AN APPROACH FOR DETECTION OF BLACK HOLE ATTACK IN SOFTWARE DEFINED NETWORKS

By

Kinza Saeed

A thesis submitted to the faculty of Information Security Department, Military College of Signals, National University of Sciences and Technology, Rawalpindi in partial fulfillment of the requirements for the degree of MS in Information Security

August 2021

<u>**THESIS ACCEPTANCE CERTIFICATE**</u>

Certified that final copy of MS Thesis written by **<u>Ms. Kinza Saeed</u>**, Registration No. **<u>00000203961</u>**, of **<u>Military College of Signals</u>** has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations/MS Policy, is free of plagiarism, errors, and mistakes and is accepted as partial fulfillment for award of MS degree. It is further certified that necessary amendments as pointed out by GEC members and local evaluators of the scholar have also been incorporated in the said thesis.

Signature: ________________________________

Name of Supervisor: ________________________

Date: ____________________________________

Signature (HOD): ___________________________

Date: ____________________________________

Signature (Dean/Principal) ___________________

Date: ____________________________________

Declaration

I hereby declare that no portion of work presented in this thesis has been submitted in support of another award or qualification either at this institution or elsewhere

Dedication

This thesis is dedicated to MY FAMILY, TEACHERS AND FRIENDS for their love, endless support and encouragement

Acknowledgement

It would not have been possible to write this thesis without the help and support of kind and cooperative people around me, some of whom deserve to be given particular mention here.

First and foremost, I want to express my gratitude to my supervisor, Asst Professor Bilal Rauf, for providing me with expert guidance, support, and patience throughout my master's research. His vast knowledge and experience have been a source of inspiration for me in every phase of the research.

I am forever indebted to my committee member Asst Proffessor Muhammad Waseem Iqbal for his immense support, encouragement and motivation throughout my academic career at Military College of Signals,NUST.

I would like to thank Asst Professor Dr Yawar Abbas for clearing my concepts about working of Software Defined Networking.

Words are not enough to appreciate the continuous support and motivation given by my colleague Lt Hafiz M. Shafeeq during challenging times of my thesis. I am extremely grateful to my colleague Mr Abdul Rehman Janjua for always being ready to help me in coding and software use.

I wish to acknowledge the cooperation of Department Head clerk, Hadayat Khattak for assistance in my thesis documentation formalities especially during my stay at Karachi.

I am forever grateful to my parents and my brothers for their encouragement and support in difficult times. Last but not the least would like to appreciate myself for keeping up the faith in hard times.

Abstract

Software Defined Network(SDN) is a novel networking architecture based on separation of data and control plane. SDN enables the controller to have a logically centralized view of the complete network [1]. It allows routing applications that run on top of the control plane to discover the best routes and to manage and design traffic flow efficiently. To do so, the controller must first know the whole SDN infrastructure's networking topology in order to attain centralized control and visibility. However, topology information of the network can be manipulated by an attacker to carry out black hole attack [2] by dropping or steering all the traffic passing through it towards itself and use the information in the packets to serve as a launching pad to carry out further lethal attacks. Therefore, it is critical to detect the attack at an earlier stage and isolate the malicious/compromised black hole node. Hence, we propose a dynamic routing framework that finds routing paths based on the behavior of hosts and then chooses the best path considering past behavior of hosts. It helps in reducing probability of attacks and multi-hop communication between hosts to confuse attackers and expand exploration space for carrying out targeted attack. Furthermore, our framework detects black hole attack from malicious node by continuously analyzing the traffic statistics on nodes so that the attack can be detected and prevented nearest to the malicious host (from where it originates) and dynamically reconfigures route after isolating the malicious node. Our simulations were performed using mininet emulator and RYU controller. Throughput, packet delivery ratio and end to end delay are recorded periodically and whenever they fall out of threshold boundaries an alert is generated and malicious node is removed from the routing path. Results show that the values of network parameters resume to normal shortly after our detection and mitigation of attack.

Contents

1 Introduction **2**

1.1 Motivation . . . 3

1.2 Scope and Objectives . . . 4

1.3 Definition and benefits of SDN . . . 4

1.3.1 Definition of SDN . . . 4

1.3.2 Benefits of SDN . . . 6

1.4 Contribution . . . 7

2 Theoretical Background and Related Work **8**

2.1 Software Defined Network Architecture . . . 8

2.1.1 Forwarding Plane . . . 8

2.1.2 Control Plane . . . 10

2.1.3 Application Plane . . . 12

2.1.4 Southbound Interfaces . . . 12

2.1.5 Northbound Interface . . . 13

2.1.6 Westbound/Eastbound Interface . . . 14

2.2 Communication Protocol . . . 14

2.2.1 Openflow 14

2.2.2 OpenFlow Architecture 15

2.2.3 OpenFlow compliant switch 16

2.2.4 Flow Tables 17

2.2.5 Matching flow 17

2.2.6 Actions taken on the flows 18

2.2.7 Counters 19

2.2.8 Flow types 19

2.2.9 Packet forwarding mechanism 21

2.2.10 OpenFlow communication messages 22

2.2.11 Explanation of messages exchanged in OpenFlow network . . . 24

2.3 Network Protocols 26

2.3.1 Adress Resolution Protocol 27

2.3.2 Link Layer Discovery Protocol(LLDP) 27

2.4 Vulnerabilities of Network Protocols and Black Hole attack 28

2.5 Related Work 31

3 Proposed Methodology 37

3.1 Methodology 38

3.1.1 Algorithm 38

3.2 System Architecture 40

3.3 Implementation 40

3.3.1 Control Layer 40

3.4 Simulation Environment 41

3.4.1 System Specifications 41
3.5 Simulation Structure 42
3.5.1 Software Tools Used in Experiments 42
3.5.2 Simulation Topology 44
3.6 Mininet Topology 44
3.6.1 Mininet Topology Code 44
3.7 Detection Metrics 45
3.7.1 Throughput 45
3.7.2 End to End Delay 46
3.7.3 Packet Delivery Ratio 46
3.8 Anomaly Detection 46

4 Implementation 48

4.1 Proposed Methodology 48
4.1.1 Algorithm 48
4.1.2 Detection Parameters 49
4.2 Simulation Environment 49
4.2.1 System Specifications 50
4.2.2 Software Tools Used For Implementation and Experiments . . . 50
4.2.3 Simulation Topology 52
4.3 Implementation Modules 52
4.3.1 Controller 53
4.4 Implementation Steps(Normal Mode) 56
4.5 Results and Analysis 58

4.5.1 Graphs and Values of Detection Parameters under Normal Operation 58
4.6 Discussion of Results 60
4.6.1 Graph for PDR 60
4.6.2 Graph for End-to-End Delay 61
4.6.3 Graph for throughput 61

5 Conclusion **73**

References **75**

A Source Codes **80**

List of Figures

2.1 Fields in a flow entry 9

2.2 Packet forwarding method in SDN 9

2.3 SDN Architecture 13

2.4 Openflow Network Architecture 16

2.5 Basic Packet Process mechanism for Openflow switch 17

2.6 Basic Packet Process mechanism for Openflow switch 22

2.7 Network Topology using mininet 24

2.8 Communication Messages between switch and controller 25

2.9 Ping process between h1 and h2 26

2.10 Network Topology Discovery in SDN using LLDP 28

3.1 Architecture of our proposed model 41

3.2 Command Run on MiniEdit GUI 43

3.3 Command To Start Ryu Controller 43

3.4 Screenshot of Simulation Topology Script 45

3.5 Topology in Miniedit 46

3.6 Mininet Topology Code Snippet 47

4.1 Command Run on MiniEdit GUI 51
4.2 Command To Start RYU Controller 51
4.3 RYU Controller started 51
4.4 Simulation Topology 1 in MiniEdit 53
4.5 Simulation Topology 2 in MiniEdit 53
4.6 Simulation Topology 3 in MiniEdit 54
4.7 Simulation Topology 4 in MiniEdit 54
4.8 Simulation Topology 5 in MiniEdit 55
4.9 Simulation Topology 6 in MiniEdit 55
4.10 Simulation Topology 7 in MiniEdit 56
4.11 Simulation Topology 8 in MiniEdit 56
4.12 Flow Management in Our Implementation 57
4.13 Host h9 requesting connection to host h1 57
4.14 Controller running 62
4.15 Running mininet topology script 62
4.16 Terminals for all hosts in topology 63
4.17 Terminal of n-1 host in ten host topology 63
4.18 Entering destination details in terminal of n-1 host in ten host topology . 63
4.19 Entering destination details in terminal of n-1 host in ten host topology . 64
4.20 Path selected in our test example 64
4.21 Message traversing from source to destination through multiple hops . . 64
4.22 Entering destination details in terminal of n-1 host in ten host topology . 64
4.23 PDR of 3 host topology 64

4.24 Throughput of 3 host topology 65

4.25 End to End Delay of 3 host topology 65

4.26 Values of 3 host topology 65

4.27 PDR of 4 host topology 65

4.28 Throughput of 3 host topology 65

4.29 End to End Delay of 3 host topology 66

4.30 Values of 4 host topology 66

4.31 Throughput of 5 host topology 66

4.32 Throughput of 5 host topology 66

4.33 End to End Delay of 5 host topology 67

4.34 Values of 5 host topology 67

4.35 Throughput of 6 host topology 67

4.36 Throughput of 6 host topology 67

4.37 End to End Delay of 6 host topology 68

4.38 Values of 6 host topology 68

4.39 PDR of 7 host topology 68

4.40 Throughput of 7 host topology 68

4.41 End to End Delay of 7 host topology 69

4.42 Values of 7 host topology 69

4.43 Throughput of 8 host topology 69

4.44 Throughput of 8 host topology 69

4.45 End to End Delay of 8 host topology 70

4.46 Values of 8 host topology 70

4.47 PDR of 9 host topology 70

4.48 Throughput of 9 host topology 70

4.49 End to End Delay of 9 host topology 71

4.50 Values of 9 host topology 71

4.51 PDR of 10 host topology 71

4.52 Throughput of 10 host topology 71

4.53 End to End Delay of 10 host topology 72

4.54 Values of 10 host topology 72

List of Tables

1.1 SDN vs Traditional Networks 6

2.1 Fields in a flow table for OpenFlow 16

2.2 List of virtual ports for the “Required” forward action 18

2.3 List of virtual ports for the “Optional” forward action 19

2.4 Required list of counters for use in statistics messages 20

3.1 System Specifications 42

3.2 Software Tools Used For Implementation and Experiments 42

4.1 Software Tools in Implementation and Experiments 50

List of Abbreviations and Symbols

Abbreviations

SDN	Software Defined Networks
IP	Internet Protocol
DOS	Denial of Service
DDOS	Distributed Denial of Service
ONF	Open Networking Foundation
TLS	Transport Layer Security
BGP	Border Gateway Protocol
OVSDB	Open vSwitch Database
ROFL	revised open-flow library
HAL	Hardware abstraction layer (HAL),
PAD	Programmable Abstraction of Data path
NAT	Network Address Translation
ARP	Address Resolution Protocol
OSI	Open Standards International

IGMP	Internet Group Management Protocol
LLDP	Link Layer Discovery Protocol
DCN	Data Center Networks
VM	Virtual Machine
OS	Operating System
ICMP	Internet Control Message Protocol
PDR	Packet Delivery Ratio

Chapter 1

Introduction

Computer networks are playing a pivotal role in the modern day life because of the variety of services that internet provides. As a result of this there is an ongoing and rapid evolution as well as integration of computing technologies. This necessitates the connectivity of a wide range of devices, including mobile phones, personal computers, handheld computers, client machines, and server machines, all of which operate on a variety of operating systems and work using different protocols. The consistent integration of so many gadgets and entities creates a slew of problems. Establishment of secure communication between these different service infrastructure involve significant issues for example how to implement management tasks of these networks is a big issue. In traditional networks, the control logic is also present on each data plane, and they share information about how packets are handled on the data plane device, making data plane setup difficult. An overall picture of the network is not available in traditional networks because there is no centralization of control plane is, and hence can't create centralized and intelligent networking and management decisions. Operators must devote a significant amount of time and effort to configure the control plane since it is spread across devices and lacks a global picture of the network, preventing it from making good network-wide decisions. To configure each device separately, operators must expend a significant amount of effort and time. It's still a work in progress to figure out how to administer these networks while maintaining data security and reliability [3]. Fortunately, the management of networks which was a very difficult task in the traditional networks has become quite easy thanks to the software defined networks(SDN) through control plane centralization. However, where on one and this centralization of control plane has brought a number of benefits it has also opened doors to new vulnerabilities and attacks. One such attack is black ole attack were a malicious/compromised user drops or sniffs off packets leading to a black hole situation were the availability of services is affected. Since SDN is a novel paradigm, the attacks on SDN also requires new detection schemes and countermeasures. Hence in this thesis we will discuss a new scheme for detection of black hole attack in SDN.

1.1 Motivation

The SDN technology brings a number of benefits compared to traditional networks, as will be discussed in detail in subsequent sections. SDN technology is encouraging data centers and businesses to adopt it because of the benefits it provides. Open Networking Foundation (ONF) formulators created it for the growth in specifications and promotion of the use of SDN[4]. This rapid shift demands security of this technology at the same rate. Security of data when sending it over networks is extremely crucial. Security of data does not only mean maintaining privacy but also security and availability. The control of flow of information on devices of forwarding plane is performed by controller but there are some limitations. Openflow is the de-facto SDN protocol and makes adoption of TLS optional and complex. Authentication of users involve overhead. The malicious node problem is considered less in research. Moreover, topology information can be spoofed affecting other dependent services like routing resulting in black hole attack where all the data passing through a node is dropped resulting in reduction of efficiency of communication, increasing packet loss and exhaustion of computing resources. This data can be discarded by the node or extracted by an attacker node with malicious intent to incur further security breach attacks in the network causing loss of confidentiality and availability. Hence establishment of safe and reliable routes to black hole attack to take full advantage of the benefits provided by SDN.

1.2 Scope and Objectives

This thesis concentrates on the black hole attack arising from malicious/compromised hosts/end devices. Application plane, northbound interface and black-hole attacks arising from compromised switches are not within the scope of this thesis. Finally, a simple scheme is proposed to detect black-hole attacks arising from malicious/compromised hosts/end devices.

1.3 Definition and benefits of SDN

1.3.1 Definition of SDN

The focus of Open Networking Foundation (ONF) [5] is centered on the research, standardization, and marketing of Software-defined Network(SDN). SDN is a novel networking technology wit idea of a separate control plane and a separate data plane and enables control plane to be directly programmed by the application plane, according to the ONF [6][7]. SDN is based on two ideas, according to this definition: control planes decoupled and separated from data plane, second; control plane programmed through applications. However, these two SDN elements aren't wholly novel in network architecture.

The authors of [8] conducted a comprehensive review of previous attempts, revealing that a number of past studies investigating network flammability had been conducted. SwitchWare [9], for example, uses active networking, which lets packets to dynamically adjust activities while traversing via the network. In similar manner, Click [10], XORP [11], Quagga [12], and BIRD [13], for example, made an effort to construct extended software based forwarding devices by allowing network components adding programmability to them. The behaviour of these network devices can be altered by using new routing software or by altering existing routing software.

Furthermore, the idea of separating the control and data planes has been investigated throughout the previous decade. [13] replaces the idea of inter-domain Border Gateway Protocol routing with centralised routing control. They believe that this method makes truly dispersed path computing easier. At the time, the IETF suggested the Forwarding and Control Element Separation (ForCES) structure. They separate the aspects of control and packet routing.

[14] offered a four-dimensional strategy. This technique is based on a four-plane network architecture paradigm. The planes are referred to as "decision," "dissemination," "discovery," and "data." Ethane work was introduced in 2007 [15], and it is based on centralized controller and Ethernet switches tat are flow based. The processing of flows can be performed this way.

The SDN concept's distinctiveness stems from the fact that it allows for programmability by decoupling the control and data planes - instead of making networking devices more complicated, SDN provides simple programmable network devices. Furthermore, in the network design, SDN allows for decoupled forwarding plane and control plane. This architecture allows for separately perform control of network at control layer without disturbing the flows of data. The network's intelligence can be taken to the controller from the switches . Meanwhile, software may control the switches without the need for on-board intelligence. Separating the control and data planes allows for a more customizable environment as well as more freedom for developers of external applications to determine a network's behaviour.

1.3.2 Benefits of SDN

SDN gives administrators dynamic control over network configurations. The primary goal of SDN is to centralise control away from network nodes and toward the controller. The server's control logic enables easy alteration of program, the protocols, and the applications, thereby improving the utilization of the resources, moreover network becomes less complicated, and total revenue rises. Controller provides a holistic picture of network because of centralized nature of controller. By utilizing clear picture, it can improve process of management of flows and offer flexibility, high speed and scalability [2]. The benefits of using SDN over traditional networks are listed in the table below.

Table 1.1: SDN vs Traditional Networks

Criteria	Traditional Networks	Software Defined Networks
Network management	Hard; devices configured individually	Easier with central Programmability
Global view	Hard	Central View at controller
Maintenance Cost	High	Low
Error processing and update time	Sometimes months	Quite Easy with Software
Attack detection and mitigation	Difficult	Easier and Quick
Controller and applications authenticity	Trivial	Non-trivial
Forwarding table Integrity and network state	Important	Important
Controller's Availablity	Irrelevant	Significant
Resources Utilization	Low	High

1.4 Contribution

The contributions of this thesis are summarized below:

• Proposed a scheme for traffic management in SDN to resist black hole attack. Instead of static routes, this scheme is based on multi-hops between hosts from source to destination. Multi-hops between hosts confuses the attacker and expands the exploration space for attacker to carry out black hole attack.

• Extended the controller's functionality to meet requirements of this thesis by adding an information gathering module, a path calculation module and a flow management module.

• Performed simulations of the proposed scheme in mininet containing RYU controller under normal traffic and also generated black hole attack.

• Evaluated performance parameters like throughput, packet delivery ratio and end to end delay to detect black hole attack. After detecting black hole attack, malicious node that caused black hole attack was isolated and new route was constructed for sending data between source and destination.

Chapter 2

Theoretical Background and Related Work

2.1 Software Defined Network Architecture

There is separated and decoupled data plane and control plane in an SDN network design, thereby rendering forwarding devices "dumb devices," and total networking logic is contained in centralised control plane that is programmed by software applications [17]. The SDN architecture is made up of three layers:

2.1.1 Forwarding Plane

The primary function of these forwarding plane devices is to route incoming packets to their intended destinations using a centralised and programmable control plane's routing policy. Each switch, in the form of a forwarding table, maintains this routing strategy. Flow rules [1] are what the items in the forwarding table are [1]. Each flow is made up of three fields, as shown in 2.1: a header field, a counter field, and an action field. The packet forwarding method is depicted in 2.2 as follows: When a packet reaches on switch, it examines its forwarding table for a rule with a pattern field that matches the header value of the packet, such as the (Ethernet port number, the source IP address, VLAN tag, destination Ethernet or IP port). The value of the counter field is incremented when a matching rule is found, and the action field associated with the rule is executed. The actions are shown in 2.1.

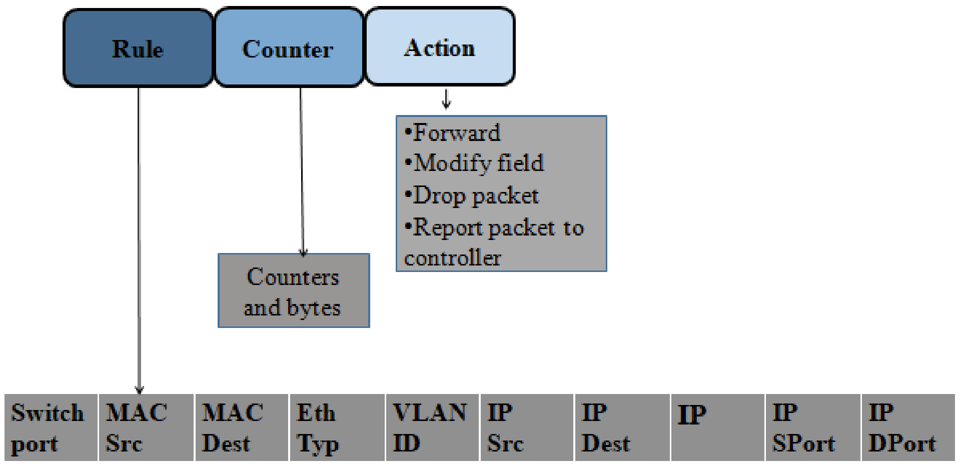


Figure 2.1: Fields in a flow entry

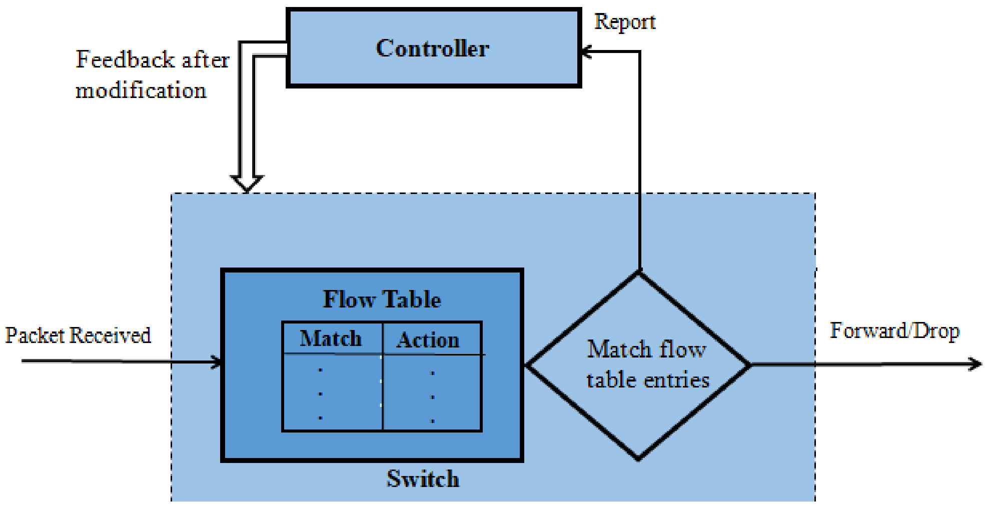


Figure 2.2: Packet forwarding method in SDN

2.1.2 Control Plane

Single/multiple controllers compose control plane, which formulates policies to regulate and operate network traffic. As a result, the control plane can be considered the "the brain of SDN network". The controller's primary job should be to provide a centralised means of managing networks and exchanging the information between the data layer and the networking applications that program the network. Dynamic access and administration are enabled by moving the control plane to software. Without having to touch individual switches or change their rules or configuration, a network administrator can shape traffic from a centralised management console [19]. On the market, there are more than twenty SDN controllers. The open source and active controllers listed below are some of the best.

Floodlight

Of all SDN controllers one of for the OpenFlow protocol is Project Floodlight. It is written in Java and is an enterprise class controller. It has support module-based apps and the RESTful ones. The former apps are Java applications that are compiled with the controller. This API can be used to get information from the controller and to send information regarding route to the controller. In terms of contact with the controller, the later API is restricted in comparision to the former API with regards to contact wit controller, but By detaching from the controller via TCP/IP communication, hazardous network application injection can be caused [20].

Ryu

It is a tiny controller based on components and implemented in Python. Ryu's core is tinier than others. Ryu provides software having APIs that are well-defined and make it straightforward to build new applications for management and control of network. It supports a number of protocols, like OpenFlow, Netconf, and OF-config) for management of networking devices [21]. This controller comes highly promising.

