in"]:
        for App in self.App_table:
            if App[1] == ip.dstip:
                to_switch = Applications(event, segmant, ip, packet, App[0], "outside")
                log.debug("Packet from outside will be sent to switch ? %s",
str(to_switch))
                if to_switch is False:
                    return False

        del self.flow_table[key]
        log.debug("UDP flow is safe. Flow is deleted from flow table")
        return "Mod"

    elif self.flow_table[key][2] < 100:
        self.flow_table[key][2] += 1

```

```
log.debug("UDP packets that might belong to another type of applications or  
retransmission")
```

```
return "Pktout"
```

```
elif self.flow_table[key][2] > 100:
```

```
del self.flow_table[key]
```

```
log.debug("A possible DOS attack")
```

```
self.Drop(event, ip.srcip, ip.dstip, segment.dstport, "dst")
```

```
return False
```

```
else:
```

```
self.flow_table[key] = [1, "out", 0]
```

```
log.debug("First UDP Packet from outside.")
```

```
return "Pktout"
```

```
def CheckPolicy(self, dpidstr, srcip="", dstip="", port=0):
```

```
"""
```

This method checks the src ip, dst ip, and port number of the packet against the rules and return the value of the corresponding policy.

It has an implicit True so that the switch can handle un-matched packets for ease of implementation.

```
"""
```

```
key = (dpidstr, srcip, dstip, port)
```

```
if key in self.firewall_table:
```

```
log.debug('Policy From (%s) to (%s) found in %s. The Action is (%s): ', srcip, dstip,  
dpidstr, str(self.firewall_table[key]))
```

```
return self.firewall_table[key]
```

```
else:
```

```
log.debug('Policy From (%s) to (%s) NOT found in %s: Pass to Switch Module', srcip,  
dstip, dpidstr)
```

```
return True # Implicit True
```

```
def Drop (self, event, src, dst, port, port_type):
```

```
"""
```

This method Drops this packet and installs a flow to continue dropping similar ones.

Since no action has been added in the flow table modification message, the default is drop.

For more info on open flow packets:

<https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-OpenFlowMessages>

```
"""
```

```
msg = of.ofp_flow_mod() #creates a flow modification message
```

```
msg.match = of.ofp_match.from_packet(event.parsed, event.port)
```



```

msg.match.dl_dst = None
msg.idle_timeout = 120
msg.priority = 65535 #priority at which a rule will match, higher is better.
msg.command = of.OFPFC_MODIFY
msg.data = event.ofp
event.connection.send(msg)# send the message to the OpenFlow switch
log.debug("Firewall dropped packet from (%s) to (%s) and installed a rule on %s", src, dst,
dpidToStr(event.dpid))

```

```

def LearningSwitch (self, event, whattodo):

```

```

    """

```

The learning switch "brain" associated with a single OpenFlow switch. When we see a packet, we'd like to output it on a port which will eventually lead to the destination. To accomplish this, we build a table that maps addresses to ports.

We populate the table by observing traffic. When we see a packet from some source coming from some port, we know that source is out that port.

When we want to forward traffic, we look up the destination in our table. If we don't know the port, we simply send the message out all ports except the one it came in on. (In the presence of loops, this is bad!).

In short, our algorithm looks like this:

For each packet from the switch:

- 1) Use source address and switch port to update address/port table
- 2) Is transparent = False and either Ether type is LLDP or the packet's destination address is a Bridge Filtered address?
 - Yes:
 - 2a) Drop packet -- don't forward link-local traffic (LLDP, 802.1x)
 - DONE
- 3) Is destination multicast?
 - Yes:
 - 3a) Flood the packet
 - DONE
- 4) Port for destination address in our address/port table?
 - No:
 - 4a) Flood the packet
 - DONE
- 5) Is output port the same as input port?
 - Yes:
 - 5a) Drop packet and similar ones for a while

- 6) Install flow table entry in the switch so that this flow goes out the appropriate port
- 6a) Send the packet out appropriate port

"""

packet = event.parsed

def flood (message = **None**):

""" Floods the packet """

msg = of.ofp_packet_out()

if time.time() - event.connection.connect_time >= _flood_delay:

Only flood if we've been connected for a little while...

if self.hold_down_expired **is** **False**:

Oh yes it is!

self.hold_down_expired = **True**

log.info("%s: Flood hold-down expired -- flooding",
dpid_to_str(event.dpid))

if message **is not None**: log.debug(message)

#log.debug("%i: flood %s -> %s", event.dpid, packet.src, packet.dst)

OFPP_FLOOD is optional; on some switches you may need to change

this to OFPP_ALL.

msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))

else:

pass

#log.info("Holding down flood for %s", dpid_to_str(event.dpid))

msg.data = event.ofp

msg.in_port = event.port

event.connection.send(msg)

def drop (duration = **None**):

"""

Drops this packet and optionally installs a flow to continue dropping similar ones for a while

"""

if duration **is not None**:

if not isinstance(duration, tuple):

duration = (duration, duration)

msg = of.ofp_flow_mod()

msg.match = of.ofp_match.from_packet(packet)

msg.match.dl_dst = **None**

msg.idle_timeout = duration[0]

msg.hard_timeout = duration[1]

msg.buffer_id = event.ofp.buffer_id

