Early Detection and Mitigation of DDoS Attacks In Software Defined Networks

by

Maryam Kia

A Thesis Presented to the School of Graduate Studies at
Ryerson University
In partial fulfillment of the
Requirements for the degree of
Master of Applied Science
In the program of
Computer Networks

Toronto, Ontario, Canada, 2015

© Maryam Kia 2015

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.
I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Early Detection and Mitigation of DDoS Attacks in Software Defined Networks

© Maryam Kia 2015

Master of Applied Science
In Computer Networks
Ryerson University

Abstract

Software Defined networking (SDN) is a new approach for the design and management of computer networks. The main concept behind SDN is the separation of the network's control and forwarding planes with the control plane moved to the centralized controller. In SDN networks with the centralized controller structure DDoS attacks can easily exhaust the controller's or the switches' computing and communication resources, thus, breakdown the network within a short time.

In this thesis, the scheme, running at the controller, can detect DDoS attacks at the early stage. The method not only can detect the attacks but also identify the attacking paths and start a mitigation process to provide some degree of protection of the network devices the moment an attack is detected. The Proposed method is based on the Entropy variation of destination IP address, Flow initiation rate and study of the Flow specifications.

Acknowledgement

I would like to express my sincere appreciation to my supervisor Dr. Ngok-Wah Ma, and my co-supervisor Dr. Xiaoli Li, for their continuous support, patience, motivation, encouragement and time throughout my graduate studies. It was a great experience to study and work under their supervision throughout my research work and thesis writing.

Special thanks to the Computer Networks Master's program and the school of Graduate studies. It was an immense opportunity to study in this great citadel of learning.

Last, but not least, I would like to thank my husband for his inspiration, encouragement and support, my mom and dad for being the best parents and grandparents in the world and taking care of my precious little girl days and nights and my little Raha for her beautiful smile that encouraged me all through the way.

Table of Contents

List of Tables

List of Figures

List of Abbreviation

ACL	Access Control List
ACK	Acknowledgement Notice
API	Application Programming Interface
CPU	Central Processing Unit
DNS	Domain Name Service
DOS	Denial of Service
DDoS	Distributed Denial of Service
FN	False Negative
FP	False Positive
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
IP	Internet Protocol
ISP	Internet Service Provider
NETAD	Network Traffic Anomaly Detector
OS	Operating System
OVS	Open vSwitch
QOS	Quality of Service
RAM	Random-access Memory
SDN	Software Defined Networks
sFlow	sampled flow
SOM	Self-Organizing Maps
SSL	Secure Socket Layer
SYN	Synchronization Message
TCP	Transport Control Protocol
TLS	Transport Layer Security
TTL	Time to Live
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
VM	Virtual Machine

Chapter 1

1 Introduction

1.1 Problem Statement

Software Defined Networking (SDN) is a new networking approach that is introduced with the goal to simplify the network management by separating the data and control planes. SDN has brought with itself programmability in the network control plane. The shift of the control logic from networking devices, such as switches and routers, in traditional networks to a centralized unit known as the controller permits the physical network hardware to be detached from the control Plane. This separation simplifies the design of new protocols and implementation of new network services such as access control, QOS, enforcement of new policies, bandwidth management, traffic engineering and etc. No longer does every small change need to come at the cost of reconfiguring all the network devices [1].

The data plane consists of devices that contain tables of flow entries. These devices use a secure transport layer protocol to securely communicate with a controller about new entries that are not currently in the flow table. Each flow entry includes matching rules and actions that guide the data plane on the action to take for the matched flow. When a packet arrives at a switch the header fields will be matched against the matching rules available in the flow table and if a match is found the action specified by the flow entry will be taken by the switch otherwise the packet header will be forwarded to the controller for further process. The controller then processes the header and constructs a flow rule to be installed in the flow tables of the switches along the chosen path.

The centralized structure of the controller could lead to many security challenges. One of such critical challenges is the impact of distributed denial of service attacks (DDoS) on SDN networks.

In a DDoS attack multiple compromised systems are usually infected with a Trojan and are used to target a single or multiple victims in the network. The attack traffic flooding the victim uses many different spoofed source IP addresses. This effectively makes it impossible to stop the attack by only blocking traffic based onsource IP addresses. It is also very difficult to distinguish legitimate user traffic from attack traffic when the attack traffic sources are spread across the Internet. Over the years, effective deployment of DDoS detection and response methods has always been critical factors for the proper network operation. This issue is even more pronounced in SDN networks.

The centralized role of the controller in SDN makes it a perfect target for the attackers. Such attacks can easily bring down the entire network by bringing down the controller. Since the attack packets are sent with many spoofed source IPs, the DDoS attack can cause problems to both switches and the controller [2]. At the arrival of each of the attack packets a new flow rule needs to be created and therefore the switch will need to store the packets in its memory and forward the header field to the controller. Receiving a large number of attack packets will use up a lot of switch memory and eventually install many new flow entries in the flow tables.This may cause the depletion of memory and slow down the flow table lookup very quickly, and in the worst case scenario, it may bring down the switch. Meanwhile all these packets from all over the network will be forwarded to the controller for processing. The large volume of packets sent to the controller with each of them consuming part of the memory and the processing power of the controller may also eventually break down the controller.

The effectiveness of DDoS attacks will be seen in a much faster pace and with a greater damage in SDN networks compared to the traditional networks. Hence it is vital to make sure an effective and trustable detection method is in place to detect such attacks and appropriate action is taken followed by an early detection.

Since SDN networks are used mainly in large data centers with many switches, it is critical to also find the targeted parts of the network through the detection process. This will reduce the time required to carry out a mitigation measure. Controllers are usually designed with backups and also are very powerful devices with huge amounts of memory but the resources in the switches are much more limited. This makes the switches to be more susceptible against these types of attacks and hence it is very important to have quick provisional methods in place to prevent the switches from breaking down as soon as first signs of an attack are detected. It is also very important to design the detection method as light weight as possible to prevent putting any extra load on the controller.

1.2 Research Objective and Contribution

This thesis concentrates on DDoS attacks on SDN networks. A method is proposed to protect the SDN switches and controller from the destructive effects of DDoS attacks by adding a lightweight detection mechanism at the controller. The main contributions of this thesis are as follows:

- Designing a DDoS detection algorithm by adding a light weight code to the SDN controller. The algorithm is designed to be effective to detect both single-victim and multiple-victim DDoS attacks with very high accuracy. Different detection methods are used including the Entropy variation of destination IP address, Flow initiation rate and study of the Flow specifications.
- A mitigation method based on the shorting of the flow idle timer is implemented at the time of an attack to help the switches survive when under attack.
- The algorithm is successfully implemented using Mininet and the pox controller.
- The effectiveness of the method is analyzed through extensive testing scenarios.

1.3 Thesis Organization

The remainder of this thesis is organized in the following order:

- **Chapter 2** covers some background information on SDN and OpenFlow architecture. The SDN security features and concerns are discussed followed by a glance at the DDoS attack operation, features and SDN related issues. Lastly some previous work in SDN DDoS attack detection is presented.

- **Chapter 3** describes the proposed DDoS attack detection and mitigation algorithm. The Step by step implementationof the algorithm will also be described**.**

- **Chapter 4** presents the simulation results, along with their detailed analysis.

- **Chapter 5** concludes the thesis and presents some future work highlights.

Chapter 2

2 Background and Related Works

2.1 What is SDN?

Software defined networking (SDN) provides a different approach in network design and management. The nature of conventional networking is static and even small changes in networking conditions would exact a high cost of re-configuring large number of switches, routers and other network resources [3].

As shown in figure 1(a), the operation of a network node in a traditional network consists of the interaction between the data and control planes. It is the control plane's responsibility to calculate the paths across the network and push them to the data plane to enable data forwarding. Therefore after a flow policy has been established in most cases any policy changes would require altering the configurations on each of the devices. In large networks this might mean that the network operators would need to spend quite a time reconfiguring the devices routinely to adapt to the changing traffic demands and network conditions.

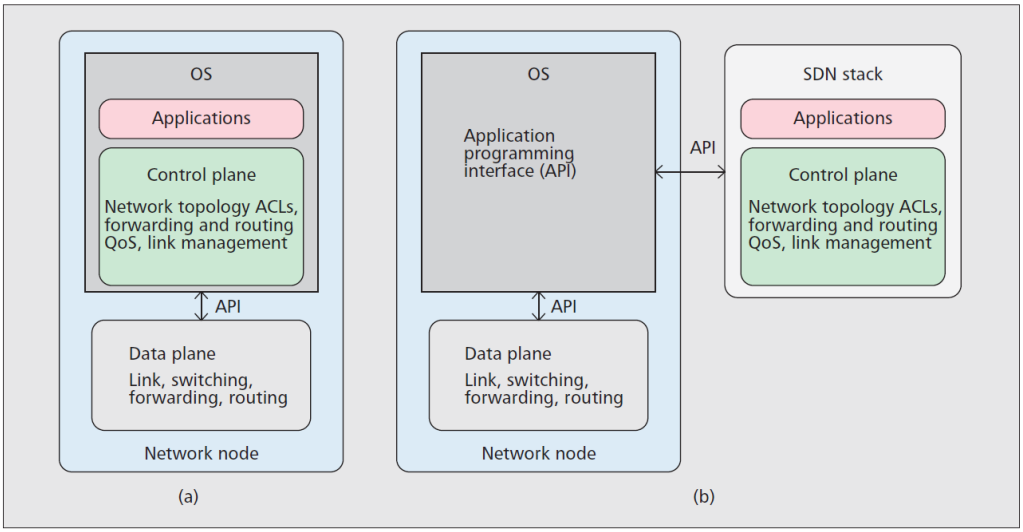

Figure 1Comparing Traditional Networks and SDN [4]

The main concept of SDN lies in the notion of decoupling the control and data planes that is shown in figure 1(b). In SDN, the control plane is no longer distributed among network nodes, as in the traditional network, but instead centralized at the controller which communicates with network nodes to setup the data plane through a southbound SDN protocol [5]. OpenFlow is a standardized southbound SDN protocol used to support communications between the controller and network nodes. Figure 2 illustrates the communication path between the application, control and data layers. Control plane applications such as load balancing at the application layer of the SDN architecture interact with the data plane through application programming interfaces (APIs). Both the application and control layers are implemented at the controller while the data layer is implemented distributed by the networking devices such as routers and switches. APIs are utilized to implement network services that are customized base on the application layer requirements (quality of service, access control, bandwidth management, energy management and etc.) [6]. More detail on different controller types is given in section 2.4.

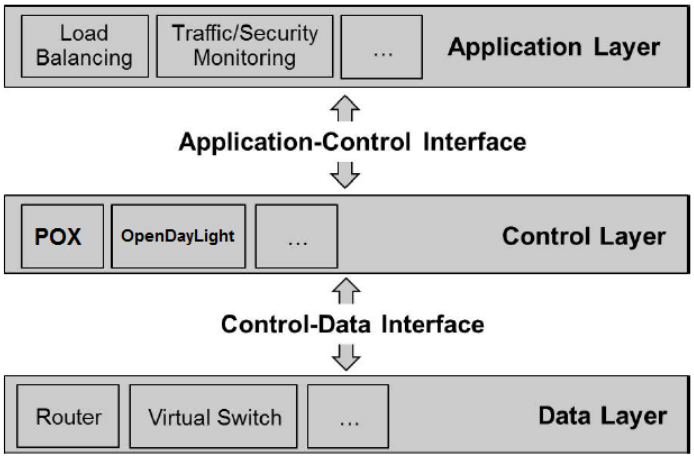

Figure 2 SDN Functional Architecture [7]

Since the controller gains a complete vision of the entire network, it can alone manage appropriate changes according to the traffic conditions. This approach significantly simplifies the implementation of some of the network functions.

As shown in figure 3, once the first packet arrives at the switch, the flow table is checked for a matching flow rule. If a match is found the flow actions specified by the flow rule will be executed and the flow statistics are updated otherwise the packet header is forwarded to the controller over a secure channel. The controller processes the packet according to the defined algorithm in the controller and specifies the actions that are to be taken by the switches along the chosen path between the source and destination. The new flow rule will be sent to the switches in the path to be installed in their flow tables. The switch will start taking appropriate actions on the data packets of the flow based on the new flow rule. If no matching flow entry is found in the controller the packets will be dropped.

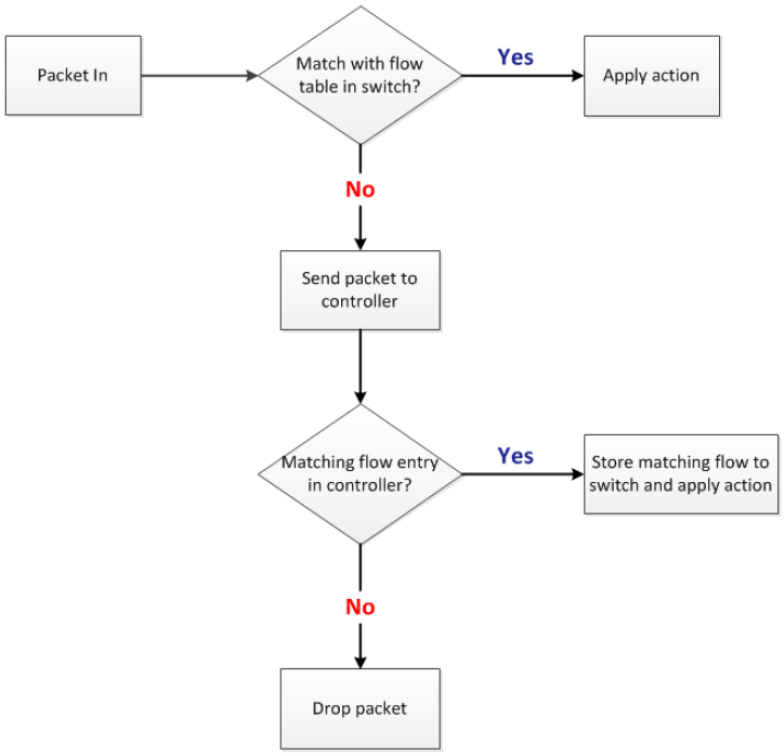

Figure 3 Flow Processing in OpenFlow Protocol [8]

2.2 Advantages and Disadvantages of SDN

As pointed out by sakir sezer et al. [9]the network programmability introduced by SDN provides continuous and smooth communication at all levels from hardware to software and eventually to the end users and builds awareness between the

network and the running applications. SDN helps to improve the use of resources by the applications and simplifies the introduction of new applications. Network changes are managed at a much lower cost and faster pace. Applications no more need to have a full view of the underlying network structure and the network complexity is hidden from the business applications [10].

In addition, having a global view of the network enhances the decision-making and consequently improves the network behavior efficiency. The introduction of new protocols and mechanisms in handling packets inside a network is more relaxed and data plane and control planes can develop impartially without having major influence on one another.

Apart from all the advantages that SDN has brought there are major concerns with the SDN implementation that need to be studied carefully before employing this newly emerging technology.

The centralized control in SDN provides a single point of failure in the network that if not well taken care of could be quite dangerous. Querying the controller to determine new flows for new connections will bring a control overhead and delay that in high traffic conditions could result in bottlenecks. Having a distributed SDN architecture and utilizing a number of controllers could help in mitigating the effects of network expansions. Using control hierarchy, implementing parallel controllers and classifying the flows according to their duration and priority are some of the approaches taken to prevent such conditions. Figure 4 illustrates the two network architectures that could be considered in the SDN networks. In the centralized architecture a single controller handles the entire controlling functionalities. To avoid the single point of failure issue in the network and provide redundancy a backup controller usually operates in parallel to the main controller. In the distributed architecture a number of controllers are running simultaneously, each controlling part of the network.

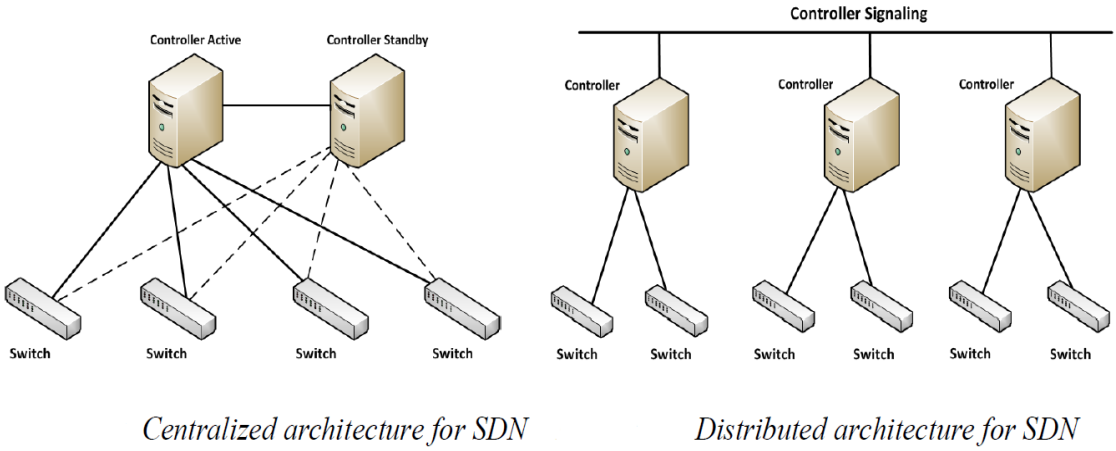

Figure 4 Comparing Centralized and Distributed SDN Architecture [6]

One other key area that requires more attention and is a crucial factor in any network is the security concerns. The concept of having a centralized controller would mean that all the traffic that is going through every node in the network could be monitored and controlled by the controller. Therefore the controller could be utilized to monitor, analyze and respond to any malicious activity in the network with minimum delay. Subsequently the required security policies can be implemented on the network by generating new flow actions or modifying the previously installed flows. On the other hand, having a centralized controller means a single point of failure/attack and therefore makes SDN very susceptible to Denial of Service attacks. Being able to control the entire network by a central programming point also means that unauthorized or malicious access to the controller could result in the loss of the entire network. Therefore it is crucial to ensure a highly sophisticated trust system resides among the data, control and application planes. A better analysis of the security concerns related to SDN networks is presented in the upcoming chapters.

Despite all the challenges that SDN is facing it has secured its place in the cloud computing and virtualization technologies and it is moving on much faster than expected. SDN has attracted the attention of many data center operators that deal

with large scale network structure and data exchanges. Google and NEC are among two of the well-known service providers that have already implemented SDN in their backbone networks [6].

2.3 OpenFlow Protocol

OpenFlow is the most well-known SDN protocol that has been chosen as the standard bearer in software defined networking. It was first created at Stanford University in 2008 [11]. The open Networking foundation is responsible for the update and development of this protocol [10]. The latest version of this protocol 1.4.0 was published in October 2013 but it is not widely implemented yet. OpenFlow version 1.3.0 is used as the baseline for the purpose of this research.

OpenFlow allows the switches to be controlled by an independent controller. The communication between the switch and controller is often over a Transport Layer Security (TLS) enabled channel. According to the specification each switch must contain one or more flow tables that will keep the flow entries specified by the controller.

A flow entry includes a header that identifies the individual flow that the packets are matched against and a set of actions that are to be taken by the switch for the matched packets. The matching is made on packet header fields. The actions taken can vary from packet forwarding, drop, further lookups in other flow tables, rewriting of the header fields and etc.

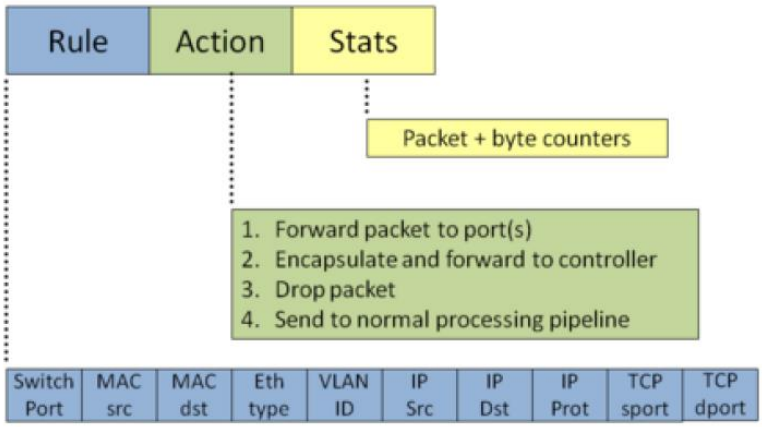

Figure 5 OpenFlow Table Entry [12]

Per table, per flow, per port and per queue counters are the four available statistic scopes kept in the flow table. The controller could query the switch to forward the statistics [12].

Scope	Values	Size (bits)
Per Table	Active Entries	32
	Packet Lookups	64
	Packet Matches	64
Per Flow	Received Packets	64
	Received Bytes	64
	Duration (seconds)	32
	Duration (nanoseconds)	32
Per Port	Received Packets	64
	Transmitted Packets	64
	Received Bytes	64
	Transmitted Bytes	64
	Receive Drops	64
	Transmit Drops	64
	Receive Errors	64
	Transmit Errors	64
	Receive Frame	64
	Alignment Errors	64
	Receive Overrun Errors	64
	Receive CRC Errors	64
	Collisions	64
Per Queue	Transmit Packets	64
	Transmit Bytes	64
	Transmit Overrun Errors	64

Figure 6Supported Counter Fields [2]

2.3.1 OpenFlow Operation

Jad Naous et al. [13] present a well described figure of the steps taken in routing a flow between two hosts across two switches in an SDN network. As we see in Figure 7 the switch flow tables are empty in the initial startup phase. When a new packet arrives in step 1, since no match is found in the switch flow table, it is forwarded to the controller (step 2). The controller observes the packet and decides on the action that should be taken (forward or drop) and creates a flow entry accordingly. This flow entry is sent to the switches in the path that the packet will traverse (step 3). The packet is then sent through to the receiving host in steps 4 and 5. In steps 6, 7, and 8 any new packets belonging to the same flow are routed directly since they would match the new entry in the flow tables.

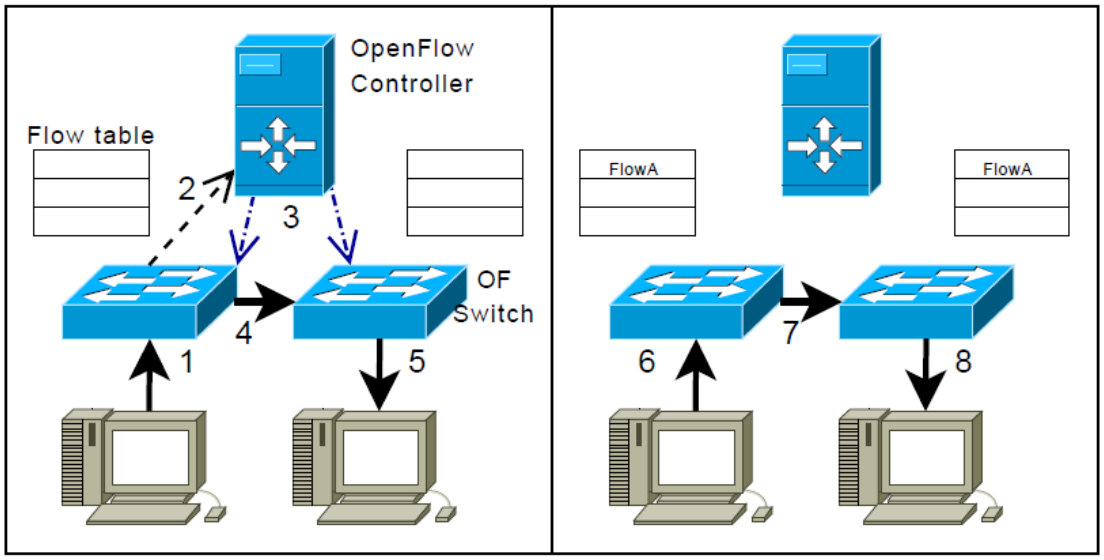

Figure 7 OpenFlow Flow Processing Procedure [13]

2.3.2 Flow Table Entry

The basic OpenFlow switch [8], the OpenFlow Type-0 switch, classifies packets into flows based on a 10- tuple. The 10-tuple entails of the following fields:

- Switch input port
- Source MAC address

- Destination MAC address
- Ethernet Type
- VLAN ID
- IP source address
- IP destination address
- IP protocol
- TCP/UDP source port
- TCP/UDP destination port

Figure 8 shows the header fields used as the match structure. If any of the fields are wild carded that field will be ignored while matching against the incoming packets.

Ingress Port	Ether source	Ether dst	Ether type	VLAN id	VLAN priority	IP src	IP dst	IP proto	IP ToS bits	src port	dst port

Figure 8 OpenFlow table entry header field [14]

Flow table entries are matched using a 10-tuple to find the corresponding actions associated with the flow. The OpenFlow Type-0 switch has three required actions:

- Forward to a specified set of output ports: This is used to move the packet across the network.