OpenDaylight

It is the largest open source collaborative project, with the goal of increasing the rate at which SDN is adopted and laying the groundwork for Network Function Virtualization (NFV). The Model-Driven Service Abstraction Layer(MD-SAL) is at the heart of the OpenDaylight controller. In this controller, all underlying network devices and network applications are shown as objects or models [7]. The YANG models do this by providing broad details of capabilities of device/application without needing any party to be aware of the other's specifications of implementation details. Furthermore, OpenDaylight supports a wide range of protocols on the SDN platform, including OpenFlow, OVSDB, NETCONF, BGP, and many others, which make today's networks more programmable and address variety of requirements of users. OpenDaylight is built on top of a Kafka container, allows it to start and stop in a dynamic fashion without interference with other modules [22].

ONOS

For establishing the controller cluster, ONOS is a native distributed SDN controller. The scalability of deployment is increased by clustering of controller and at the same time avoids the bane of one-point-of-failure issue of SDN. The control plane capacity of an ONOS cluster can be increased as required. In case where more switches have to be added to the network, more ONOS replicas can readily be installed to the cluster and services wont be disturbed. ONOS, as OpenDaylight, is based on Apache Karaf framework [24]. (OpenContrail. OpenContrail is based on industry protocols and includes all of the parts required in virtualization of network. ONOS facilitate the development specifically virtual network management. It also focuses on management of issues of large-scale environments for eg issues of multi-tenancy,management and segmentation of network,access management etc [25].

More controllers are also available **NOX** [24] and **POX** [25] are the most popular among them. NOX is the initial OpenFlow controller, which was created concurrently with the protocol [9]. As a result, it drew a lot of research interest. POX is a Python-based brother of NOX that is commonly used in research for rapid prototyping of network applications. **Open Mul** [26] is a C-based OpenFlow controller that focuses on performance and reliability. This controller, they argue, is appropriate for "mission-critical" situations [28]. We will use RYU controller in our thesis because it can operate on any platform that supports Python and requires no extra configuration [29].

2.1.3 Application Plane

The application set that programme the controller to implement network control and operation logic is known as the application plane. As depicted in 2.3, application plane applications include routing apps, firewalls, access control lists (ACLs), monitoring, load balancers,intrusion detection systems (IDS), and DDoS attack mitigation. Network applications provide rules, which are then translated into southbound (data plane to control plane)-related commands that programme the forwarding component's behaviour[30]. The IDS network programme, for example, may keep track of network traffic data, host migration, and payload of packet etc. When malicious traffic is detected, the IDS application may be able to halt it before it infects the network. However, there are still a number of security vulnerabilities with the network application's implementation.

2.1.4 Southbound Interfaces

These interfaces (also known as APIs) provide communication between data plane and control plane and acts as a link between the two planes. This interface allows for effective network control and enable modifications in networks dynamically in response to real-time network demands. In contrast with other southbound interfaces he Open Networking Foundation's (ONF) OpenFlow is the mostly used as southbound interface, and it will be addressed in the following subsections. It's industry based standard that specifies how the SDN controller communicates with switches/routers for improving

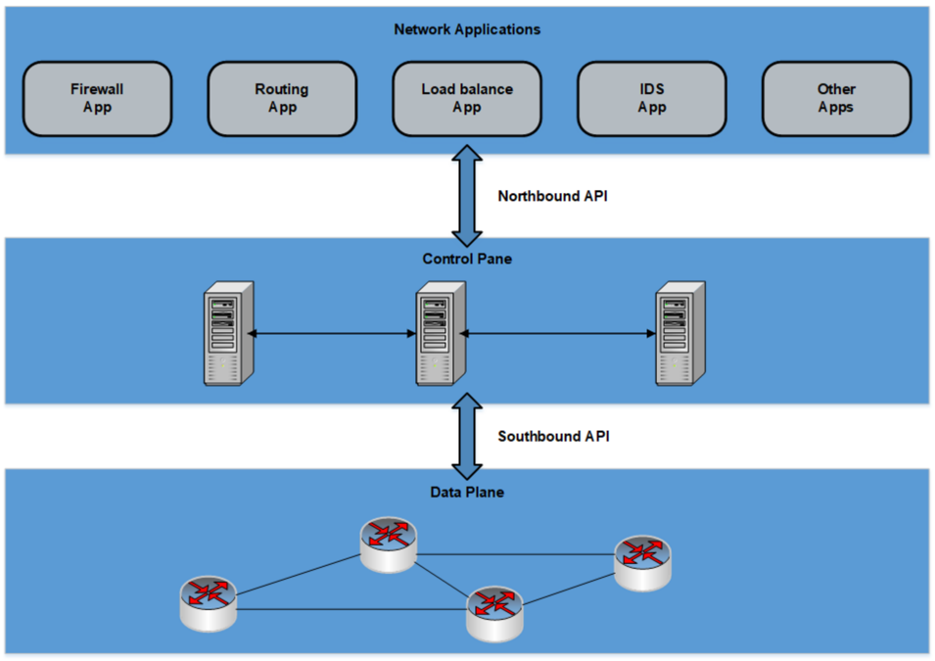


Figure 2.3: SDN Architecture

network reactivity as per real time need. OpenFlow allows users to add and remove entries from switches' and routers' internal flow tables [27]. Other southbound interface designs exist, and OpenFlow isn't the only one that's available or in the works. This thesis will be focused on OpenFlow-based SDN because it is the highly used southbound interface.

2.1.5 Northbound Interface

It is utilised for communication of network applications in application plane with the control layer and vice versa. Its programmabe nature helps in innovations and automate the network for meeting the demands of various applications. It must be able to handle a wide variety of applications. The northbound interface, for example, might be used to optimise network applications such as load balancers, IDS, firewalls etc. The northbound interfaces of SDN controller is also used for connecting it to NFV orchestrators or cloud systems like Puppet, Chef, Ansible, and OpenStack. For SDN northbound interfaces, a range of interfaces such as RESTful APIs, Java APIs, Python APIs, or message queues are available [28].

2.1.6 Westbound/Eastbound Interface

They interaction boundary remains still undecided. They are used for communication across separate SDN/non-SDN domains in general. This interface is like a communication link between control layer of SDN and different network domains that are controlled via different controllers in SDN. Establishing of network flows becomes easy across many domains while also allowing the interchange of network status information to impact routing decisions of each controller. Control planes communicate with non-SDN domains using the eastbound interface [29]. In ideal scenario both domains must seem to be entirely consistent with each other in this way. Routing protocol used by non-SDN domains, for example, should be compatible with the SDN domain.

2.2 Communication Protocol

For management of the the routing of network devices on an SDN, many communication protocols have been established. OpenFlow is the most widely used communication protocol.

2.2.1 Openflow

Traditional networks have made overall revolutionary improvement in terms of security, speed, and dependability. In terms of the physical network, network layer devices now have high-capacity links and more computing power. A number of tools for monitoring and inspection have arisen as a result of these uses. However, since its inception, the structure hasn't changed significantly.

The primary function of a network is to carry out operations such as switching, routing, and access control. In today's networks, this function is performed by network devices from several vendors running proprietary firmware, leaving little room for fresh research concepts such as routing algorithms.This impediment is one of the reasons why current network architecture is stagnant and inflexible, with no major change in this direction.

To address this problem, OpenFlow is an open standard protocol. Now, network administrators may install and regulate the desired functionality within the software. Furthermore, it enables academics to conduct real-world trials with novel protocols and algorithms without requiring companies to open up their products.

OpenFlow makes use of flow-tables to construct firewalls, NAT, QoS, and network management statistics regardless of which vendor is involved. A centralised controller can develop and modify these flow-tables, which contain match/action rules. Using flow-based packet forwarding processing, the controller allows network managers to programmatically govern flows and construct a customised route from source to destination. It reduces power consumption and network management costs by removing router packet handling and allocating paths with the help of centralised control plane.

2.2.2 OpenFlow Architecture

Three basic characteristics of the OpenFlow network design are:

1. The data plane is made up of Openflow switches (those that follow the Openflow protocol/compatible with it).
2. Single/multiple OpenFlow controllers make up control layer.
3. The switches connect with the control plane via a safe control communication channel .

As indicated in fig. 2.4, data path communication is given by software, whereas control

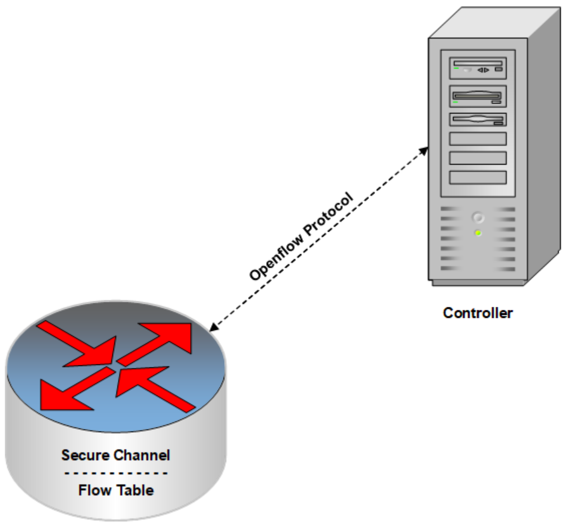


Figure 2.4: Openflow Network Architecture

path communication is provided by a controller and can be made secure by optional cryptographic protocols such as SSL/TLs, in which both entities are mutually authenticated using symmetric keys. Even though it is a secure method, the controller is prone to attacks like Man-in-the-Middle and DoS attack such as the Black Hole attack. Therefore, a good security strategy must be present to protect against these kind of attacks.

2.2.3 OpenFlow compliant switch

A switch that is OpenFlow compliant or based on the protocol is the most basic data forwarding device. It is in charge of the flowtable and keeps it up to date. It communicates openflow messages with the controller via a secure control pat/channel, as shown in 2.5.

On the OpenFlow Switch Specification ONF, many versions are available, but in this thesis, OpenFlow version 1.0 is used.

Header Fields	Counters	Actions

Table 2.1: Fields in a flow table for OpenFlow

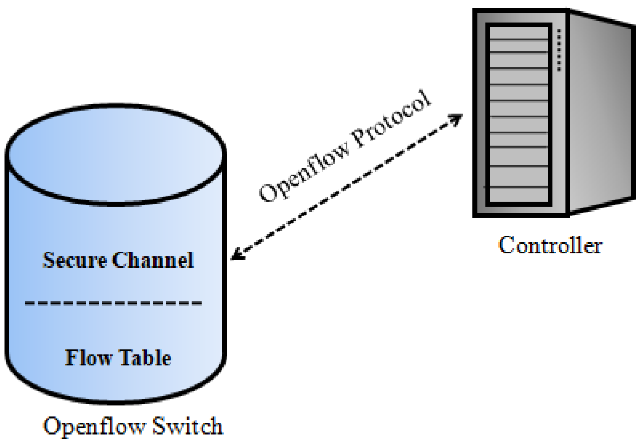


Figure 2.5: Basic Packet Process mechanism for Openflow switch

2.2.4 Flow Tables

An openflow switch keeps track of entries in one or more flow tables. As shown in table 2.1 above, each item has three fields: match, action, and counter table 2.1 above. After receiving a packet, switch looks into its forwarding/flow table for a matching rule for the packet's header. The relevant action field (drop/forward/modify and forward) is then executed, and the counter is incremented. Counters are saved and used to maintain record of the statistics of flow, for example the packets counts/bytes counts received, the duration of the flow, and so on. If the flow table of switch doesnot have any matching rule, the packet is passed to controller, which then delivers it back to the switch with a feedback.

2.2.5 Matching flow

The header field in the openflow switch's flow table contains multiple fields with. Flows arriving at the switch are compared on each tier of the OSI model as well as on the switch port. The list of those fields is as follows:

- Incoming switch port
- IEEE 802.3 Ethernet source and destination address

- IEEE 802.3 Ethernet type
- IEEE 802.1Q VLAN ID and priority
- IP source and destination address
- IP proto field
- IP Type Of Service (TOS) bits
- TCP/UDP source and destination ports

2.2.6 Actions taken on the flows

If any of the match fields of flow table matches with an incoming packet, an action must be done on that packet according to flow table. The OpenFlow switch must be capable of forwarding packets to each physical port. OpenFlow standard also define virtual ports as special targets to which packets may be routed, as shown in table 2.2 and table 2.3. There are two types of actions: “required” and “optional”. To be OpenFlow-compliant, all switches must provide both the required and optional actions, which are useful but are not in every case in an OpenFlow switch.

Virtual Port	Description
All	Send the packet to all ports except the one where it was received.
Controller	Send the packet to the controller in encapsulated form.
Local	Send the packet to the switch’s local networking stack.
Table	Execute actions in the flow table. Only for packet-out messages.
In-port	Send the packet to the port where it was received.

Table 2.2: List of virtual ports for the “Required” forward action

There are a number of more actions in the flow table of switch other than the forwarding action:

- Drop: An empty action list indicates a required action. Every packet matching an empty list of action is dropped.

Virtual Port	Description
Normal	Forward the packet on the basis of traditional forwarding mechanisms, i.e. traditional L2, VLAN, and L3 processing.
Flood	Send the packet along the minimum spanning tree, not including the incoming interface. All ports in the OpenFlow enabled switch has a NO_FLOOD-bit, which indicates that the port doesn't belong to the minimum spanning tree. The packets that match that flow entry are e forwarded to the ports that have a NO_FLOOD-bit.

Table 2.3: List of virtual ports for the "Optional" forward action

- Enqueue: This is an optional action for queueing packets related to a port for providing QoS.
- Modify-field: It is an optional operation used for changing the header field of a packet that has arrived. The changes that can be made are as follows:
  - Stetting up of VLAN ID and priority.
  - Stripping off the VLAN header.
  - Modification of Ethernet source MAC address and destination MAC address.
  - Modification of the IP source and destination.
  - Modification of IP TOS bits
  - Modification in the source and destination ports of transport layer.

2.2.7 Counters

Flow-table of switch have counters that allow it to store statistics such as queue, flow, and port, as shown in table 2.4

2.2.8 Flow types

The flows are generally classified into micro flows and aggregated flows [31]:

Counters	Bits
Per table	
Active Entries	32
Packet Lookup	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 alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64
Per Queue	
Transmit Errors	64
Transmit Bytes	64
Transmit Overrun Error	64

Table 2.4: Required list of counters for use in statistics messages

- Microflows: Small networks, such as campus networks, benefit from these types of flows. In this form of flow, each flow table entry corresponds to a single flow.
- Aggregated: This sort of flow is appropriate for big networks, backbone networks for example, where a huge amount of entries of flow table are needed. Here, a single flow entry (Wild-carded) matches more than one flow from a single category.

Both these type of flows can further be classified into Proactive flows and Reactive flows.

- Reactive: Until the first packet is received by controller from OpenFlow switch it remains in idle state in this configuration. A new flow table entry is created after parsing the incoming packet. A little amount of time for setup is required for every new flow entry. In case of failure of the connection between the switch and controller the switch is unable to forward packets like a regular switch, the hosts will not be forwarded the packets.
- Proactive: Controller pre-installs entries into flow table of switch without requiring the first packet of the flow to be received. No additional time is required for setting up of the flow, and traffic will not be disrupted if the link between the controller and the switch fails.

2.2.9 Packet forwarding mechanism

Upon receiving a packet switch analyses the header fields to analyze if there a match entry is present in flow table. In case a match is found, action is performed according to field. If multiple matches exist for single entry, the flow entry having the highest priority is chosen. The counter of that flow entry is then incremented, and the packet is transmitted to a port. In case of no matching flow entry, packet is sent to controller, which determines which logic/action should be applied to that packet and any subsequent packets of a similar nature. The flowchart 2.6 depicts the packet forwarding method.

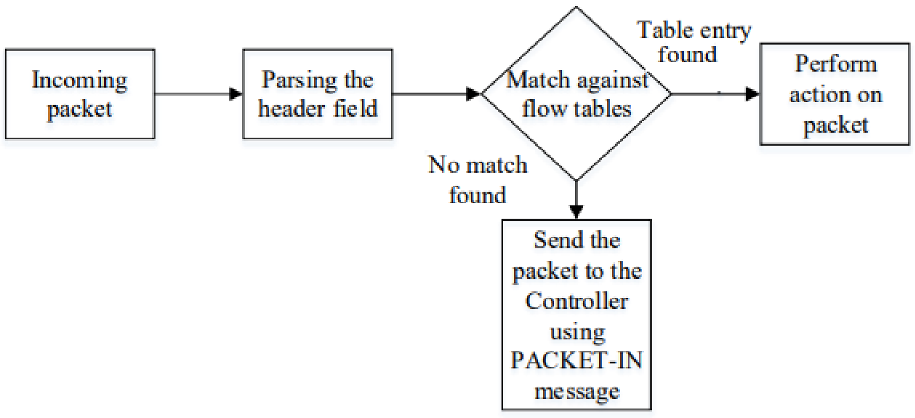


Figure 2.6: Basic Packet Process mechanism for Openflow switch

2.2.10 OpenFlow communication messages

In OpenFlow protocol there are following types of communication messages i.e., Controller-to-switch, Asynchronous and Symmetric messages.

Controller-To-Switch Messages Messages sent to switch from controller are known as Controller-To-Switch Messages. Following are its types:

- Features: In order to know the features supported by a switch, a 'feature request' message is sent by controller to it and it replies with a 'features reply' message for specifying features it supports.
- Configuration: By this message configuration parameters in the switch are set and queried by controller.
- Modify-State: For adding/deleting or modifying flow table's flows and for setting properties of switch port this message is used.

- Read-State: The controller uses these messages for collecting statistics from flow-tables and/or ports and/or individual flow entries.
- Send-Packet: For sending packets out of specific switch port a controller uses this message.
- Barrier: In order to check whether previous messages had arrived controller uses this message. If previous messages have successfully arrived a Barrier Reply message is sent by switch.

Asynchronous Messages These messages are sent to the controller by switch. Following are four main asynchronous messages:

- Packet_in: Controller sends this message in case a switch there is no matching entry with a received packet. It may also be sent to controller if the matching flow rule orders the switch to send packet_in message to controller.
- Flow-Removed: The flow table entries have two types of timeout :The idle timeout means that if no packets match it after this time flow entry is deleted from the flow table whereas hard timeout is time after which time the flow is deleted from flow table regardless of whether packets match or not. The controller indicated them via Flow-Removed messages.
- Port-status: In case status of an existing port changes or if new port is added or if a port is removed or is modified this message is sent to controller.
- Error: This message is used to inform a controller in case of a switch error.

Symmetric Messages The switch as well as the controller can send symmetric messages in both directions. Following are the types of these messages:

- Hello: Upon connection start up controller and switch send these messages to each other.

- Echo: They are of two types echo request and echo reply when a switch or controller send an echo message ,echo reply message is sent to show the liveness, bandwidth or latency of connection between controller and switch.

- Vendor: The vendors can create custom message and provide more functions with the help of these messages.

2.2.11 Explanation of messages exchanged in OpenFlow network

Mininet [30] was used for explaining process of exchange of messages mentioned above for simulating host h1 and h2 connected with a switch and controller, as shown in fig 2.7 below. For this demonstration, we'll go through how to connect to the Switch Controller, then how to communicate from one host to another using the OpenFlow switch and controller.

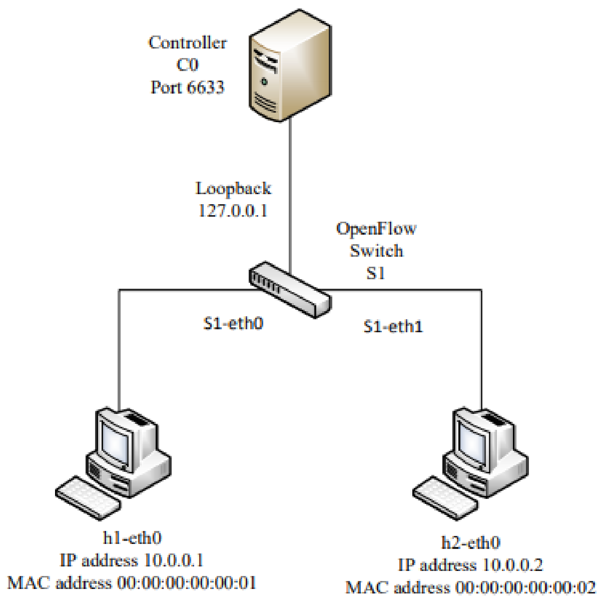


Figure 2.7: Network Topology using mininet

Message establishment between a switch and a controller

Upon joining an OpenFlow network by a switch, a TCP handshake is done between controller having IP Loopback interface 127.0.0.1 and port 6633(default), as illustrated in fig 2.8. After that, both parties begin exchanging Hello messages with the greatest OpenFlow version that they support. Controller send a ’Feature request’ message to switch for knowing the available ports , and switch responds with a Feature reply message with port’s list,speed and tables and actions supported. The controller then sends a set config message to the switch, instructing it to deliver flow expirations. Finally, the switch and the controller send echo requests and replies periodically to communicate information about the bandwidth, latency, and liveness of their connection.

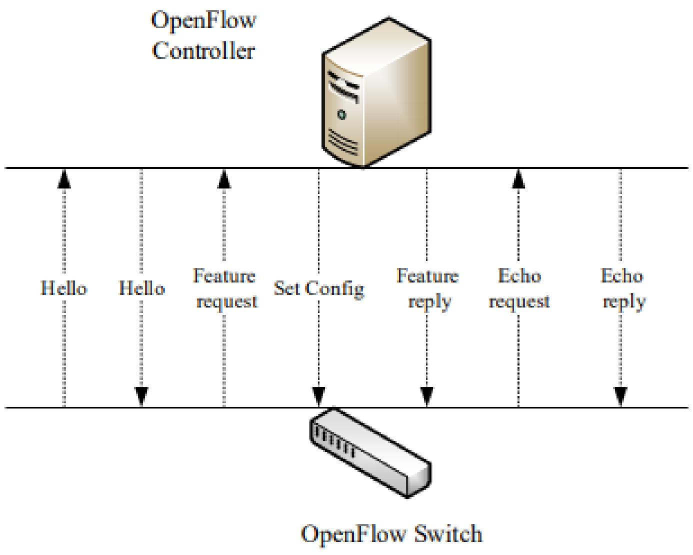


Figure 2.8: Communication Messages between switch and controller

Messages exchanged between two hosts

In order to show how the link between two hosts work in SDN network we utilised ping tool for sending ICMP messages from host h1 to h2 as well as from h2 to h1 . Process begin when an ARP request is made to the switch by h1, to ask for h2’s MAC address; since switch is unaware of how to process flow, therefore it sends it to controller as a packet_in message. After this, the controller sends Feature request message to switch for knowing about ports available, then switch responds with ’Feature reply’ message having a list of ports, port speeds, and supported tables and actions. The controller then sends a list of configuration message to switch, instructing to deliver expiration values of flows. The OpenFlow controller will then install new flow entries in the flow table of switch, as seen in Fig 2.9.