```

        event.connection.send(msg)
    elif event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out()
        msg.buffer_id = event.ofp.buffer_id
        msg.in_port = event.port
        event.connection.send(msg)

def Packet_out(port):
    """
    Send this packet out without installing a flow entry.
    """
    msg = of.ofp_packet_out()
    msg.actions.append(of.ofp_action_output(port = port))
    msg.data = event.ofp
    event.connection.send(msg)

def Mod_Message(packet, port):
    """
    Send this packet out and install a flow entry for subsequent packets
    """
    msg = of.ofp_flow_mod()
    msg.match = of.ofp_match.from_packet(packet, event.port)
    msg.match.dl_dst = None
    msg.idle_timeout = 60
    msg.actions.append(of.ofp_action_output(port = port))
    msg.buffer_id = event.ofp.buffer_id
    event.connection.send(msg)

self.macToPort[packet.src] = event.port # 1

if not self.transparent: # 2
    if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered():
        drop() # 2a
        return

if packet.dst.is_multicast:
    flood() # 3a
else:
    if packet.dst not in self.macToPort: # 4
        flood("Port for %s unknown -- flooding" % (packet.dst,)) # 4a
    else:
        port = self.macToPort[packet.dst]
        if port == event.port: # 5

```

```

# 5a
log.warning("Same port for packet from %s -> %s on %s.%s. Drop."
           % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
drop(10)
return
# 6
if whattodo == "Mod":
    Mod_Message(packet, port)
elif whattodo == "Pktout":
    Packet_out(port)

def launch (transparent=False, hold_down=_flood_delay):
    """
    Starts The Master class.
    """
    try:
        global _flood_delay
        _flood_delay = int(str(hold_down), 10)
        assert _flood_delay >= 0
    except:
        raise RuntimeError("Expected hold-down to be a number")

core.registerNew(Master, str_to_bool(transparent))

```

B. Pica8 Configuration

1. reboot
2. Choose OpenFlow
3. Choose manual configuration
4. Type the following command:
`ovsdb-tool create /ovs/ovs-vswitchd.conf.db /ovs/bin/vswitch.ovsschema`
5. Statically configure the IP address through the command:
`ifconfig eth0 192.168.100.2 netmask 255.255.255.0 up`
6. Start the OVS Database Server:
`ovsdb-server /ovs/ovs-vswitchd.conf.db --remote=ptcp:6633:192.168.100.2 &`
7. Start the OVS Daemon:
`ovs-vswitchd tcp:192.168.100.2:6633 --pidfile=pica8 --overwrite-pidfile >/var/log/ovs.log 2>/dev/null &`
8. Create the bridge and add ports to it:
`ovs-vsctl --db=tcp:192.168.100.2:6633 add-br br0 -- set bridge br0 datapath_type=pica8`
`ovs-vsctl --db=tcp:192.168.100.2:6633 add-port br0 ge-1/1/1 -- set Interface ge-1/1/1 type=pica8`
`ovs-vsctl --db=tcp:192.168.100.2:6633 add-port br0 ge-1/1/13 -- set Interface ge-1/1/13 type=pica8`
`ovs-vsctl --db=tcp:192.168.100.2:6633 add-port br0 ge-1/1/14 -- set Interface ge-1/1/14 type=pica8`
`ovs-vsctl --db=tcp:192.168.100.2:6633 add-port br0 ge-1/1/15 -- set Interface ge-1/1/15 type=pica8`
9. Add the controller to the bridge:
`ovs-vsctl --db=tcp:192.168.100.2:6633 set-controller br0 tcp:192.168.100.1:6633`

A useful show commands:

- `ovs-ofctl show br0`
- `ovs-ofctl dump-flows br0`
- `ovs-ofctl dump-ports br0`