- Encapsulate and send to the controller: The packet is sent via the secure channel to the remote OpenFlow controller. This is typically used for the first packet of a flow to establish a path in the network.

- Drop: Can be used for security, to curb denial of service attacks, or to reduce spurious broadcast discovery traffic from end-hosts [13].

2.3.3 OpenFlow Message Types

OpenFlow protocol consists of a number of message types that are used for communication between the switch and the controller. These messages can be classified into three groups: controller to switch messages, Asynchronous messages and Symmetric messages.

2.3.3.1 Controller to switch messages

The controller to switch messages, such as Packet_out and modify messages, are inducted by the controller and are used to configure the switch, manage the flow table and gather statistics about flow entries in the switch flow table [10].

Packet_Out messages are sent from the controller to the data plane to inject the packets generated by the controller into the data plane. These packets are either a raw packet ready to be injected into the switch to install a new flow or indicate a local buffer on the switch containing a raw packet to modify an existing flow [8].

Flow_Mod messages are sent from the controller to the data plane to add, delete or modify the flow entries in the switch flow table. To identify which flow entries are to be modified the Flow_Mod messages contain a match structure. This match structure has the same format of the match structure used in matching the packets with the incoming packets in the switch [8].

Information about tables is requested with the OFPMP_TABLE multipart request type. The request does not contain any data in the body.

The body of the reply sent from the switch to the controller consists of following information [8]:

```
/* Body of reply to OFPMP_TABLE request. */
struct ofp_table_stats {
    uint8_t table_id; /* Identifier of table.  Lower numbered tables    are consulted first. */
```

```
uint8_t pad[3]; /*Align to 32-bits. */
    uint32_t active_count; /* Number of active entries. */

    uint64_t lookup_count; /*Number of packets looked up in the table. */
    uint64_t matched_count;  /* Number of packets that hit table. */
};
OFP_ASSERT(sizeof(struct ofp_table_stats) == 24);
```

A flow is defined by a group of packets that share the same matching structure at a certain time interval. As we see in the above reply sent from the switch to the controller flow statistics can be considered based on three characteristics:

1- *Received Packets:* The number of packets that match the flow entry matching structure.
2- *Received Bytes:* The amount of Bytes received by the flow.
3- *Flow Duration:* the time interval a flow entry has been in the flow table since its initial installation.

These characteristics will be used in the initial detection stage of the proposed algorithm.

2.3.3.2 Asynchronous messages

Asynchronous messages are sent from the switch to the controller and inform the controller of an event that is a change in the switch or network state. A sample of the asynchronous messages is the Packet_In messages.

Packet_In messages are sent from the data plane to the controller whenever a flow entry could not be found for a received packet on the switch or if the flow entry action asks for the packet to be forwarded to the controller. Packet_In messages could contain the whole packet or only part of the received packet. After receiving the Packet_In message the controller must either act on the packet itself or install a flow entry on the switches that must act on the packet [15].

2.3.3.3 Symmetric messages

Symmetric messages are used to assist in diagnosing problems in the data plane and controller connection. Hello and echo messages are among this group of messages.

2.3.4 Flow Duration

idle_timeout and hard_timeout fields are used to control the duration a flow entry kept in the flow table. The fields idle_timeout and hard_timeout are set when a flow entry is being pushed to the flow table. When a flow entry is modified, the idle_timeout and hard_timeout fields do not change.

If the idle_timeout is set, the flow will expire if no traffic is received during the set time. If the hard_timeout is set, the flow entry will expire regardless of whether or not packets are being received during the set time. If both idle_timeout and hard_timeout are set, the flow entries will timeout if any of the two timers expires. If both idle_timeout and hard_timeout are not set, the flow entry will be a permanent entry and can only be removed with a deleting flow_mod message [8].

2.4 SDN Controllers

Controllers are the focal point in the SDN networks also known as the brain of the network. The southbound application programming interfaces (APIs) are used to communicate with the OpenFlow switches [16]and the northbound APIs are used to communicate with the SDN applications.

Different jobs are performed by the controller using a variety of modules. Some of these tasks include identifying the devices within the network and the capabilities of each, gathering network statistics, etc. add-ons can be installed on the controller that will improve and widen the controller functionalities such as network monitoring and traffic anomaly detections.

Currently a number of well-known controller implementations are available that are open source and are written with different programming languages such as python, C++ and Java. Table 1, summarizes the current implementations of different available controllers. The table provides a brief overview of the controller characteristics.

Table 1 Current Controller Implementations complient with the OpenFlow Standard [17]

Controller	Implementation	Open Source	Developer	Overview
POX	Python	Yes	Nicira	General, open-source SDN controller written in Python.
NOX	Python/C++	Yes	Nicira	The first OpenFlow controller written in Python and C++.
MUL	C	Yes	Kulcloud	OpenFlow controller that has a C-based multi-threaded infrastructure at its core. It supports a multi-level north-bound interface (see Section III-E) for application development.
Maestro	Java	Yes	Rice University	A network operating system based on Java; it provides interfaces for implementing modular network control applications and for them to access and modify network state.
Trema	Ruby/C	Yes	NEC	A framework for developing OpenFlow controllers written in Ruby and C.
Beacon	Java	Yes	Stanford	A cross-platform, modular, Java-based OpenFlow controller that supports event-based and threaded operations.
Jaxon	Java	Yes	Independent Developers	a Java-based OpenFlow controller based on NOX.
Helios	C	No	NEC	An extensible C-based OpenFlow controller that provides a programmatic shell for performing integrated experiments.
Floodlight	Java	Yes	BigSwitch	A Java-based OpenFlow controller (supports v1.3), based on the Beacon implementation, that works with physical- and virtual- OpenFlow switches.
SNAC	C++	No	Nicira	An OpenFlow controller based on NOX-0.4, which uses a web-based, user-friendly policy manager to manage the network, configure devices, and monitor events.
Ryu	Python	Yes	NTT, OSRG group	An SDN operating system that aims to provide logically centralized control and APIs to create new network management and control applications. Ryu fully supports OpenFlow v1.0, v1.2, v1.3, and the Nicira Extensions.
NodeFlow	JavaScript	Yes	Independent Developers	An OpenFlow controller written in JavaScript for Node.JS [65].
ovs-controller	C	Yes	Independent Developers	A simple OpenFlow controller reference implementation with Open vSwitch for managing any number of remote switches through the OpenFlow protocol; as a result the switches function as L2 MAC-learning switches or hubs.
Flowvisor	C	Yes	Stanford/Nicira	Special purpose controller implementation.
RouteFlow	C++	Yes	CPqD	Special purpose controller implementation.

POX controller is one of the most popular in the research and academic field. POX is a lightweight OpenFlow controller that is a suitable platform for SDN research, academic work, education, and experimentation. Having a straightforward and lightweight design a great amount of tasks have been performed to prototype SDN projects or do academic research in POX [18]. The proposed algorithm in this research project is implemented over POX but this does not mean that the proposed algorithm is dependent to a particular type of controller In order to simplify the setup, as well as to improve repeatability; a virtualized network setup based on Mininet will be used [2].

2.5 SDN Security

The main characteristics of securing a network would be to ensure the confidentiality and integrity of the transmitted data and availability of network resources even when the network is under attacked.

The new networking architecture introduced by SDN has raised many debates on whether having a centralized controller would be a security threat or could help in achieving the security goals. An example of the advantages that SDN can simplify the security provisioning within the network is the easier use of random virtual IP addresses within the SDN. The real IP of the hosts can easily be hidden from the external world by the OpenFlow controller which will make it much harder for the attackers to find vulnerable ends within the network [7].

Since all network traffic is guided through the controller, at least for the flow initiation, the controller is the best place to monitor the flows to detect any abnormal events [7]. Furthermore, with the feature of programmability the mitigation process after identifying the attack will be deployed much quicker and more efficiently. The controller can easily install new flow rules or delete/modify the previous flows in order to reduce the effect of the attack. The other positive feature of SDN networks is that most of the proposed attack detection and mitigation methods are open source, light weight and usually don't require the installation of separate devices. As mentioned in the overview of the SDN networks by sakir Sezer et al. [4] SDN can support:

- Network Forensic: enables quick and predefined threat identification and management mechanisms.
- Security Policy Alteration: diminishing the rate of misconfiguration and conflicting policies within the network through the easy installation of new policies throughout the whole infrastructure with minimum delay.

- Security Service insertion: a required service insertion (e.g. firewalls and intrusion detection system) within the network is simplified due to the SDN structure.

On the downside to the aforementioned facts, Controllers are an appealing target for the attackers. If an attacker is able to use the network vulnerabilities to gain unauthorized access to the network resources and cover-up as a controller it can control the entire network. Introduction of open interfaces and protocols within SDN also opens the doors to malicious attackers to plan and execute their attacks with a better overview and knowledge of the network structure and actions.

One method to prevent the breakdown of the controller within a network is the use of multiple controllers in the network. In such cases, authorization and access controls become much more complex.

SDN is a growing platform especially for data centers and cloud computing centers [3]. In such setup multiple organizations will be accessing the network resources simultaneously and they each demand for their data protection. Providing a comprehensive security prototype to support the appropriate level of network privileges to each application comes with its own challenges. The controller must be able to handle conflicting flow rules received from different applications while it must isolate the applications from one another.

DoS and DDoS attacks are another major security concern. As explained previously in section 2.3.1once a switch receives a new packet that does not match any of the flow table entries, it will either forward the whole packet or a partial segment of the packet, the header field, to the controller so a new flow rule could be created and installed on the switch. By generating high level of traffic with spoofed source IPs the attacking traffic could consume most of the bandwidth if the complete packet is forwarded to the controller. If only the packet header is forwarded to the controller

the packet must be buffered in the switch. The attacker could easily target the switch by over loading the switch memory.

The aim of this project is to propose a high efficiency, lightweight DDoS detection mechanism with minimum delay that could be easily integrated into the SDN controller. In addition, a mitigation approach is proposed that could help the SDN network administrators to better handle the attack.

2.6 DDoS Attacks

Denial of Service attacks or DoS is one of the major issues in today's network security. The devastating effects of such attacks are well documented in many cases. It is also well understood that the structure of SDN is vulnerable to such attacks.

The goal of DoS attacks is to block network services by limiting the access to the nodes that provide these services in the network. This could be achieved by using up the entire network bandwidth or by consuming the resources available on the service provider nodes such as memory and CPU. The attacker simply sends high volumes of packets to occupy the entire channel bandwidth or breaks down the service by taking the entire processing capacity available on the service provider nodes. As an example this attacking traffic could be generated by sending high volume of UDP packets that take away the entire bandwidth so that the legitimate traffic cannot reach its destination [12]. Distributed Denial of Service (DDoS) attacks add a many-to-one characteristic to DOS attacks. Consequently, higher level of attack prevention, detection and mitigation complexity are required.

DDoS attacks are conducted by sending many packet streams from sources that are hijacked by the attacker. An experienced attacker could vary the packet fields and traffic characteristics to avoid detections which are based only on traffic classification policies [12].

2.6.1 DDoS Attack Operation

A DDoS attack consists of four elements [19]:

1. The main attacker that is behind all the attack planning and intelligence.
2. The handlers or masters are compromised hosts that have special programs running on them that control multiple agents.
3. The compromised hosts that run the attacking program and generate the packet streams destined for the targeted victims. These agents are also known as zombie hosts as the owner of the agent system is usually unaware of the malicious program that is running on his/her computer.
4. The targeted destination

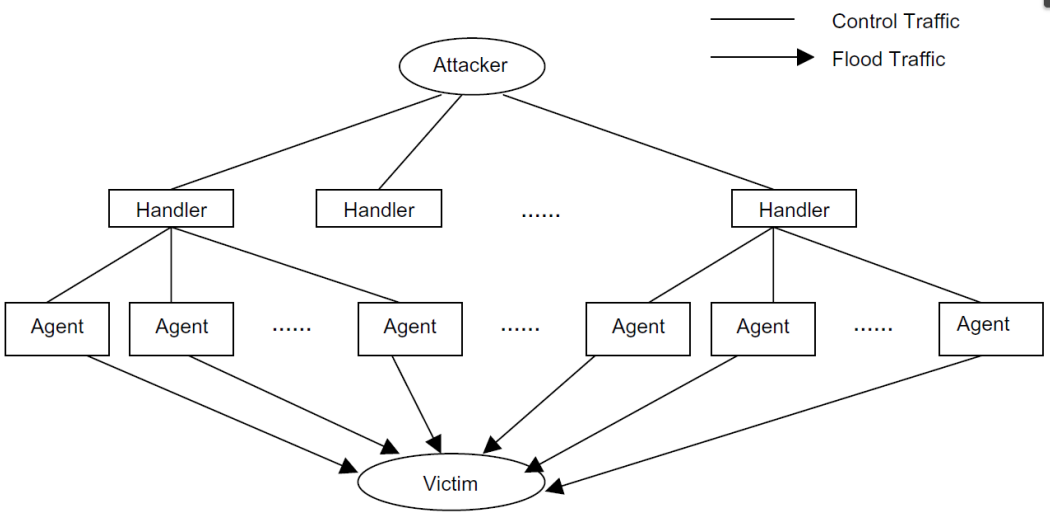

Figure 9 DDos attack structure [19]

The attacker finds the machines that have security vulnerabilities and that can be compromised to gain access to them through the internet. The attacker must make sure that the chosen hosts have the required system requirements to run the attacking program. This task is now done by available programs that make the job easy for the attacker. By compromising the found vulnerabilities the attacker can install its code on the host. These codes are designed in a format that consumes a share of their host system resources so that no major issues can be detected on the system performance by their owners. On the other hand the codes are designed to hide in the best way to avoid being detected by the applications that are running on the host system. The attacker needs to communicate with the handlers to find out

which hosts can take part in the attacking process or use them to update the codes on the hosts. Each host is controlled by one or a number of handlers and this communication can run over UDP, TCP or ICMP protocols [19].

The attacker schedules the attack, the destination victim address, characteristics of the attack traffic sent such as the port, duration, TTL, traffic type and etc. All these features could vary during the attack phase to help avoid detection.

2.6.2 Various DDoS Attack Types

Three example DDoS attacks that have had the highest reported attack incidents are the UDP flood attack, ICMP flood attack and TCP flood attack. These attacks will be briefly explained below.

In UDP flood attack a large volume of UDP packets are sent to random or specified port forcing the system to look for the application attached to this port. Since no waiting application is usually found an ICMP destination unreachable message is sent back to the spoofed source address. The processing of the attack UDP packets and generation of ICMP responses may cause the targeted host to run out of resources and crash [20].

In ICMP flood attack the zombie hosts, send a large number of ICMP_ECHO_REQUEST packets also known as ping packets to the target address. The target shall reply back to all the requests simultaneously which causes it to crash [21].

TCP SYN flood attack takes advantage of the nature of TCP three way connection setup handshakes. Upon receiving an initial SYN the server replies back with a SYN/ACK and waits for the final ACK that is never replied back by the attacking host. Exhausting the network resources using heavy traffic loads is the mutual aspect of all the aforementioned attacks [22].

2.6.3 DDoS Attacks in SDN

In addition to the aforementioned attacks that target specific services and servers another type of DDoS attack is more hazardous to the SDN. These attacks are independent of the traffic type. Every new packet received with no matching flow is processed by the switches and the controller. After a path is installed for the flow the switches will forward the packets of the flow along the installed path. The inside servers might be targeted by the attackers using the characteristics of different traffic types but the switches and the controller take no notice of the type of traffic that is passed through the network. Therefore if the attacker targets the switches or the controller the detection methods applied that are based on identifying the traffic characteristics will not be effective.

When a DDoS attack targets a destination within the SDN network the huge number of different spoofed source addresses will result in initiation of many flow entries by the controller. Each packet header must be sent to the controller while the packet waits in the switch till a new flow entry is installed in the flow table. Both the size of the packet queue and flow table is restricted within the switch. Receiving a large number of entry packets will cause the overflow in both the queue and flow table very quickly and breakdown the switch. Meanwhile all these packets from all over the network will be forwarded to the controller for processing. No matter how powerful the controller, it will eventually run out of resources and cannot handle any Packet_In requests. The crash of the controller will mean the crash of the entire SDN network. This single point of failure in SDN networks is the vulnerability that the DDoS attacks can use to cause the utmost damages. This thesis concentrates on the switch and controller attacks although the server specific attacks will also be detected effectively.

2.7 Previous Work in SDN DDoS Attack Detection

It is very interesting to know that implementing SDN architecture is proposed by Seungwon shin et al. [23]as a method for the intrusion detection in cloud

environment. In the proposed scheme OpenFlow is integrated into the network structure to control the network flows and diverts the traffic through a path that it is inspected by the preinstalled security devices (e.g. network intrusion detection system (NIDS), firewall, etc.). Employing the SDN infrastructure will simplify the network operator's job in a huge cloud infrastructure. The changes in the flow directions and network policies can easily be performed by running simple scripts on the controller that will install new flow entries on the switches. The controller itself is not involved in the abnormal activity detection but it is responsible for calculating the best and shortest paths that will guide the traffic through the NIDS.

In a similar approach Snort, an Intrusion Detection Systems (IDS), is used to monitor network traffic and measures to identify mischievous activities in the network. Intrusion prevention System (IPS) is an IDS that has the power to spontaneously take action towards the suspect events upon attack detection. Tianyi Xing et al. [24] have implemented an IPS called snortflow by integrating Snort and OpenFlow modules. In this approach the cloud networking environment is dynamically reconfigured utilizing the power of OpenFlow switches in real time to dynamically detect and prevent the attacks.

Kreutz et al. [25] reveal the need of building protected and trustworthy SDNs in the design phase. Bringing replication, diversity and dynamic switch association to SDN control platform design are the main arguments described as mitigation methods for several threat vectors that enable the exploit of SDN vulnerabilities. In the proposed example by implementing a number of replicated controllers the backup controller will take over if one controller malfunctions. The controllers must be designed with interoperation capabilities. Meanwhile the switches must have the ability to dynamically associate to the controllers. To prevent simultaneous attack on all controllers, controllers' diversity must be considered to improve the robustness of the system. FRESCO [25] is an extension of this work that makes it easy to create and deploy SDN security services.

FRESCO [23] is a framework proposed for easier design of secure SDN networks. FRESCO presents an OpenFlow security application development framework that assists in prototyping new compassable security services in OpenFlow networks. FRESCO offers a library of reusable security modules that can detect and mitigate different attacks. The scripting API offered by FRESCO enables the rapid design and development of these modular libraries. Essential security functions (e.g. firewalls, IDS, attack deflector, etc.) can be simulated by assigning values to the interfaces and connecting the necessary modules. The modules can produce flow rules used to enforce the security directives.

Braga et al. [26] propose a DDoS detection method built into the NOX controller based on Self-Organizing Maps (SOM). SOM is an unsupervised artificial neural network trained with the features of the network flow that is periodically collected from the switches. The traffic is classified as either normal or abnormal based on the SOM pattern. This detection method as shown in figure 10 runs in three modules running periodically within a loop in the NOX controller:

- The flow collector module queries the switches periodically for their flow tables.
- The feature extractor module extracts the main features that are studied for DDoS attack detection and gathers them in 6-tuples. The main elements that are calculated based on the collected features and will be studied in the next module for the traffic classification include average of packets per flow, average of bytes per flow, average of duration per flow, percentage of pair flows, growth of single-flows and growth of different ports.
- The classifier module must analyze and decide whether the given 6-tuple corresponds to a DDoS attack.

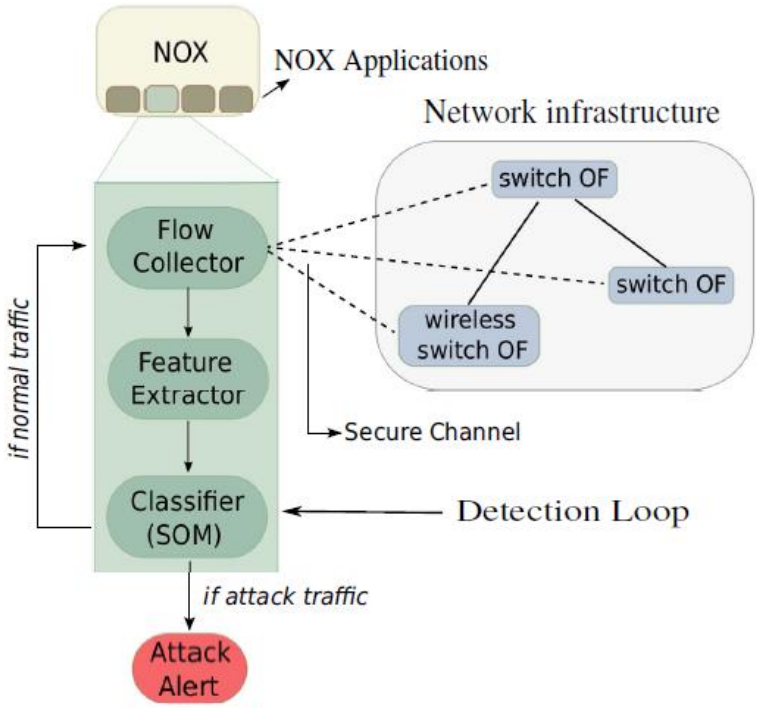

Figure 10Detection Loop Operation [26]

Querying the switches periodically especially in the large scale cloud architecture with large number of switches will put an extreme overhead on the system and will eventually affect the performance of the controller. Processing that high volume of flows in the flow tables is another issue that must also be well-thought-out.

Tamihiro Yuzawa [27] implements new generation database that does the heavy lifting of sFlow data processing for DDoS attack detection. sFlow or "sampled flow", is an industry standard for packet export at Layer 2 of the OSI model. To keep the legitimate traffic running and provide source-and-destination-based filtering OpenFlow or more specifically Floodlight's static flow pusher API is executed. Static Flow Pusher is a Floodlight module that allows a user to manually insert flows into an OpenFlow network. This is known as the proactive approach to flow insertion. To do DDoS mitigation in this way requires lots of preparation, and a strong understanding of the network flows.

YuHunag et al. [28] from Chungwa Telecom Co. proposes an OpenFlow DDoS Defender that monitors flows on an open flow switch. If the number of packets received in 5 seconds exceeds 3000 then the number of packets will be studied in per second duration. If the number packets per second exceed 800 for 5 continuous times then an attack is detected and the DDoS defender will start dropping the incoming packets until the flow entry times out.

Syed Akbar Mehdi et al. [29] argue that network security tasks should be delegated to the home and office networks instead of ISPs. In the presented work security policy implementation is delegated to the downstream networks. Four prominent traffic anomaly detection algorithms, threshold random walk with credit based rate limiting, rate-limiting, maximum entropy detector and Network Traffic Anomaly Detector(NETAD) are implemented in NOX controller and it is observed that the anomaly detection can function well at line rates without any performance degradation in the home network traffic. It is suggested that this approach can monitor the network activities without the need of the excessive sampling.

C.Dillon and M.Berkelaar [12] monitor the flow statistics sent from the open flow switch to the controller to find the large spikes in traffic that could be signs of an attack. The OpenFlow controller then finds the sources of the attack traffic and as a mitigation method flows are installed on the switches to drop the traffic from the suspected sources. The proposed detection techniques include using packet symmetry and temporary blocking of the traffic. In routine traffic state a symmetrical behavior exists between the two sides of a communication. In the learning phase the symmetry ratio is analyzed in the network and sources with high asymmetric ratio are suspected of an attack. In temporary blocking the flows are blocked for a short period and the traffic behavior to this blocking is used to analyze if the traffic is legitimate or not. The three phases of this strategy include: sampling, blocking and analysis.

Entropy variation of destination IP address is used as an early detection method in the pox controller in a work done by Seyed Mohammad Mousavi [30]. Entropy is known as a measure of randomness. The maximum entropy happens when each incoming packet is destined for a different host and the minimum entropy is seen when all the packets are destined to the same destination address. As explained in the previous chapters a characteristic of DDoS attack is sending high volume of packets to the same destination. In the proposed method the destination IP is used for entropy computation. The algorithm flow chart is shown in figure 11. A window of packets is studied and the entropy is calculated for their destination IP addresses. If the calculated entropy is less than the threshold for a continuous number of times, an attack will be reported. There are a number of limitations to this method. When the number of hosts under attack within the network rise or when the entire network is under attack the entropy detection will fail. On the other hand when the load of the traffic increases in the network with legitimate traffic in the peak times using the proposed entropy detection mechanism alone will result in false positive attack detections. This is because the provided algorithm does not adapt to the traffic load changes dynamically. The work done in this project is an improvement to the previous work done in [32].