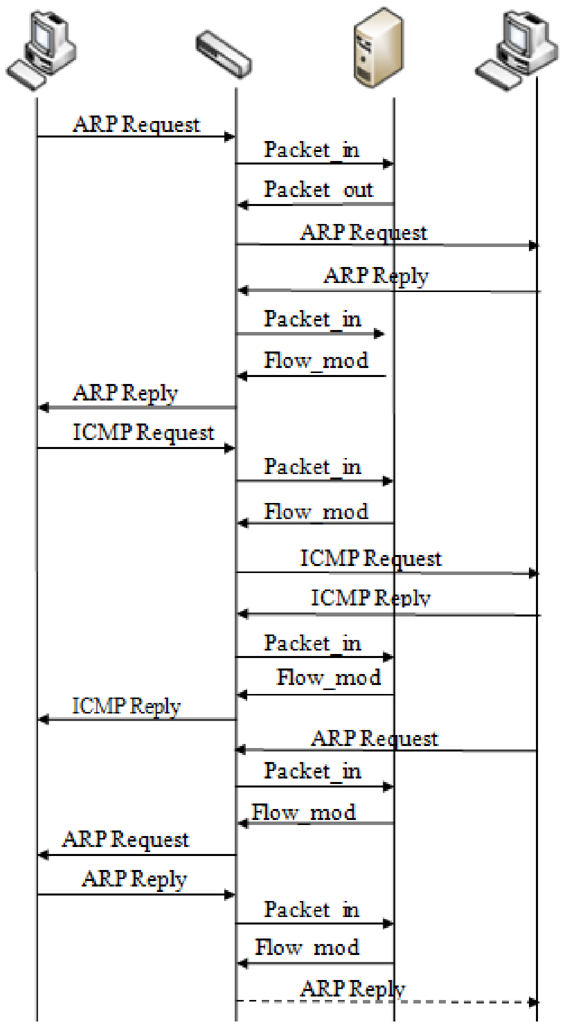


Figure 2.9: Ping process between h1 and h2

2.3 Network Protocols

To build its perspective of the network topology, different kinds of packets of protocols like ARP and the LLDP are sent by the switches in form of packet_in messages. Controllers manage LLDP and IGMP messages for topology discovery and multicast group maintenance, as well as ARP queries and answers, for enabling hosts to construct ARP caches [31] for facilitation of network communication.

2.3.1 Adress Resolution Protocol

ARP Proxy SDN, like IP networks, enables Address Resolution Protocol (ARP), which determines the relationship between a destination IP address and its corresponding hardware(MAC) address so that hosts can transmit and receive IP packets appropriately. Layer two switches in IP networks flood an ARP request submitted by a host in order to receive an ARP reply. A router functions as an ARP proxy, sending back an ARP reply with the hardware address of its own interface if the target IP address in the ARP request is not in the local network. An ARP proxy application in the SDN controller handles ARP packets in SDN. When a host sends an ARP request to a switch, it is transmitted to the controller as a packet in messages. We used the Ping programme to send ICMP packets from h1 to h2 and vice versa to demonstrate how the host-to-host link works in an OpenFlow network. The procedure begins when h1 makes an ARP request to the switch, requesting h2's MAC address; the switch is unsure how to handle the packet, so it sends it to the controller as a packet_in message.Packet_out messages are used to generate ARP reply packets, which are then routed to the ingress switch. The original host receives an ARP response as a result.

2.3.2 Link Layer Discovery Protocol(LLDP)

To create their view of the network , SDN controllers process a number of protocol packets sent by switches using OpenFlow packet_in messages. During process of connection discovery, the controller sends LLDP packets as Packet-out messages to all switches in the network. When the SDN switch receives an LLDP packet from the controller, it sends it to all of the other switches connected to it. Because there is no similar forwarding rule in the switch's FlowTable, when a switch gets an LLDP packet from another switch, it sends the LLDP packet to the controller as a packet_in message for assistance. [33] as shown in fig 2.10.

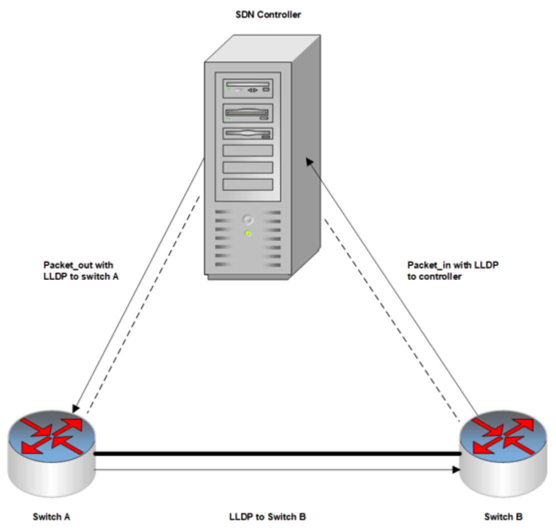

SDN Controller
Packet_out with LLDP to switch A
Packet_in with LLDP to controller
Switch A
LLDP to Switch B
Switch B

Figure 2.10: Network Topology Discovery in SDN using LLDP

2.4 Vulnerabilities of Network Protocols and Black Hole attack

The described link discovery approach is insecure because LLDP control packets are not authenticated [34] As a result, any LLDP packet that the controller receives is regarded as valid. As a result, an attacker can poison the controller's topology information by sending counterfeit LLDP control messages. The attacker can forge the content of LLDP packets or fake link discovery by constructing a link that does not exist [35].

Consider the situation where a host has been compromised. The controller generates LLDP packets for all of switch S1's active ports, specifically LLDP packet(S1, P3) and LLDP packet(S1, P4) (S1, P4). Then, to relay the necessary LLDP packets, it sends packet_out messages to all active ports, specifically packet_out(S1, P3) and packet_out(S1, P4). The attacker captures the LLDP packet(S1, P4) and changes it to LLDP packet(S2, P3) before sending the faked packet to S1. When switch S1 receives an LLDP packet (through Port P4), it sends a packet_in message to the controller that includes the sender's and receiver's information, as determined by the discovery technique (Step 3). When LLDP packet(S2, P3) is transferred, S1 transmits packet_in (S2, P3, S1, P4), with the first two parameters from the original. The controller assumes that a link exists between Switch S2 (through Port P3) and Switch S1 (via Port P4) when it does not, as the attacker's goal require [36]. Packet loss happens when the controller tries to route traffic through these bogus links, and if the link is on a critical path, a black hole may result.

Attacks like ARP spoofing, which might affect behaviour in a traditional network in a specific way, can present differently in SDN topologies depending on the controller implementation. For example, bogus links can be created using Link Layer Discovery Protocol (LLDP) signals to trick the controller into thinking they exist, resulting in black hole routing.

ARP spoofing attacks, in which a rogue node sends ARP packets, are also seen on SDN. Successful attacks have the potential to contaminate network topology data, which is a key component of fundamental SDN components. As a result of the poisoned network visibility, the SDN controller's top layers services and applications may be completely misconfigured and badly impacted.

SDN also sees ARP spoofing attacks, in which a rogue node transmits ARP packets. Successful attacks have the potential to poison network topology information, which is a critical building ingredient for basic SDN components. The top layers services and applications of the SDN controller may be fully misconfigured and negatively effected as a result of the poisoned network visibility. In some circumstances, this condition results in major hijacking, denial of service attacks, and network collapse. Several SDN investigations reveal that these attacks affect all current main SDN controllers (e.g., Floodlight, Open Daylight, Beacon, and POX). If such core network topology information is tainted, all dependent network services are damaged instantaneously, resulting in catastrophic problems [37]. As a result, the controller's routing services/apps can be altered to create a black hole route/attack. The controller then sends a Feature request message to the switch to see which ports are available, and the switch responds with a Feature reply message with a list of ports, port speeds, and supported tables and actions. The controller then sends a set config message to the switch, instructing it to deliver flow expirations. When a switch receives an LLDP packet from another switch, it sends the LLDP packet to the controller as a Packet-in message for assistance because there is no analogous forwarding rule in the switch's FlowTable. The SDN controller can check whether switches are directly connected to one another and design the global topology after receiving Packet-in notifications via LLDP [33].

Certain communications can be dropped if malicious or aggressive nodes refuse to route them. A Black Hole Attack occurs when all of the packets are dropped via them. It's one of the most damaging routing attacks out there. It generates network traffic and dissipates all network interactions. It made use of a variety of routing metrics. Fake link quality, shortest path, and other criteria are used. It is a direct attack on the network's service availability. If the malicious node does not forward all messages between network devices and the controller, communication will inevitably break down, with data plane devices unable to contact the controller when necessary.

The black hole attack is a type of DoS attack that generates and disseminates forged routing information. The rogue node advertises itself as having a proper routing path between the source and destination via the routing protocol. It then drops the captured packets rather than forwarding them to other nodes.

Attackers with access to a host directly connected to an SDN, such as a server or a user workstation, can execute a blackole attack to fool the SDN controller about the network topology and location of target hosts, allowing them to hijack a target host or traffic of interest. These are usually ARP-based attacks because the Address Resolution Protocol (ARP) is one of the few protocols that hosts can use to change the SDN control plane. According to prior research, hosts can also inject or tunnel LLDP packets.

Attackers with access to a host directly connected to an SDN, such as a server or a user workstation, can execute a blackole attack to fool the SDN controller about the network topology and location of target hosts, allowing them to hijack a target host or traffic of interest. These attacks include denial of service (DoS)/DDoS assaults, data manipulation, repudiation, blackhole attacks, and side channel attacks.

It's also possible that the controller will be subjected to a false information attack. Nodes in the network may provide the controller with fake route information, which the controller may use to construct a false route for certain hosts, resulting in a black hole. It is relatively easy to create false rules if a controller becomes wicked, resulting in network disputes.

A hacked program can use OF rules to adjust forwarding behavior invisibly, resulting in active network exploits like the black hole attack.

2.5 Related Work

Being previously stated, as the key component of the SDN network, the SDN controller becomes more exposed to a variety of assaults. Successful attacks have the potential to contaminate network topology information, which is a key component of core SDN components. If the topology view of the network is altered ten the services of the upper layer might be totally causing DDoS attacks and breakdown of network. According to several SDN investigations, these attacks affect all of the current key SDN controllers (e.g., Floodlight, Open Daylight, Beacon, and POX). All reliant network services are immediately harmed if such essential network design information is corrupted, perhaps leading in catastrophic challenges. Even a black hole attack can be launched by corrupting the routing services.

The fundamental risk of LDS in SDN is absence of proper mechanisms of authentication of openflow discovery protocol (OFDP), a standard utilized for implementing the link discovery process in mainstream controllers such as Floodlight, OpenDayLight, POX, and Ryu. OFDP cannot ensure the legitimacy of incoming network packets like

LLDP packets, that can cause packets to being received at random, even those that are manufactured. Attackers can obtain and analyze LLDP packets using network monitor tools in order to carry out tampering. As a result, the controller is likely to update the view of network on the basis of forged LLDP packets, and would result in wrong view of network adversely affecting the management and may also result in collapse of network. Nonetheless, the different methods that have been proposed so far have not been able to handle the problem effectively.

If an attacker successfully hijacks an OpenFlow-based SDN switch, they gain access to all flow rules and data [36]. If an attacker gains control of a host, they can impersonate a legitimate user and gather data from other hosts (e.g., tokens and passwords). [36]developed a False LLDP Injection technique in which an adversary sends fake LLDP packets onto an OpenFlow network to inject bogus internal links between two switches. By monitoring traffic from OpenFlow switches, the adversary can recover the genuine LLDP packet. [36] has created a host location hijacking attack that may fake a target host's identity and use its location information without their knowledge. Hong et al. [38] presented TopoGuard, a security module for OpenFlow controllers that detects Network Topology Poisoning Attacks automatically and in real time.

A study in [39] presents a solution based on the pre-condition and post-condition of the host migration to overcome the faked identity problem in SDN. Once the host has migrated from its current location, it must notify the SDN controller of its prior port shutdown. The controller confirms that the host is not reachable in the post-condition by sending ping packets to its prior location. As a result, the controller can efficiently follow the host's true location and detect the malicious host's faked identity. Abdou et al [3] in their work presented the specific features to differentiate between benign and malicious users.

Malicious hosts can produce topology poisoning, which results in a black hole attack. SDN controllers must be accessible to end users, however they are subject to end-user assaults, particularly black hole attacks, in which an attacker can drop/sniff packets, causing a loss of availability and computational resources, making the controller unavailable to legitimate users. One way to avoid black hole attacks from end users is to authenticate them.However it results in large over head. Shin et al. [40] presented ways for improving network security by using an SDN controller to dynamically route flows through static security devices for attack detection. Attacks, on the other hand, can only be detected after the Controller has established the flows. As a result, the controller may be subject to attacks from malicious end hosts. A security management application for the controller is proposed in [41]. SDN security policies can be specified and evaluated using this application. These regulations can then be enforced on network flows by security modules in the switches. This work enables proactive identification of malicious hosts prior to the controller receiving flow requests.

The use of TLS (Transport Layer Security) has been made optional in the openflow specification, making it a weak-point and plainly vulnerable to numerous assaults such as black-hole attacks, according to [42]. As a result, they recommend that an intrusion detection system (IDS) be used to detect rogue hosts.

Sphinx [31] framework that drew attention to a slew of security issues One issue was the construction of a bogus topology in the network, among other things. Fake LLDP packets can be used by a node to alert the controller of a fake route. A switch black hole is formed by a malevolent host supplying false information about a route in order to create a black hole. Graph blackholes could be identified by Sphinx by validating the byte consistency of flow.

The SDN controller is intended to be accessible to end users, however it is vulnerable to end-user assaults, particularly the black hole attack, in which an attacker can drop/sniff packets, resulting in a loss of availability and computing resources, making the controller unavailable to end users.Shin et al. [40] offered approaches for improving network security by using the SDN Controller to dynamically route flows through static security devices in order to identify assaults. Attacks, on the other hand, can only be detected after the Controller has established the flows. As a result, the Controller may be subject to attacks from malicious end hosts.A security management application for the controller is presented by Varadharajan and Tupakula [43]. SDN security policies can be specified and evaluated using this application. These regulations can then be enforced on network flows by security modules in the switches. This work enables proactive identification of malicious hosts prior to the controller receiving flow requests.

The fundamental solution for the data layer malicious node/end user dilemma is encryption and authorization authentication. However it caused undue overhead and complex computationn. To authenticate the host identification, AuthFlow [37] employs a RADIUS server. It implements access control for each host based on its level of authority by mapping host credentials to a set of flows. To ensure the entire SDN communication process, MM et al. [11] suggested a hybrid control security approach based on TLS. On the other hand, this method involves the usage of a centralised trust management module and increases the signature and authentication overhead, both of which have an impact on system performance. The controller has SLICOTS installed, which monitors active TCP connection requests and prevents malicious hosts.

WedgeTail [36] is an Intrusion Prevention System (IPS) that monitors a network for malicious devices. They rely on collecting all OpenFlow messages sent between switches and controllers in an SDN network. They build a virtual network copy in geometrical space using this information, with each forwarding device represented as a dot and each packet as an edge linking the dots.The expected route trajectory for each packet is enumerated in geometrical space based on historical data. The attack detection engine flags packets that adopt unusual trajectories or courses in real time, implying that one or more of the devices that the packet crossed along the trajectory is/are malicious. This study has been shown to be superior to the SPHINX system, which was an attack detection method based on a graphical depiction of network traffic flows and a threat model that ignored trusted devices.

SDN's traffic monitor continuously collects traffic data and flow characteristics from each switch/router. The controller employs the weighted KNN with GA to classify whether each switch/router is abnormal or not after gathering these data and parameters. Then, in the next phase, creates a suspicious list for further testing and confirmation.

Encryption of communication is an old method for preventing packet sniffing attack. However, availability loss due to blackole attack disrupts the communication. Furthermore, actual applications have a variety of constraints. To begin, both communicating parties must support the encryption protocol; otherwise, communication would fail. Second, several common protocols, including HTTP, FTP, Telnet, and SMTP, do not use encryption, posing a severe security risk to communication using these protocols. Finally, some encryption systems have security weaknesses that allow attackers to decrypt transmission data.

Collaborative modifications to numerous network configurations place greater demands on network management capabilities. In a typical IP network, where routing protocols are used to configure the routing table, distributed control is used. The changing network configuration might have major effects in this paradigm, such as service outages and route inflation [9]. It's also difficult for traditional networks to alter several network configurations at the same time. Due to the lack of a global view and flexible resource allocation, MPLS, a high-speed networking technique utilised in traditional networks, finds it difficult to execute dynamic resource adjustments.

[9] attempts to enable dynamic changing of host IP configuration in a typical network. However, because various new gadgets are being released, the cost is substantial. As a result, collaborative modifications across many network configurations necessitate effective network management. SDN (software-defined network) [13] is a new way for achieving dynamic network configuration. SDN uses logic centralised control by decoupling the control plane from the forwarding plane (data plane). SDN's robust network management and control capabilities provide for additional flexibility in the

implementation of dynamic network configurations. Because of SDN's programmability, it can directly regulate the flowtable of forwarding devices, avoiding service interruptions and inflation of routing. Because of SDN's centralised control, a global view of the network is possible. As a result, it is possible to make cooperative changes to numerous network configurations.

Chapter 3

Proposed Methodology

The static route configuration is a serious threat to communication security as the attackers can obtain packets and cause black hole attack. Therefore, we propose a hopping communication approach in which the routing paths are dynamic and pass through multiple users/hosts that are selected based on their health. This periodic changing of path increases the ability of network to resist black hole attack.

In hopping communication, our source and destination mac address changes on every hop making it difficult for attackers to capture traffic and hence drop/absorb it.

The most widely used topology in data center networks is the multi-rooted topology. In this kind of topology, more than one path are available between source and destination host. To take full advantage potential of multi-rooted topology, we employ our multi-hop multiple path routing implementation method instead. It allows for selection three best paths based on the health of the nodes. The SDN approach is then employed to achieve control of flow at a finer level: The network state is collected in control plane, and path allocation is done individually with the help of real-time link information. The routing rules are then saved in flow table of switch. This method maximizes the use of network resources and reduces congestion at the same time. This on demand multi-path routing is done only when a new flow is sent by host.

3.1 Methodology

Mostly the topologies used in data-centers are multi-rooted topologies. In such topologies there are more tan one path between two hosts. In our scheme we therefore employ a multi-hop multiple path routing approach in order to resist black hole attack. This dynamic routing with hopping of route will make it less easier for attackers to carry out targeted packet drop/black hole attack because the data will pass through a number of hosts and the attacker will have to take control of more hosts in order to carry out attack. In this approach we select tree best paths between a source and destination host based on the fitness of the hosts. This fitness is based on past behavior of hosts and is assumed that they will behave in a similar manner next time. The controller of SDN gathers real-time statistics and link information and ten selects the path/link with the hosts with maximum fitness. The fitness of link/routing path is calculated in our approach by taking product of the fitness of all the hosts involved in that link/routing path.

3.1.1 Algorithm

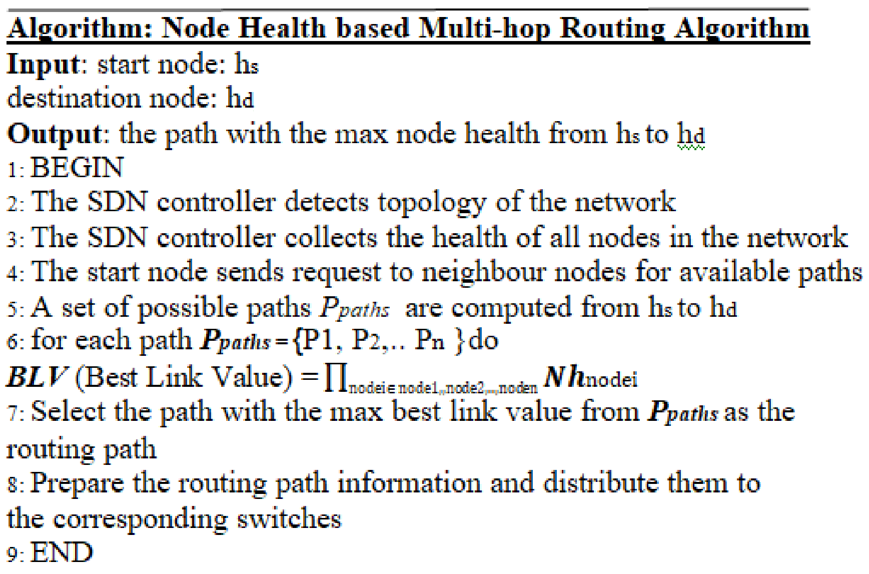
In order to explain how our multi-hop multiple path scheme works we use an example to show the main messages exchanged in a typical scenario

- Suppose host h9 wants to communicate with host h1. First of all packet in message is sent by both hosts to the controller for registering to its services. When this message is received by the controller, it stores this information in a host registration module and assigns a unique ID to the hosts. In our implementation the host ID is h1 and h9.
- Next, as host h9 is unaware of the best routing path to host h1, a connection request message is sent to the controller in order to inquire about the the route information of host h1. This request is in the form of route request message and

is broadcasted to all the hosts.

- When destination i.e., host h9 receives the RREQ message it responds with a route reply packet
- Controller finds out three best paths available paths as per topology of network and decides one best path after computing the product of the fitness of hosts involved in path.
- Host h9 then sends a DATA message to host h1 it then replies back with to host h9 with a REPLYDATA message. All these 4 messages shared serve as connection establishment.
- After connection establisment, h9 is now able to communicate with host h1 and it communicates with our hopping communication approach in which the source and destination MAC addresses change at every hop.
- The values of detection parameters are recorded for the path and if they there is an anomaly in the range of values alert is generated and connection is terminated.

The algorithm of our routing approach is shown below:



```
Algorithm: Node Health based Multi-hop Routing Algorithm
Input: start node: $h_s$
destination node: $h_d$
Output: the path with the max node health from $h_s$ to $h_d$
1: BEGIN
2: The SDN controller detects topology of the network
3: The SDN controller collects the health of all nodes in the network
4: The start node sends request to neighbour nodes for available paths
5: A set of possible paths $P_{paths}$ are computed from $h_s$ to $h_d$
6: for each path $P_{paths}=\{P1, P_2,.. P_n\}$do
$BLV$ (Best Link Value) = $\prod_{nodei \in node1,,node2,,,noden} Nh_{nodei}$
7: Select the path with the max best link value from $P_{paths}$ as the
routing path
8: Prepare the routing path information and distribute them to
the corresponding switches
9: END
```

3.2 System Architecture

The purpose of writing this research is to give an effective routing mechanism for packets in SDN that as the capability of detecting black hole attack. This mechanism is based on collaborative trust with hosts. The trust means that the hosts will behave in a similar manner next time as they behaved in the past. With the help of multiple paths through multiple trusted hops in our solution the controller can select the best path keeping in view the performance of the hosts involved. This will decrease the packet loss and delay. Whenever a new flow arrives a switch ,the switch sends it to the SDN controller which then our routing algorithm is used for calculation of the routing path by controller .

3.3 Implementation

Here, we show the three layer construction of our routing mechanism. We consider a software defined network consisting of a control layer which performs the task of pat allocation in coordination with the hosts, a data layer consists of switches and a host cluster supported by our routing approach tat transmit data and communicated with control layer for allocation of best path for the flow. Management of the flow used in our approach is shown in 3.1.