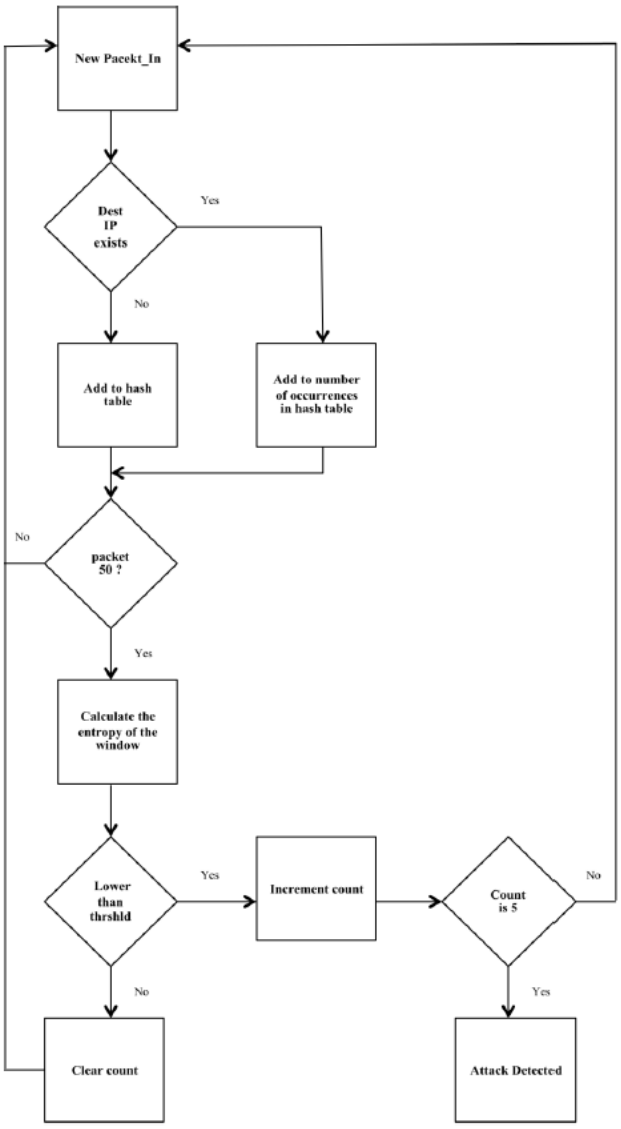

Figure 11Entropy Variation of Destination IP address DDOS Detection Flowchart [30]

Chapter 3

3 Proposed DDoS Detection and Mitigation Algorithm

This detection algorithm is designed based on three main concepts including Entropy variation of destination IP address, Flow Initiation Rate and study of Flow Specification. The proposed detection algorithm can be broken into seven phases.

1- Data Collection.
2- Entropy calculation and comparison.
3- Flow initiation rate computation and comparison.
4- Finding the possible attack path(s) if an attack is suspected in steps 2 or 3, otherwise returning to step 1.
5- Polling the switches in the attack path for flow statistics and studying the returned results from the switches to confirm or cancel an attack state.
6- Updating the thresholds according to the detection result in step 5.

After an attack is detected a mitigation method will be applied to prevent the network switches from breaking down and to give enough time to network administrators to take the required actions. The mitigation approach set the flow idle_timer to a small value for the new flows.

All the above steps are performed by adding a set of lightweight codes to the controller. The flowchart in figure 12 illustrates the proposed detection and mitigation method.

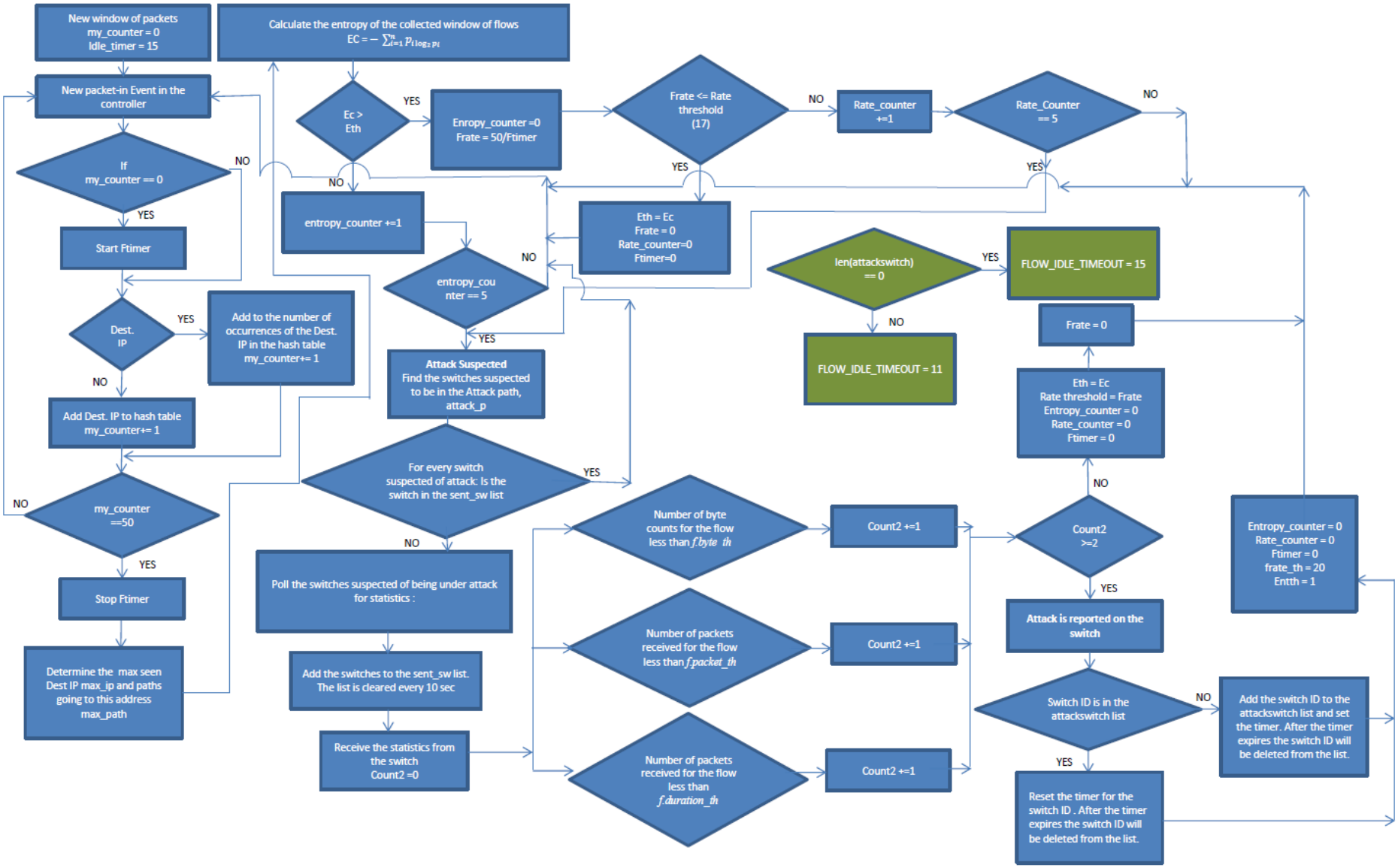

Figure 12 Algorithm Flowchart

3.1 Entropy Variation of Destination IP address

Entropy or Shannon-Wiener index [31] is an important concept in information theory. Entropy is a measure of uncertainty or randomness associated with a random variable which in this case is the destination address. A higher randomness will result in higher entropy. The entropy value lies in the range of $[0,\log_2 m]$ where m is the number of destination IP addresses. The entropy value is at its minimum when all the traffic is heading to the same destination. On the other hand the entropy value is at its maximum when the traffic is equally distributed to all the possible destinations.

The entropy-based detection algorithm used here is similar to that in [32]. To collect packets for entropy analysis, we use a fixed size window. Using a fixed size window simplifies the entropy computation complexity. The window size can be fixed in terms of Elapsed time or Number of received packets. Since during light traffic load period, at a fixed-time window might degrade the accuracy of the entropy calculations. Instead of using a fixed-time window, the algorithm uses a window that is measured by *n* number of packets, where *n* is the window size. For each window, the packets will be classified into groups based on their destination IP addresses. All the packets in each group will have the same destination address but they may have distinct source addresses. The destination IP address is used as the characteristic metric and the frequency of each distinct destination IP address in the window is adopted as a measure of randomness. Let *m* be the total number of destination IP addresses associated with these *n* packets. The relative frequency of destination IP address IP_i is calculated in equation 3.1:

$$F_i = \frac{n_i}{n} \qquad (3.1)$$

where n_i is the number of packets with destination IP address IP_i.

The entropy formula is calculated according to equation 3.2:

$$H = -\sum_{i=1}^{m} F_i \log_2 F_i \qquad (3.2)$$

Since $0 \leq F_i \leq 1 \Rightarrow \mathrm{H} \geq 0$

The entropy value is maximum when the relative frequencies associated with the m IP destination addresses are equal (F_i = $1/m$ for all i). For instance if we have fifty packets and each packet is destined to a distinct destination the calculated probability for each destination address will be $F_i = \frac{1}{50}$ and H = $-\sum_1^{50} \frac{1}{50} * \log_2 \frac{1}{50}$ = 5.643. If, instead, 10 of the 50 packets are delivered to one of the destination addresses then the entropy value is reduced s to 5.213.

In the normal network state we expect that the traffic spreads out to many different hosts. During a DDoS attack the number of packets destined for a specific host or a small set of hosts rises suddenly and the entropy decreases. A decrease in the entropy is an alarm for the network to watch out for a possible attack.

It is vital in SDN networks to have a fast detection method and to detect the attacks at its early stages. SDN networks are more vulnerable against the DDoS attacks than the traditional networks. If the detection time takes too long the attacker could break the switches or the controller and so an early detection is extremely important. For an early detection the window should not be too large. On the other hand a small window will add to the computational overhead. As proposed by S. Oshima et al. [32] in this thesis we will use the window size of fifty to balance the two concerns.

A module is added to the pox controller for the entropy calculations. For every fifty packets that arrive in the controller the relative frequencies are calculated. The calculated entropy is compared against the threshold value. If the calculated entropy is less than the threshold for five consecutive entropy calculations an attack is suspected and further analysis will be performed to determine if the attack is real.

3.1.2 Implementation of Entropy Variation of Destination IP address

The steps related to the application of Entropy Variation of Destination IP address are highlighted in the below algorithm flowchart.

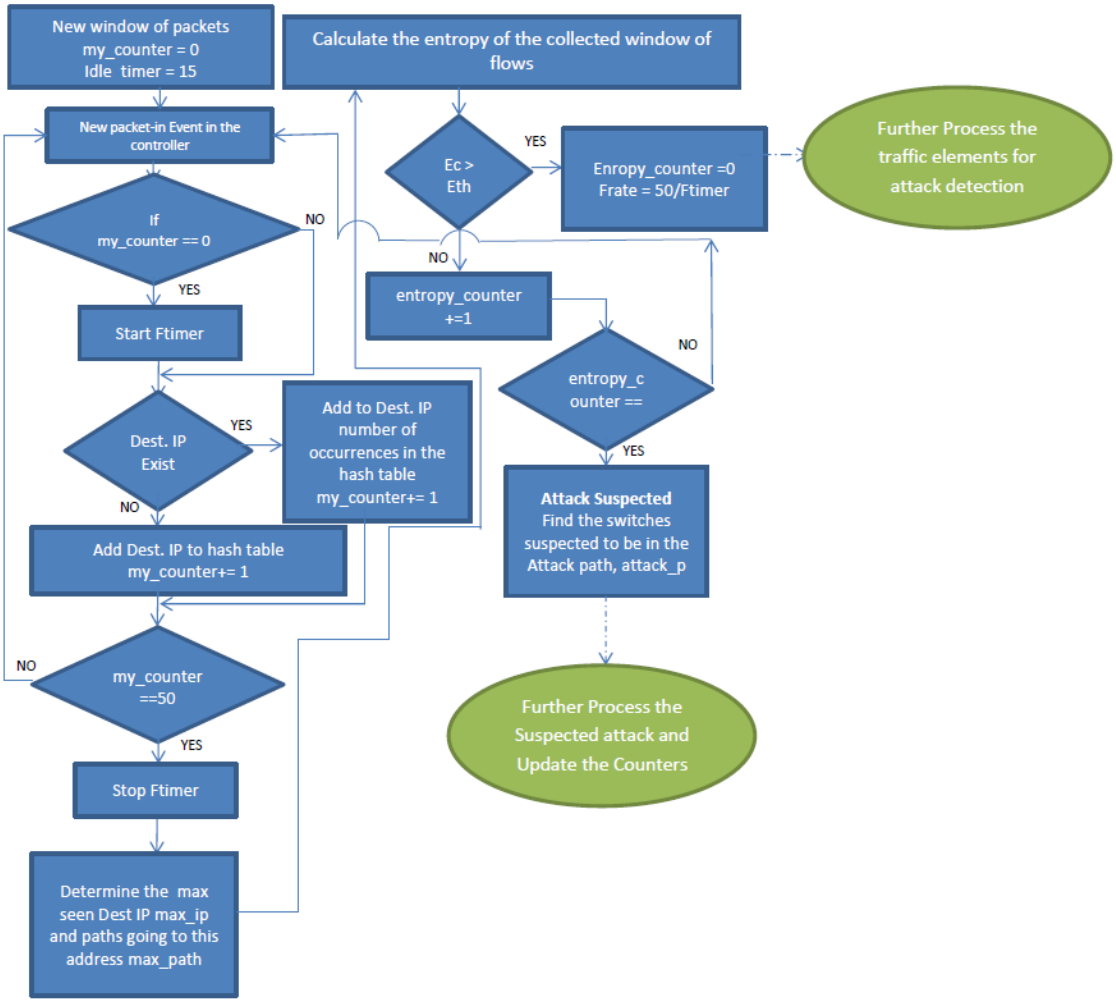

Figure 13 Implementation of Entropy Variation of Destination IP address in the proposed Algorithm

In step 1 a window of fifty packets is collected on the controller. These packets represent a window of fifty flow initiation requests that was sent to the controller by the switches. A timer (ftimer) calculates the duration it takes for this window of fifty packets to be collected. This timer will be used in the next phase of detection based on the flow initiation rate.

As we know the controller computes the shortest path for each flow and installs the flows on the path. By default the controller does not keep the calculated paths. However, in order to easily find the attack path in later stages of the detection algorithm a module is added to record the calculated paths (*path_for_stat*). After reaching a window of fifty collected packets the destination IP address for each

packet is examined to determine how many times each IP address is repeated within the collected window. The destination IP address that is repeated the most number of times is confirmed as the *max_ip* and the paths that are used to reach this IP address are stored for later processing (*max_path*).

The entropy function is then called to calculate the entropy. This function takes the destination IP addresses and the number of times they are repeated as input, and calculates the frequency of each destination IP address. The calculated frequencies are used to derive the current entropy (*Ec*) using eq. (3.2). When an attacker targets a host within the network the number of new flows destined for certain address will rise dramatically. Accordingly the entropy will start to decline. In the beginning of the algorithm a default entropy value is calculated under normal to low network traffic condition that is considered as the initial entropy threshold value (*Eth*). The calculated entropy (*Ec*) is compared to this threshold (*Eth*). If for five consecutive times the calculated entropy is lower than the threshold an attack is suspected and the switches in the possible attack path are identified and their flow tables and statistics are polled and analyzed by the controller.

If an attack is not detected after the study of the switches' flow tables and statistics the entropy threshold will be updated to the current calculated entropy to prevent further false positive detections. This approach will enable the detection algorithm to adjust itself dynamically with the current traffic flow pattern. Moreover, if an attack is confirmed, the entropy threshold will return to its default value to raise the level of sensitivity and awareness in the detection process. Since simple changes in the traffic pattern could change the entropy in short spikes of time, that is why the detection algorithm only suspect possible attack if the computed entropy is below the threshold in five consecutive windows.

Although entropy has proven to be a successful detection method, using entropy alone cannot detect many attack scenarios. For example at peak times with the sudden rise of legitimate traffic, the demand to a certain network destination such as the web server or email server grows and the entropy based detection method

may continuously report false positive alarms. On the other hand when the attacker distributes the attack among many victims the entropy may not show a significant decrease and so it will result in a false negative report. To overcome the mentioned limitations of the entropy detection the proposed detection method in this project also incorporates other detection algorithms.

3.2 Flow Initiation Rate

As mentioned in the previous chapter entropy is an approach used in other DDoS attack systems and has been an effective element in single-victim DDoS attack detections. But due to the limitations of the entropy method, especially in multiple victim attacks, it cannot be considered as a standalone, effective scheme for DDoS attack detection. It is because in the multiple victim attack, the attack traffic targets many different destinations resulting possibly insignificant changes of entropy. A more efficient detection algorithm in this case will be based on the flow initiation rate.

When a DDoS attack is underway the attacker will be sending a large volume of packets through its agents to the targeted destination(s). The results obtained in experiments Braga et al. [26] suggest that the flow initiation follows a linear growth during a DDoS attack.

Similar to the entropy computation, the algorithm computes the flow initiation rate every 50 packets , using equation 3.3 to compute the rate

$$FR = \frac{n}{T_W} \qquad (3.3)$$

Where FR is the flow rate, n is the window size and T_W is the duration of the window.

If the calculated flow initiation rate is above the threshold then an attack is suspected and further investigation is performed. If the calculated rate is below the threshold the system is considered to be in a safe state.

In the proposed algorithm the initial threshold is fixed at a level that we know is an acceptable traffic level for our network and that at such rate the network will be able to operate safely. This threshold can be calculated in an initial learning phase by studying the controller while legitimate traffic is running at an average to low rate. The threshold, however, must be adaptive to the network traffic load to reflect the current traffic flow pattern.

When the flow initiation rate starts to grow, the calculated rate will start to increase and at some point it will go over the threshold. If this behavior continues to occur for five consecutive times, it will be considered as a sign of attack. In such cases the network traffic statistics need to be further studied to confirm the attack. If a further study of the flow statistics does not indicate an attack then the threshold must be updated to the current calculated rate to avoid false positive attack reports in the system.

When the traffic load decreases and flow rate starts to drop the threshold must again be updated to the current value of the flow rate to avoid any false negative scenarios. In the case of an attack being confirmed the network will move in to an alert state and the threshold will drop to the initial value to increase the sensitivity of the detection algorithm.

3.2.2 Implementation of Flow Initiation Rate

The steps related to the application of Flow Initiation Rate are highlighted in the algorithm flowchart of figure 14.

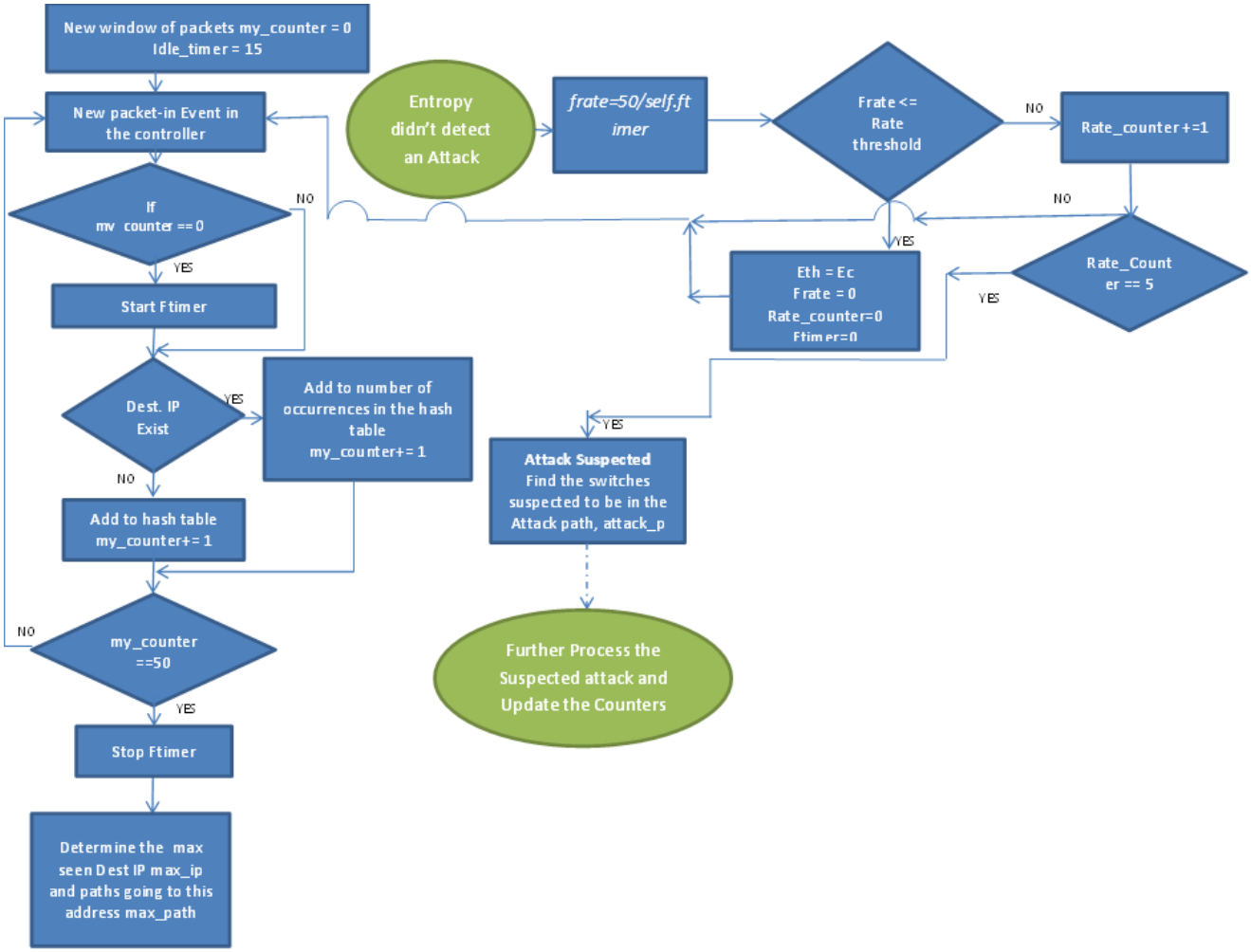

Figure 14 Implementation of Flow Initiation Rate in the proposed Algorithm

As mentioned in the previous section the flow initiation rate is calculated by dividing the window size by the window duration given by equation (3.3).

If the calculated flow initiation rate,*FR*, is higher than the threshold *frate_th*, for five continuous times we suspect that there may be an attack to the network and will look at the flow statistics for confirmation. Using five consecutive windows to identify a suspected attack will help in lowering probability of false positive detections

The *frate_th* value should be chosen to match the network flow initiation rate of the normal network traffic. The default threshold is set to a value such that the network will be able to easily handle the traffic at that rate.

The threshold value will be constantly updated according to the current traffic load.

When using flow initiation rate as a detection method we should always keep in mind that high volume of flow initiations may be caused by the sudden increase of legitimate traffic. Relying on only the flow initiation rate to report an attack may cause high probability of false positive. Therefore in the proposed algorithm the flow initiation rate monitoring is only used as a mean of detecting the sign of an attack but not confirming the attack. If an attack is suspected, the flow statistics from flow tables of the switches that are suspected to be in the attack path are analyzed to confirm the attack.

3.3 Study of Flow specification

In this part of the detection process, the controller will examine the flow statistics of the switches that are suspected in the paths of the attack traffic. Through the use of OpenFlow protocol, the controller is able to poll any of the switches for the flow table and the flow statistics. Three flow statistics are analyzed for the attack detection:

1- Received Packets per flow.
2- Received Bytes per flow.
3- Flow duration.

In what follows, we will study the characteristics of these three statistics associated with malicious flows.

In order to maximize the attack to the controller and the switches under a given attack traffic load, the attacker will try to create as many attack flows as possible. It is because the controller only needs to process the first packet of the flow. The data rate and duration of the flows do not affect the performance of the controller. In the case of the switch, the size of the flow table corresponds to the number of flows currently handled by the switch. Again, the data rate and duration has no effect on the processing resources of the switch. An example should clarify this point. Suppose the attacker can generate in total 100 Mbps of attack traffic and suppose

the minimum packet size is 100 bytes. The maximum flows (and the maximum damage) generated by the attacker per second are 125,000. In this case, each flow only consists of one packet with the smallest packet size, but the controller needs to handle 125,000 flow request per sec and a switch may have to setup 125,000 entries per sec in its flow table. This kind of attack can crash the controller or overflow the switch flow table. On the other end of the spectrum, if an attacker only generates 100 Mbps of attack traffic in one attack flow, the attack will have no effect on the controller and the flow table.

Based on the above argument, we can conclude that attack traffic usually has the characteristics of small packet size, small number of packets per flow and short flow duration. Also note that if the attack traffic is TCP-based, then the number of packets per flow is just one and the flow duration depends on the connection wait time, which is usually small. If the traffic is UDP based, the duration of the flow depends on the idle timer, which is also small, set at the switch.

Thus, a large volume of flows with small number of packets (short flows) and small payloads may imply a DDoS attack. We should note that legitimate external cloud traffic may also consist of many flows with short duration. What distinguishes legitimate and malicious traffic flows is that legitimate traffic flow usually involves a higher number of packets and/or larger payload.

There are a number of previous studies on the DDoS attack detection that rely on the study of the statistics polled from the network switches but since polling the switches and processing their fairly large flow tables consume a lot of bandwidth and resources, consequently detection algorithms that only rely on the analysis of the flow table statistics are not recommended as they could result in the breakdown of either, the controller, switches or both. In the proposed algorithm the study of the flow tables is only used as a mechanism for final assurance of an attack after an attack is suspected based on the analyses of entropy and flow initiation rate. Furthermore, the proposed algorithm does not examine the flow table statistics

from all the switches. Instead, it only examines the flow table statistics of some of the switches that are presumably under attack.

3.3.2 Implementation of Study of Flow Specification

The attack path is determined by finding the destination IP addresses with highest relative frequency found in the packets processed by the controller. In DDoS, there are two main types of attacks. In the first type, the attacker targets a specific host. In the second type, the attacker might just distribute the attack traffic evenly over the network. As the external attackers can attack a limited number of IP addresses that are known to the outside, it is most likely that the attack can only target a restricted list of addresses, thus, limited set of hosts. In both cases, the controller should be able to locate the hosts that are under attack by examining the destination IP address frequency, F_i, computed in section 3.1. Our algorithm will determine the target host(s) as follows:

1. Find the host with the highest frequency, F_h. The host will be put in the target host list.
2. All the other hosts whose frequencies that is greater than 0.5*F_h will also be put into the target host list.

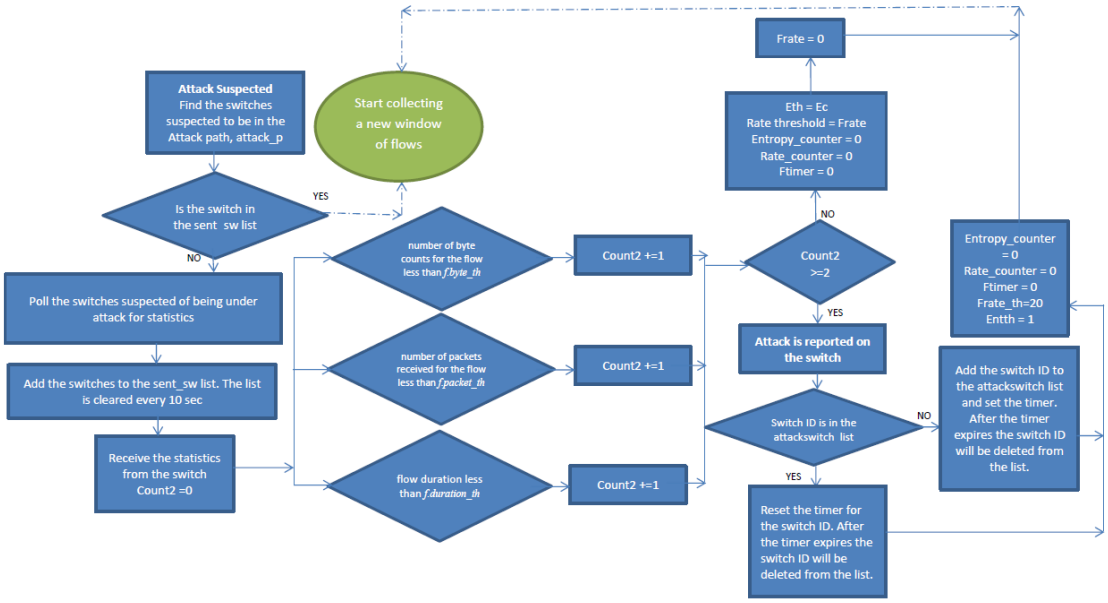

Figure 15 Implementation of Study of Flow Specification in the proposed Algorithm

After deriving the targeted host list, the algorithm determines a set of switches that are suspected under attack. The derivation can be formulated as follows:

Let m be the total number of paths to the hosts in the target host list. In other words, m is the total number of paths whose destination(s) is (are) suspected under attack. Let P_i be the set of switches that the i^{th} of the m paths traverses

$$P_i = \{S_{i1}, S_{i2}, \dots, S_{in}\} \qquad (3.4)$$

Where S_{ij} is the j^{th}switch of path *i* and *n* is the length of the path in terms of number of switches.

Let *SA* be the set of switches that could be under attack, then

$$SA = P_1 \cup P_2 \cup \dots \cup P_m \qquad (3.5)$$

After finding the set SA, the algorithm will send a request to download the flow tables from all the switches in the set. In order to prevent from creating excessive flow table download, the request is limited to at most once in every 10 second. Every switch that is queried for flow tables from the controller is added to a list called *sent_sw* list. The controller will not make a flow-table download request to the switches in the *sent_sw* list. The switches remain in this list for 10 seconds before being removed.

Processing the flow tables also puts a load on the controller and the network. This is the reason that the proposed algorithm does not perform periodic querying as some other detection methods do. This algorithm only queries the switches when an attack is suspected and the same switch will not be queried more than once within a certain time interval (10 sec in this thesis).

The handle_flowstats_received is the function used to analyze the flow tables received from switches. Having too many short flows or flows with small number of bytes or packets is considered as indications of an attack. For every received flow table every flow is checked for these three features and a counter (*count2*) is incremented by one when a flow matches two out of three of the following conditions:

1- Is the number of byte counts of the flow, *f.byte_count,* less than *f.byte_th*
   (*f.byte_count < f.byte_th*)
2- Is the number of packets received *of* the flow, *f.pack_count,* less than *f.packet_th* packets
   (*f.packet_count < f.packet_th)*
3- Is the flow duration, *f.duration_sec.* less than *f.duration_th*
   (f.duration_sec < f.duration_th)

The threshold values are chosen based on the averages reported by the algorithm under normal traffic flow in the learning phase with a 30% safety margin. This is shown in equation 3.6:

*threshold_values = Average_values *0.7* (3.6)

It is important to make sure that as the average values change under different network traffic load the threshold values are also updated.

When all the flows in the flow table are examined, an attacking rate, A_{rate}, is calculated to represent the probability that this switch might be under attack. This attacking rate is calculated (equation 3.7) by dividing the number of flows that satisfy two of the above conditions by the total number of flows in the flow table:

$$A_{rate} = \frac{count2}{number\_flows} \quad (3.7)$$

Finally, the proposed detection method will declare there is an attack if

$$A_{rate} > frate_{th}$$

where $frate_{th}$is the threshold computed (equation 3.8) by multiplying the average rate of A_{rate}during the learning phase by 1.4(a 40% safety margin)

$$Frate_{th} = average\ of\ (A_{rate}) * 1.4\ ;\ (3.8)$$

Note that the value of $Frate_{th}$ depends on how we choose *f.byte_th, f.packet_th and f.duration_th.* Normally, these thresholds are chosen such that $Frate_{th}$is relatively small, in the range between *0* and *0.5*. In general choosing an appropriate value of $Frate_{th}$ is very vital in the false positive and false negative detection rates. If this threshold is too high it might result in high rate of false negative especially in its early stage. If the threshold is too low it will result in high rate of false positive. Choosing this value very much depends on the level of sensitivity we require in our network protection. If the network is very sensitive and an early detection is crucial, it is better to lower the value of $Frate_{th}$; otherwise, we set it at a higher value.

3.4 Attack Mitigation

If a switch is reported as being under attack the algorithm should try to mitigate the attack. A number of possible attack mitigation approaches include installing flows in the attack paths to drop packets until the attack is stopped or blocking the incoming ports where the attack traffic is arriving at.

Although all these methods will mitigate the attack and will buy time for the network operators to find the attack sources before the break down of the controller or switches the adoptions of these methods will also affect the legitimate traffic as much as the attack traffic and the network services will become unavailable or respond slowly to legitimate traffic.

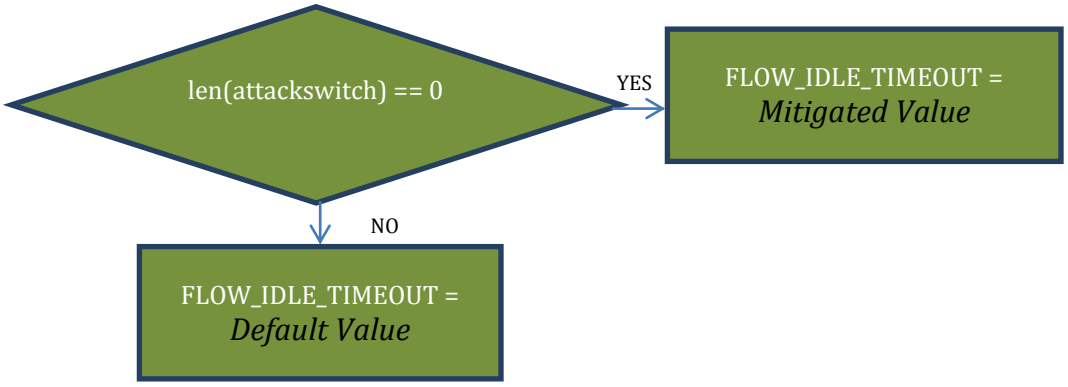

Figure 16 Implementing Attack Mitigation in the proposed Algorithm

The controller is usually designed with high capacities and therefore it will not crash very easily. The switches on the other hand have limited resources and are not very robust against attacks. When an attack is underway the flow table on the switches will be filled with a large number of short flows that will eventually break the switch. In the proposed mitigation algorithm the flow idle timer will be changed from the default value to the mitigated value to prevent the breakdown of the switches. The mitigated value is smaller than the default value; consequently, the short malicious flows will time out quickly and are deleted from the switch flow tables. The legitimate traffic flows on the other hand are expected to have a longer connection with a larger number of packets. If the *mitigated value* is chosen correctly it will not affect the legitimate flow entries significantly but will clear out the malicious flows quickly.

Great care must be taken to choose the mitigated value. If the *mitigated value* is shorter than the time out set for the sent_sw list the algorithm may oscillate. This can be explained as follows. As soon as the attack is detected the default value of the *idle_timer* will shorten to the mitigated value. In the next check for attack most of the malicious flows have timed out and been cleared from the flow table and so no more attack might be reported resulting in a false negative. Consequently the *idle_timer* will return back to its default value that will result in the overload of the flow tables of the switches that are still under attack. In a short while the algorithm

will detect an attack again and the same process will repeat. This behavior results in instability in the detection system. Therefore we must always ensure that:

Mitigated value of the idle_timer> Timer set for the sent_sw list (10 sec in this thesis)

Chapter 4

4 Simulation and Results

The simulation and testing of the proposed method for DDoS attack detection is explained through the following sections. The algorithm is implemented on the python based pox controller in the Mininet virtualized network environment. Scapy scripts [33] are used to generate the legitimate and attack traffics on the network hosts during the simulation.

4.1 Mininet

Mininet [34] is a tool used to simulate the Software Defined Networks, allowing a simple and quick approach to create, interact and customize prototypes for Software Defined Networks. Mininet allows network topologies to be specified parametrically [2]. It also allows configuration of a range of performance parameters for every virtual link. This is necessary for simulating real world systems and a requirement to implement most attack scenarios simulated in this thesis [11].

4.2 Traffic Generation

The tool used in this project to generate both the legitimate and attack traffic is scapy [35]. Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the network, simulate attacks, capture packets, match requests and replies, and much more. Scapy can be run in two different modes, interactively from a terminal window and programmatically from a Python script. The normal and attack traffic scripts used in this study are written in python. Three different scripts are written for the normal and attack traffic flows.

In Mininet the IP addresses are assigned incrementally starting from 10.0.0.1. For the normal traffic the destination IP address is generated based on the range we

have specifies in the script (e.g. 10.0.0.1 – 10.0.0.64). The source IP addresses are generated using a random function “randrange(1,256)”.

Two different attack scripts are used to simulate single victim attack and multiple victim attacks. In the single victim attack script all the packets will be forwarded to a single victim’s destination address that is specified while calling the attack script (e.g. python attack.py 10.0.0.13). The source IP addresses of attack traffic are generated using the same mechanism used for normal traffic.

The other parameters that are set in scapy are the packets type, the number of packets to be sent, packet payload and the traffic interval. The type of packets chosen is UDP packets for both attack and normal traffic. Normal traffic packets carry a payload while the attack traffic does not carry a payload. Normal traffic flows have a longer connection time than attack traffic flows and also have longer transmission interval.

4.3 Simulation Scenarios and Results

The simulation is performed on a laptop. Dell Inspiron 5, 7000 series with Intel Core i7-4500U CPU, 1.80 GHZ and 16 GB RAM. As shown in figure 17 the network structure built for the current simulation is a tree type network of depth two and fan-out of eight that creates sixty four hosts. A tree structure is chosen based on the fact that it is the network structure widely used in data centers. The switches used in the network are open virtual switches or OVS.

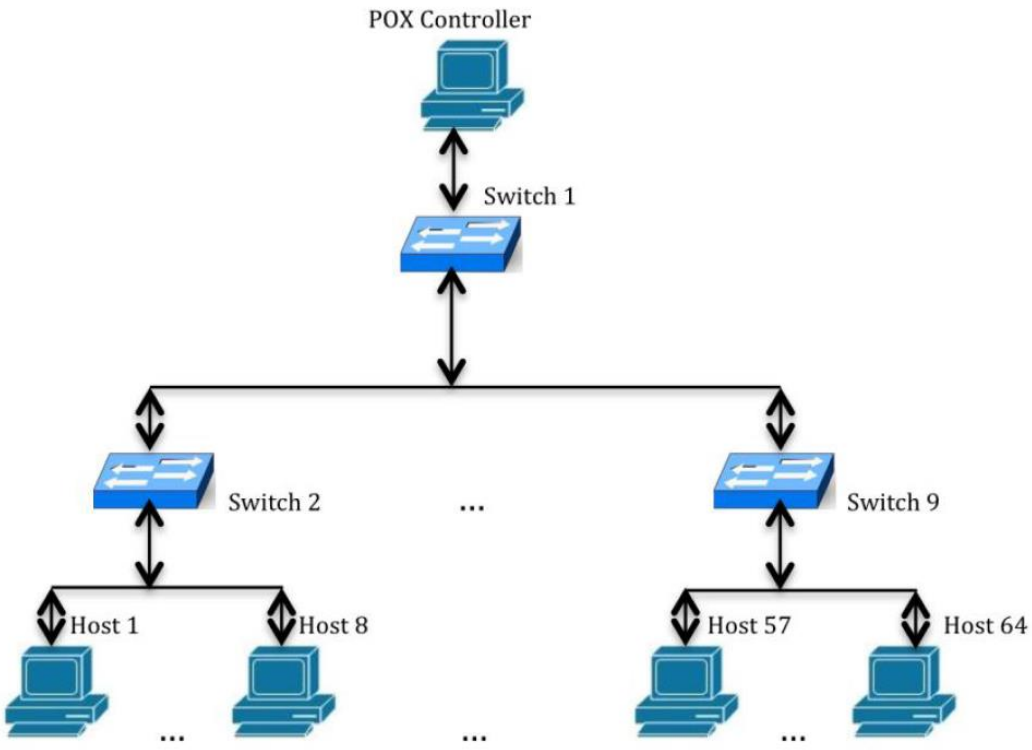

Figure 17 Network Topology

In the first step, in section 4.3.1 we will have a look at the overall algorithm behavior in detecting attacks under different legitimate traffic patterns. Three traffic patterns will be tested for the legitimate traffic running in the network and the algorithm behavior is observed under different traffic loads. In section 4.3.2 more detailed analysis is performed on how effective the algorithm proves to be in identifying the exact attack paths and minimizing the detection delay. A study is also performed on how effective each applied detection technique is in detecting the reported attacks. Section 4.3.3 performs a study over the algorithm detection performance with different legitimate and attack flow types. A study of the effectiveness of the mitigation method is covered in section 4.3.4.

4.3.1 False Positive and False Negative Attack Detections

In this section the algorithm is run under three different traffic patterns and the False Positive and False negative detection rates are observed in each case. Three different traffic patterns are tested under two attacking types: Single victim attack and Multiple victim attack.

The characteristics of each traffic pattern are as follows:

Traffic Pattern A

In this traffic pattern there is a distinct difference in the characteristics of legitimate and attack traffic parameters. Legitimate traffic is defined as traffic with long duration flows, large number of packets and bytes whereas the attack traffic has short duration flows with small number of packets and no payload. Table 2 summarizes the traffic specifications in each attack scenario.

LEGITIMATE TRAFFIC SPECIFICATION	
Packets Type	UDP
Packet Payload	21 Bytes
Number of Packets to be sent	7
Traffic Interval	0.2 sec
Traffic Rate	5 Packet/sec
Flow Rate	0.6 Flow/sec

SINGLE VICTIM ATTACK TRAFFIC SPECIFICATION	
Packets Type	UDP
Packet Payload	-
Number of Packets to be sent	1
Traffic Interval	0.08 sec
Traffic Rate	12.5 Packet/sec
Flow Rate	12.5 Flow/sec

MULTIPLE VICTIM ATTACK TRAFFIC SPECIFICATION	
Packets Type	UDP
Packet Payload	-
Number of Packets to be sent	1
Traffic Interval	0.03 sec
Traffic Rate	33.3 Packet/sec
Flow Rate	33.3 Flow/sec

Table 2 Traffic Pattern "A" Legitimate and Attack Traffic Specifications

The interval set for normal traffic is 0.2 sec whereas the interval for attack traffic is set to 0.08 sec for single victim attack and 0.03 sec for multiple victim attacks. The reason to change the interval between the single victim attack and multiple victim attack is to generate the same amount of attack traffic for both attack scenarios.

Let's define Attack Traffic Ratio as the ratio of the attack traffic rate to the total traffic rate as is described in equation 4.1:

Attack Traffic Ratio = Attack Traffic Rate / Total Traffic Rate *(4.1)*

The total number of traffic sources is fixed at twenty, and with n attackers, the number of legitimate traffic sources is 20-n. In the single victim attack when one host is sending attack traffic with traffic rate of 12.5 packets/sec (1/0.08 = 12.5 packets per second) nineteen other hosts will be sending normal traffic with individual traffic rate of 5 packets/sec (1/0.2 = 5 packets per second) that will result in 13% attack traffic ratio. With two attacking hosts we will have 28% attack traffic

ratio, with three attacking hosts 45% attack traffic ratio and with four attacking hosts 63% of attack traffic ratio.

In the multiple victim attack scenario with an interval of 0.03 approximately 33.3 packets will be generated per second with nineteen hosts generating 5 packets of legitimate traffic per second. This results in 26% attack traffic ratio with a single host sending attack traffic to four different destinations. In the next phase when two different hosts start the attack destined to eight different destinations the attack traffic ratio changes to 42.5% and when three hosts attack twelve destinations the attack traffic ratio changes to 54%.

Table 3 summarizes the threshold values set for the study of the flow specifications.

f.byte_th	*f.packet_th*	*f.duration_th*
120 bytes	5 packets	9.999999999 seconds

Table 3Threshold values set for the Study of the Flow Specifications in Traffic Pattern "A"

As explained in section 3.3.2 the threshold values are calculated according to equation 3.7 based on the average rates reported by the algorithm under normal traffic flow in the learning phase with a 30% safety margin.

Traffic Pattern B

In this traffic pattern there is less difference in the characteristics of legitimate and attack traffic parameters. Legitimate traffic is defined as traffic with shorter duration flows, less number of packets but large payload. The attack traffic has short duration flows with small number of packets and no payload. Table 4 summarizes the traffic specifications in each attack scenario.

LEGITIMATE TRAFFIC SPECIFICATION	
Packets Type	UDP
Packet Payload	21 Bytes
Number of Packets to be sent	4
Traffic Interval	0.1 sec
Traffic Rate	10 Packet/sec
Flow Rate	2.5 Flow/sec

SINGLE VICTIM ATTACK TRAFFIC SPECIFICATION	
Packets Type	UDP
Packet Payload	-
Number of Packets to be sent	1
Traffic Interval	0.08 sec
Traffic Rate	12.5 Packet/sec
Flow Rate	12.5 Flow/sec

MULTIPLE VICTIM ATTACK TRAFFIC SPECIFICATION	
Packets Type	UDP
Packet Payload	-
Number of Packets to be sent	1
Traffic Interval	0.03 sec
Traffic Rate	33.3 Packet/sec
Flow Rate	33.3 Flow/sec

Table 4Traffic Pattern "B" Legitimate and Attack Traffic Specifications

Table 5 summarizes the threshold values set for the study of the flow specifications.

f.byte_th	*f.packet_th*	*f.duration_th*
120 bytes	2 packets	5.699999999 seconds

Table 5Threshold values set for the Study of the Flow Specifications in Traffic Pattern "B"

Traffic Pattern C

This traffic pattern is a mix of traffic patterns "A" and "B". Half of the hosts with legitimate traffic will be sending traffic according to pattern "A" and the other half will send traffic according to pattern B. This traffic pattern generates a combination of different flow types. The attack traffic will remain unchanged.

Table 6 summarizes the threshold values set for the study of the flow specifications.

f.byte_th	*f.packet_th*	*f.duration_th*
120 bytes	2 packets	7.309999999 seconds

Table 6Threshold values set for the Study of the Flow Specifications in Traffic Pattern "C"

The simulation consists of two attack types: Single victim attack and Multiple victim attack.

In the single victim attack twenty simulation runs are performed. Each simulation run consists of four attack scenarios with one, two, three or four hosts attacking a single victim with destination IP (10.0.0.13).

In the multiple victim attack, twenty simulation runs are performed. Each simulation run consists of four attack scenarios. In these scenarios the attacker distribute the attacks among a group of four, eight, twelve and sixteen victims while sending the attack traffic from one, two, three or four hosts, respectively.

4.3.1.1Single Victim Attacks

In Single victim attacks simulation run, Normal traffic is run on 20 hosts for at least one minute to let the network stabilize and the algorithm to derive appropriate

values for different thresholds before any attacks are carried out. This period can also be regarded as the initial learning phase for the algorithm.

Each run covers four attacks scenarios. Each simulation run continues between twenty to forty minutes and each attack scenario lasts for about three minutes. While the simulation is running if an attack is suspected either due to entropy changes or changes in the traffic rate an event is created within the log file and time stamped. The switches suspected of being in the attack path are also identified and a list of these switches is written to the logfile along with a time stamp. Further along the analysis of this event if an attack is confirmed on any of the switches a warning is created that warns of an attack with the exact switch number mentioned. This attack report is simultaneously printed on the screen, written to the logfile and emailed to the network administrator with the exact time stamp mentioned. The rate of the suspected attack being an attack is also logged that could be significant information for the network administrator to set the threshold value used for distinguishing between legitimate and malicious flows in the study of flow specifications. After an attack is cleared the clearance of the attack is also reported and written to the logfile.

The table below summarizes the number of False Positive (FP) and False Negative (FN) reports on while traffic patterns "A", "B" and "C" are tested.

	Traffic Pattern A		Traffic Pattern B		Traffic Pattern C	
Total Number of Attacks on the System	80		80		80	
	Error Counts	Error Probability	Error Counts	Error Probability	Error Counts	Error Probability
FP	0	0%	1	1.3%	5	6.3%
FN	0	0%	0	0%	3	3.8%
Algorihm Detection Rate	**100.00%**		**98.75%**		**90.00%**	
Algorithm Failure Rate	**0.00%**		**1.25%**		**10.00%**	

Table 7FP and FN Reports under different Traffic Patterns in Single Victim Attacks

As shown in table 7 when the attack and legitimate traffic have distinct characteristics as it was in traffic pattern "A", the algorithm is able to perform with a 100% detection rate. The results and log files obtained under this traffic pattern will be studied under much more detail in the next section.

The algorithm also performs very well in traffic pattern "B". Even though the legitimate traffic and attack traffic characteristics start to move closer and look alike still only one case of false positive report is seen in eighty simulation runs that results in an error probability of less than 1.3%. This high rate of detection is a result of dynamic threshold selection values and the ability of the algorithm to adjust to different traffic characteristics.