3.3.1 Control Layer

To realize gathering of information, calculation of path, selection of optimum path, and managing flow, the control layer has following modules:

- Information Gathering Module: This module is responsible for collecting information about topology, node and link discovery by sending periodic LLDP messages. It also collects information about the link for example delay ,throughput

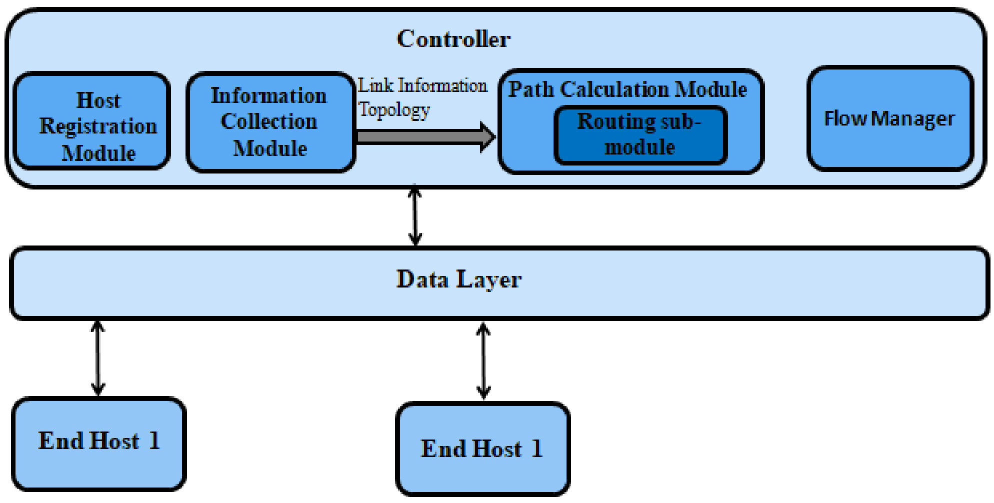


Figure 3.1: Architecture of our proposed model

etc for detection of anomalies.

- Path Calculating Module: This module takes the information from the information collection module about the topology and link information and utilizes this information to find out the fitness of the nodes and calculates three available paths.
- Flow Management Module: This module keeps track of flow status and allocates best path in the current topology.

3.4 Simulation Environment

The details of our simulation environment used in our implementation and experiments are as follows:

3.4.1 System Specifications

The test was performed on an Intel® CoreTMDELL laptop with i7-7500U processor running at 2.70GHz and 2.90GHz and 8GB of RAM. For the Linux environment, we used virtual machines to generate Ubuntu 14.04. We installed the Oracle VM Virtual box as the virtual machine emulator. For making the testbed, we used Mininet to create the testbed where one "remote controller" was used with a custom topology with four switches and ten hosts. The switches were Open vSwitches with OpenFlow protocol. The controller was Ryu.

3.5 Simulation Structure

The simulation structure used for our implementation and experiment is shown in table 3.1.

Table 3.1: System Specifications

Processor	Intel Core i7
System Type	64-bit Operatin System
Oracle	Virtual Macine
Ubuntu	14.04
Mininet	Installed on Ubuntu 14.04
POX Controller	Run on Ubuntu 14.04

3.5.1 Software Tools Used in Experiments

Table 3.2: Software Tools Used For Implementation and Experiments

Software	Function	Version
Mininet	Network Emulator	2.2.2
MiniEdit	Graphical User Interface	2.2.0.1
Open vSwitch	Software SDN Switch	2.3.90
Ryu	SDN Controller Platform	0.5.0
Oracle VirtualBox	Virtualisation	6.0.4
Linux(Ubuntu)	Host Operatin System	14.04
Python	Programming Language	2.7.6

VirtualBox

We may make a virtual network environment using Oracle's free software [34]. With its assistance, an entire operating system can operate within another. We also installed

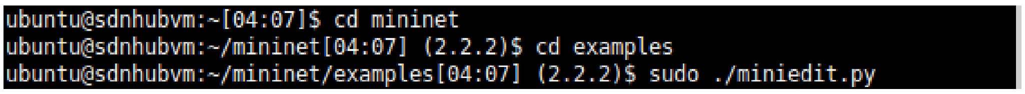
Mininet VM using VirtualBox, which makes switching from Windows to Ubuntu and vice versa instantly.

Mininet

It's a software emulator that uses SDN to simulate a practical topology [35]. It helps in simulating large networks on single workstation without the need for a network connection. Mininet can be used to simulate network topology in OpenFlow switches. It comes with a virtual machine, OpenFlow binaries, and a kernel pre-installed.

MiniEdit

The Mininet network simulator includes MiniEdit, a basic GUI editor for Mininet. MiniNet can be expanded with MiniEdit, which is a demonstration tool. Mininet's Python API was used to construct a graphical user interface (GUI) application. It's a simple network editor that lets us drag and drop switches and hosts, wiring them up together, and to produce a real, functional network just by hitting the "run" button. The MiniEdit script can be found in Mininet's examples folder. Fig 4.1 shows the command to run it.

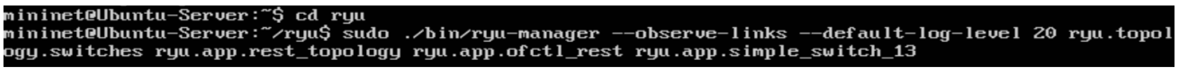


```
ubuntu@sdnhubvm:~[04:07]$ cd mininet
ubuntu@sdnhubvm:~/mininet[04:07] (2.2.2)$ cd examples
ubuntu@sdnhubvm:~/mininet/examples[04:07] (2.2.2)$ sudo ./miniedit.py
```

Figure 3.2: Command Run on MiniEdit GUI

RYU It's a Python-based open controller that can turn OpenFlow into switch,a load balancer, a hub, a firewall, and various other devices. We can pass parameters into the topology in real time. There are provisions for developing custom components as well as stock components. RYU is a framework for establishing OpenFlow communication between SDN switches.

To start the RYU Controller, the command in fig 4.2 starts the controller.



```
mininet@Ubuntu-Server:~$ cd ryu
mininet@Ubuntu-Server:~/ryu$ sudo ./bin/ryu-manager --observe-links --default-log-level 20 ryu.topol
ogy.switches ryu.app.rest_topology ryu.app.ofctl_rest ryu.app.simple_switch_13
```

Figure 3.3: Command To Start Ryu Controller

Ping It's an utility for system administrators. It sends periodic echo request packets to a target network address and then returns the packet to the source using the Internet Control Message Protocol (ICMP). It calculates the round-trip time and displays the total number of packets transmitted, received, and lost, as well as the lowest, maximum, and average response times.

Python Scripting We have used python programming language for scripting with SDN controller application for detection and Mininet. Some of associated python distributions and libraries ,modules etc, we utilised are: Random(accessing iterable object randomly) ,socket API of python(for TCP and UDP host communication interfaces) , blessings(API to manipulate terminals for hosts), matplotlib(library for graphical plotting), subprocess (module that to run external programs and inspect their outputs), datetime (module to work with dates as date objects(it is not python data type however can be imported), NumPy library(to work with arrays)

3.5.2 Simulation Topology

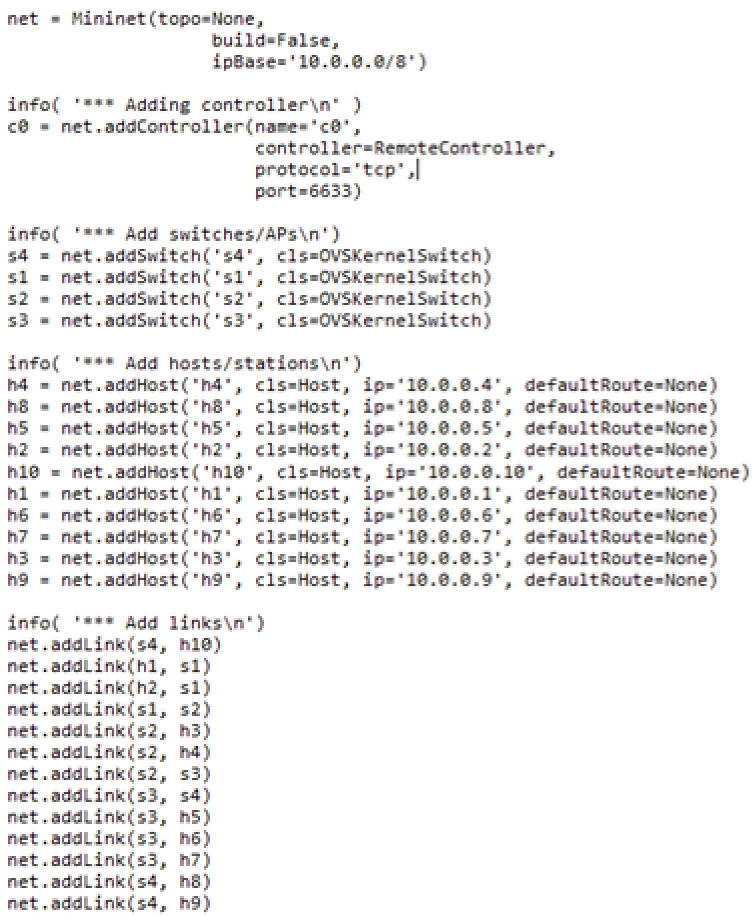
In our mininet script, we've defined a custom topology with four switches and ten hosts, as shown in fig 3.4. In the miniedit GUI, the topology is presented in fig 3.5.

3.6 Mininet Topology

As shown in fig 3.5 flat topology is created in Mininet graphical interface Miniedit consisting of 10 hosts and 4 switches and a Ryu controller.

3.6.1 Mininet Topology Code

For controlling the network and gathering of data from hosts, hosts must register their information(IP address and port) with the SDN controller. It is saved by the SDN con-



```
net = Mininet(topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

info( '*** Adding controller\n' )
c0 = net.addController(name='c0',
                       controller=RemoteController,
                       protocol='tcp',
                       port=6633)

info( '*** Add switches/APs\n')
s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
s3 = net.addSwitch('s3', cls=OVSKernelSwitch)

info( '*** Add hosts/stations\n')
h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
h8 = net.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)
h5 = net.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
h10 = net.addHost('h10', cls=Host, ip='10.0.0.10', defaultRoute=None)
h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
h6 = net.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)
h7 = net.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
h9 = net.addHost('h9', cls=Host, ip='10.0.0.9', defaultRoute=None)

info( '*** Add links\n')
net.addLink(s4, h10)
net.addLink(h1, s1)
net.addLink(h2, s1)
net.addLink(s1, s2)
net.addLink(s2, h3)
net.addLink(s2, h4)
net.addLink(s2, s3)
net.addLink(s3, s4)
net.addLink(s3, h5)
net.addLink(s3, h6)
net.addLink(s3, h7)
net.addLink(s4, h8)
net.addLink(s4, h9)
```

Figure 3.4: Screenshot of Simulation Topology Script

troller, and each host is given a unique ID for identification.

3.7 Detection Metrics

The metrics used to detect black hole attack are as follows:

3.7.1 Throughput

Throughput refers to the speed at which a node may send packets over the network. It is, in fact, the total number of packets successfully transmitted from the origin to the destination. Following is the formula for throughput calculation:

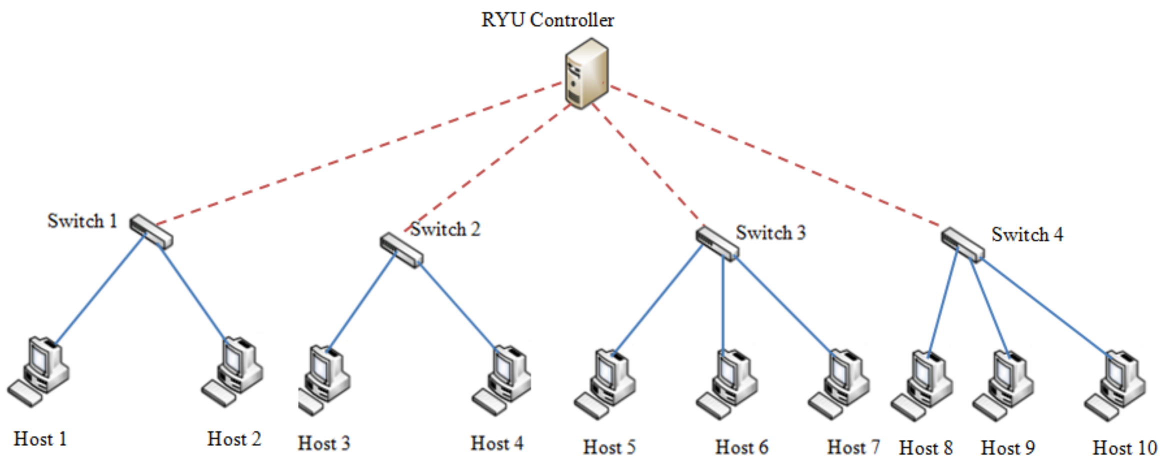


Figure 3.5: Topology in Miniedit

$$\text{Throughput} = \frac{(Total\ packets\ successfully\ delivered\ to\ destination)}{(Total\ packets\ sent\ by\ source)}$$

3.7.2 End to End Delay

It is the time taken by a packet to travel from its sender to its receiver , including routing, release, and transmission delays is known as end to end delay.

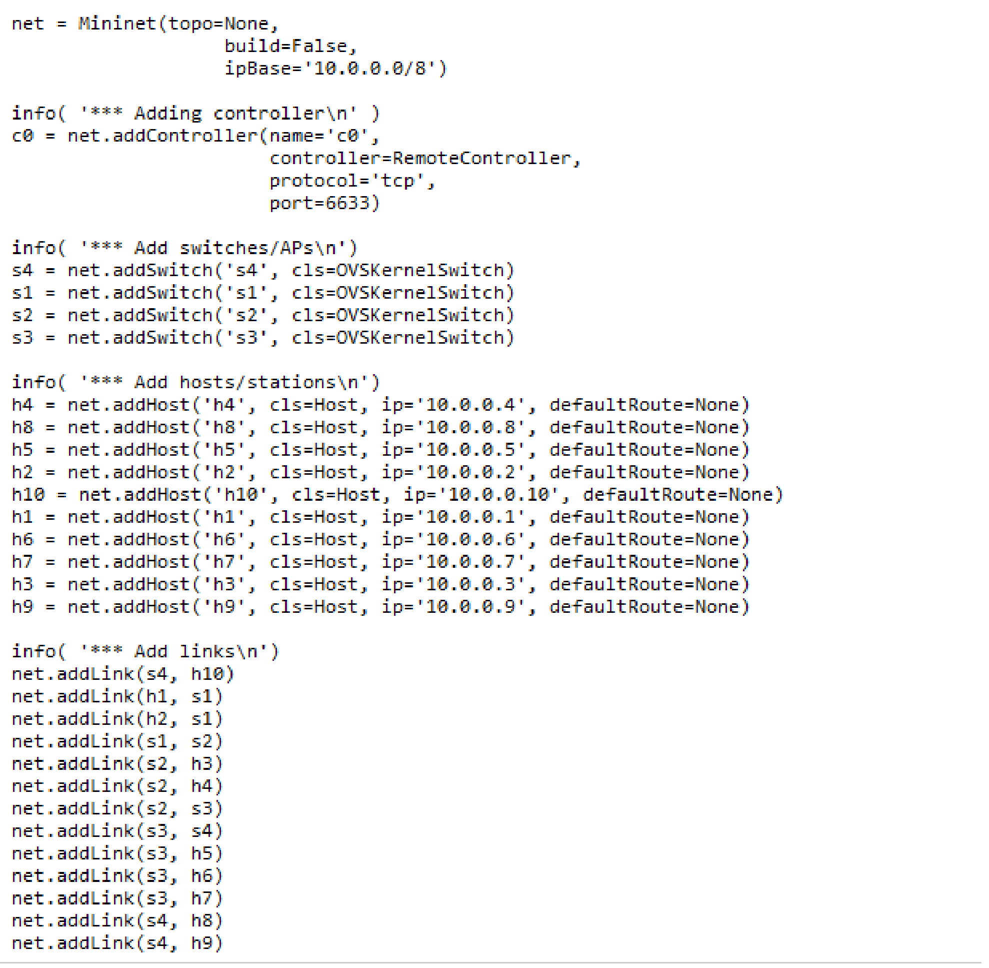
3.7.3 Packet Delivery Ratio

Packet Delivery Ratio refers to the average number of packets that successfully reach their destination in a given amount of time. The formula for PDR is :

$$\text{Packet Delivery Ratio} = \frac{(Total\ packets\ at\ source)}{(Total\ packets\ received\ at\ destination)}$$

3.8 Anomaly Detection

Anomaly detection is technique of identifying events in dataset that do not follow the expected pattern. These occurrences are referred to as anomalies or outliers. In real-time applications, anomaly detection algorithms have been widely employed to discover



```
net = Mininet(topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

info( '*** Adding controller\n' )
c0 = net.addController(name='c0',
                       controller=RemoteController,
                       protocol='tcp',
                       port=6633)

info( '*** Add switches/APs\n')
s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
s3 = net.addSwitch('s3', cls=OVSKernelSwitch)

info( '*** Add hosts/stations\n')
h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
h8 = net.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)
h5 = net.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
h10 = net.addHost('h10', cls=Host, ip='10.0.0.10', defaultRoute=None)
h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
h6 = net.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)
h7 = net.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
h9 = net.addHost('h9', cls=Host, ip='10.0.0.9', defaultRoute=None)

info( '*** Add links\n')
net.addLink(s4, h10)
net.addLink(h1, s1)
net.addLink(h2, s1)
net.addLink(s1, s2)
net.addLink(s2, h3)
net.addLink(s2, h4)
net.addLink(s2, s3)
net.addLink(s3, s4)
net.addLink(s3, h5)
net.addLink(s3, h6)
net.addLink(s3, h7)
net.addLink(s4, h8)
net.addLink(s4, h9)
```

Figure 3.6: Mininet Topology Code Snippet

unexpected patterns in a system. Establishing areas of regular behaviour in the data is a straightforward approach of detecting anomalies, and cases that do not fit into this category are referred to as anomalies. The anomaly detection technique selected is influenced by the type of accessible information, its format (labeled/unlabeled), and nature of anomalies to be identify. We employed a method in which we set detection metric thresholds and repeated our experiment 100 times in both normal and black hole attack circumstances. When these metrics exceed our predefined threshold, an alert is generated, and the malicious node that is dropping/absorbing packets is removed from the routing path, and a new route is chosen.

Chapter 4

Implementation

4.1 Proposed Methodology

4.1.1 Algorithm

The essential messages exchanged in a typical scenario have been included to provide a clear idea of our suggested solution.

- Let's say that host h9 wishes to communicate with host h1. To register, both hosts h9 and h1 must send Packet-In messages to SDN controller which logs information about the host in host registration module and assigns a unique ID to each host when it receives the Packet-In message (h9 and h1 in our implementation).
- Secondly, as host h9 has no idea about how to contact host h1, it makes a connection request to the controller to inquire about host1's details. This request is sent as an RREQ message to all hosts.
- When the RREQ message is received by the destination, i.e., host h9, a RREP packet is sent back to oriinator i.e., 1 in tis example.
- According to the network topology, the controller obtains three possible paths and selects the best path after estimating path latency and capacity etc.

- After sending a DATA message to host h1, host h1 responds with a REPLYDATA message to host h9. All four of the messages are used to establish a connection.
- After connection establishment, h9 is now able to communicate with host h1 through a series of trusted ops and in the meanwhile values detection parameters are recorded and if they there is an anamoly in the range of values alert is generated and connection is terminated.

4.1.2 Detection Parameters

The metrics used in our work are as follows

Throughput The average number of packets that are successfully delivered is called throughput. We calculate throughput by formula: $\frac{(Packet\ size)(Number\,of\ packets\ received)(8)}{End\ to\ End\ Delay}$

End to End Delay The average time from the time of transmission of packet from the origin to the time it is deduced in the destination, which includes routing, release, and transmission time [33]. It is given by: End time - Start time

Packet Delivery Ratio The average number of packets that successfully reach destination in a given time is refered to as Packet Delivery Ratio(PDR). It is given by: $\frac{(Number\,of\ packets\ received)(100)}{Number\,of\ packets\ sent}$

4.2 Simulation Environment

The details of our simulation environment used in our implementation and experiments are as follows:

4.2.1 System Specifications

The test setup was done on Intel® Core^{TM} DELL laptop with i7-7500U CPU @2.70GHz and 2.90GHz processor with 8GB of RAM. We used a virtual machine to create Ubuntu 14.04 for the Linux environment. We used the Oracle VM Virtual box as the virtual machine emulator. For making the testbed, we used Mininet to create the testbed where one "remote controller" was used with a custom topology with four switches and ten hosts. The switches were Open vSwitches with OpenFlow protocol. The controller was POX.

4.2.2 Software Tools Used For Implementation and Experiments

Table 4.1: Software Tools in Implementation and Experiments

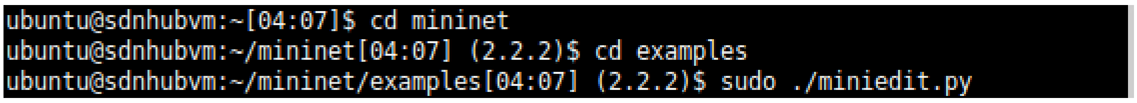
Software	Function	Version
Mininet	Network Emulator	2.2.2
MiniEdit	Graphical User Interface	2.2.0.1
Open vSwitch	Software SDN Switch	2.3.90
POX	SDN Controller Platform	0.5.0
Oracle VirtualBox	Virtualisation	6.0.4
Linux(Ubuntu)	Host Operatin System	14.04
Python	Programming Language	2.7.6

VirtualBox We can use Oracle's free software to help us create a virtual network environment[34]. A full operating system can run within another with its help. We also used VirtualBox to instal Mininet VM, which makes switching from Windows to Ubuntu a breeze.

Mininet It's an SDN-based software emulator for creating a realistic virtual topology in an SDN environment [35]. It allows larger network prototypes to be created on a single workstation without the requirement for an actual network connection. Mininet can be used in OpenFlow switches to imitate network topology. It includes a virtual machine, OpenFlow binaries, and kernel configuration tools pre-installed.

MiniEdit MiniEdit, a small network editor, is included in the Mininet network simulator. MiniEdit is a demonstration tool that shows how Mininet can be expanded. By pressing the "run" button, you can easily drag and drop switches and hosts to build a live, functioning network. MiniEdit is launched in an xterm.

The MiniEdit script is located in Mininet's examples folder,fig 4.1 shows the command to run it.

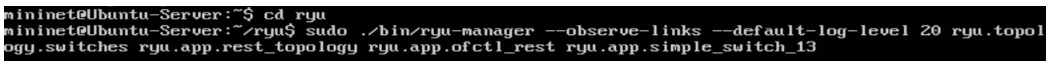


```
ubuntu@sdnhubvm:~[04:07]$ cd mininet
ubuntu@sdnhubvm:~/mininet[04:07] (2.2.2)$ cd examples
ubuntu@sdnhubvm:~/mininet/examples[04:07] (2.2.2)$ sudo ./miniedit.py
```

Figure 4.1: Command Run on MiniEdit GUI

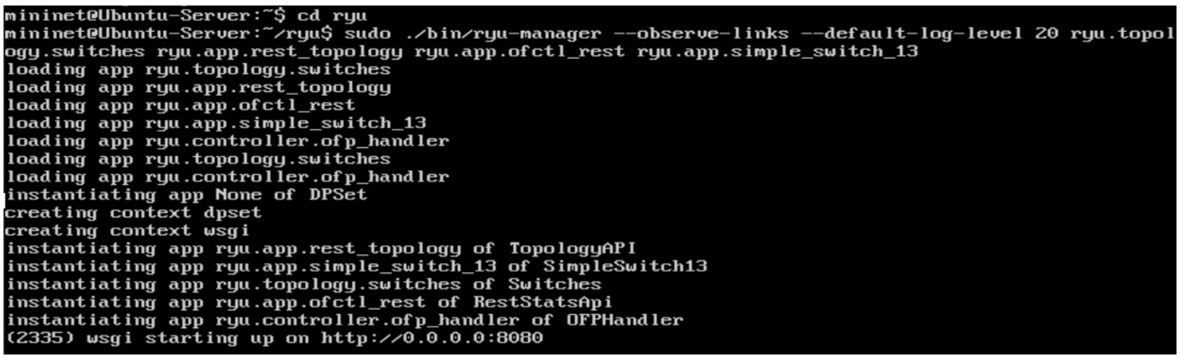
RYU It's a Python-based open controller that can turn OpenFlow into switch,a load balancer,a hub,a firewall, and various other devices. We can pass parameters into the topology in real time. There are provisions for developing custom components as well as stock components. RYU is a framework for establishing OpenFlow communication between SDN switches[39].