Traffic pattern C is the point that although it still has a high detection rate of 90.00% but starts to show an increase in both false positive and false negative detections. In traffic pattern C the assumption is that the legitimate traffic is a mix of short and long flows. The shorter flows in this traffic pattern are very similar to the flows generated by the attack traffic. In this case finding the appropriate threshold is very difficult.

4.3.1.2 Multiple Victim Attacks

In this part of the simulation twenty different runs are performed. Each run consists of three scenarios. In the first scenario nineteen hosts will be running normal traffic and one host is generating attack traffic 26% attack traffic ratio under traffic pattern "A"). The attacker will be attacking four different destinations. In the second scenario two hosts will send their attack traffic to eight victims. Two attacking hosts will leave eighteen hosts generating legitimate traffic (generating a 42.5% attack traffic ratio under traffic pattern "A"). In the third scenario three hosts will send attack traffic leaving seventeen hosts generate legitimate traffic while attacking twelve victims (resulting in 54% attack traffic ratio under traffic pattern "A"). In the fourth scenario four hosts will send attack traffic leaving sixteen hosts generate legitimate traffic while attacking sixteen victims.

Each simulation starts with a two-minute learning period with all twenty hosts running legitimate traffic. Each simulation lasts around ten to twenty minutes. All the detecting events including suspecting of an attack, list of switches on the suspected paths, confirmation of an attack on detected switches and clearing of an

attack are written to a log file. The other elements of the attack remain the same as the single victim attack.

The table below summarizes the number of FP and FN reports under the traffic patterns “A”, “B” and “C”.

	Traffic Pattern A		Traffic Pattern B		Traffic Pattern C	
Total Number of Attacks on the System	80		80		80	
	Error Counts	Error Probability	Error Counts	Error Probability	Error Counts	Error Probability
FP	0	0%	1	1.3%	4	5.0%
FN	0	0%	2	0%	5	6.3%
Algorihm Detection Rate	100.00%		96.25%		88.75%	
Algorithm Failure Rate	0.00%		3.75%		11.25%	

Table 8FP and FN Reports under different Traffic Patterns in Single Victim Attacks

As we see in table 8, the results obtained in the multiple victim attacks show a high detection rate for all three types of traffic patterns. Similar to the single victim attack, traffic pattern “A” is detected with 100% detection rate and the algorithm proves to perform extremely well in these cases.

Although the legitimate and attack traffic characteristics are close to each other in traffic pattern B, the threshold update helps to keep the detection rate at 96.25% that is a very high detection rate especially in multiple victim attacks. When the attacks are distributed among different victims the chances of an attack not being detected, FN, increases as the attack traffic is distributed among different destinations and the effect of the attack might not reach the threshold values.

As we see in the above table the detection rate in Traffic pattern “C” shows a little decline to 88.75%. Traffic pattern “C”, is the complicated case where finding the appropriate threshold is very difficult. Since this traffic pattern is a mix of different flow types that some are very long, with higher number of packet counts and some are very short with limited number of packets it is difficult to find the appropriate margin to fit both flow types.

In the case of single victim attacks while running traffic pattern "A" the proposed algorithm was 4% more effective compared to the algorithm proposed in [30] which only chooses entropy as the alone detection method and has a detection rate of 96%. Through the experiments performed in section 4.3.2.2 we see that comparing our algorithm with the entropy variation detection algorithm [30] the proposed algorithm is 52.16% more effective in detecting multiple victim attacks in traffic pattern "A".

The other SDN based technique in detecting DDOS attacks is the use of Self Organizing Maps (SOM) in the NOX controller for DDOS attack detections [26]. The detection results presented show a detection rate between 98.57% and 99.11%.This method requires a very long training period that will take several hours. The switches are polled in short periods (every 3 sec) for their flow statistics and complicated computations are performed to generate the required comparison data. This consumes a lot of network and controller resources, thus, it might not be always practical especially in larger network structures.

The proposed algorithm in this thesis provides a light weight detection method that requires minimal network resources while it delivers a very high detection rate. This detection method has proven to function well under different traffic behaviors with minimal detection delays. This method is the only proposed method that in addition to detecting the attacks it is able to detect the attack paths and report the switches affected by the malicious traffic with high accuracy.

4.3.2Detailed Analysis of Attack Path Detection and Detection Delays

4.3.2.1 Single Victim Attacks

In this section, we study the performance of the proposed methods on identifying the switches that are under attack.

Figures 18 to 21 illustrate the false negative and false positive rates under 13%, 28%, 45% and 63% attack traffic ratio. If a switch is not in the attack path but it is reported as being under attack it is counted as an instance of FP report. If a switch is in the attack path but is not reported as being under attack it is counted as a FN report.

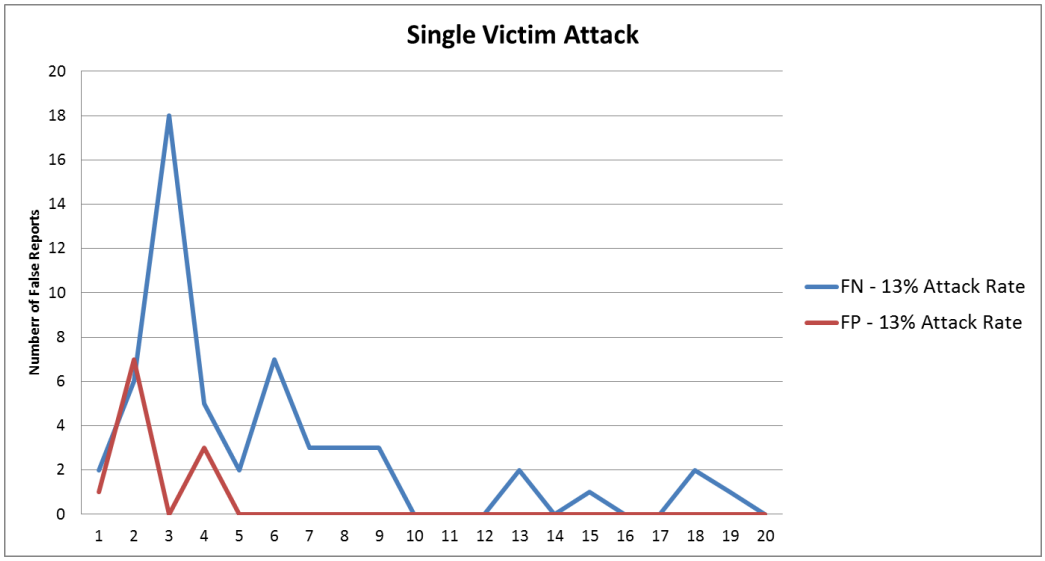

Figure 18 False Positive / False Negative reports in Traffic Pattern "A" Single Victim Attack scenario under 13% Attack Rate

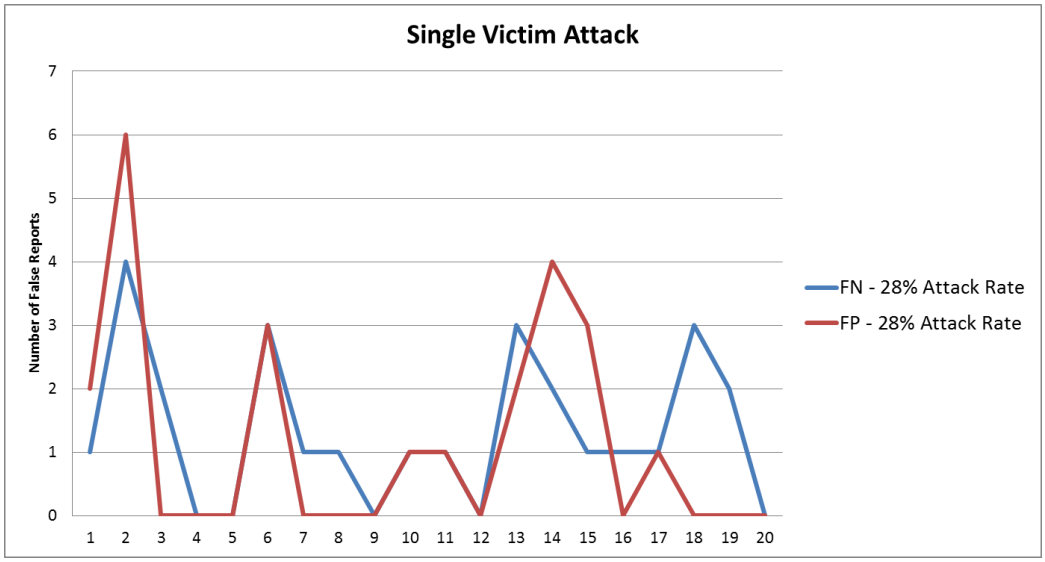

Figure 19 False Positive / False Negative reports in Traffic Pattern "A" Single Victim Attack scenario under 28% Attack Rate

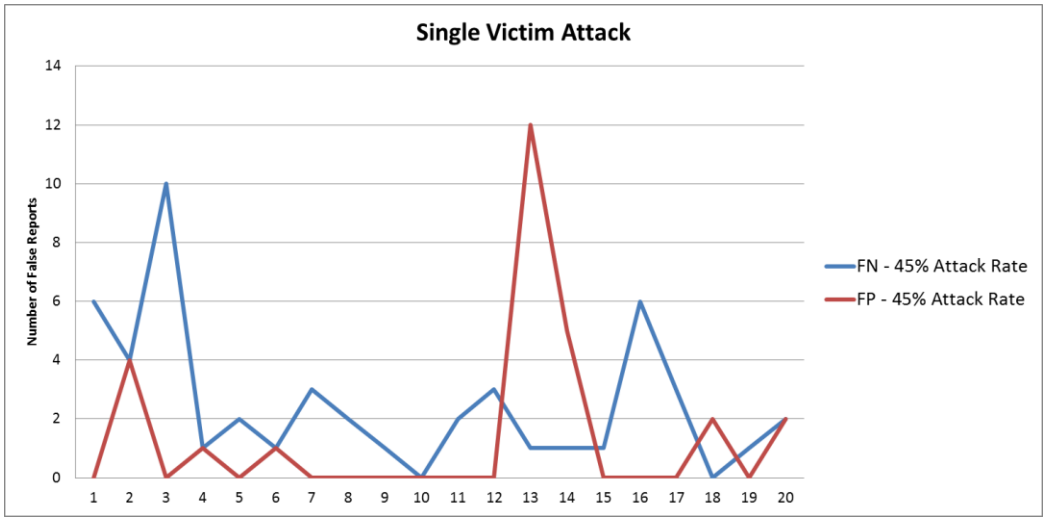

Figure 20 False Positive / False Negative reports in Traffic Pattern "A" Single Victim Attack scenario under 45% Attack Rate

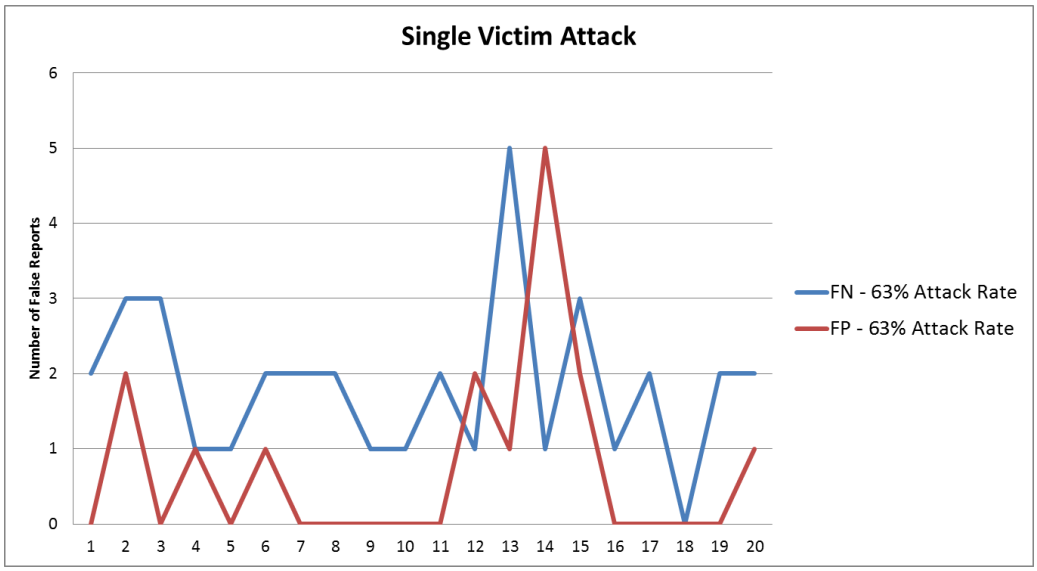

Figure 21 False Positive / False Negative reports in Traffic Pattern "A" Single Victim Attack scenario under 63% Attack Rate

As we can see in the above figures the highest rate of false negative reports are detected while the attack traffic is at its lowest rate. At 13% attack rate the FN reports reaches its peak at 18 reports in scenario 3. This is expected for the lower the attack traffic is, the harder the attack will be detected.

Table 9 shows the average false negative reports under different attacking rates. Although the highest FN rate belongs to the lowest attack rate but as we can see in the figure 22, the FN rate does not follow a linear behavior. Statistically, the result

shows that the algorithm gives a very low FN rate in all range of attacks. Although the diagram shows a small drop and peak with the change of attack traffic but the fluctuation is less than 0.93%.

Table 9 False Negative Report Statistics in Traffic Pattern "A" Single Victim Attack scenario

	13% Attack Traffic	28% Attack Traffic	45% Attack Traffic	63% Attack Traffic
Total Reports	3069	3069	3168	3177
FN	55	27	50	37
%	1.792114695	0.879765396	1.578282828	1.164620711

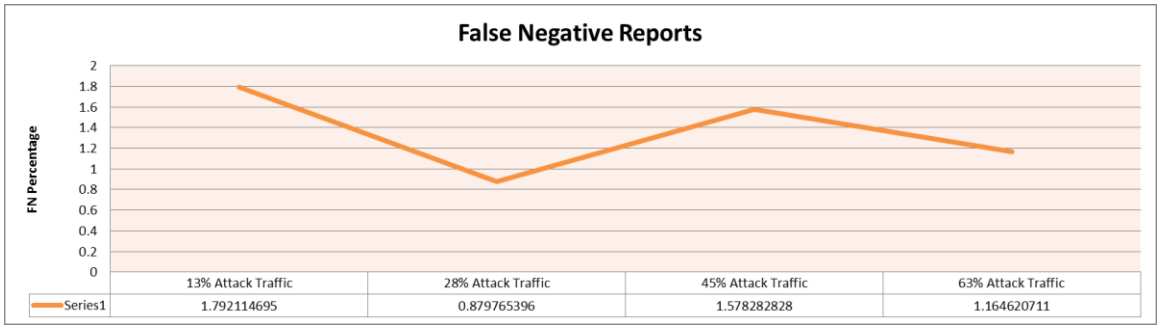

Figure 22 False Negative Reports Behaviour in Traffic Pattern "A" Single Victim Attack scenario

Table 10 shows the average false positive reports under different attack rates. For the first three attacking rates the false positive reports tend to increase with the traffic rate but as the traffic grows more in the network the FP reports begin to fall back to its minimum value. The result shows that the algorithm gives a very low FP rate in all range of attacks. The fluctuation in the peak and lowest probability of FP reports is less than 0.5%. The average probability of FP report in a single victim attack is 0.6%.

Table 10 False Positive Report Statistics inTraffic Pattern "A" Single Victim Attack scenario

	13% Attack Traffic	28% Attack Traffic	45% Attack Traffic	63% Attack Traffic
Total Reports	3069	3069	3168	3177
FP	11	23	27	15
%	0.358422939	0.749429782	0.852272727	0.472143532

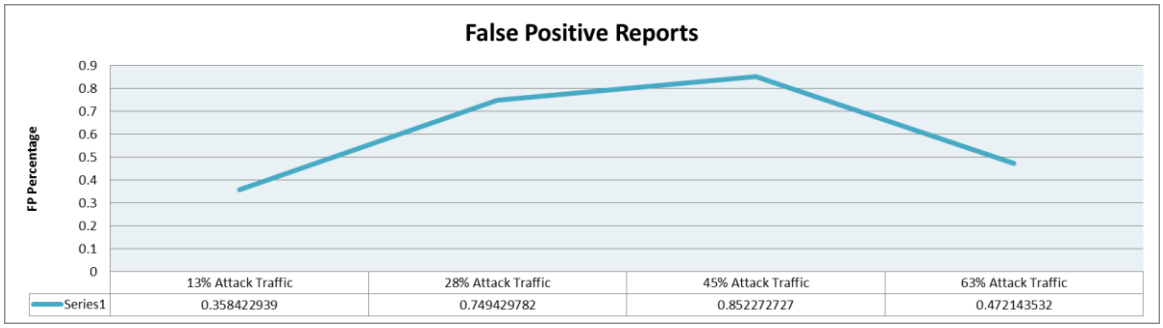

Figure 23 False Positive Reports Behaviour in Traffic Pattern "A" Single Victim Attack scenario

The total probability of error in identifying the exact attacking path inclusive of FN or FP reports is approximately 1.3%.

Not only are the error probability of false reporting (both negative and positive) so low but it could easily be reduced by small changes in the algorithm. In the results obtained from the simulations it can be observed that as long as the controller and the switches are not taken down by the attacks a FP or FN report will not be repeated more than two times. This means that in the pattern shown in all the simulations run through this thesis a FN or FP report is continuously repeated for maximum two times and the error is corrected through the results generated in maximum third rounds of consecutive reports. For example if a switch is reported as being under attack and this is a FP report it will not be reported as being under attack after two more rounds of reports. As the switch list is cleared after each ten seconds we could approximately say that an error will be corrected after maximum of thirty seconds as long as the switches and the controller are still operational through the attacks.

Figure 24 shows part of the logged events by the algorithm. The red switch IDs indicate a report of an attack on a switch and the cells highlighted in blue indicate a false negative report. Switches 2, 4 and 5 are reported as being under attack at 5:37:34 PM that are FN reports. In the next round of reports in around 5 seconds later at 5:37:40 PM another report confirms that there's no attack running on these switches and the switches show as being safe in the reports generated afterwards too.

Likewise switches 4 and 5 have FN attack reports on them at 5:38:39 PM. The switches are reported as being under attack in the next round of reports at 5:38:50. In the next round of reports at 5:39:02 that is around 23 seconds later the switches are reported as being safe and the FN reports are corrected by the algorithm.

Attack from	Attack Path	Attack Suspected at:	Switches list	Attack Reported at:
h35,h42,h51	s6,s7,s8,s3,s1	5:37:34 PM	00-00-00-00-00-01',	5:37:34 PM
			00-00-00-00-00-02',	5:37:34 PM
			00-00-00-00-00-03',	5:37:34 PM
			00-00-00-00-00-04',	5:37:34 PM
			00-00-00-00-00-05',	5:37:34 PM
			00-00-00-00-00-06'	5:37:34 PM
			00-00-00-00-00-07',	5:37:34 PM
			00-00-00-00-00-08',	5:37:34 PM
			00-00-00-00-00-09',	
h35,h42,h51	s6,s7,s8,s3,s1	5:37:39 PM	00-00-00-00-00-01',	5:37:40 PM
			00-00-00-00-00-02',	
			00-00-00-00-00-03',	5:37:39 PM
			00-00-00-00-00-04',	
			00-00-00-00-00-05',	
			00-00-00-00-00-06'	5:37:39 PM
			00-00-00-00-00-07',	5:37:39 PM
			00-00-00-00-00-08',	5:37:39 PM
			00-00-00-00-00-09',	
h35,h42,h51	s6,s7,s8,s3,s1	5:38:22 PM	00-00-00-00-00-01',	5:38:23 PM
			00-00-00-00-00-02',	
			00-00-00-00-00-03',	5:38:23 PM
			00-00-00-00-00-04',	
			00-00-00-00-00-05',	
			00-00-00-00-00-06'	5:38:22 PM
			00-00-00-00-00-07',	5:38:22 PM
			00-00-00-00-00-08',	5:38:22 PM
h35,h42,h51	s6,s7,s8,s3,s1	5:38:33 PM	00-00-00-00-00-01',	5:38:34 PM
			00-00-00-00-00-02',	
			00-00-00-00-00-03',	5:38:34 PM
			00-00-00-00-00-04',	
			00-00-00-00-00-05',	
			00-00-00-00-00-06'	5:38:34 PM
			00-00-00-00-00-07',	5:38:34 PM
			00-00-00-00-00-08',	5:38:34 PM
			00-00-00-00-00-09',	
h35,h42,h51	s6,s7,s8,s3,s1	5:38:39 PM	00-00-00-00-00-01',	5:38:39 PM
			00-00-00-00-00-02',	
			00-00-00-00-00-03',	5:38:39 PM
			00-00-00-00-00-04',	5:38:39 PM
			00-00-00-00-00-05',	5:38:39 PM
			00-00-00-00-00-06'	5:38:39 PM
			00-00-00-00-00-07',	5:38:39 PM
			00-00-00-00-00-08',	5:38:39 PM
			00-00-00-00-00-09',	
h35,h42,h51	s6,s7,s8,s3,s1	5:38:50 PM	00-00-00-00-00-01',	5:38:50 PM
			00-00-00-00-00-02',	5:38:50 PM
			00-00-00-00-00-03',	5:38:50 PM
			00-00-00-00-00-04',	5:38:50 PM
			00-00-00-00-00-05',	5:38:50 PM
			00-00-00-00-00-06'	5:38:50 PM
			00-00-00-00-00-07',	5:38:50 PM
			00-00-00-00-00-08',	5:38:50 PM
			00-00-00-00-00-09',	5:38:50 PM
h35,h42,h51	s6,s7,s8,s3,s1	5:39:01 PM	00-00-00-00-00-01',	5:39:02 PM
			00-00-00-00-00-02',	
			00-00-00-00-00-03',	5:39:02 PM
			00-00-00-00-00-04',	
			00-00-00-00-00-05',	
			00-00-00-00-00-06'	5:39:02 PM
			00-00-00-00-00-07',	5:39:02 PM
			00-00-00-00-00-08',	5:39:02 PM
			00-00-00-00-00-09',	

Figure 24 Sample FN Reporting from Attack Scenario 13

One of the goals of this thesis is to offer a solution to detect an attack at its early stages or in other words to have the minimum detection time. Figure 25 illustrates the average attack detection time under each attack traffic load. As we can see in the diagram with the increase in the attack load the detection time starts to drop. This is because the higher the traffic load the quicker the packet sampling window will be collected and therefore the faster the attack is detected. The average detection time for the single victim attack is 14.86 seconds.

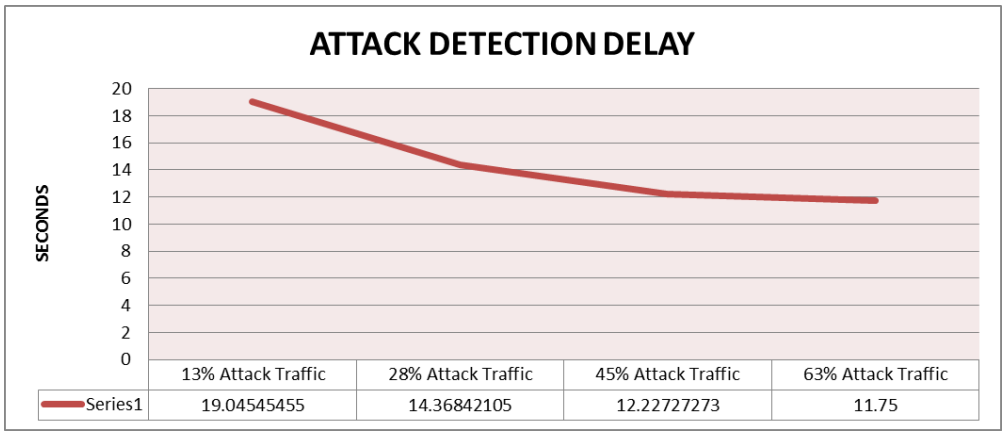

Figure 25 Attack Detection Delay in Traffic Pattern "A" Single Victim Attacks

As discussed in the previous chapter the attack detection procedure in the proposed algorithm is based on two measurements: entropy variations and sudden changes in the flow initiation rate. It is very interesting to have a study over the algorithm performance and see which parts of the algorithm are more effective on detecting certain types of attacks. Table 11illustrates the probability of detecting single victim attacks through each of the two detection techniques.

Table 11Comparing Entropy and Flow Initiation Rate Effectiveness in Detecting Attacks in Traffic Pattern "A" Single Victim Attacks

	13% Attack Traffic	%	28% Attack Traffic	%	45% Attack Traffic	%	63% Attack Traffic	%
Report of Attack being suspected	341		341		278		353	
Attack Detected through Entropy Changes	331	97.07	336	98.53	277	99.64	353	100
Attack Detected through Flow Initiation Rate Changes	10	2.93	5	1.47	1	0.36	0	0

As we can see chances of single victim attacks being detected through entropy changes is much higher than through flow initiation rate changes. With the probability of detection being between 97% and100% we can definitely say that entropy variation is the most effective method in detecting attacks that target a limited number of destinations in the network. This is because in such attacks most of the attack traffic is destined to a certain destination address and this lowers the entropy significantly.