To start the RYU Controller, the command in fig 4.2 starts the controller. This makes controller up in our VM1 as shown in fig4.3



```
mininet@Ubuntu-Server:~$ cd ryu
mininet@Ubuntu-Server:~/ryu$ sudo ./bin/ryu-manager --observe-links --default-log-level 20 ryu.topol
ogy.switches ryu.app.rest_topology ryu.app.ofctl_rest ryu.app.simple_switch_13
```

Figure 4.2: Command To Start RYU Controller



```
mininet@Ubuntu-Server:~$ cd ryu
mininet@Ubuntu-Server:~/ryu$ sudo ./bin/ryu-manager --observe-links --default-log-level 20 ryu.topol
ogy.switches ryu.app.rest_topology ryu.app.ofctl_rest ryu.app.simple_switch_13
loading app ryu.topology.switches
loading app ryu.app.rest_topology
loading app ryu.app.ofctl_rest
loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_topology of TopologyAPI
instantiating app ryu.app.simple_switch_13 of SimpleSwitch13
instantiating app ryu.topology.switches of Switches
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app ryu.controller.ofp_handler of OFPHandler
(2335) wsgi starting up on http://0.0.0.0:8080
```

Figure 4.3: RYU Controller started

Ping Internet Control Message Protocol (ICMP) is a system administrator's tool. It calculates the round-trip time and displays the total number of packets transmitted, received, and lost, as well as the lowest, maximum, and average response times.

Python Scripting We have used python programming language for scripting with SDN controller application for detection and Mininet simulator. Some of the python libraries ,modules and API which are used in our work are Random(accessing iterable python object randomly) , python socket API(for TCP and UDP host communication interfaces) , blessings(API to manipulate terminals for hosts),matplotlib(library for graphical plotting), subprocess (module that to run external programs and inspect their outputs). datetime (module to work with dates as date objects,NumPy library(to work wit arrays)

4.2.3 Simulation Topology

We have defined 8 custom topologies. In first topology (fig 4.4), we have 3 hosts and two switches. In second topology (fig 4.5), we have 4 hosts and 2 switches. In the third topology (fig 4.6), we have 5 hosts and 3 switches.In the fourth topology (fig 4.7), we have 6 hosts and 3 switches. In fifth topology (fig 4.8), we have 7 hosts and 3 switches. In sixth topology (fig 4.9), we have 8 hosts and 4 switches.In our seventh topology (fig 4.10), we have 9 hosts and 4 switches and our last topology (fig 4.11), has 10 hosts with 4 switches These topologies are made in miniedit GUI.

4.3 Implementation Modules

The flow module is described in this section in three tiers. We investigate an SDN-based network architecture with a control layer, a data layer, and a host cluster, as depicted in Figure. The control layer is in charge of the SDN controller's route allocation of flows. SDN switches make up the data layer. The host cluster communicates with the control layer and transmits data.

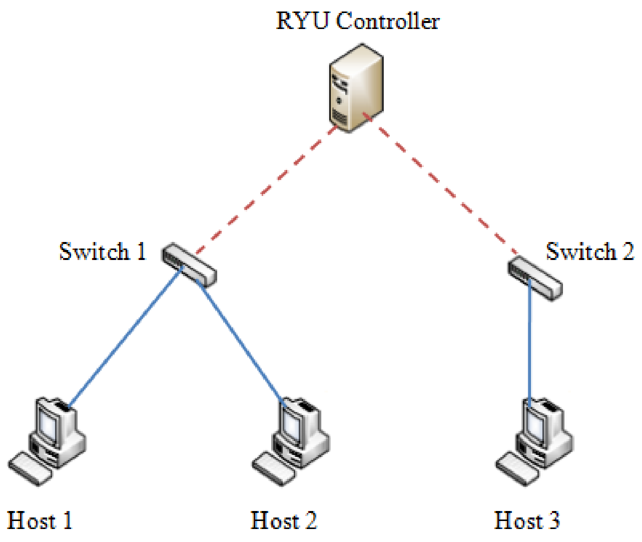


Figure 4.4: Simulation Topology 1 in MiniEdit

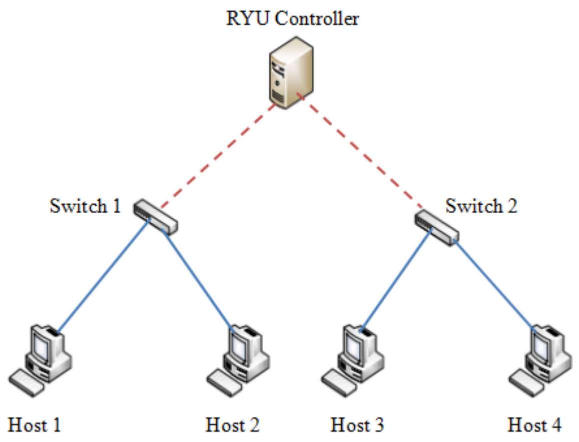


Figure 4.5: Simulation Topology 2 in MiniEdit

4.3.1 Controller

The SDN controller is used to implement information collecting, path calculation, path selection, flow management, and attack detection in the control layer. Our control layer's modules are as follows:

In order to establish a connection, the connection initiator sends a message to the controller, inquiring about the destination hosts. According to several SDN investigations, these attacks affect all of the current key SDN controllers (e.g., Floodlight, Open Day-

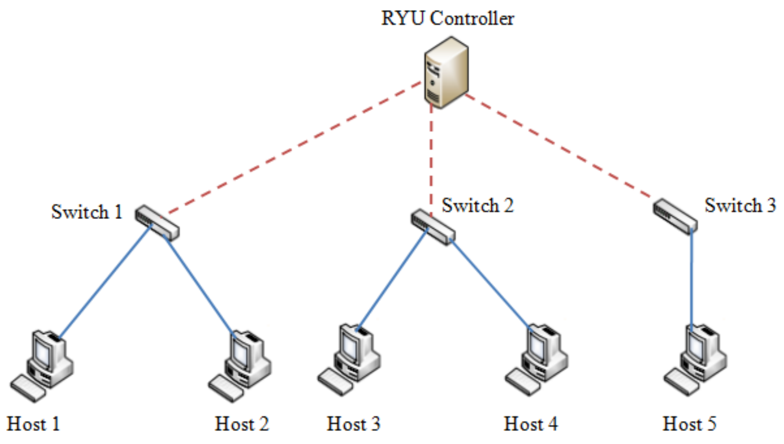


Figure 4.6: Simulation Topology 3 in MiniEdit

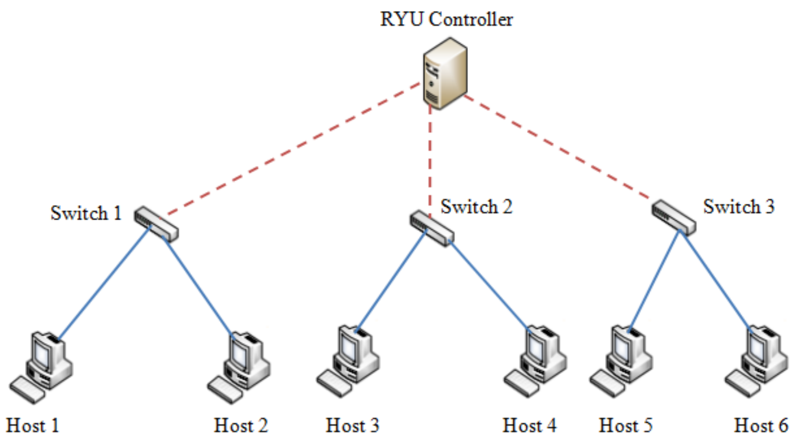


Figure 4.7: Simulation Topology 4 in MiniEdit

light, Beacon, and POX). All reliant network services are immediately harmed if such essential network design information is corrupted, perhaps leading in catastrophic challenges. The controller's routing services/apps, for example, can be changed to start a black hole. When a switch receives an LLDP packet from another switch, it sends the LLDP packet to the controller as a packet_in message for assistance because there is no analogous forwarding rule in the switch's FlowTable. The SDN controller can check whether switches are directly connected to one another and design the global topology after receiving Packet-in notifications via LLDP[33].

This module's main function is to discover network topology and collect link informa-

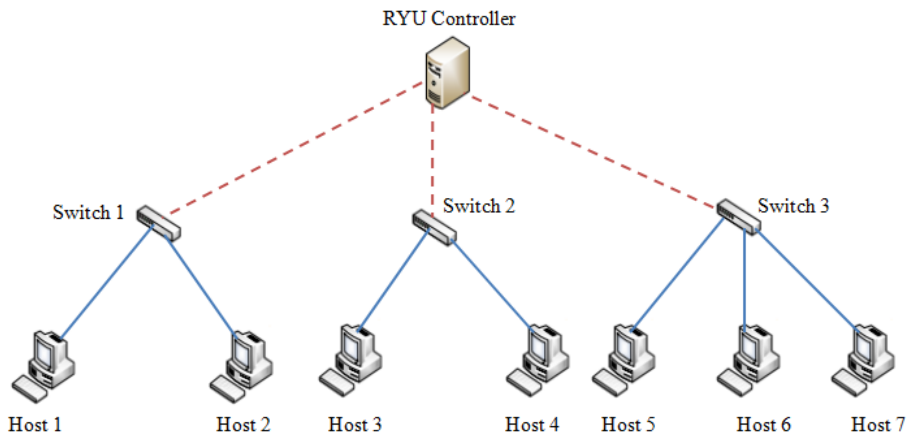


Figure 4.8: Simulation Topology 5 in MiniEdit

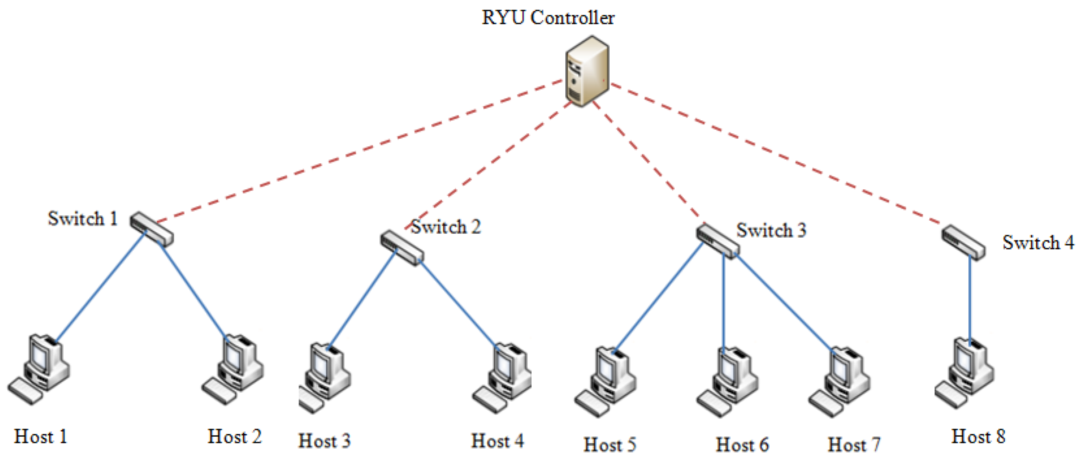


Figure 4.9: Simulation Topology 6 in MiniEdit

tion.

Because hosts have no idea how to get to destination hosts, when they want to connect to another host, they contact the controller for the destination hosts' information. In order to establish a connection, the connection initiator sends a message to the controller, inquiring about the destination hosts. The Connection Request Handler is a program that processes packets requesting information from hosts. In the diagram, host h9 is requesting a connection with host h1.

The path of transmission of the flow is calculated with the help of our routing mech-

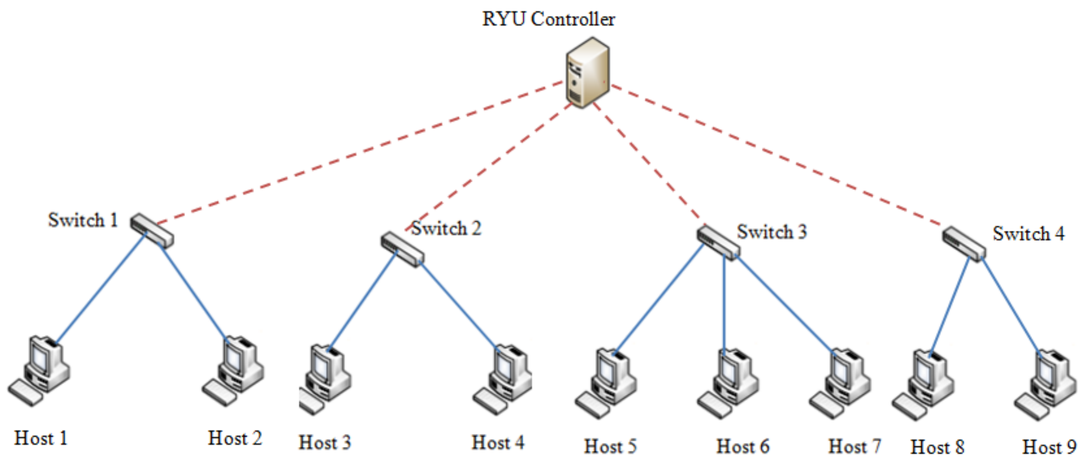


Figure 4.10: Simulation Topology 7 in MiniEdit

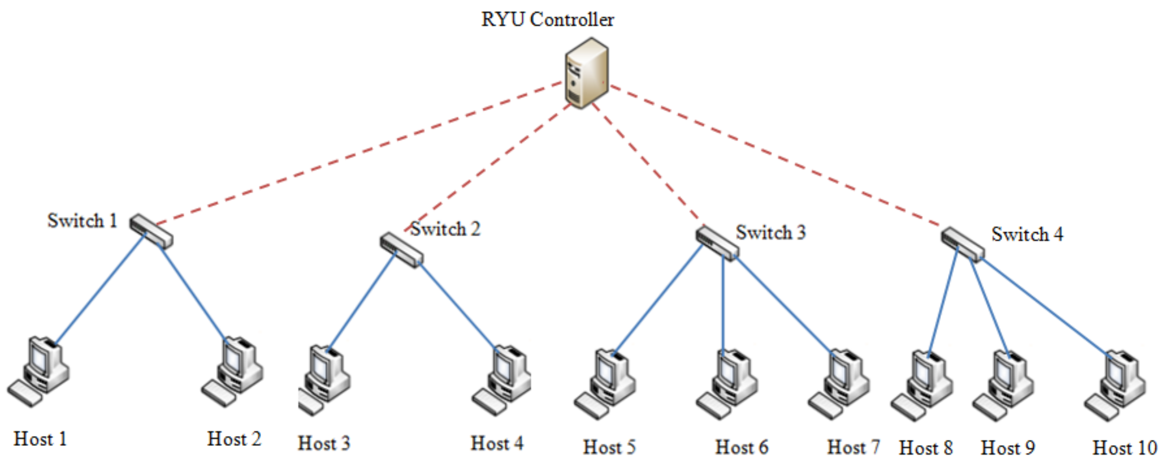


Figure 4.11: Simulation Topology 8 in MiniEdit

anism. On basis of paths available in present topology and information of the link collected from the Information Collection Module, paths of transmission having the best performance is selected.

4.4 Implementation Steps(Normal Mode)

First of all the controller is started in VM1 as shown in fig 4.14

Next we run one of our mininet topology scripts. Fig 4.15 shows screen-shot of our

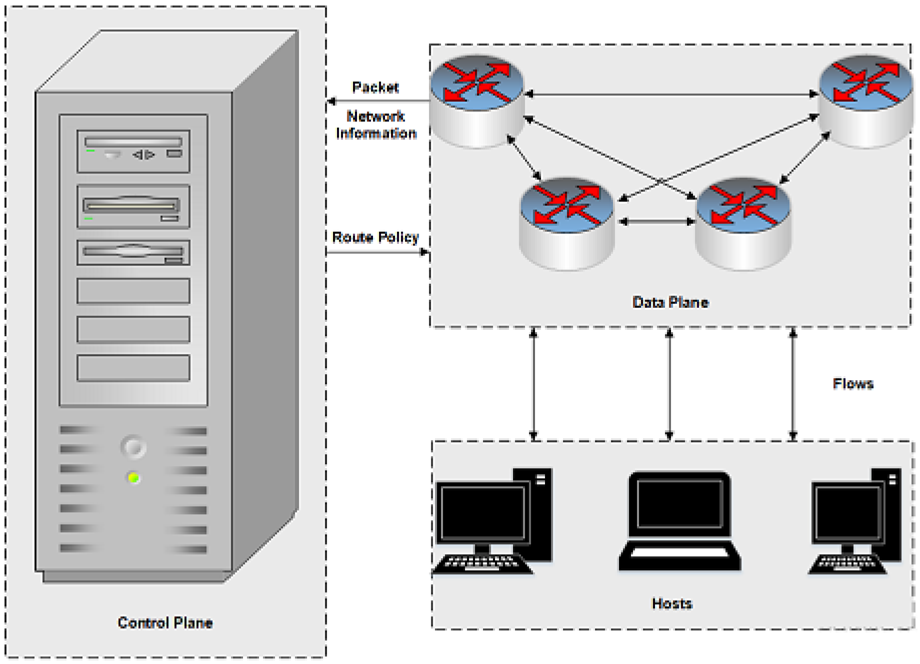
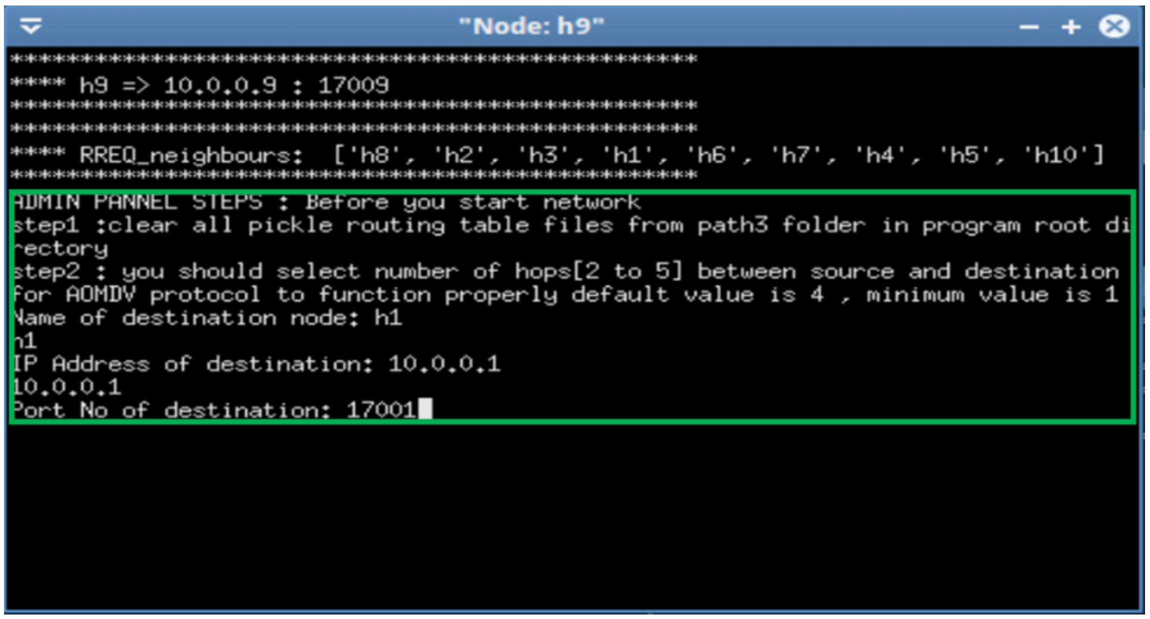


Figure 4.12: Flow management in our implementation



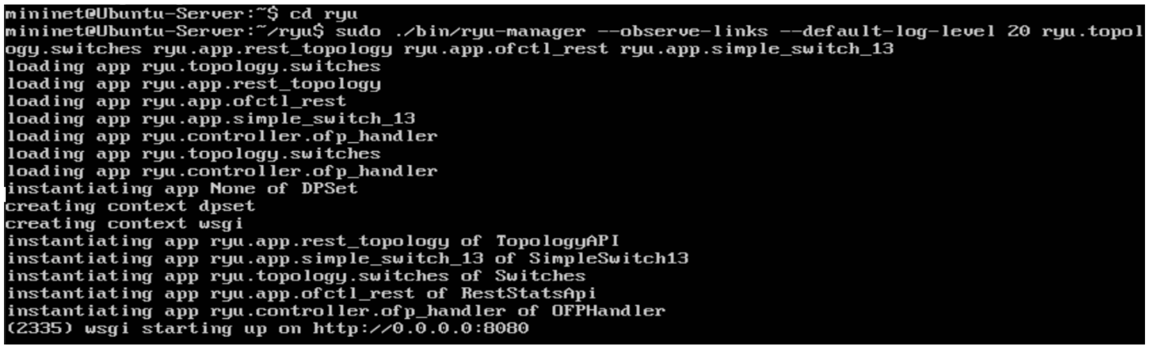
```
***************************************************
**** h9 => 10.0.0.9 : 17009
***************************************************
***************************************************
**** RREQ_neighbours:  ['h8', 'h2', 'h3', 'h1', 'h6', 'h7', 'h4', 'h5', 'h10']
***************************************************
ADMIN PANNEL STEPS : Before you start network
step1 :clear all pickle routing table files from path3 folder in program root di
rectory
step2 : you should select number of hops[2 to 5] between source and destination
for AOMDV protocol to function properly default value is 4 , minimum value is 1
Name of destination node: h1
h1
IP Address of destination: 10.0.0.1
10.0.0.1
Port No of destination: 17001
```

Figure 4.13: Host h9 requesting connection to host h1

collected from the Information Collection Module, paths of transmission having the best performance is selected.