4.3.2.2 Multiple Victim Attacks

As we saw in section 4.3.1.2 while running the simulations with traffic pattern "A" that is distinct legitimate traffic and attack traffic characteristics no false negative or false positive reports were perceived in the attack detection stage as in the single-victim case; but in finding the exact attack path and reporting the exact switches that are under attack there were instances that the algorithm reported false positive or false negative detections.

When studying the FN reports in multiple victim attacks, only the reports on the end switches (switches that are attached to the attacking hosts) are considered. This is to avoid querying and examining a large number of switches that could put a great load on the network and the controller. Figures 26 to 28 illustrate the false negative and false positive report instances under 26%, 42.5%, 54% attack traffic rates.

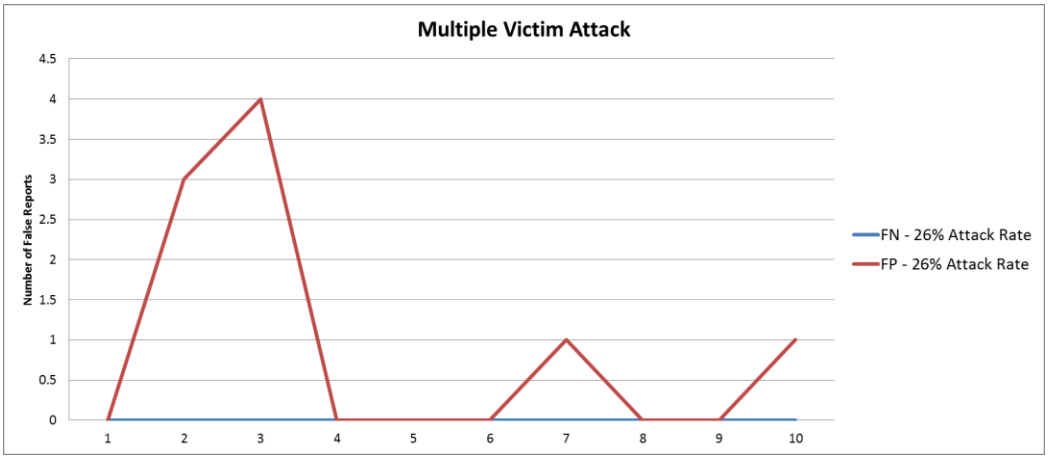

Figure 26 False Positive / False Negative reports in Traffic Pattern "A" Multiple Victim Attack scenario under 26% Attack Rate

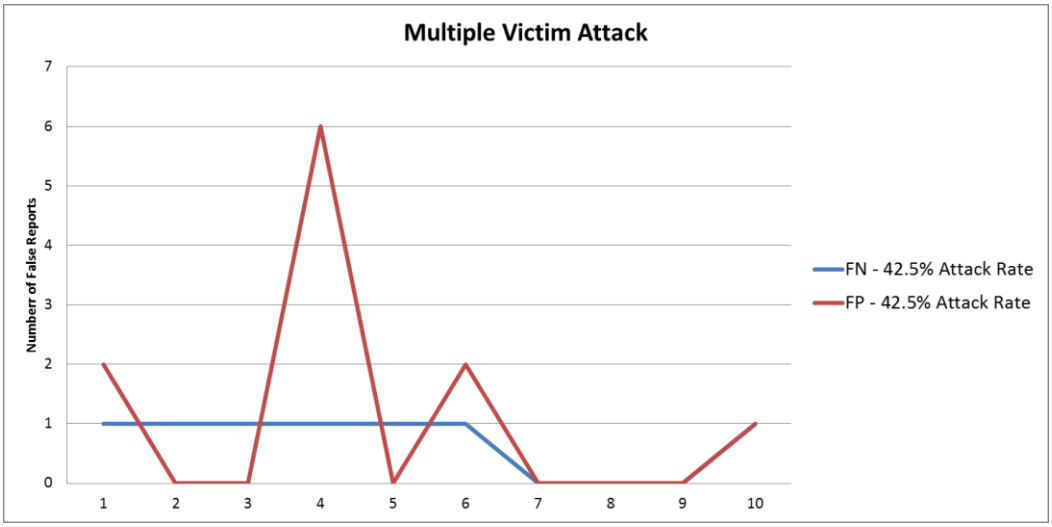

Figure 27 False Positive / False Negative reports in Traffic Pattern "A" Multiple Victim Attack scenario under 42.5% Attack Rate

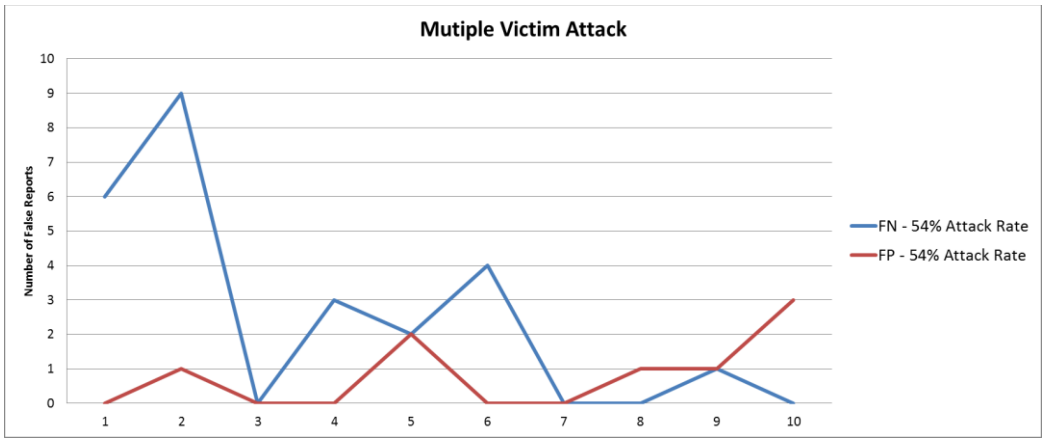

Figure 28 False Positive / False Negative reports in Traffic Pattern "A" Multiple Victim Attack scenario under 54% Attack Rate

As we can see in the above figures as the attack rate increases so does the chances of having false negative reports. This behavior can be explained as follows. In the experiments as the attack traffic increases so do the number of attacking victims and consequently the number of switches reported as being under attack also increases. Since the attack traffic is distributed to multiple victims, not all switches will receive a high volume of attack flows and the number of false negative reports starts to rise. Table 12 shows the FN reporting percentage. As we see the fluctuation is less than 1.3% and the average is less than 0.6% which is a low and acceptable error rate.

Table 12 False Negative Report Statistics in Traffic Pattern "A" Multiple Victim Attack scenario

	%26 Attack Traffic	%42.5 Attack Traffic	%54 Attack Traffic
Total Reports	1746	2016	1863
FN	0	7	25
%	0	0.347222222	1.341921632

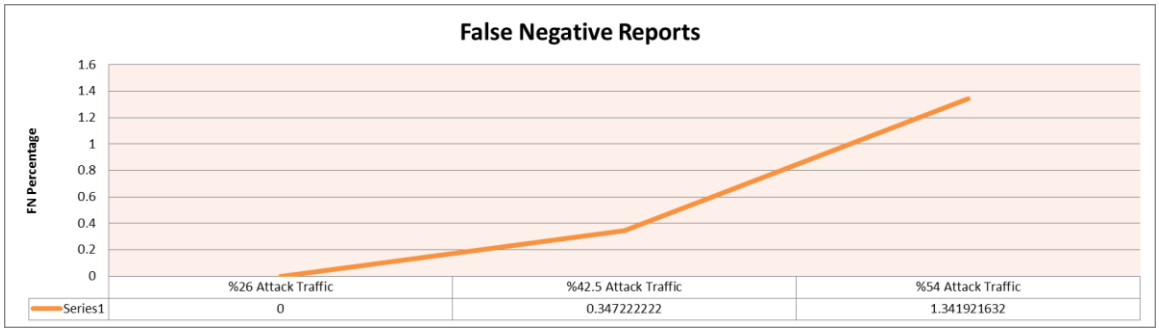

Figure 29False Negative Reports Behaviour in Traffic Pattern "A" Multiple Victim Attack scenario

The false positive reporting percentage shows a relative stable behavior with a slight decline at the attack traffic increases. The average FP percentage is 0.5% and the fluctuation is only around 0.1%.

Table 13 False Positive Report Statistics in Traffic Pattern "A" Multiple Victim Attack scenario

	26% Attack Traffic	42.5% Attack Traffic	54% Attack Traffic
Total Reports	1746	2016	1863
FP	10	11	8
%	0.572737686	0.545634921	0.429414922

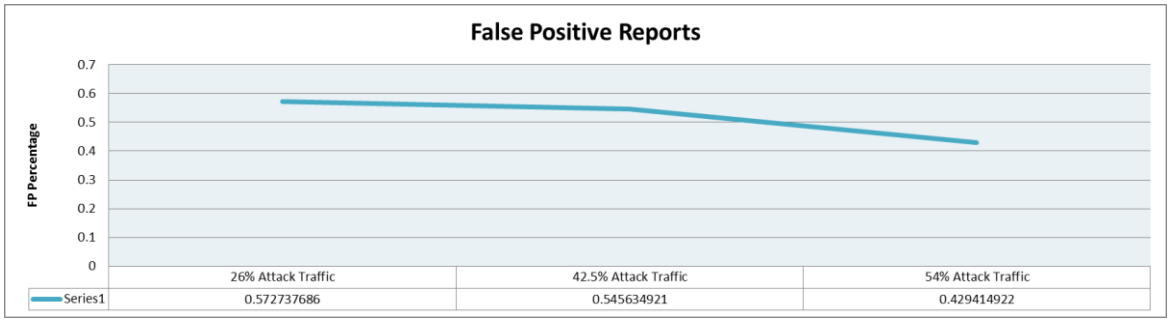

Figure 30 False Positive Reports Behaviour in Traffic Pattern "A" Multiple Victim Attack scenario

Figure 31 illustrates the average attack detection times under different attack traffic loads. As we can see in the figure with the increase in the attack load the detection time starts to drop. Again this is because the higher the traffic load the shorter duration of the packet sampling window and thus the faster the attack is detected.

Compared to single victim attacks the drop in attack detection time continues at a much slower pace. The average detection time for the multiple victim attack is 20.97 seconds that is 6.11 seconds longer than the case of single-victim attacks. Distributing the attack among different victims will result in less generation of short flows in each switch and also will reduce the effectiveness of entropy detection , both of which will result in a longer detection time.

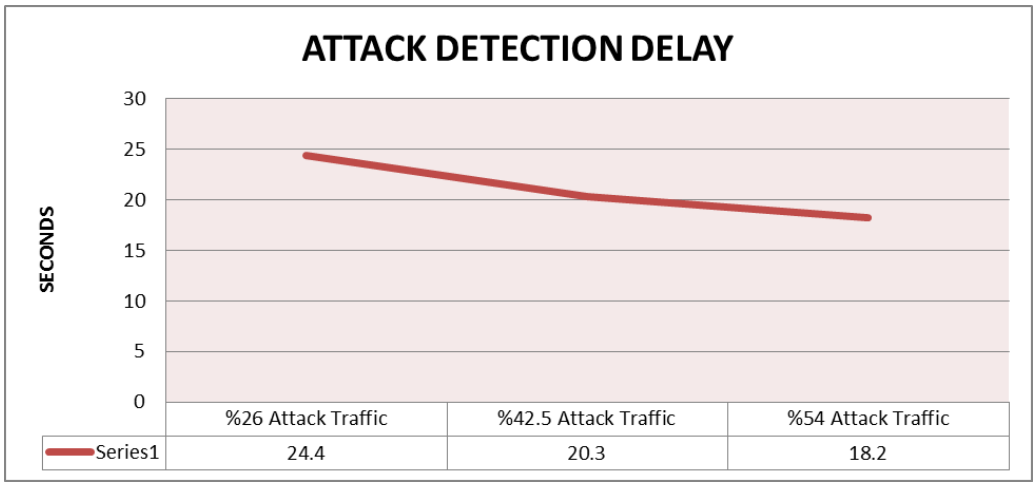

Figure 31 Attack Detection Delay in Traffic Pattern "A" Multiple Victim Attacks

Table 14 illustrates the probability of detecting multiple victim attacks through entropy and flow initiation rate variation detection techniques.

	26% Attack Traffic	%	42.5% Attack Traffic	%	54% Attack Traffic	%
Report of Attack being suspected	194		224		207	
Attack Detected through Entropy Changes	98	50.52	106	47.32	95	45.89
Attack Detected through Flow Initiation Rate Changes	96	49.48	118	52.68	112	54.11

Table 14Comparing Entropy and Flow Initiation Rate Effectiveness in Detecting Attacks in Traffic Pattern "A" MultipleVictim Attacks

As we can see in multiple victim attack, the entropy technique is not as effective as in single victim attacks. In fact chances of multiple victim attacks being detected through flow initiation rate changes is higher than through entropy changes. Comparing this algorithm with the entropy variation detection algorithm [30] the proposed algorithm is 52.16% more effective in detecting multiple victim attacks in

traffic pattern “A”. A detection rate of 52.16% through flow initiation rate changes shows that the entropy variation technique cannot be considered as a standalone detection method for the multiple victim attack.

4.3.3 Algorithm Detection Changes with Changes of Legitimate and Attack Flow Types

As we saw in the above discussed results the proposed algorithm is proven to detect attacks with minimal detection errors. Although the effort in choosing the parameters throughout the algorithm and in generating the attack and legitimate traffic flows has been based on the known attack characteristics and the observable differences between legitimate traffic and malicious traffic, we should always keep in mind that some legitimate traffic flows might be short and although they are legitimate they show the characteristics of malicious flows. Therefore in studying the proposed algorithm detection level it is also important to compare the detection rates while the legitimate traffic has similar characteristics to the attack traffic. In other words when the legitimate flows consists of many short flows instead of longer flows. In order to achieve shorter legitimate flows the number of packets sent in each flow and the traffic interval is decreased gradually. Table 15 shows the parameters set in each scenario. In all cases the legitimate flow is run over 35 hosts in the network.

10% Shorter Flows		40% Shorter Flows	
Packets Type	UDP	Packets Type	UDP
Packet Payload	21 Bytes	Packet Payload	21 Bytes
Number of Packets to be sent	7	Number of Packets to be sent	5
Traffic Interval	0.18	Traffic Interval	0.15
Traffic Rate in the Newtork (Packet /Sec)	194.44	Traffic Rate in the Newtork (Packet /Sec)	233.33
Flow Rate in the Newtork (Flow/sec)	27.78	Flow Rate in the Newtork (Flow/sec)	46.67
20% Shorter Flows		**50% Shorter Flows**	
Packets Type	UDP	Packets Type	UDP
Packet Payload	21 Bytes	Packet Payload	21 Bytes
Number of Packets to be sent	6	Number of Packets to be sent	4
Traffic Interval	0.17	Traffic Interval	0.14
Traffic Rate in the Newtork (Packet /Sec)	205.88	Traffic Rate in the Newtork (Packet /Sec)	250.00
Flow Rate in the Newtork (Flow/sec)	34.31	Flow Rate in the Newtork (Flow/sec)	62.50
30% Shorter Flows		**60% Shorter Flows**	
Packets Type	UDP	Packets Type	UDP
Packet Payload	21 Bytes	Packet Payload	21 Bytes
Number of Packets to be sent	6	Number of Packets to be sent	3
Traffic Interval	0.16	Traffic Interval	0.12
Traffic Rate in the Newtork (Packet /Sec)	218.75	Traffic Rate in the Newtork (Packet /Sec)	291.67
Flow Rate in the Newtork (Flow/sec)	36.46	Flow Rate in the Newtork (Flow/sec)	97.22

Table 15 Changing Legitimate Traffic Parameters

It is clear from table 16 that as the flows become shorter we start to see more false positive reports in the network. The algorithm proves to be quite reliable till a 30% margin where we start to see the first signs of false positive reports. The growth is very slow until the flows are fifty percent shorter where it starts to change very rapidly. As the flows are shortened by ten percent the false positive reports are doubled.

	10% Shorter Flows	20% Shorter Flows	30% Shorter Flows	40% Shorter Flows	50% Shorter Flows	60% Shorter Flows
Report of Attack being Suspected	0	8	26	25	58	104
Report of Attack being Detected	0	0	2	4	23	73
FP Error Probability	0%	0%	7.69%	16.00%	39.66%	70.19%

Table 16 FP Error Probability changes with Change of Legitimate Traffic Characteristics

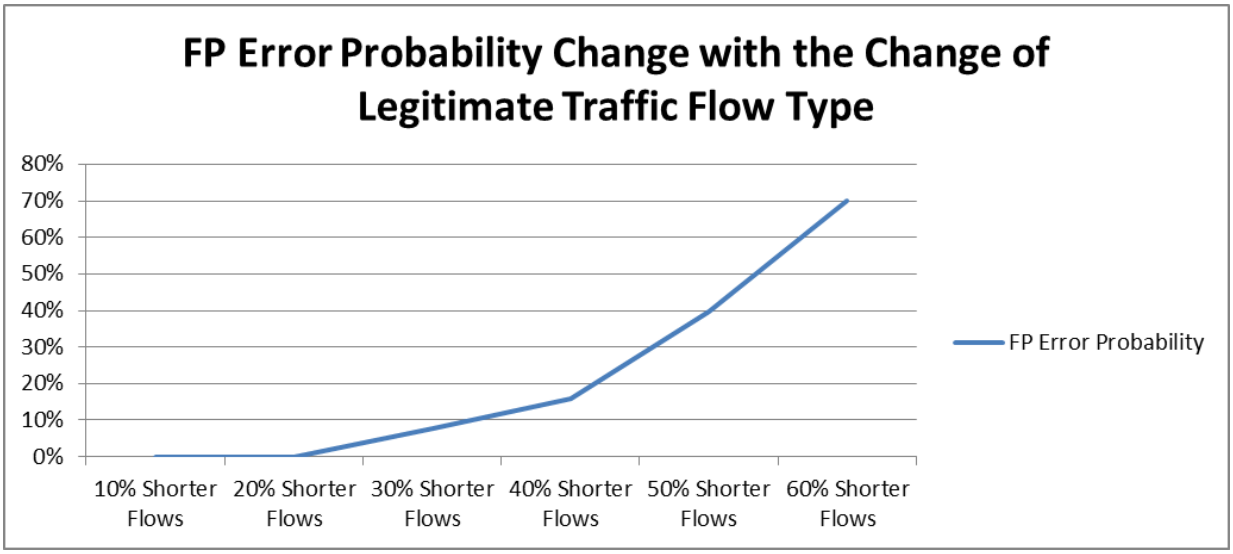

Figure 32 FP Error Probability Change with the Change of Legitimate Traffic Flow Type

In addition, the attackers are always seeking new attack methods and therefore their attack traffic characteristics might also adapt to defeat the detection method. One of such methods is to make the malicious flows look like legitimate flows by making them appear as longer flows. It is very important to see how the algorithm starts to behave when the malicious traffic starts to generate longer flows with more number of packets in other words when attack flows start to look like legitimate flows. In order to achieve longer attack flows the number of packets sent in each flow and the traffic interval is increased gradually. Table 17 shows the parameters set in each scenario. In all cases the attack flow is run over 4 hosts all attacking a single victim in the network and ten attacks are tested for each attack traffic pattern.

10% Longer Flows		40% Longer Flows	
Packets Type	UDP	Packets Type	UDP
Packet Payload	-	Packet Payload	-
Number of Packets to be sent	2	Number of Packets to be sent	4
Traffic Interval	0.092	Traffic Interval	0.128
Traffic Rate in the Newtork (Packet /Sec)	43.48	Traffic Rate in the Newtork (Packet /Sec)	31.25
Flow Rate in the Newtork (Flow/sec)	21.74	Flow Rate in the Newtork (Flow/sec)	7.81
20% Longer Flows		**50% Longer Flows**	
Packets Type	UDP	Packets Type	UDP
Packet Payload	-	Packet Payload	-
Number of Packets to be sent	2	Number of Packets to be sent	4
Traffic Interval	0.104	Traffic Interval	0.14
Traffic Rate in the Newtork (Packet /Sec)	38.46	Traffic Rate in the Newtork (Packet /Sec)	28.57
Flow Rate in the Newtork (Flow/sec)	19.23	Flow Rate in the Newtork (Flow/sec)	7.14
30% Longer Flows		**60% Longer Flows**	
Packets Type	UDP	Packets Type	UDP
Packet Payload	-	Packet Payload	-
Number of Packets to be sent	3	Number of Packets to be sent	5
Traffic Interval	0.116	Traffic Interval	0.152
Traffic Rate in the Newtork (Packet /Sec)	34.48	Traffic Rate in the Newtork (Packet /Sec)	26.32
Flow Rate in the Newtork (Flow/sec)	11.49	Flow Rate in the Newtork (Flow/sec)	5.26

Table 17 Changing Attack Traffic Parameters

Looking at table 18, as the flows become longer some attacks are not reported. Again the algorithm proves to be quite reliable till a 30% margin where we start to see the first signs of false negative reports. The FN error rate shows an approximate 10% growth as the flows are longer by ten percent. The thresholds set in the algorithm play an important key in the algorithm tolerance level.

	10% Longer Flows	20% Longer Flows	30% Longer Flows	40% Longer Flows	50% Longer Flows	60% Longer Flows
Number of Attacks	10	10	10	10	10	10
Number of Not Reported Attacks	0	0	1	2	4	5
FN Error Probability	0%	0%	10.00%	20.00%	40.00%	50.00%

Table 18 FN Error Probability changes with Change of Attack Traffic Characteristics

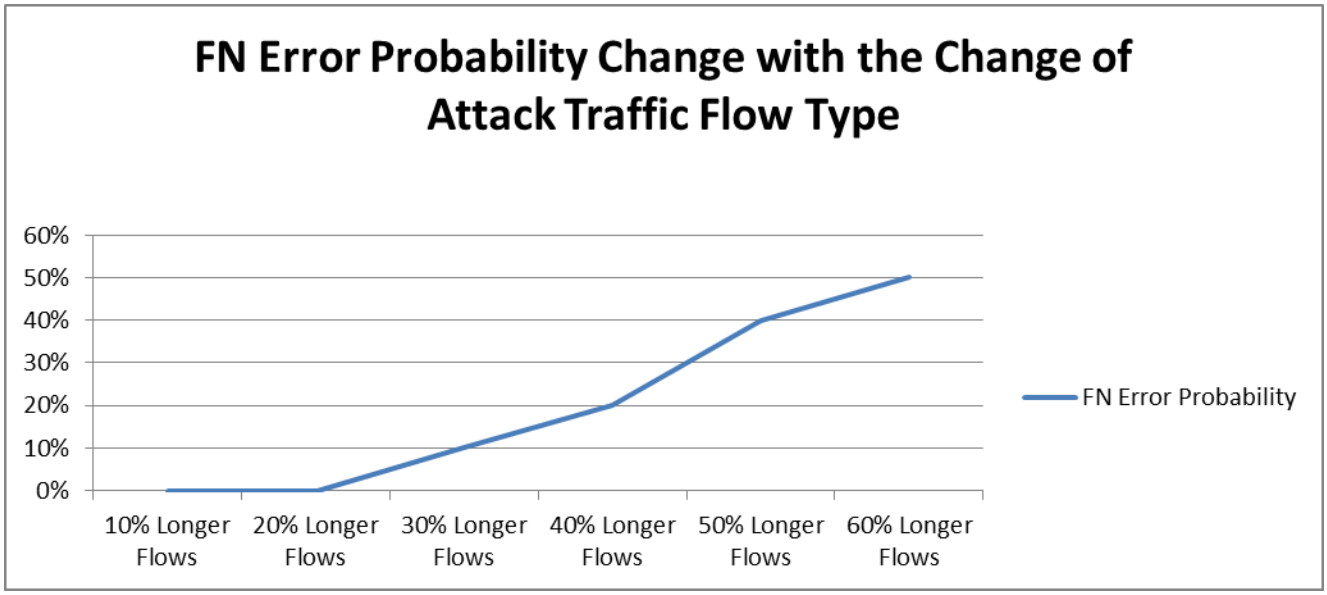

Figure 33 FN Error Probability Change with the Change of Attack Traffic Flow Type

4.3.4 Attack Mitigation Effectiveness

To test the effectiveness of the mitigation method the attack traffic is run in five simulation tests over 9, 10, 11, 12 and 13 hosts in the network. In the first attack scenario nine random hosts will run the attack traffic. In the second scenario ten random hosts run the attack traffic and so on. A single-victim attack is deployed in these tests. In the first set of tests the mitigation method is in place on the controller and as soon as the attack is detected the idle timer's timeout period is changed from the default value (15 sec) to mitigated value (11 sec) on the new flows. In the next set of tests the mitigation method will not be performed in the controller and the idle_timer's timeout period remains at 15 seconds before and after an attack detection. Table 19 illustrates test results.

	With Mitigation Method	Without Mitigation Method	Mitigation Effectiveness	
Number of Attackers	Time it takes to Congest the First Switch	Time it takes to Congest the First Switch	**sec**	**%**
9	3 min 38 sec = 218 sec	3 min 21 sec = 191 sec	19	8.72%
10	3 min 10 sec = 190 sec	2 min 50 sec = 170 sec	20	10.53%
11	2 min 40 sec = 160 sec	2 min 26 sec = 142 sec	18	11.25%
12	2 min 5 sec = 125 sec	1 min 38 sec = 98 sec	27	21.60%
13	1 min 14 sec = 74 sec	57 sec	17	22.97%
		AVG	20	15.01%

Table 19 Effectiveness of the Applied Mitigation Method

As we can see in table 19 on average the mitigation method is able to delay the congestion of the switch flow table by 15%. Although this number is calculated about an average of 20 seconds delay in our simulation runs but we could be sure that in real networks with larger idle_timers and greater timer changes the mitigation method could be much more effective.

Chapter 5

5 Conclusion and Future work

5.1 Conclusion

The initial objectives of this thesis were to propose a reliable and light weight solution for detecting a range DDoS attacks independent of their structure in its early stages. The types of DDOS attacks performed in SDN networks have a wider range. In traditional networks the attacker tries to bring down a service running in the network by congesting it with excessive traffic loads or to reduce the network speed by congesting the available bandwidths, therefore a huge load of attack traffic need to target a certain destination. In SDN this is not a required characteristic to make the attacks effective. The attack traffic could be distributed as much as possible to skip the detection mechanisms in place and still target the controller and switches. The detection method used in SDN must be able to detect both isolated and multiple victim attacks. The detection delay must be very short to provide enough time to establish a mitigation method.

With around 300 lines of coding added to the controller not only any mischievous activities will be detected but also the affected attack paths are identified. The high detection rates for different traffic patterns in our results show that the algorithm is able to perform well under different network conditions and it is not limited to a specific network condition. In a detailed study of the results obtained under traffic pattern “A” the algorithm shows to detect the Single victim attacks with an average delay of 14.86 seconds and False Negative probability/False Positive of less than 1.3%. Similarly under same traffic pattern multiple victim attacks are detected with an average delay of 20.97 seconds with False Negative / False Positive probability of less than 1%. Based on the results, it seems that the proposed algorithm has been successful in achieving the intended goals.

5.2 Future Work

As mentioned in section 4.3.1 when the traffic flows running in the network have different characteristics, using a single threshold margin could increase the chances of FP and FN reports. Hence it is advised to create a system that will use a range of thresholds that will each be used with their matching flow types when implementing the study of flow specifications. In order to implement this, the legitimate flows in the system first need to be distinguished and the appropriate thresholds must be calculated in relation to each flow group specifications.

Appendix

Controller Codes

```
# coding=utf-8
# 2012 James McCauley
#
# This file is part of POX.
#
# POX is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as
published by
# the Free Software Foundation, either version 3 of the License,
or
# (at your option) any later version.
#
# POX is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public
License
# along with POX.  If not, see <http://www.gnu.org/licenses/>.

"""
A shortest-path forwarding application.

This is a standalone L2 switch that learns ethernet addresses
across the entire network and picks short paths between them.

You shouldn't really write an application this way -- you should
keep more state in the controller (that is, your flow tables),
and/or you should make your topology more static.  However, this
does (mostly) work. :)

Depends on openflow.discovery
Works with openflow.spanning_tree
"""


from colors import red,green,yellow
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.recoco import Timer
from collections import defaultdict
from pox.openflow.discovery import Discovery
from pox.lib.util import dpid_to_str
import time
import datetime
from Mail_Handler import Mail
```

```
from pox.lib.util import dpidToStr
from pox.openflow.of_json import *

import math
import test_flow_stat3
log = core.getLogger()
pointer = 0
# Adjacency map.  [sw1][sw2] ->port from sw1 to sw2
adjacency = defaultdict(lambda:defaultdict(lambda:None))
# Switches we know of.  [dpid] -> Switch
switches = {}
# ethaddr -> (switch, port)
mac_map = {}
# [sw1][sw2] -> (distance, intermediate)
path_map = defaultdict(lambda:defaultdict(lambda:(None,None)))
# Waiting path.  (dpid,xid)->WaitingPath
waiting_paths = {}
# Time to not flood in seconds
FLOOD_HOLDDOWN = 5
# Flow timeouts
FLOW_IDLE_TIMEOUT = 15
FLOW_HARD_TIMEOUT = 30
# How long is allowable to set up a path?
PATH_SETUP_TIME = 4
#keeps track of the number of flows until we reach 50
my_counter = 0
#used as the starting point of the timer to calculate the time it
takes to receive 50 packets. used to calculate and monitor the
receiving rate
my_start = 0
#stores the IP destination for each flow so that the suspected
attack destination could be identified
ipList = []
#saves the path for detecting the switches in the attack path
when an attack is suspected
path_for_stat = []
#keeps track of the switches that a statistic request has been
sent to. This is to prevent sending duplicate requests.
sent_sw = []
#If the entropy change is repeated for 5 times in a row we
suspect an attack and poll the switches in the attack path to
send their statistics.
entropy_counter = 0
#The variable used as the entropy threshold.
Entc2 = 0
#The variable used as the receiving rate threshold.
frate_th = 20
frate2 = 0
attackswitch = {}
def _calc_paths ():
  """
  Essentially Floyd-Warshall algorithm
  """
```

```
  def dump ():
    for i in sws:
      for j in sws:
        a = path_map[i][j][0]
        #a = adjacency[i][j]
        if a is None: a = "*"
        print a,
      print
  sws = switches.values()
  path_map.clear()
  for k in sws:
    for j,port in adjacency[k].iteritems():
      if port is None: continue
      path_map[k][j] = (1,None)
    path_map[k][k] = (0,None) # distance, intermediate
  #dump()
  for k in sws:
    for i in sws:
      for j in sws:
        if path_map[i][k][0] is not None:
          if path_map[k][j][0] is not None:
            # i -> k -> j exists
            ikj_dist = path_map[i][k][0]+path_map[k][j][0]
            if path_map[i][j][0] is None or ikj_dist <
path_map[i][j][0]:
              # i -> k -> j is better than existing
              path_map[i][j] = (ikj_dist, k)
  #print "--------------------"
  #dump()

def _get_raw_path (src, dst):
  """
  Get a raw path (just a list of nodes to traverse)
 """
  if len(path_map) == 0: _calc_paths()
  if src is dst:
    # We're here!
    return []
  if path_map[src][dst][0] is None:
    return None
  intermediate = path_map[src][dst][1]
  if intermediate is None:
    # Directly connected
    return []
  return _get_raw_path(src, intermediate) + [intermediate] + \
         _get_raw_path(intermediate, dst)

def _check_path (p):
  """
  Make sure that a path is actually a string of nodes with
connected ports
  returns True if path is valid
  """
```

```
  for a,b in zip(p[:-1],p[1:]):
    if adjacency[a[0]][b[0]] != a[2]:
      return False
    if adjacency[b[0]][a[0]] != b[2]:
      return False
  return True

def _get_path (src, dst, first_port, final_port):
  """
  Gets a cooked path -- a list of (node,in_port,out_port)
  """
  # Start with a raw path...
  if src == dst:
    path = [src]
  else:
    path = _get_raw_path(src, dst)
    #keep a copy of the path to find the switches suspected of
being in the attack path
    path_for_stat = path
    if path is None: return None
    path = [src] + path + [dst]
  # Now add the ports
  r = []
  in_port = first_port
  for s1,s2 in zip(path[:-1],path[1:]):
    out_port = adjacency[s1][s2]
    r.append((s1,in_port,out_port))
    in_port = adjacency[s2][s1]
  r.append((dst,in_port,final_port))
  assert _check_path(r), "Illegal path!"
  return r

class WaitingPath (object):
  """
  A path which is waiting for its path to be established
  """
  def __init__ (self, path, packet):
    """
    xids is a sequence of (dpid,xid)
    first_switch is the DPID where the packet came from
    packet is something that can be sent in a packet_out
    """
    self.expires_at = time.time() + PATH_SETUP_TIME
    self.path = path
    self.first_switch = path[0][0].dpid
    self.xids = set()
    self.packet = packet
    if len(waiting_paths) > 1000:
      WaitingPath.expire_waiting_paths()
  def add_xid (self, dpid, xid):
    self.xids.add((dpid,xid))
    waiting_paths[(dpid,xid)] = self
```

```
  @property
  def is_expired (self):
    return time.time() >= self.expires_at
  def notify (self, event):
    """
    Called when a barrier has been received
    """
    self.xids.discard((event.dpid,event.xid))
    if len(self.xids) == 0:
      # Done!
      if self.packet:
        log.debug("Sending delayed packet out %s"
                  % (dpid_to_str(self.first_switch),))
        msg = of.ofp_packet_out(data=self.packet,
            action=of.ofp_action_output(port=of.OFPP_TABLE))
        core.openflow.sendToDPID(self.first_switch, msg)

      core.l2_multi.raiseEvent(PathInstalled(self.path))

  @staticmethod
  def expire_waiting_paths ():
    packets = set(waiting_paths.values())
    killed = 0
    for p in packets:
      if p.is_expired:
        killed += 1
        for entry in p.xids:
          waiting_paths.pop(entry, None)
    if killed:
      log.error("%i paths failed to install" % (killed,))

class PathInstalled (Event):
  """
  Fired when a path is installed
  """
  def __init__ (self, path):
    Event.__init__(self)
    self.path = path

class Cleanswitch (object):
  def __init__(self):
    pass
  def _do_remove(self):
    t = time.time()
    if len(attackswitch) == 0:
      print red('no switch under attack')
      return False
    else:
      for dpid, switch_ts in attackswitch.items():
        print green('checking the attack switch list')
        time_t = int(t - switch_ts)
        print time_t
        if time_t >= 12:
```

```
          print time_t
          del attackswitch[dpid]
          print 'Deleted switch from the attack list: %s', dpid
        if len(attackswitch) == 0:
          print green('Hurray We are Secured')
          return False
      return True

  def _do_sleep(self):
    r = self._do_remove()
    if r == False:
      core.callDelayed(12, self._do_sleep)
    else:
      s = min(attackswitch, key=attackswitch.get)
      sleep_time = 12 - int(time.time() - attackswitch[s])
      core.callDelayed(sleep_time, self._do_sleep)

class Switch (EventMixin):
  entDic = {} #Table for the IP address and its occurrence
  all_ip = {}
  dstEnt = [] #List of entropies
  count1 = 0
  start_time = 0
  end_time = 0
  ftimer = 0
  count3 = 0
  max_path = []
  Entth = 1
  Entc = 0

  @staticmethod
  def cleaning_sent_sw ():
    del sent_sw[:]
    print "deleting sent_sw"
  def statcolect(self, path_stat, element, element_src):
    global my_counter
    # my-counter : counts the number of packets. We collect 50
packets
    global ipList
    #my_start is used as the starting point for the timer.
    global my_start
    global entropy_counter
    global frate_th
    global frate2
    global Entc2
    print "Packet Counter:", my_counter
    #This function collects IP statistics
    ipList.append(element)
    #Increment until we reach 50
    if my_counter == 0:
      #we need to calculate the time it takes o collect 50
packets so we could use in calculating the rate
      self.start_time = time.time()
```

```
      my_start = self.start_time
      print "start time" ,my_start
    my_counter +=1
    #keep the path statistics so that we could find the switches
in the attack path when an attack is suspected
    if element in self.all_ip:
       self.all_ip[element].append(path_stat)
    else:
       self.all_ip[element]= (path_stat)
    if my_counter == 50:
      self.end_time = time.time()
      self.ftimer = self.end_time - my_start
      print "we reach 50 and our start_time %s end_time %s and
timer is %s" % (str(my_start), str(self.end_time),
str(self.ftimer))
      self.start_time = 0
      self.entDic = {}
      for i in ipList:
          if i not in self.entDic:
            self.entDic[i] =0
          self.entDic[i] +=1
      #print the hash table and clear all
      print self.entDic
      max_ip = max(self.entDic, key=self.entDic.get)
      print "max seen ip=", max_ip
      self.max_path = self.all_ip[max_ip]
      #call the entropy function
      self.Entc = self.entropy(self.entDic)
      print "Entc", self.Entc
      print "Entth", self.Entth
      #using math.floor to compare the integer part of the
entropies
      if math.floor(self.Entc) >= math.floor(self.Entth):
        frate = 50 / self.ftimer
        #frate2 is used to pass the receiving rate. frate is
reset before being passed so a new variable is defined to pass
the value
        frate2 = frate
        Entc2 = self.Entc
        print "frate2 is updated:",frate2
        if frate <= frate_th:
          print "Be happy frate<=frate_th frate= ",frate
          print "frate_th=",frate_th
          self.Entth Entc = self.
          print "Entth is updated to Entth=",self.Entth
          if frate >= 20:
            self.frate_th = frate
            print "frate_th is updated to",frate_th
          frate = 0
          entropy_counter = 0
          print "entropy_counter is reset",entropy_counter
          self.count1 = 0
          self.ftimer = 0
```

```
        else:
          self.count1 +=1
          print "frate=", frate
          print "frate_th=",frate_th
          print "count1=",self.count1
          #count1 is used to detect attacks using the receiving
rate of new flows. when count1 is 5 we suspect an attack.
          if self.count1 == 5:
            self.max_path = self.all_ip[max_ip]
            #eliminating duplicate paths
            self.max_path = sorted(self.max_path)
            dedup = [self.max_path[i] for i in
range(len(self.max_path)) if i == 0 or self.max_path[i] !=
self.max_path[i-1]]
            print "we suspect an attack because count1=5 so we
will go to test_flow_stat3"
            print ""
            dtm = datetime.datetime.now()
            msg = "we suspect an attack because counter1=5, we
will query switches" + " Time:" + str(dtm)
            with
open('/home/mininet/pox/pox/forwarding/logfile.txt','a+') as
flog:
               flog.write('\n')
               flog.write(msg)
               flog.write('\n')
            #although the duplicate paths are eliminated but
still a list of individual switches apear in the list. Since
these switches will be also in the switch path list we will not
consider them and will only look a the list type members of our
sitch path list.
            for raha in dedup:
              if type(raha) == type(list()):
                dtm = datetime.datetime.now()
                msg= "The switches suspected of being in the
Attack path are:" +str(raha) +" Time:" + str(dtm)
                with
open('/home/mininet/pox/pox/forwarding/logfile.txt','a+') as
flog:
                   flog.write(msg)
                   flog.write('\n')
                print "The switches suspected of being in the
Attack path are:", raha
                print ""
            for raha in dedup:
              if type(raha) == type(list()):
                  self.flow_stat(raha) #calling the flow_stat
function that will send request to all the switches in the Raha
list to send their flow tables to the controller
            self.count1 = 0
            self.ftimer = 0
            frate = 0
          else:
```

```
            self.ftimer = 0
            self.frate = 0
      else:
        self.count1 = 0
        self.ftimer = 0
        frate = 0
        entropy_counter +=1
        print "count3=",entropy_counter
        #The entropy changes continue for 5 times so we suspect
an attack.
        if entropy_counter == 5:
          self.max_path = self.all_ip[max_ip]
          self.max_path = sorted(self.max_path)
          dedup = [self.max_path[i] for i in
range(len(self.max_path)) if i == 0 or self.max_path[i] !=
self.max_path[i-1]] #deleting the duplicate paths
          dtm = datetime.datetime.now()
          msg = "we suspect an attack because entropy_counter=5,
we will query switches" + " Time:" + str(dtm)
          with
open('/home/mininet/pox/pox/forwarding/logfile.txt','a+') as
flog:
             flog.write('\n')
             flog.write(msg)
             flog.write('\n')
          print "we suspect an attack because entropy_counter=5
so we will go to test_flow_stat3"
          print ""
          for raha in dedup:
            if type(raha) == type(list()):
                dtm = datetime.datetime.now()
                msg= "The switches suspected of being in the
Attack path are:" +str(raha) +" Time:" + str(dtm)
                with
open('/home/mininet/pox/pox/forwarding/logfile.txt','a+') as
flog:
                   flog.write(msg)
                   flog.write('\n')
                print "The switches suspected of being in the
Attack path are:",raha
                print""
          for raha in dedup:
            if type(raha) == type(list()):
                self.flow_stat(raha)
          self.count1 = 0
          entropy_counter = 0
          self.ftimer = 0
          frate = 0
      self.entDic = {}
      ipList = []
      #l =0
      my_counter = 0
  def entropy (self, lists):
```

```
    #this function computes entropy
    #l = 50
    elist = []
    print lists.values()
    print sum(lists.values())
    for p in lists.values():
      print p
      c = float(p)/50
      print "c=",c
      elist.append(-c * math.log(c, 2))
    Ec = sum(elist)
    print 'Entropy = ',sum(elist)
    self.dstEnt.append(sum(elist))
    print len(self.dstEnt)
    return Ec
  # handler for timer function that sends the requests to the
switches in  the attack path that a request is not sent to them
in the last 10 seconds.
  def _timer_func (self, attack_p):
   sent_connection = 0
   for connection in core.openflow._connections.values():
       for item in attack_p:
         if dpidToStr(connection.dpid) == str(item[0]):
             if dpidToStr(connection.dpid) not in sent_sw:
                 print"sending flow request to switch",
dpidToStr(connection.dpid)
                 dtm = datetime.datetime.now()
                 msg= "Sending flow request to:"
+dpidToStr(connection.dpid) + " Time:" + str(dtm)
                 with
open('/home/mininet/pox/pox/forwarding/logfile.txt','a+') as
flog:
                   flog.write(msg)
                   flog.write('\n')
                 connection.send(of.ofp_stats_request(body=of.ofp
_flow_stats_request()))
                 sent_connection +=1
                 sent_sw.append(dpidToStr(connection.dpid))#the
sent_sw is the list used to prevent sending duplicate request for
statistics to same switch. This list is cleared every 10 sec.
   log.info("Sent %i flow stats request(s)", sent_connection)
   dtm = datetime.datetime.now()
   msg= "Sent Switches list:" +str(sent_sw) +" Time:" + str(dtm)
   with open('/home/mininet/pox/pox/forwarding/logfile.txt','a+')
as flog:
     flog.write(msg)
     flog.write('\n')
   print "sent switches",sent_sw
 #function used to analyze the flow tables received from
switches. Having too many short flows, flows with small number of
bytes or packets are considered as signs of attack.
  def _handle_flowstats_received (self, event):
    global frate2
```

```
    global Entc2
    global frate_th
    stats = flow_stats_to_list(event.stats)
    log.info("FlowStatsReceived from
%s",dpidToStr(event.connection.dpid))
    flowlist = []
    for flow in event.stats:
        flowlist.append({
        "table_id": flow.table_id,
        "duration_sec": flow.duration_sec,
        "duration_nsec": flow.duration_nsec,
        "idle_timeout": flow.idle_timeout,
        "hard_timeout": flow.hard_timeout,
        "packet_count": flow.packet_count,
        "byte_count": flow.byte_count,
        })
   # print flowlist
    count_flow = 1
    count_3 = 0
    for f in event.stats:
      count_2 = 0
      count_flow +=1
      if f.byte_count <20:
        count_2 +=1
      if f.packet_count <4:
        count_2 +=1
      if ((f.duration_sec*pow(10,9)) + f.duration_nsec)
<9999999999:
        count_2 +=1
      if count_2 >=2:
        count_3 +=1
    rate = (float(count_3)/count_flow) * 100
    log.info("on switch %s: we have count_3 %s count_flow %s with
a rate of %s percent",
      dpidToStr(event.connection.dpid), count_3, count_flow,
rate)
    if rate>87:
      dtm = datetime.datetime.now()
      print "WE HAVE AN ATTACK!!!"
      msg = "There is an attack at switch :" +
dpidToStr(event.connection.dpid) + "with  rate of:" + str(rate) +
" Time: " + str(dtm)
      attackswitch[dpidToStr(event.connection.dpid)] =
time.time()
      #sub = "Attack!!!"
      #m = Mail(msg, sub)
      #m.send_email()
      with
open('/home/mininet/pox/pox/forwarding/logfile.txt','a+') as
flog:
        flog.write(msg)
        flog.write('\n')
      frate_th = 20
```

```
Entth = 1# Since we have an attack the system is in alert status
and so the threshold values are reset.
      print "frate_th is updated to:",frate_th
    else:
      self.Entth = Entc2
      print "we didnt have an attack on switch %s so the Entth is
updated=",self.Entth
      print "frate2",frate2
      frate_th = frate2
      print "frate_th is updated to",frate_th
      dtm = datetime.datetime.now()
      msg= "We didn't have an attack on switch" +
dpidToStr(event.connection.dpid) + "rate=" +str(rate) + "and the
new entth=" +str(self.Entth) + "New frate_th=" +str(frate_th) +
"Time:" + str(dtm)
      with
open('/home/mininet/pox/pox/forwarding/logfile.txt','a+') as
flog:
        flog.write(msg)
        flog.write('\n')
  # main functiont to launch the module
  def flow_stat (self, attack):
   from pox.lib.recoco import Timer
   self._timer_func(attack)

  def __init__ (self):
    self.connection = None
    self.ports = None
    self.dpid = None
    self._listeners = None
    self._connected_at = None
  def __repr__ (self):
    return dpid_to_str(self.dpid)
  def _install (self, switch, in_port, out_port, match, buf =
None):
    if len(attackswitch) == 0:
      FLOW_IDLE_TIMEOUT = 15
    else:
      FLOW_IDLE_TIMEOUT = 11
    msg = of.ofp_flow_mod()
    msg.match = match
    msg.match.in_port = in_port
    msg.idle_timeout = FLOW_IDLE_TIMEOUT
    msg.hard_timeout = FLOW_HARD_TIMEOUT
    msg.actions.append(of.ofp_action_output(port = out_port))
    msg.buffer_id = buf
    switch.connection.send(msg)
  def _install_path (self, p, match, packet_in=None):
    wp = WaitingPath(p, packet_in)
    for sw,in_port,out_port in p:
      self._install(sw, in_port, out_port, match)
      msg = of.ofp_barrier_request()
```

```
      sw.connection.send(msg)
      wp.add_xid(sw.dpid,msg.xid)
  def install_path (self, dst_sw, last_port, match, event):
    """
    Attempts to install a path between this switch and some
destination
    """
    p = _get_path(self, dst_sw, event.port, last_port)
    if p is None:
      log.warning("Can't get from %s to %s", match.dl_src,
match.dl_dst)
      import pox.lib.packet as pkt
      if (match.dl_type == pkt.ethernet.IP_TYPE and
          event.parsed.find('ipv4')):
        # It's IP -- let's send a destination unreachable
        log.debug("Dest unreachable (%s -> %s)",
                  match.dl_src, match.dl_dst)

        from pox.lib.addresses import EthAddr
        e = pkt.ethernet()
        e.src = EthAddr(dpid_to_str(self.dpid)) #FIXME: Hmm...
        e.dst = match.dl_src
        e.type = e.IP_TYPE
        ipp = pkt.ipv4()
        ipp.protocol = ipp.ICMP_PROTOCOL
        ipp.srcip = match.nw_dst #FIXME: Ridiculous
        ipp.dstip = match.nw_src
        icmp = pkt.icmp()
        icmp.type = pkt.ICMP.TYPE_DEST_UNREACH
        icmp.code = pkt.ICMP.CODE_UNREACH_HOST
        orig_ip = event.parsed.find('ipv4')
        d = orig_ip.pack()
        d = d[:orig_ip.hl * 4 + 8]
        import struct
        d = struct.pack("!HH", 0,0) + d #FIXME: MTU
        icmp.payload = d
        ipp.payload = icmp
        e.payload = ipp
        msg = of.ofp_packet_out()
        msg.actions.append(of.ofp_action_output(port =
event.port))
        msg.data = e.pack()
        self.connection.send(msg)
      return
    log.debug("Installing path for %s -> %s %04x (%i hops)",
        match.dl_src, match.dl_dst, match.dl_type, len(p))
    print "maryam dest ip is" , match.nw_dst
    #calling the statcolect function when a new flow is to be
installed. This function collects statisctics to monitor the
network behavior to detect DDOS attacks.
    send_path = p
    tuple(send_path)
    self.statcolect(send_path, match.nw_dst, match.nw_src)
```

```
    # We have a path -- install it
    self._install_path(p, match, event.ofp)
    # Now reverse it and install it backwards
    # (we'll just assume that will work)
    p = [(sw,out_port,in_port) for sw,in_port,out_port in p]
    self._install_path(p, match.flip())

  def _handle_PacketIn (self, event):
    def flood ():
      """ Floods the packet """
      if self.is_holding_down:
        log.warning("Not flooding -- holddown active")
      msg = of.ofp_packet_out()
      # OFPP_FLOOD is optional; some switches may need OFPP_ALL
      msg.actions.append(of.ofp_action_output(port =
of.OFPP_FLOOD))
      msg.buffer_id = event.ofp.buffer_id
      msg.in_port = event.port
      self.connection.send(msg)
    def drop ():
      # Kill the buffer
      if event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out()
        msg.buffer_id = event.ofp.buffer_id
        event.ofp.buffer_id = None # Mark is dead
        msg.in_port = event.port
        self.connection.send(msg)
    packet = event.parsed
    loc = (self, event.port) # Place we saw this ethaddr
    oldloc = mac_map.get(packet.src) # Place we last saw this
ethaddr
    if packet.effective_ethertype == packet.LLDP_TYPE:
      drop()
      return
    if oldloc is None:
      if packet.src.is_multicast == False:
        mac_map[packet.src] = loc # Learn position for ethaddr
        log.debug("Learned %s at %s.%i", packet.src, loc[0],
loc[1])
    elif oldloc != loc:
      # ethaddr seen at different place!
      if loc[1] not in adjacency[loc[0]].values():
        # New place is another "plain" port (probably)
        log.debug("%s moved from %s.%i to %s.%i?", packet.src,
                  dpid_to_str(oldloc[0].connection.dpid),
oldloc[1],
                  dpid_to_str(   loc[0].connection.dpid),    loc[
1])
        if packet.src.is_multicast == False:
          mac_map[packet.src] = loc # Learn position for ethaddr
          log.debug("Learned %s at %s.%i", packet.src, loc[0],
loc[1])
      elif packet.dst.is_multicast == False:
```

```
        # New place is a switch-to-switch port!
        #TODO: This should be a flood.  It'd be nice if we
knew.  We could
        #      check if the port is in the spanning tree if it's
available.
        #      Or maybe we should flood more carefully?
        log.warning("Packet from %s arrived at %s.%i without
flow",
                    packet.src, dpid_to_str(self.dpid),
event.port)
        #drop()
        #return

    if packet.dst.is_multicast:
      log.debug("Flood multicast from %s", packet.src)
      flood()
    else:
      if packet.dst not in mac_map:
        log.debug("%s unknown -- flooding" % (packet.dst,))
        flood()
      else:
        dest = mac_map[packet.dst]
        match = of.ofp_match.from_packet(packet)
        self.install_path(dest[0], dest[1], match, event)
  def disconnect (self):
    if self.connection is not None:
      log.debug("Disconnect %s" % (self.connection,))
      self.connection.removeListeners(self._listeners)
      self.connection = None
      self._listeners = None
  def connect (self, connection):
    if self.dpid is None:
      self.dpid = connection.dpid
    assert self.dpid == connection.dpid
    if self.ports is None:
      self.ports = connection.features.ports
    self.disconnect()
    log.debug("Connect %s" % (connection,))
    self.connection = connection
    self._listeners = self.listenTo(connection)
    self._connected_at = time.time()
  @property
  def is_holding_down (self):
    if self._connected_at is None: return True
    if time.time() - self._connected_at > FLOOD_HOLDDOWN:
      return False
    return True
  def _handle_ConnectionDown (self, event):
    self.disconnect()

class l2_multi (EventMixin):
```

```
  _eventMixin_events = set([
    PathInstalled,
  ])
  def __init__ (self):
    # Listen to dependencies
    def startup ():
      core.openflow.addListeners(self, priority=0)
      core.openflow_discovery.addListeners(self)
    core.call_when_ready(startup,
('openflow','openflow_discovery'))
  def _handle_LinkEvent (self, event):
    def flip (link):
      return Discovery.Link(link[2],link[3], link[0],link[1])
    l = event.link
    sw1 = switches[l.dpid1]
    sw2 = switches[l.dpid2]
    # Invalidate all flows and path info.
    # For link adds, this makes sure that if a new link leads to
an
    # improved path, we use it.
    # For link removals, this makes sure that we don't use a
    # path that may have been broken.
    #NOTE: This could be radically improved! (e.g., not *ALL*
paths break)
    clear = of.ofp_flow_mod(command=of.OFPFC_DELETE)
    for sw in switches.itervalues():
      if sw.connection is None: continue
      sw.connection.send(clear)
    path_map.clear()
    if event.removed:
      # This link no longer okay
      if sw2 in adjacency[sw1]: del adjacency[sw1][sw2]
      if sw1 in adjacency[sw2]: del adjacency[sw2][sw1]
      # But maybe there's another way to connect these...
      for ll in core.openflow_discovery.adjacency:
        if ll.dpid1 == l.dpid1 and ll.dpid2 == l.dpid2:
          if flip(ll) in core.openflow_discovery.adjacency:
            # Yup, link goes both ways
            adjacency[sw1][sw2] = ll.port1
            adjacency[sw2][sw1] = ll.port2
            # Fixed -- new link chosen to connect these
            break
    else:
      # If we already consider these nodes connected, we can
      # ignore this link up.
      # Otherwise, we might be interested...
      if adjacency[sw1][sw2] is None:
        # These previously weren't connected.  If the link
        # exists in both directions, we consider them connected
now.
        if flip(l) in core.openflow_discovery.adjacency:
          # Yup, link goes both ways -- connected!
```

```
          adjacency[sw1][sw2] = l.port1
          adjacency[sw2][sw1] = l.port2
      # If we have learned a MAC on this port which we now know
to
      # be connected to a switch, unlearn it.
      bad_macs = set()
      for mac,(sw,port) in mac_map.iteritems():
        #print sw,sw1,port,l.port1
        if sw is sw1 and port == l.port1:
          if mac not in bad_macs:
            log.debug("Unlearned %s", mac)
            bad_macs.add(mac)
        if sw is sw2 and port == l.port2:
          if mac not in bad_macs:
            log.debug("Unlearned %s", mac)
            bad_macs.add(mac)
      for mac in bad_macs:
        del mac_map[mac]
  def _handle_ConnectionUp (self, event):
    sw = switches.get(event.dpid)
    if sw is None:
      # New switch
      sw = Switch()
      switches[event.dpid] = sw
      sw.connect(event.connection)
    else:
      sw.connect(event.connection)
  def _handle_BarrierIn (self, event):
    wp = waiting_paths.pop((event.dpid,event.xid), None)
    if not wp:
      #log.info("No waiting packet %s,%s", event.dpid, event.xid)
      return
    #log.debug("Notify waiting packet %s,%s", event.dpid,
event.xid)
    wp.notify(event)

def launch ():
  core.registerNew(l2_multi)
  core.registerNew(Cleanswitch)
  core.Cleanswitch._do_sleep()
  timeout = min(max(PATH_SETUP_TIME, 5) * 2, 15)
  Timer(timeout, WaitingPath.expire_waiting_paths,
recurring=True)
  print "will go to execute the timer for sent_sw"
  #we will call the cleaning_sent_sw function in the switch class
to erase the list of the switches that have been already polled
for statistics. As long as the switches are in the sent_sw list
no statistics request will be sent to them.
  Timer(10, Switch.cleaning_sent_sw, recurring=True)
  core.openflow.addListenerByName("FlowStatsReceived",
Switch()._handle_flowstats_received)
```