4.4 Implementation Steps (Normal Mode)

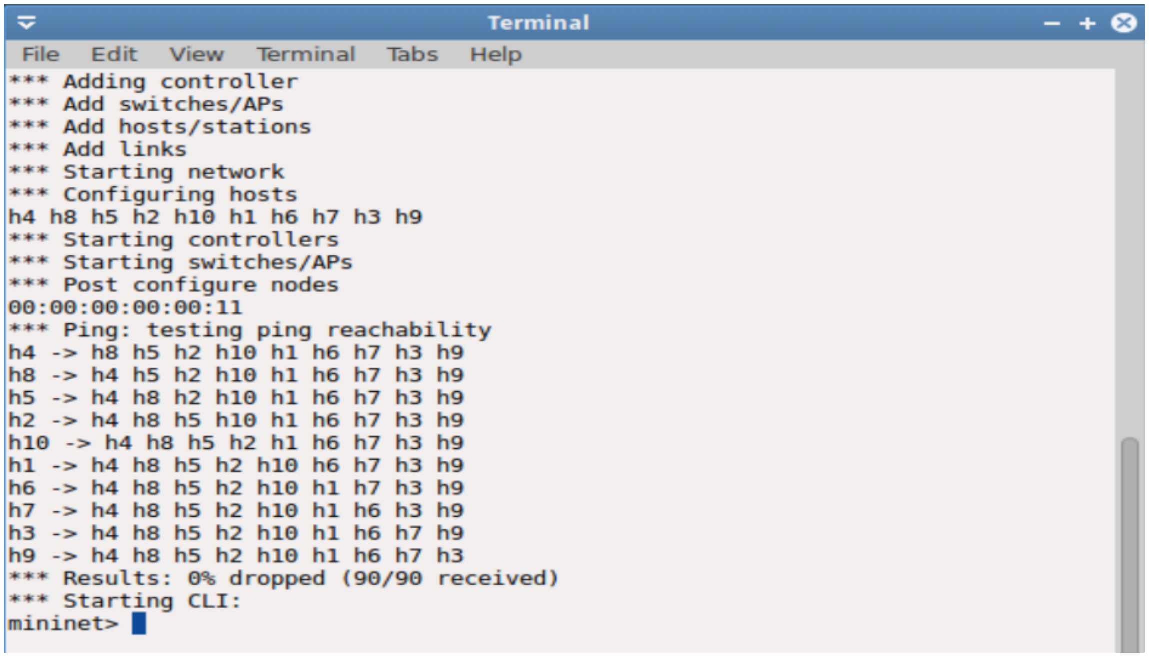
First of all the controller is started in VM1 as shown in fig 4.14



```
mininet@Ubuntu-Server:~$ cd ryu
mininet@Ubuntu-Server:~/ryu$ sudo ./bin/ryu-manager --observe-links --default-log-level 20 ryu.topol
ogy.switches ryu.app.rest_topology ryu.app.ofctl_rest ryu.app.simple_switch_13
loading app ryu.topology.switches
loading app ryu.app.rest_topology
loading app ryu.app.ofctl_rest
loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_topology of TopologyAPI
instantiating app ryu.app.simple_switch_13 of SimpleSwitch13
instantiating app ryu.topology.switches of Switches
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app ryu.controller.ofp_handler of OFPHandler
(2335) wsgi starting up on http://0.0.0.0:8080
```

Figure 4.14: Running controller

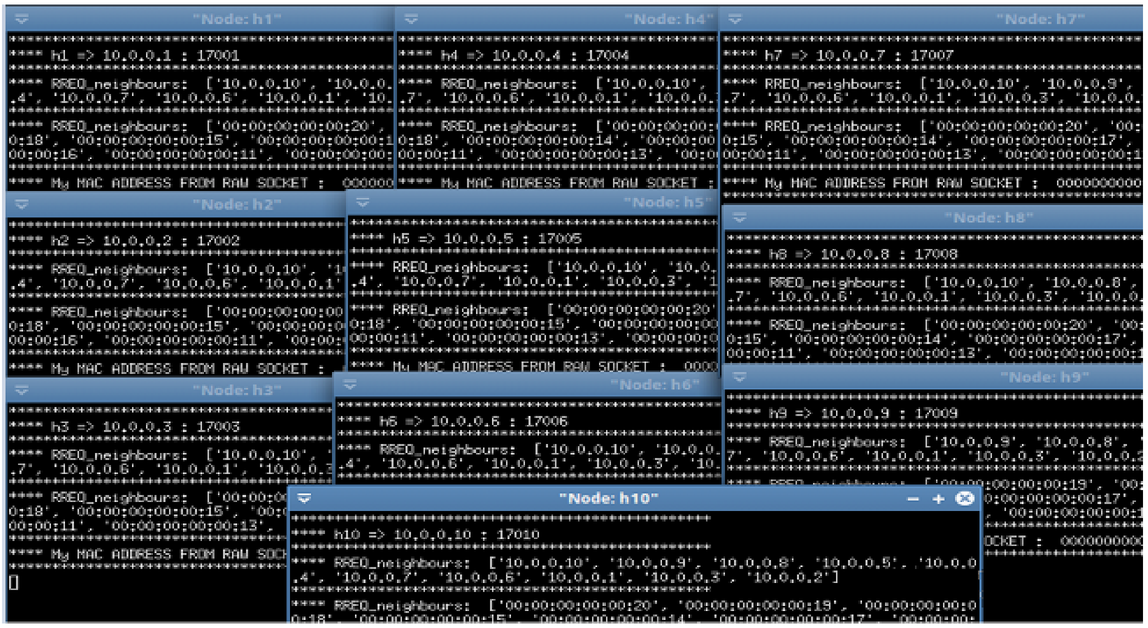
Next we run one of our mininet topology scripts. Fig 4.15 shows screen-shot of our topology with 10 hosts.



```
*** Adding controller
*** Add switches/APs
*** Add hosts/stations
*** Add links
*** Starting network
*** Configuring hosts
h4 h8 h5 h2 h10 h1 h6 h7 h3 h9
*** Starting controllers
*** Starting switches/APs
*** Post configure nodes
00:00:00:00:00:11
*** Ping: testing ping reachability
h4 -> h8 h5 h2 h10 h1 h6 h7 h3 h9
h8 -> h4 h5 h2 h10 h1 h6 h7 h3 h9
h5 -> h4 h8 h2 h10 h1 h6 h7 h3 h9
h2 -> h4 h8 h5 h10 h1 h6 h7 h3 h9
h10 -> h4 h8 h5 h2 h1 h6 h7 h3 h9
h1 -> h4 h8 h5 h2 h10 h6 h7 h3 h9
h6 -> h4 h8 h5 h2 h10 h1 h7 h3 h9
h7 -> h4 h8 h5 h2 h10 h1 h6 h3 h9
h3 -> h4 h8 h5 h2 h10 h1 h6 h7 h9
h9 -> h4 h8 h5 h2 h10 h1 h6 h7 h3
*** Results: 0% dropped (90/90 received)
*** Starting CLI:
mininet>
```

Figure 4.15: Running mininet topology script

After that terminals open up for each of the host in the topology executed , as shown in fig 4.16.



```
**********************************************
**** h1 => 10.0.0.1 : 17001
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '10.0.0.
.4', '10.0.0.7', '10.0.0.6', '10.0.0.1', '10.
**********************************************
**** RREQ_neighbours:  ['00:00:00:00:00:20',
0:18', '00:00:00:00:00:15', '00:00:00:00:00:1
00:00:16', '00:00:00:00:00:11', '00:00:00:00:
**********************************************
**** My MAC ADDRESS FROM RAW SOCKET :  000000
```

```
**********************************************
**** h4 => 10.0.0.4 : 17004
**********************************************
**** RREQ_neighbours:  ['10.0.0.10',
.7', '10.0.0.6', '10.0.0.1', '10.0.0.
**********************************************
**** RREQ_neighbours:  ['00:00:00:00:
0:18', '00:00:00:00:00:14', '00:00:00
00:00:11', '00:00:00:00:00:13', '00:0
**********************************************
**** My MAC ADDRESS FROM RAW SOCKET :
```

```
**********************************************
**** h7 => 10.0.0.7 : 17007
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '10.0.0.9',
.7', '10.0.0.6', '10.0.0.1', '10.0.0.3', '10.0.0.
**********************************************
**** RREQ_neighbours:  ['00:00:00:00:00:20', '00:
0:15', '00:00:00:00:00:14', '00:00:00:00:00:17',
00:00:11', '00:00:00:00:00:13', '00:00:00:00:00:1
**********************************************
**** My MAC ADDRESS FROM RAW SOCKET :  0000000000
**********************************************
```

```
**********************************************
**** h2 => 10.0.0.2 : 17002
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '1
.4', '10.0.0.7', '10.0.0.6', '10.0.0.1'
**********************************************
**** RREQ_neighbours:  ['00:00:00:00:00
0:18', '00:00:00:00:00:15', '00:00:00:0
00:00:16', '00:00:00:00:00:11', '00:00:
**********************************************
**** My MAC ADDRESS FROM RAW SOCKET :
```

```
**********************************************
**** h5 => 10.0.0.5 : 17005
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '10.0.
.4', '10.0.0.7', '10.0.0.1', '10.0.0.3', '1
**********************************************
**** RREQ_neighbours:  ['00:00:00:00:00:20'
0:18', '00:00:00:00:00:15', '00:00:00:00:00
00:00:11', '00:00:00:00:00:13', '00:00:00:0
**********************************************
**** My MAC ADDRESS FROM RAW SOCKET :  0000
```

```
**********************************************
**** h8 => 10.0.0.8 : 17008
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '10.0.0.8',
.7', '10.0.0.6', '10.0.0.1', '10.0.0.3', '10.0.0.
**********************************************
**** RREQ_neighbours:  ['00:00:00:00:00:20', '00:
0:15', '00:00:00:00:00:14', '00:00:00:00:00:17',
00:00:11', '00:00:00:00:00:13', '00:00:00:00:00:1
```

```
**********************************************
**** h3 => 10.0.0.3 : 17003
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '
.7', '10.0.0.6', '10.0.0.1', '10.0.0.3
**********************************************
**** RREQ_neighbours:  ['00:00:0
0:18', '00:00:00:00:00:15', '00:0
00:00:11', '00:00:00:00:00:13',
**********************************************
**** My MAC ADDRESS FROM RAW SOCK
**********************************************
[]
```

```
**********************************************
**** h6 => 10.0.0.6 : 17006
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '10.0.0.
.4', '10.0.0.6', '10.0.0.1', '10.0.0.3', '10.
**********************************************
```

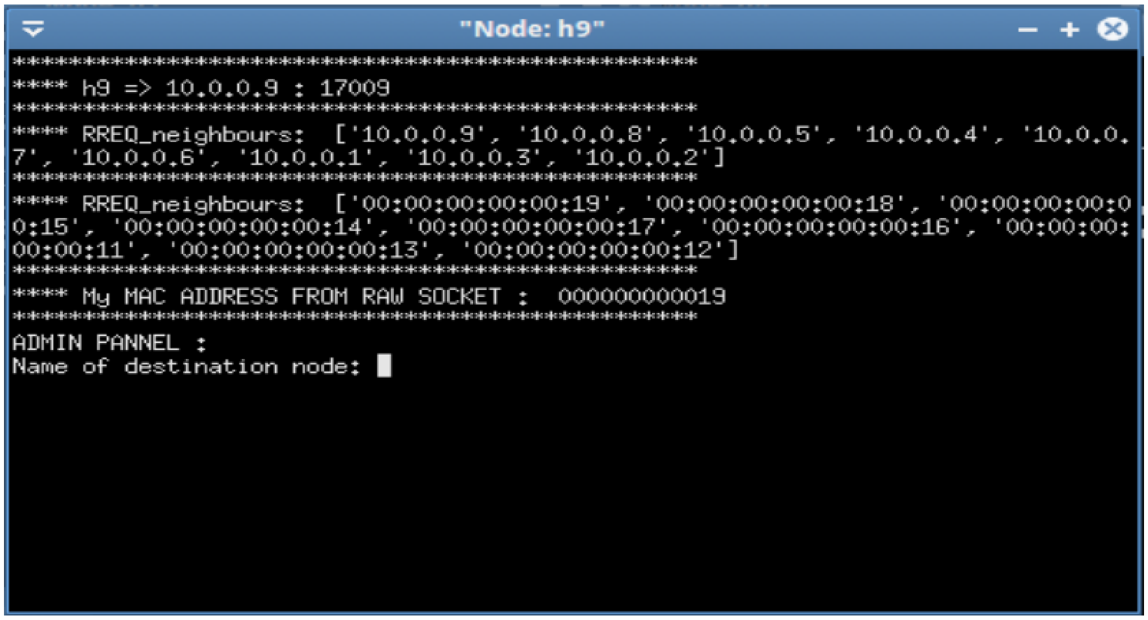
```
**********************************************
**** h9 => 10.0.0.9 : 17009
**********************************************
**** RREQ_neighbours:  ['10.0.0.9', '10.0.0.8', '
7', '10.0.0.6', '10.0.0.1', '10.0.0.3', '10.0.0.2
**********************************************
                                 :00:00:00:19', '00:
                         0:00:00:00:00:17',
                         , '00:00:00:00:00:1
**********************************************
                         OCKET :  0000000000
**********************************************
```

```
**********************************************
**** h10 => 10.0.0.10 : 17010
**********************************************
**** RREQ_neighbours:  ['10.0.0.10', '10.0.0.9', '10.0.0.8', '10.0.0.5', '10.0.0
.4', '10.0.0.7', '10.0.0.6', '10.0.0.1', '10.0.0.3', '10.0.0.2']
**********************************************
**** RREQ_neighbours:  ['00:00:00:00:00:20', '00:00:00:00:00:19', '00:00:00:00:0
0:18', '00:00:00:00:00:15', '00:00:00:00:00:14', '00:00:00:00:00:17', '00:00:00:
```

Figure 4.16: Terminals for all hosts in topology

On each terminal is printed its IP address, MAC address and its port number and the IP address and Mac address of the neighbors.

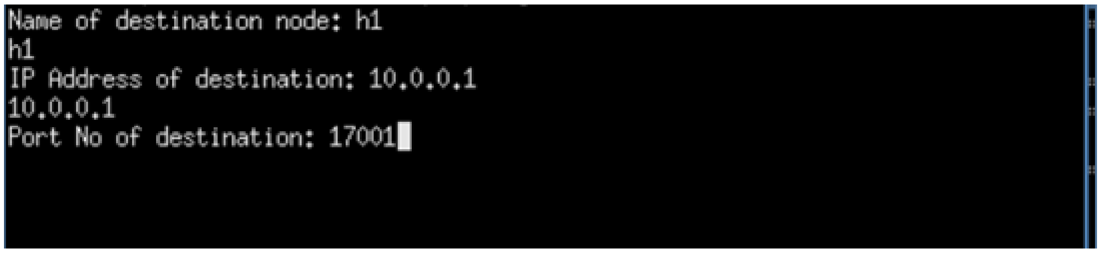
On the n-1 terminal, where n is the total number of hosts in the topology executed, the user has to input the ID, IP address and port number of the destination. Terminal n-1 i.e., h9 for topology with 10 hosts is shown in fig 4.17.



```
"Node: h9"
*****************************************************
**** h9 => 10.0.0.9 : 17009
*****************************************************
**** RREQ_neighbours:  ['10.0.0.9', '10.0.0.8', '10.0.0.5', '10.0.0.4', '10.0.0.
7', '10.0.0.6', '10.0.0.1', '10.0.0.3', '10.0.0.2']
*****************************************************
**** RREQ_neighbours:  ['00:00:00:00:00:19', '00:00:00:00:00:18', '00:00:00:00:0
0:15', '00:00:00:00:00:14', '00:00:00:00:00:17', '00:00:00:00:00:16', '00:00:00:
00:00:11', '00:00:00:00:00:13', '00:00:00:00:00:12']
*****************************************************
**** My MAC ADDRESS FROM RAW SOCKET :  000000000019
*****************************************************
ADMIN PANNEL :
Name of destination node: █
```

Figure 4.17: Terminal of n-1 host in ten host topology

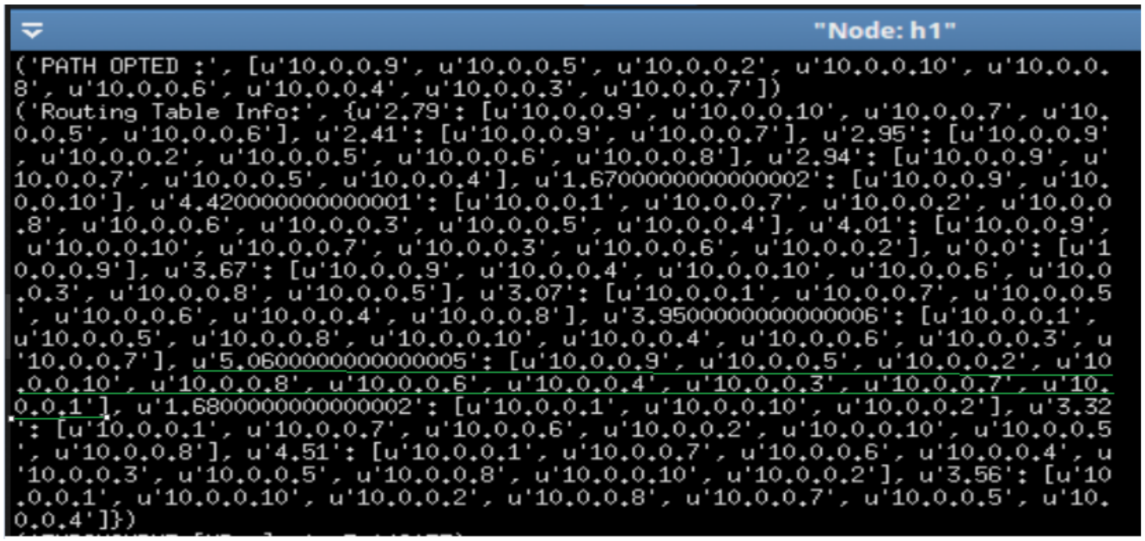
For explaining, we entered destination ID as h1, as shown in fig 4.18.



```
Name of destination node: h1
h1
IP Address of destination: 10.0.0.1
10.0.0.1
Port No of destination: 17001█
```

Figure 4.18: Entering destination details in terminal of n-1 host in ten host topology

After this routes are computed with their BL values. As underlined in fig 4.19, the path with maximum BL value is selected.

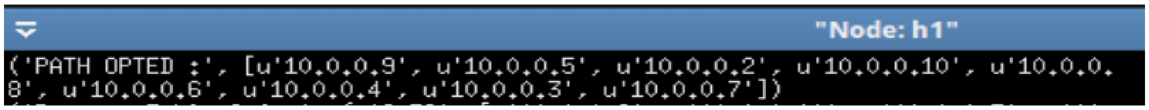


"Node: h1"

```
('PATH OPTED :', [u'10.0.0.9', u'10.0.0.5', u'10.0.0.2', u'10.0.0.10', u'10.0.0.
8', u'10.0.0.6', u'10.0.0.4', u'10.0.0.3', u'10.0.0.7'])
('Routing Table Info:', {u'2.79': [u'10.0.0.9', u'10.0.0.10', u'10.0.0.7', u'10.
0.0.5', u'10.0.0.6'], u'2.41': [u'10.0.0.9', u'10.0.0.7'], u'2.95': [u'10.0.0.9'
, u'10.0.0.2', u'10.0.0.5', u'10.0.0.6', u'10.0.0.8'], u'2.94': [u'10.0.0.9', u'
10.0.0.7', u'10.0.0.5', u'10.0.0.4'], u'1.6700000000000002': [u'10.0.0.9', u'10.
0.0.10'], u'4.420000000000001': [u'10.0.0.1', u'10.0.0.7', u'10.0.0.2', u'10.0.0
.8', u'10.0.0.6', u'10.0.0.3', u'10.0.0.5', u'10.0.0.4'], u'4.01': [u'10.0.0.9',
 u'10.0.0.10', u'10.0.0.7', u'10.0.0.3', u'10.0.0.6', u'10.0.0.2'], u'0.0': [u'1
0.0.0.9'], u'3.67': [u'10.0.0.9', u'10.0.0.4', u'10.0.0.10', u'10.0.0.6', u'10.0
.0.3', u'10.0.0.8', u'10.0.0.5'], u'3.07': [u'10.0.0.1', u'10.0.0.7', u'10.0.0.5
', u'10.0.0.6', u'10.0.0.4', u'10.0.0.8'], u'3.9500000000000006': [u'10.0.0.1',
u'10.0.0.5', u'10.0.0.8', u'10.0.0.10', u'10.0.0.4', u'10.0.0.6', u'10.0.0.3', u
'10.0.0.7'], u'5.0600000000000005': [u'10.0.0.9', u'10.0.0.5', u'10.0.0.2', u'10
.0.0.10', u'10.0.0.8', u'10.0.0.6', u'10.0.0.4', u'10.0.0.3', u'10.0.0.7', u'10.
0.0.1'], u'1.6800000000000002': [u'10.0.0.1', u'10.0.0.10', u'10.0.0.2'], u'3.32
': [u'10.0.0.1', u'10.0.0.7', u'10.0.0.6', u'10.0.0.2', u'10.0.0.10', u'10.0.0.5
', u'10.0.0.8'], u'4.51': [u'10.0.0.1', u'10.0.0.7', u'10.0.0.6', u'10.0.0.4', u
'10.0.0.3', u'10.0.0.5', u'10.0.0.8', u'10.0.0.10', u'10.0.0.2'], u'3.56': [u'10
.0.0.1', u'10.0.0.10', u'10.0.0.2', u'10.0.0.8', u'10.0.0.7', u'10.0.0.5', u'10.
0.0.4']})
```

Figure 4.19: Entering destination details in terminal of n-1 host in ten host topology

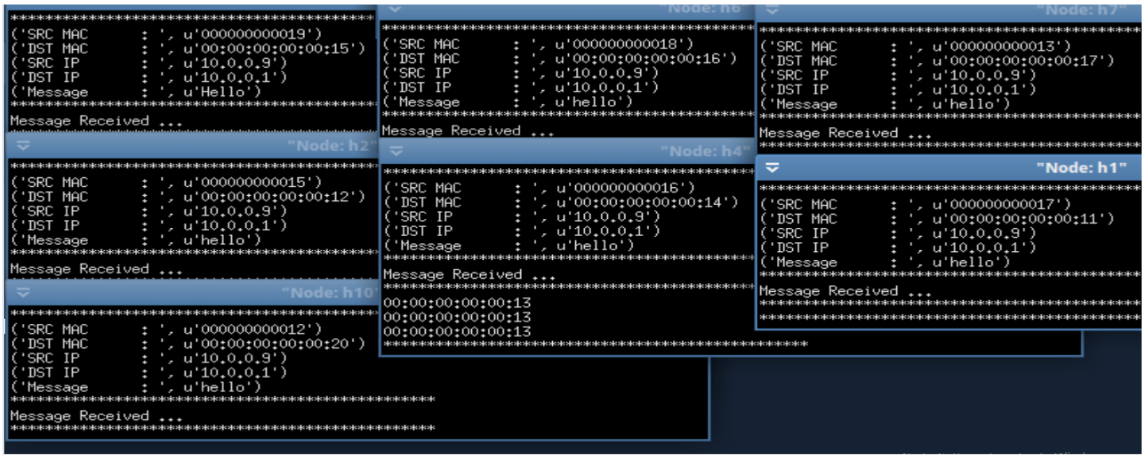
Next the packets pass from hop to hop in such a way that the MAC addresses of source and destination change on every hop. As shown in fig 4.20, the path opted is from h9(source)->h5->h2->h10->h8->h6->h4->h3->h7->h1(destination). Fig 4.21 shows message going from source terminal to destination terminal through the hops included in path opted in such a way that the MAC address of source and destination address are such that the MAC address of current hop is the source MAC address and the MAC address of the next hop is the destination MAC address. In this way the MAC addresses change at every hop.