Normal Traffic Code

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

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

def sourceIPgen():
  #this function generates random IP addresses these values are
not valid for first octet of IP address
  not_valid = [10,127,254,255,1,2,169,172,192]

  first = randrange(1,256)

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

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

  return ip

# host IPs start with 10.0.0. the last value entered by user

def gendest(start, end):


  #this function randomly generates IP addresses of the hosts
based on
  #entered start and end values


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

  return ip

  #send the generated IPs

def main():
  start = 2
  end = 30
  #main method
  try:
      opts, args =
getopt.getopt(sys.argv[1:],'s:e:',['start=','end='])
```

```
  except getopt.GetoptError:
      sys.exit(2)
  for opt, arg in opts:
      if opt =='-s':
          start = int(arg)
      elif opt =='-e':
          end = int(arg)
  if start == '':
      sys.exit()
  if end == '':
      sys.exit()

  # open interface eth0 to send packets

  interface = popen('ifconfig | awk \'/eth0/ {print
$1}\'').read()
  # send normal traffic to the destination hosts

  for i in xrange(1000):

      # form the packet
      payload = "my name is maryam kia"
      packets = Ether()/IP(dst=gendest(start,
end),src=sourceIPgen())/UDP(dport=80,sport=2)/payload
      print(repr(packets))
      m = 0
      # send packet with the defined interval (seconds)
      while m <= 8:
          sendp(packets,iface=interface.rstrip(),inter=0.2)
          m +=1

  #main

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

Single Victim Attack Traffic Code

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

def sourceIPgen():

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

  first = randrange(1,256)

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

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

  print ip
  return ip

  #send the generated IPs

def main():

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

  # open interface eth0 to send packets

  interface = popen('ifconfig | awk \'/eth0/ {print
$1}\'').read()
  print (repr(interface))
  for i in xrange(0,2000):
      # form the packet
      packets =
Ether()/IP(dst=dstIP,src=sourceIPgen())/UDP(dport=dst_port,sport=
src_port)
      print(repr(packets))

      # send packet with the defined interval (seconds)

      sendp( packets, iface=interface.rstrip(), inter=0.08)
```

```
  #main

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

Mutiple Victim Attack Traffic Code

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

def sourceIPgen():

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

  first = randrange(1,256)

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

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

  print ip
  return ip

  #send the generated IPs

def main():

  #getting the ip address to send attack packets
  print "here"
  dstIP1 = sys.argv[1:]
  dstIP2 = sys.argv[1:]
  dstIP3 = sys.argv[1:]
  dstIP4 = sys.argv[1:]
  #print dstIP
  src_port = 80
  dst_port = 1

  # open interface eth0 to send packets

  interface = popen('ifconfig | awk \'/eth0/ {print
$1}\'').read()
  print (repr(interface))
  for i in xrange(0,2000):
      # form the packet
      packets =
Ether()/IP(dst=dstIP1,src=sourceIPgen())/UDP(dport=dst_port,sport
=src_port)
      print(repr(packets))
```

```
      packets =
Ether()/IP(dst=dstIP2,src=sourceIPgen())/UDP(dport=dst_port,sport
=src_port)
      print(repr(packets))
      packets =
Ether()/IP(dst=dstIP3,src=sourceIPgen())/UDP(dport=dst_port,sport
=src_port)
      print(repr(packets))
      packets =
Ether()/IP(dst=dstIP4,src=sourceIPgen())/UDP(dport=dst_port,sport
=src_port)
      print(repr(packets))


      # send packet with the defined interval (seconds)

      sendp( packets, iface=interface.rstrip(), inter=0.03)

  #main

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

Single Victim Attack Result Page

		%13 Attack Traffic		%28 Attack Traffic		%45 Attack Traffic		%63 Attack Traffic		TOTAL
log_fi1	Total Reports	198	198	117	117	162	162	162	162	639
	FN	2		1		6		2		11
	FP		1		2		0		0	3
	Percentage	1.010101	0.505051	0.854701	1.709402	3.703704	0	1.234568	0	
log_fi2	Total Reports	153	153	162	162	162	162	153	153	630
	FN	6		4		4		3		17
	FP		7		6		4		2	19
	Percentage	3.921569	4.575163	2.469136	3.703704	2.469136	2.469136	1.960784	1.30719	
log_fi3	Total Reports	279	279	171	171	180	180	171	171	801
	FN	18		2		10		3		33
	FP		0		0		0		0	0
	Percentage	6.451613	0	1.169591	0	5.555556	0	1.754386	0	
log_fi4	Total Reports	162	162	153	153	162	162	162	162	639
	FN	5		0		1		1		7
	FP		3		0		1		1	5
	Percentage	3.08642	1.851852	0	0	0.617284	0.617284	0.617284	0.617284	
log_fi5	Total Reports	153	153	153	153	153	153	153	153	612
	FN	2		0		2		1		5
	FP		0		0		0		0	0
	Percentage	1.30719	0	0	0	1.30719	0	0.653595	0	
log_fi6	Total Reports	162	162	153	153	153	153	153	153	621
	FN	7		3		1		2		13
	FP		0		3		1		1	5
	Percentage	4.320988	0	1.960784	1.960784	0.653595	0.653595	1.30719	0.653595	
log_fi7	Total Reports	180	180	153	153	144	144	153	153	630
	FN	3		1		3		2		9
	FP		0		0		0		0	0
	Percentage	1.666667	0	0.653595	0	2.083333	0	1.30719	0	
log_fi8	Total Reports	153	153	135	135	153	153	153	153	594
	FN	3		1		2		2		8
	FP		0		0		0		0	0
	Percentage	1.960784	0	0.740741	0	1.30719	0	1.30719	0	
log_fi9	Total Reports	135	135	144	144	153	153	162	162	594
	FN	3		0		1		1		5
	FP		0		0		0		0	0
	Percentage	2.222222	0	0	0	0.653595	0	0.617284	0	
log_fi10	Total Reports	153	153	153	153	153	153	153	153	612
	FN	0		1		0		1		2
	FP		0		1		0		0	1
	Percentage	0	0	0.653595	0.653595	0	0	0.653595	0	
log_fi11	Total Reports	153	153	162	162	153	153	153	153	621
	FN	0		1		2		2		5
	FP		0		1		0		0	1
	Percentage	0	0	0.617284	0.617284	1.30719	0	1.30719	0	
log_fi12	Total Reports	153	153	162	162	162	162	153	153	630
	FN	0		0		3		1		4
	FP		0		0		0		2	2
	Percentage	0	0	0	0	1.851852	0	0.653595	1.30719	
log_fi13	Total Reports	108	108	162	162	180	180	171	171	621
	FN	2		3		1		5		11
	FP		0		2		12		1	15
	Percentage	1.851852	0	1.851852	1.234568	0.555556	6.666667	2.923977	0.584795	
log_fi14	Total Reports	126	126	180	180	180	180	171	171	657
	FN	0		2		1		1		4
	FP		0		4		5		5	14
	Percentage	0	0	1.111111	2.222222	0.555556	2.777778	0.584795	2.923977	
log_fi15	Total Reports	135	135	153	153	144	144	162	162	594
	FN	1		1		1		3		6
	FP		0		3		0		2	5
	Percentage	0.740741	0	0.653595	1.960784	0.694444	0	1.851852	1.234568	
log_fi16	Total Reports	144	144	162	162	153	153	171	171	630
	FN	0		1		6		1		8
	FP		0		0		0		0	0
	Percentage	0	0	0.617284	0	3.921569	0	0.584795	0	
log_fi17	Total Reports	153	153	144	144	153	153	153	153	603
	FN	0		1		3		2		6
	FP		0		1		0		0	1
	Percentage	0	0	0.694444	0.694444	1.960784	0	1.30719	0	
log_fi18	Total Reports	144	144	144	144	153	153	162	162	603
	FN	2		3		0		0		5
	FP		0		0		2		0	2
	Percentage	1.388889	0	2.083333	0	0	1.30719	0	0	
log_fi19	Total Reports	126	126	153	153	153	153	144	144	576
	FN	1		2		1		2		6
	FP		0		0		0		0	0
	Percentage	0.793651	0	1.30719	0	0.653595	0	1.388889	0	
log_fi20	Total Reports	99	99	153	153	162	162	162	162	576
	FN	0		0		2		2		4
	FP		0		0		2		1	3
	Percentage	0	0	0	0	1.234568	1.234568	1.234568	0.617284	

Multiple Victims Attack Result Page

		26% Attack Traffic		42.5% Attack Traffic		54% Attack Traffic		TOTAL
log2_fi1	Total Reports	198	198	279	279	261	261	738
	FN	0		1		6		7
	FP		0		2		0	2
	Percentage	0	0	0.358423	0.716846	2.298851	0	
log2_fi2	Total Reports	288	288	270	270	243	243	801
	FN	0		1		9		10
	FP		3		0		1	4
	Percentage	0	1.041667	0.37037	0	3.703704	0.411523	
log2_fi3	Total Reports	270	270	324	324	216	216	810
	FN	0		1		0		1
	FP		4		0		0	0
	Percentage	0	1.481481	0.308642	0	0	0	
log2_fi4	Total Reports	144	144	162	162	162	162	468
	FN	0		1		3		4
	FP		0		6		0	6
	Percentage	0	0	0.617284	3.703704	1.851852	0	
log2_fi5	Total Reports	162	162	144	144	171	171	477
	FN	0		1		2		3
	FP		0		0		2	0
	Percentage	0	0	0.694444	0	1.169591	1.169591	
log2_fi6	Total Reports	108	108	162	162	117	117	387
	FN	0		1		4		5
	FP		1		2		0	0
	Percentage	0	0.925926	0.617284	1.234568	3.418803	0	
log2_fi7	Total Reports	126	126	225	225	189	189	540
	FN	0		0		0		0
	FP		1		0		0	0
	Percentage	0	0.793651	0	0	0	0	
log2_fi8	Total Reports	153	153	117	117	189	189	459
	FN	0		0		0		0
	FP		0		0		1	0
	Percentage	0	0	0	0	0	0.529101	
log2_fi9	Total Reports	171	171	126	126	135	135	432
	FN	0		0		1		1
	FP		0		0		1	0
	Percentage	0	0	0	0	0.740741	0.740741	
log2_fi10	Total Reports	126	126	207	207	180	180	513
	FN	0		1		0		1
	FP		1		1		3	0
	Percentage	0	0.793651	0.483092	0.483092	0	1.666667	

Bibliography

[1] A. Gerrity and F. Hu, Network Innovation through OpenFlow and SDN: Principles and Design (SDN/OpenFlow:Concepts and Applications), Taylor and Francis Group, 2014.

[2] R. Kl¨oti, "OpenFlow: A security analysis," in *21st IEEE International Conference on Network Protocols (ICNP)*, 2013.

[3] M. S. Malik, M. Montanari, J. H. Huh, R. B. Bobba and R. H. Campbell, "Towards SDN Enabled Network Control Delegation in Clouds," *IEEE,* 2013.

[4] Sezer, Sakir; Scott-Hayward, Sandra; Chouhan, Pushpinder Kaur; Fraser, Barbara; Lake, David; Finnegan, Jim; Viljoen, Niel; Miller, Marc; Rao, Navneet;, "Are we Ready for SDN? Implementation Challenges for Software-Defined Networks," *IEEE Communication Magazine,* pp. 36-43, July 2013.

[5] N. Feamsterr, J. Rexford and E. Zegura, *The Road to SDN,* USA: acmqueue, 2013.

[6] K. Vikramjeet, Aanalysis of OpenFlow Protocol in Local Area Networks - Master of Science Thesis, Tampere , Finland: Tampere University of Technology, 2013.

[7] Sandra Scott-Hayward , Gemma O'Callaghan , Sakir Sezer, "SDN Security: A Survey," in *IEEE SDN for Future Networks and Services (SDN4FNS)*, Trento , Italy, 2013.

[8] "openflow-spec-v1.3.0," Open Networking Foundation, 2012.

[9] S. Sezer, S. Scott-Hayward and P. K. Chouhan, "Are We Ready for SDN? Implementation Challenges for Software-Defined Networks," *IEEE Communication Magazine,* pp. 36-43, July 2013.

[10] M. Vahlenkamp, "Design and Implementation of a Software-Defined Approach to Information-Centric Networking," Hamburg University of Applied Science, Germany, 2013.

[11] L. R. Prete, C. M. Schweitzer, A. A. Shinoda and R. L. Santos de Oliveira, "Simulation in an SDN network scenario using the POX Controller," *IEEE,* 2014.

[12] M. B. C. Dillon, "OpenFlow DDoS Mitigation," Amsterdam, 2014.

[13] J. Naous, D. Erickson, G. Convington, G. Apenzeller and N. McKeown, "Implementing an OpenFlow Switch on the NetFPGA Platform," in *ACM/IEEE*

Symposium on Architecture for Networking and Communications Systems, San Jose, CA, USA, 2008.

[14] "OpenFlow V1.0," OpenFlow, 2009.

[15] S. Lim, J. Ha, Y. Kim and S. Yang, "ASDN-Oriented DDOS Blocking Scheme for Botnet-Based Attacks," *IEEE,* pp. 63-68, 2014.

[16] "Production Quality, Multilayer Open Virtual Switch," 2014. [Online]. Available: http://openvswitch.org/. [Accessed 22 June 2015].

[17] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *IEEE Communications Surveys & Tutorials,* vol. 16, no. 3, pp. 1617-1634, 2014.

[18] L. R. Prete, C. M. Schweitzer, A. A. Shinoda and R. L. Santos de Oliveira, "Simulation in an SDN network scenario using the POX Controller," *IEEE,* 2014.

[19] C. Douligeris and A. Mitrokotsa, "DDOS attacks and defense mechanisms: classification and state-of-the-art," *Computer Networks 44,* pp. 643-666, 2004.

[20] L. Garber, "Denial-of-service attacks rip the Internet," *IEEE Computer Society,* vol. 33, no. 04, pp. 12-17, 2000.

[21] U. Tariq, M. Hong and K.-S. Lhee, "A Comprehensive Categorization of DDoS Attack and DDoS Defense Techniques," in *Advanced Data Mining and Applications*, Springer Berlin Heidelberg, 2006, pp. 1025-1036 .

[22] W. M. Eddy, "Defenses Against TCP SYN Flooding Attacks," *The Internet Protocol Journal ,* vol. 9, no. 4, 2006.

[23] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," *ISOC Network and Distributed System Security Symposium,* 2013.

[24] T. Xing, D. Huang, L. Xu, C.-J. Chung and P. Khatkar, "SnortFlow:A OpenFlow-based Intrusion Prevention System in Cloud Environment," *IEEE,* pp. 89-92, 2013.

[25] D. Kreutz, F. M. V. Ramos and P. Verissimo, "Towards Secure and Dependable

Software-Defined Networks," LaSIGE/FCUL, University of Lisbon, Lisbon, Portugal, 2008.

[26] R. Braga and E. Mota, "Lightweight DDOS Flooding Attack Detection using NOX/OpenFlow," in *2010 IEEE 35th Conference on Local Computer Networks (LCN)*, Denver, CO, 2010.

[27] T. Yuzawa, "OpenFlow 1.0 Actual Use-Case: RTBH of DDoS Traffic While Keeping the Target Online," April 20113. [Online]. Available: http://packetpushers.net/openflow-1-0-actual-use-case-rtbh-of-ddos-traffic-while-keeping-the-target-online/. [Accessed 22 June 2015].

[28] C. Y. Hunag, T. MinChi, C. YaoTing, C. YuChieh and C. YanRen, "A Novel Design for Future On-DemandService and Security," *IEEE,* pp. 385-388, 2010.

[29] S. A. Mehdi, J. Khalid and S. A. Khayam, Revisiting Traffic Anomaly Detection Using Software Defined Networking, Springer Berlin Heidelberg, 2011.

[30] S. M. Mousavi, *Early Detection of DDOS Attacks in Software Defined Networks Controller,* Ottawa, Canada: Carleton University, 2014.

[31] L. Li, J. Zhou and N. Xiao, "DDOS Attack Detection Algorithm Based on Entropy Computing," *ICICS 2007,* pp. 452-466, 2007.

[32] T. Nakashima, S. T. and S. Oshima, "Early DoS/DDoS Detection Method using Short-term Statistics," in *2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, Krakow , 2010.

[33] P. Biondi, "Scapy," 2003. [Online]. Available: http://www.secdev.org/conf/scapy_lsm2003.pdf.

[34] M. Team, "Mininet," Octopress , 2015. [Online]. Available: http://mininet.org/. [Accessed 07 July 2015].

[35] secdev.org, "Scapy," [Online]. Available: http://www.secdev.org/projects/scapy/. [Accessed 22 June 2015].

[36] D. Erickson, "The beacon openflow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* , New York, USA, 2013.

[37] L. Foundation, "OpenDayLight," 2015. [Online]. Available: http://www.opendaylight.org/. [Accessed 22 June 2015].

[38] nox, "NOXRepo.org," [Online]. Available: http://www.noxrepo.org/site/about/. [Accessed 22 June 2015].