"Node: h1"

```
('PATH OPTED :', [u'10.0.0.9', u'10.0.0.5', u'10.0.0.2', u'10.0.0.10', u'10.0.0.
8', u'10.0.0.6', u'10.0.0.4', u'10.0.0.3', u'10.0.0.7'])
```

Figure 4.20: Path selected in our test example



```
****************************************************
('SRC MAC      : ', u'000000000019')
('DST MAC      : ', u'00:00:00:00:00:15')
('SRC IP       : ', u'10.0.0.9')
('DST IP       : ', u'10.0.0.1')
('Message      : ', u'Hello')
****************************************************
Message Received ...
```

```
****************************************************
('SRC MAC      : ', u'000000000018')
('DST MAC      : ', u'00:00:00:00:00:16')
('SRC IP       : ', u'10.0.0.9')
('DST IP       : ', u'10.0.0.1')
('Message      : ', u'hello')
****************************************************
Message Received ...
```

```
****************************************************
('SRC MAC      : ', u'000000000013')
('DST MAC      : ', u'00:00:00:00:00:17')
('SRC IP       : ', u'10.0.0.9')
('DST IP       : ', u'10.0.0.1')
('Message      : ', u'hello')
****************************************************
Message Received ...
****************************************************
```

```
****************************************************
('SRC MAC      : ', u'000000000015')
('DST MAC      : ', u'00:00:00:00:00:12')
('SRC IP       : ', u'10.0.0.9')
('DST IP       : ', u'10.0.0.1')
('Message      : ', u'hello')
****************************************************
Message Received ...
```

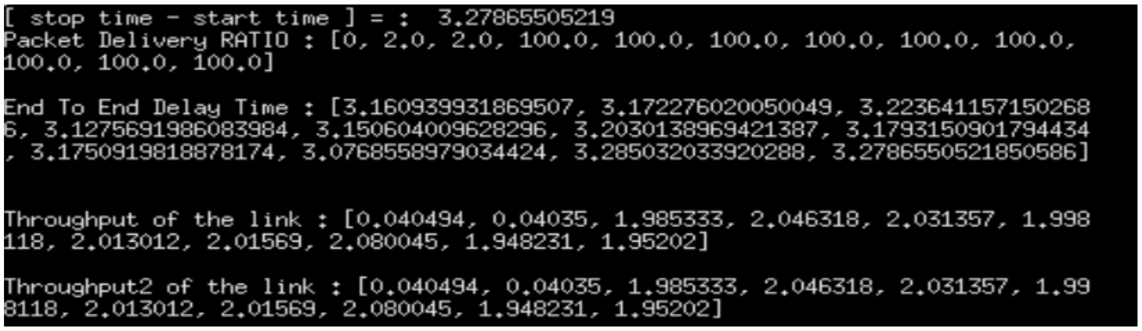
```
****************************************************
('SRC MAC      : ', u'000000000016')
('DST MAC      : ', u'00:00:00:00:00:14')
('SRC IP       : ', u'10.0.0.9')
('DST IP       : ', u'10.0.0.1')
('Message      : ', u'hello')
****************************************************
Message Received ...
****************************************************
00:00:00:00:00:13
00:00:00:00:00:13
00:00:00:00:00:13
************************************************************
```

```
****************************************************
('SRC MAC      : ', u'000000000017')
('DST MAC      : ', u'00:00:00:00:00:11')
('SRC IP       : ', u'10.0.0.9')
('DST IP       : ', u'10.0.0.1')
('Message      : ', u'hello')
****************************************************
Message Received ...
****************************************************
****************************************************
```

```
****************************************************
('SRC MAC      : ', u'000000000012')
('DST MAC      : ', u'00:00:00:00:00:20')
('SRC IP       : ', u'10.0.0.9')
('DST IP       : ', u'10.0.0.1')
('Message      : ', u'hello')
************************************************************
Message Received ...
************************************************************
```

Figure 4.21: Message traversing from source to destination through multiple hops

Measurements of parameters are done fig 4.22 and a graph gets plotted for the throughput, PDR and End to End Delay. We run experiments a number of times and record values to get our dataset for each of our topology.



```
[ stop time - start time ] = :  3.27865505219
Packet Delivery RATIO : [0, 2.0, 2.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,
100.0, 100.0, 100.0]

End To End Delay Time : [3.160939931869507, 3.172276020050049, 3.223641157150268
6, 3.1275691986083984, 3.150604009628296, 3.2030138969421387, 3.1793150901794434
, 3.1750919818878174, 3.0768558979034424, 3.285032033920288, 3.2786550521850586]


Throughput of the link : [0.040494, 0.04035, 1.985333, 2.046318, 2.031357, 1.998
118, 2.013012, 2.01569, 2.080045, 1.948231, 1.95202]

Throughput2 of the link : [0.040494, 0.04035, 1.985333, 2.046318, 2.031357, 1.99
8118, 2.013012, 2.01569, 2.080045, 1.948231, 1.95202]
```

Figure 4.22: Parameter Measurements

Black hole mode

Our goal is to simulate a Black-hole attack. Since we have to make a node work selfishly and drop packets so we set variable ‘black hole’ as true in our routing script and it starts dropping packets. Once the values of parameters fall below the threshold a black hole alert gets generated and black hole node is removed from path.

4.5 Results and Analysis

We sent packets from source to destination using our multi-hop routing scheme under normal operation and in case of black hole attack. The values of the performance parameters were recorded and their graphs were plotted using matplotlib. It was seen that in case of black hole attack the values of parameters fall out of the threshold boundaries. When the black hole attack was detected and malicious node was removed the graphs show that traffic resumed its normal operation after a short period of time.

4.5.1 Graphs and Values of Detection Parameters under Normal Operation

The experiment was run 10 times under normal operation for normal topology and under black hole attack. The values of detection parameters for ten parameters are shown in table below:

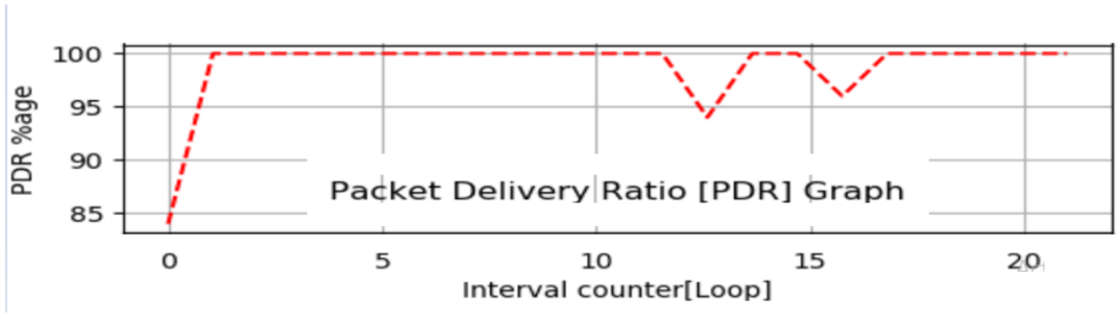


Figure 4.23: PDR of 3 host topology

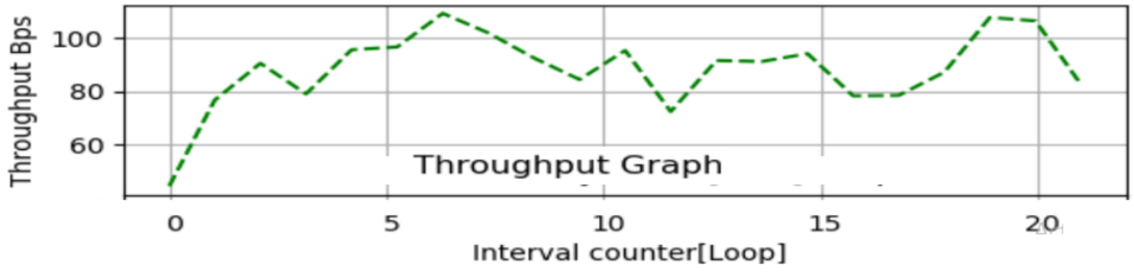


Figure 4.24: Throughput of 3 host topology

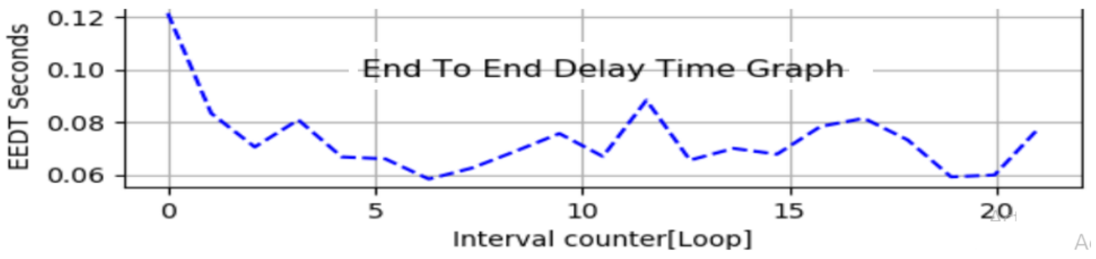
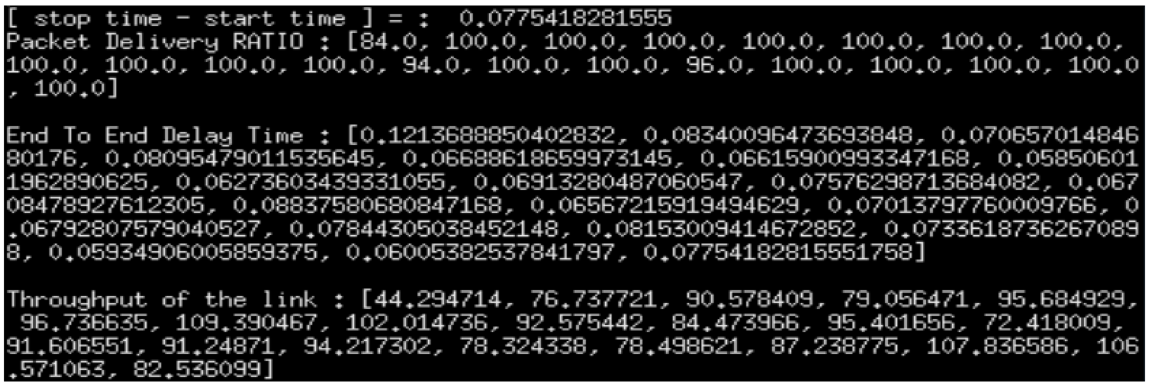


Figure 4.25: End to End Delay of 3 host topology



```
[ stop time - start time ] = :  0.0775418281555
Packet Delivery RATIO : [84.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,
100.0, 100.0, 100.0, 100.0, 94.0, 100.0, 100.0, 96.0, 100.0, 100.0, 100.0, 100.0
, 100.0]

End To End Delay Time : [0.1213688850402832, 0.08340096473693848, 0.070657014846
80176, 0.08095479011535645, 0.06688618659973145, 0.06615900993347168, 0.05850601
1962890625, 0.06273603439331055, 0.06913280487060547, 0.07576298713684082, 0.067
08478927612305, 0.08837580680847168, 0.06567215919494629, 0.07013797760009766, 0
.06792807579040527, 0.07844305038452148, 0.08153009414672852, 0.0733618736267089
8, 0.05934906005859375, 0.06005382537841797, 0.07754182815551758]

Throughput of the link : [44.294714, 76.737721, 90.578409, 79.056471, 95.684929,
 96.736635, 109.390467, 102.014736, 92.575442, 84.473966, 95.401656, 72.418009,
91.606551, 91.24871, 94.217302, 78.324338, 78.498621, 87.238775, 107.836586, 106
.571063, 82.536099]
```

Figure 4.26: Parameter values of 3 host topology

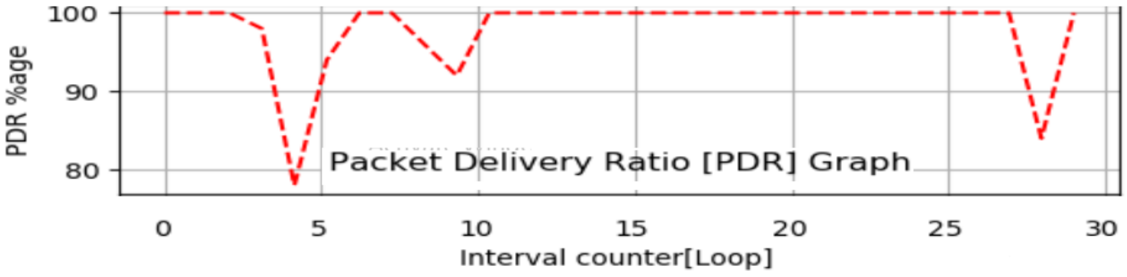


Figure 4.27: PDR of 4 host topology

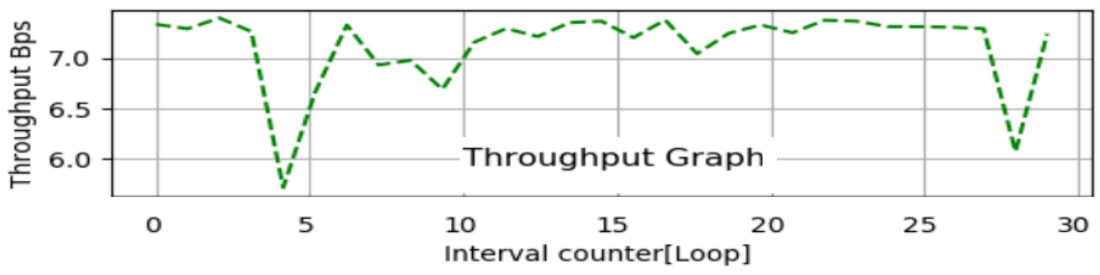


Figure 4.28: Throughput of 4 host topology

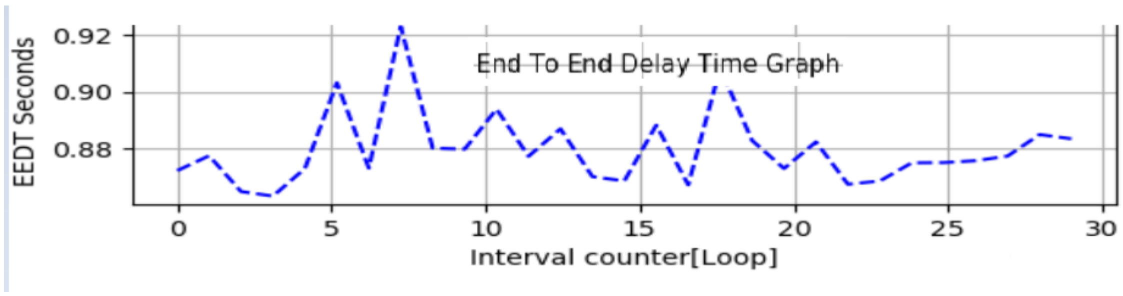
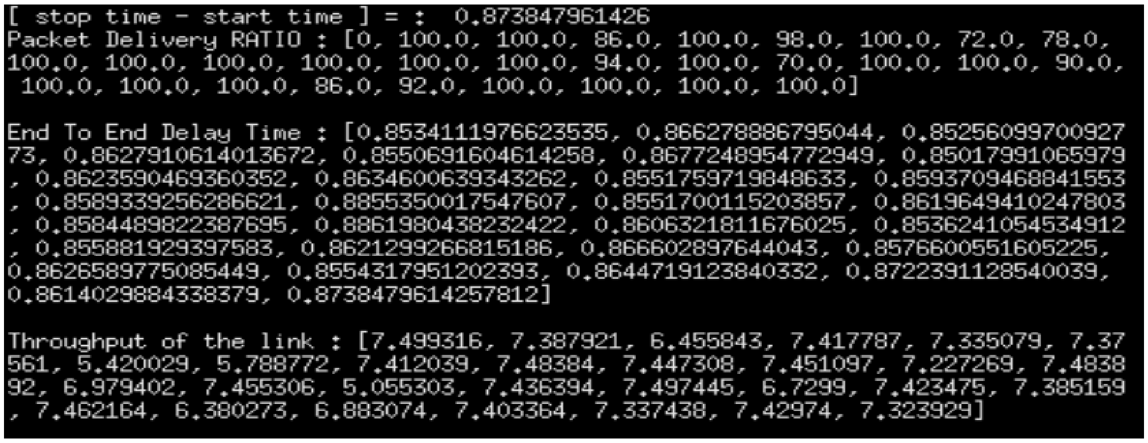


Figure 4.29: End to End Delay of 4 host topology



```
[ stop time - start time ] = :  0.873847961426
Packet Delivery RATIO : [0, 100.0, 100.0, 86.0, 100.0, 98.0, 100.0, 72.0, 78.0,
100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 94.0, 100.0, 70.0, 100.0, 100.0, 90.0,
 100.0, 100.0, 100.0, 86.0, 92.0, 100.0, 100.0, 100.0, 100.0]

End To End Delay Time : [0.8534111976623535, 0.866278886795044, 0.85256099700927
73, 0.8627910614013672, 0.8550691604614258, 0.8677248954772949, 0.85017991065979
, 0.8623590469360352, 0.8634600639343262, 0.8551759719848633, 0.8593709468841553
, 0.858933925628662l, 0.8855350017547607, 0.8551700115203857, 0.8619649410247803
, 0.8584489822387695, 0.8861980438232422, 0.8606321811676025, 0.8536241054534912
, 0.855881929397583, 0.8621299266815186, 0.866602897644043, 0.8576600551605225,
0.8626589775085449, 0.8554317951202393, 0.8644719123840332, 0.8722391128540039,
0.8614029884338379, 0.8738479614257812]

Throughput of the link : [7.499316, 7.387921, 6.455843, 7.417787, 7.335079, 7.37
561, 5.420029, 5.788772, 7.412039, 7.48384, 7.447308, 7.451097, 7.227269, 7.4838
92, 6.979402, 7.455306, 5.055303, 7.436394, 7.497445, 6.7299, 7.423475, 7.385159
, 7.462164, 6.380273, 6.883074, 7.403364, 7.337438, 7.42974, 7.323929]
```

Figure 4.30: Parameter values of 4 host topology

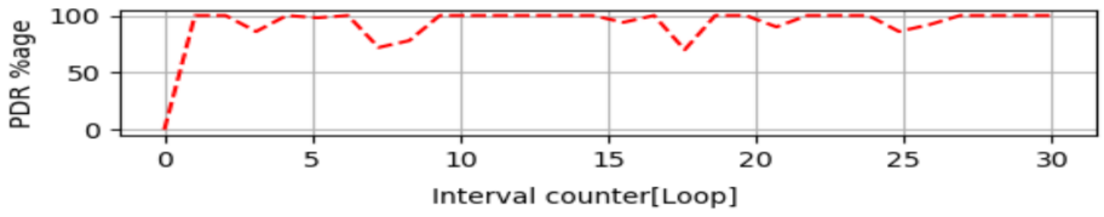


Figure 4.31: PDR of 5 host topology

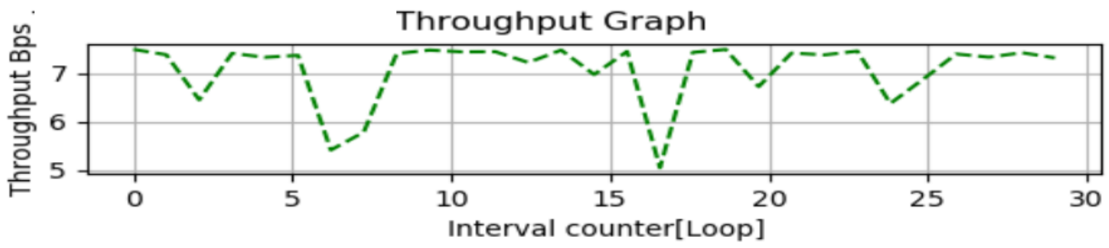


Figure 4.32: Throughput of 5 host topology

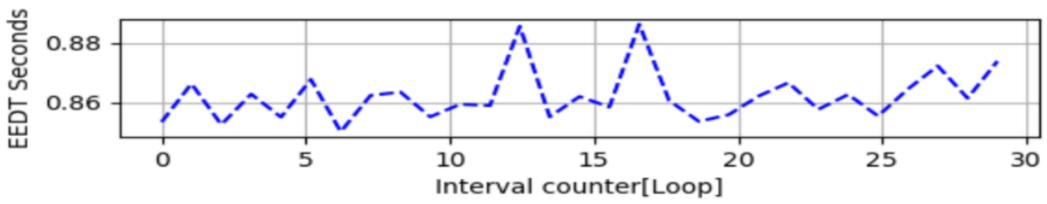
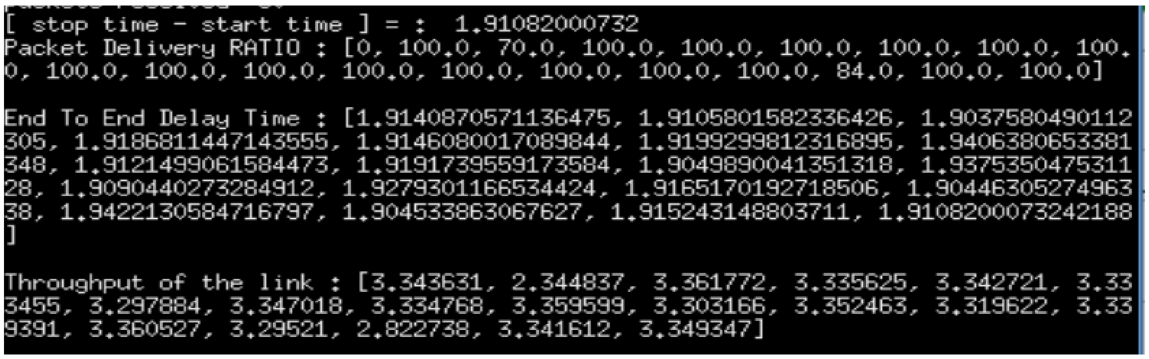


Figure 4.33: End to End Delay of 5 host topology



```
[ stop time - start time ] = :  1.91082000732
Packet Delivery RATIO : [0, 100.0, 70.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.
0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 84.0, 100.0, 100.0]

End To End Delay Time : [1.9140870571136475, 1.9105801582336426, 1.9037580490112
305, 1.9186811447143555, 1.9146080017089844, 1.9199299812316895, 1.9406380653381
348, 1.9121499061584473, 1.9191739559173584, 1.9049890041351318, 1.9375350475311
28, 1.9090440273284912, 1.9279301166534424, 1.916517019271850б, 1.9044630527496З
38, 1.9422130584716797, 1.904533863067627, 1.915243148803711, 1.9108200073242188
]

Throughput of the link : [3.343631, 2.344837, 3.361772, 3.335625, 3.342721, 3.33
3455, 3.297884, 3.347018, 3.334768, 3.359599, 3.303166, 3.352463, 3.319622, 3.33
9391, 3.360527, 3.29521, 2.822738, 3.341612, 3.349347]
```

Figure 4.34: Parameter values of 5 host topology

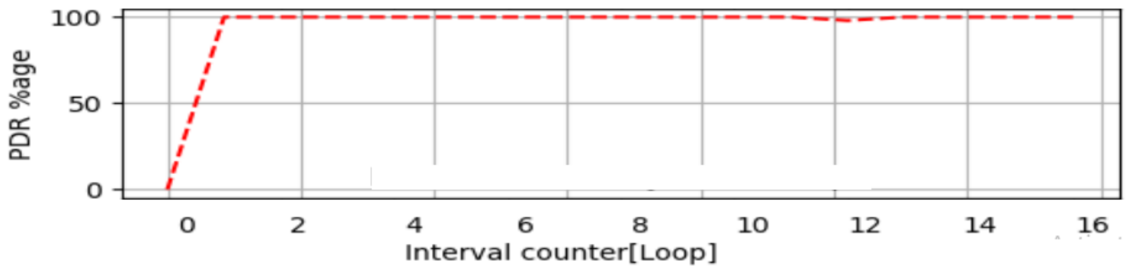


Figure 4.35: PDR of 6 host topology

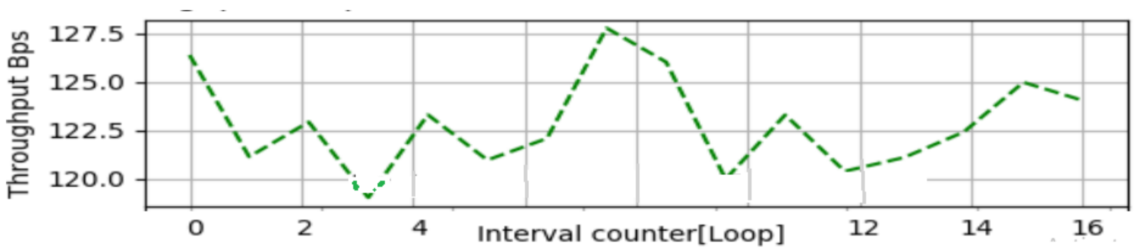


Figure 4.36: Throughput of 6 host topology

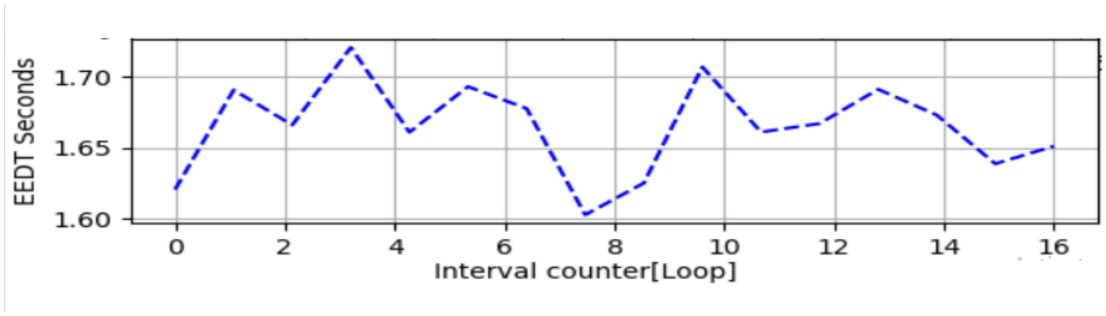
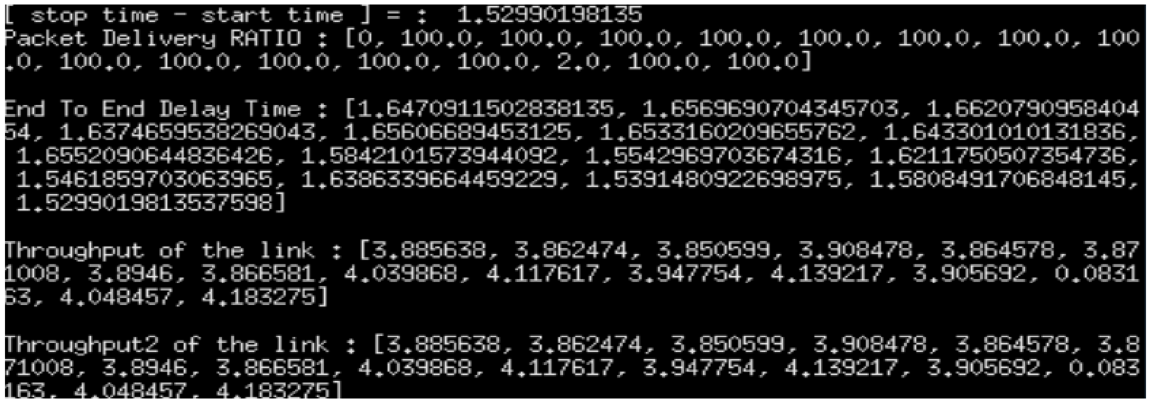


Figure 4.37: End to End Delay of 6 host topology



```
[ stop time - start time ] = :  1.52990198135
Packet Delivery RATIO : [0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100
.0, 100.0, 100.0, 100.0, 100.0, 100.0, 2.0, 100.0, 100.0]

End To End Delay Time : [1.6470911502838135, 1.6569690704345703, 1.6620790958404
54, 1.6374659538269043, 1.65606689453125, 1.6533160209655762, 1.643301010131836,
 1.6552090644836426, 1.5842101573944092, 1.5542969703674316, 1.6211750507354736,
 1.5461859703063965, 1.6386339664459229, 1.5391480922698975, 1.5808491706848145,
 1.5299019813537598]

Throughput of the link : [3.885638, 3.862474, 3.850599, 3.908478, 3.864578, 3.87
1008, 3.8946, 3.866581, 4.039868, 4.117617, 3.947754, 4.139217, 3.905692, 0.0831
63, 4.048457, 4.183275]

Throughput2 of the link : [3.885638, 3.862474, 3.850599, 3.908478, 3.864578, 3.8
71008, 3.8946, 3.866581, 4.039868, 4.117617, 3.947754, 4.139217, 3.905692, 0.083
163, 4.048457, 4.183275]
```

Figure 4.38: Parameter values of 6 host topology

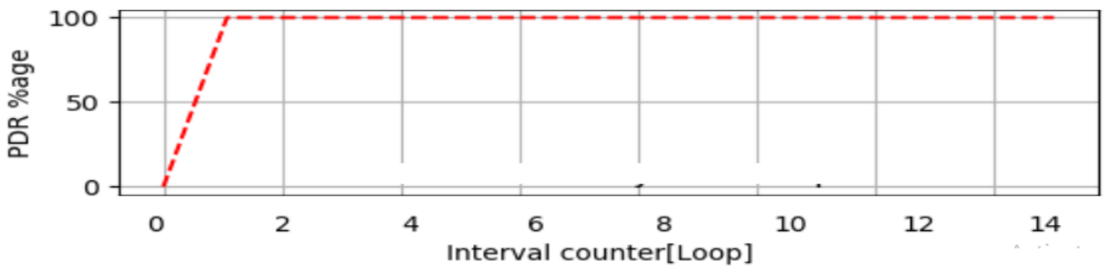


Figure 4.39: PDR of 7 host topology

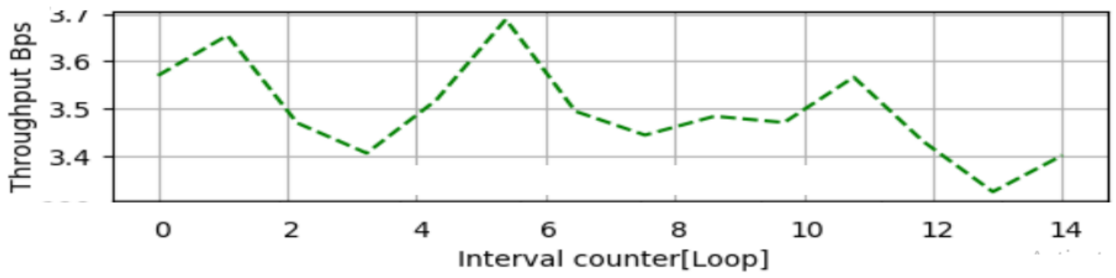


Figure 4.40: Throughput of 7 host topology

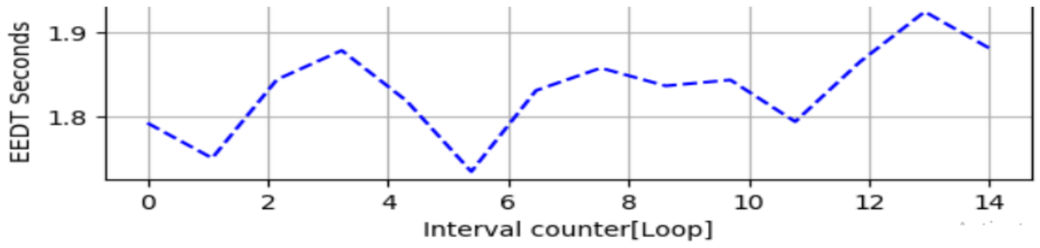
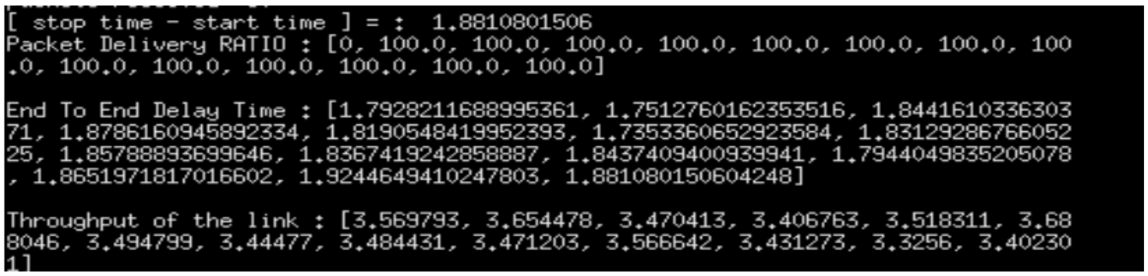


Figure 4.41: End to End Delay of 7 host topology


```
[ stop time - start time ] = :  1.8810801506
Packet Delivery RATIO : [0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100
.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]

End To End Delay Time : [1.7928211688995361, 1.7512760162353516, 1.844161033630З
71, 1.8786160945892334, 1.8190548419952393, 1.7353360652923584, 1.83129286766052
25, 1.857888936996446, 1.8367419242858887, 1.8437409400939941, 1.794404983520507８
, 1.8651971817016602, 1.9244649410247803, 1.881080150604248]

Throughput of the link : [3.569793, 3.654478, 3.470413, 3.406763, 3.518311, 3.68
8046, 3.494799, 3.44477, 3.484431, 3.471203, 3.566642, 3.431273, 3.3256, 3.40230
1]
```

Figure 4.42: Parameter values of 7 host topology

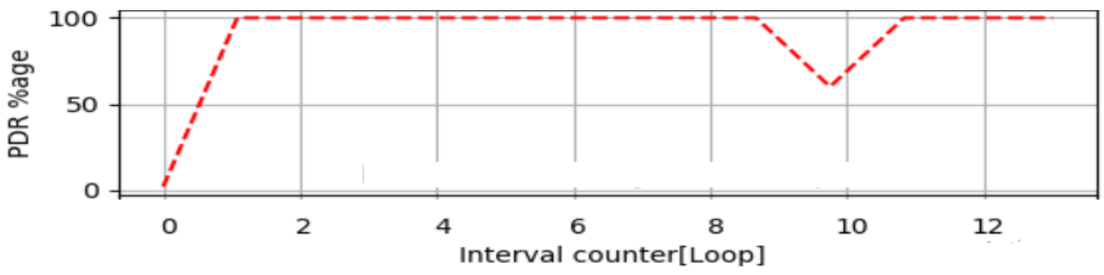


Figure 4.43: PDR of 8 host topology

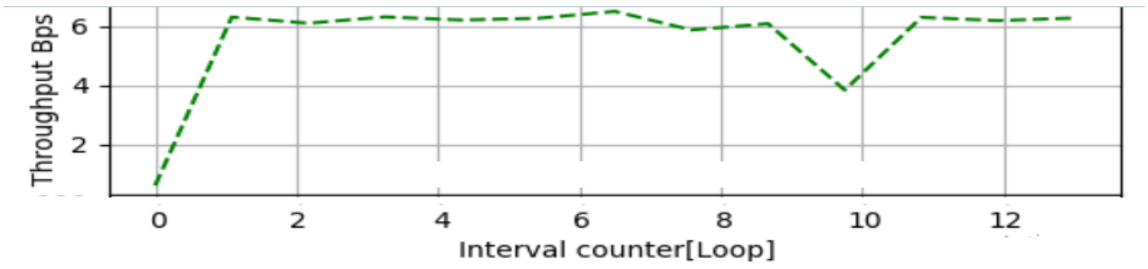


Figure 4.44: Throughput of 8 host topology

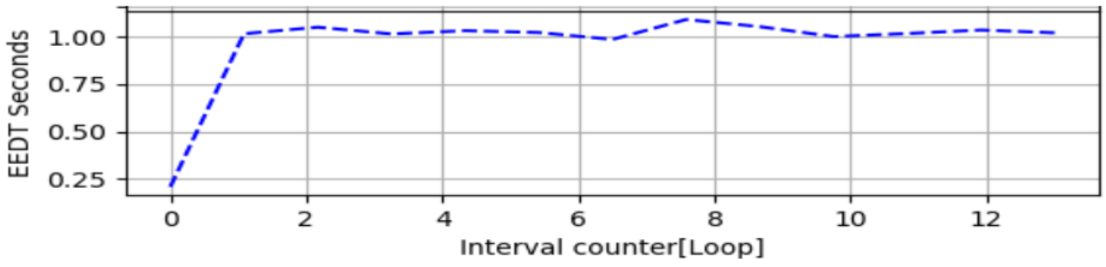
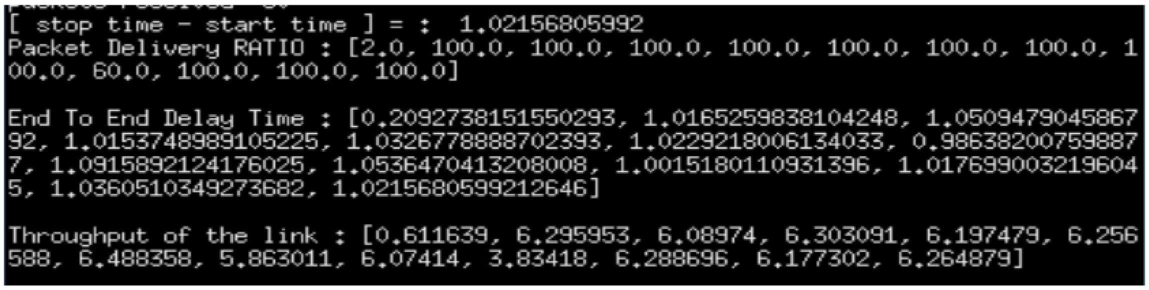


Figure 4.45: End to End Delay of 8 host topology



```
[ stop time - start time ] = :  1.02156805992
Packet Delivery RATIO : [2.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 1
00.0, 60.0, 100.0, 100.0, 100.0]

End To End Delay Time : [0.2092738151550293, 1.0165259838104248, 1.0509479045867
92, 1.0153748989105225, 1.0326778888702393, 1.0229218006134033, 0.98638200759887
7, 1.0915892124176025, 1.0536470413208008, 1.0015180110931396, 1.017699003219604
5, 1.0360510349273682, 1.0215680599212646]

Throughput of the link : [0.611639, 6.295953, 6.08974, 6.303091, 6.197479, 6.256
588, 6.488358, 5.863011, 6.07414, 3.83418, 6.288696, 6.177302, 6.264879]
```

Figure 4.46: Parameter values of 8 host topology

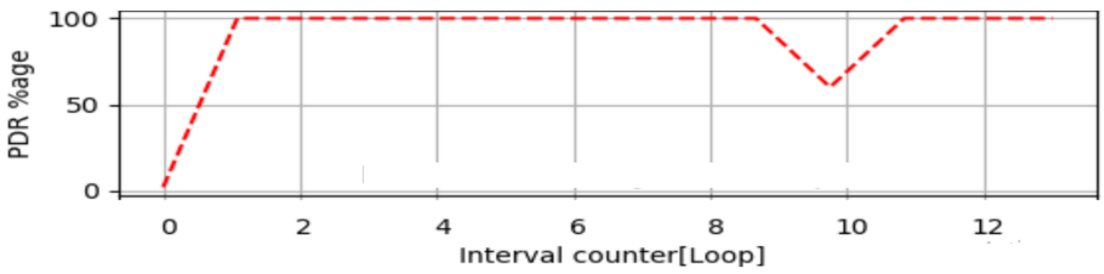


Figure 4.47: PDR of 9 host topology

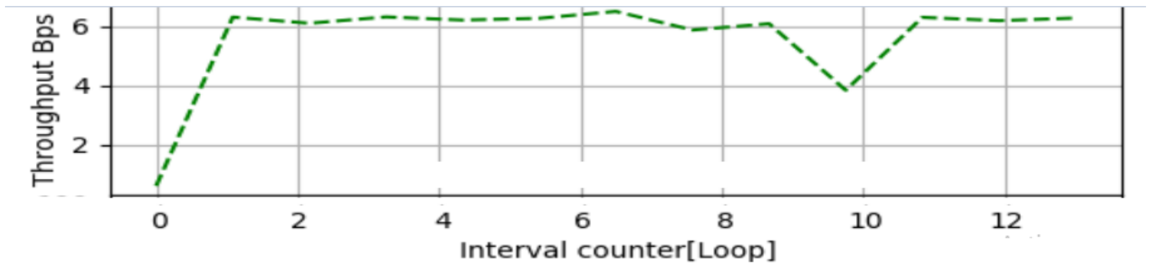


Figure 4.48: Throughput of 9 host topology

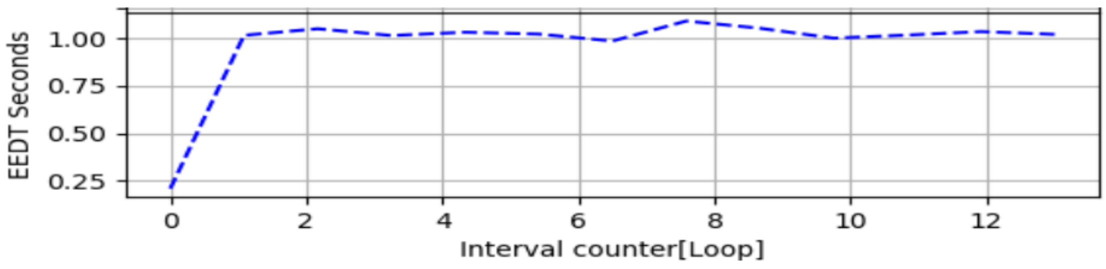
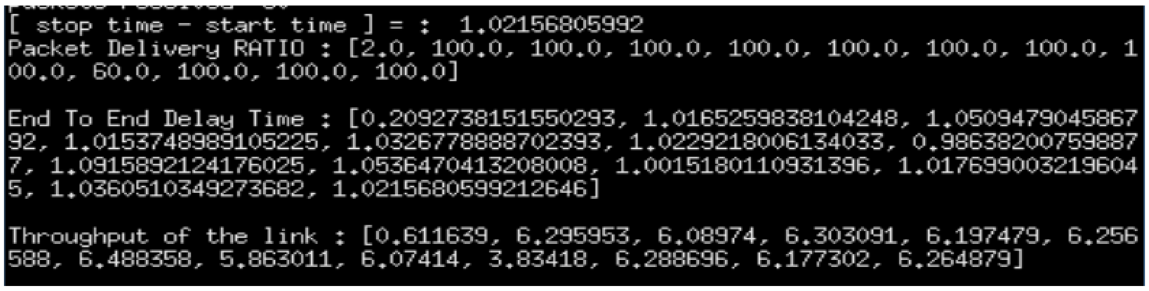


Figure 4.49: End to End Delay of 9 host topology



```
[ stop time - start time ] = :  1.02156805992
Packet Delivery RATIO : [2.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 1
00.0, 60.0, 100.0, 100.0, 100.0]

End To End Delay Time : [0.2092738151550293, 1.0165259838104248, 1.0509479045867
92, 1.0153748989105225, 1.0326778888702393, 1.0229218006134033, 0.98638200759887
7, 1.0915892124176025, 1.0536470413208008, 1.0015180110931396, 1.017699003219604
5, 1.0360510349273682, 1.0215680599212646]

Throughput of the link : [0.611639, 6.295953, 6.08974, 6.303091, 6.197479, 6.256
588, 6.488358, 5.863011, 6.07414, 3.83418, 6.288696, 6.177302, 6.264879]
```

Figure 4.50: Parameter values of 9 host topology

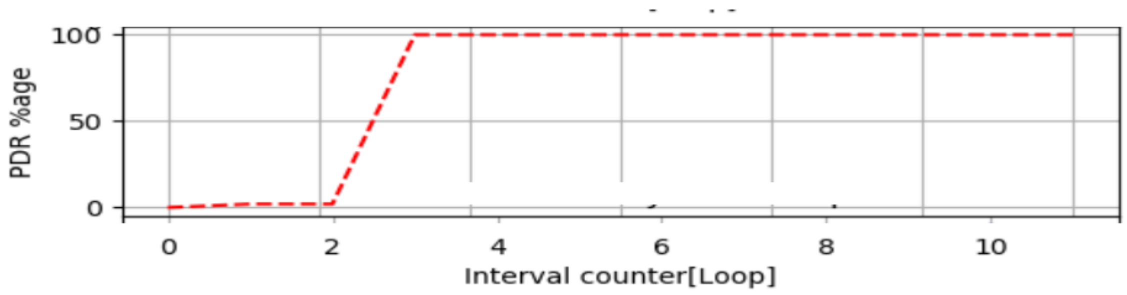


Figure 4.51: PDR of 10 host topology

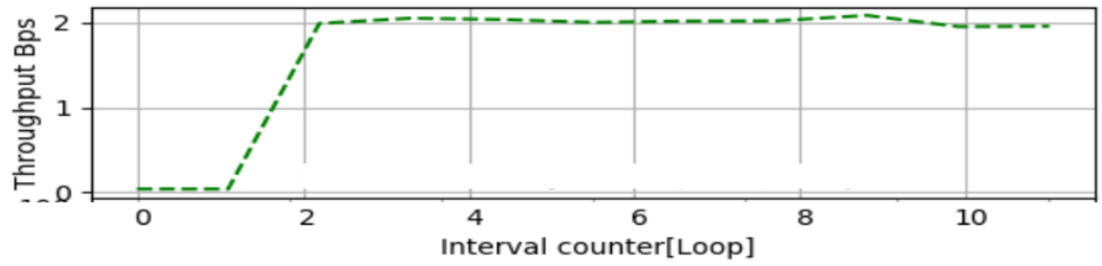


Figure 4.52: Throughput of 10 host topology

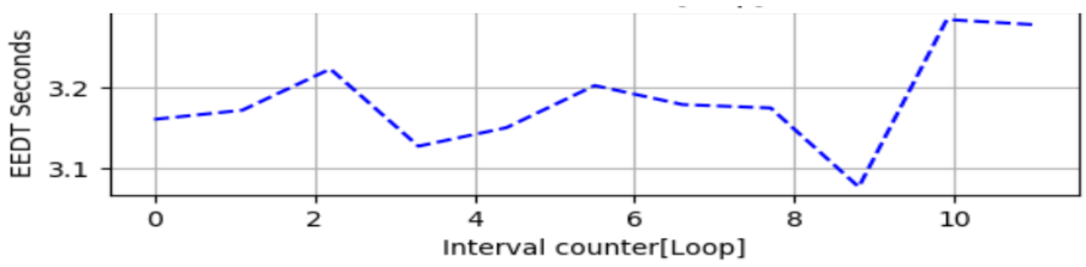


Figure 4.53: End to End Delay of 10 host topology

Chapter 5

Conclusion

SDN offers a number of advantages over traditional network. However, security of SDN is major issue. The global view of the network maintained by the controller can be spoofed by topology poisoning attacks. Malicious hosts can poison this network visibility of the controller and carry out black hole attack where availability of services gets seriously disrupted. The packets absorbed by black hole nodes may also be used to carry out further lethal attacks. Previous works to solve the problem of malicious nodes is done by authentication and permission based systems. However, such a solution is not viable because a node may get compromised after permission is granted. Moreover, static routes in the routing schemes provide ease to the attackers to carry out targeted black hole attack. Therefore, in our work we used dynamic routes by using multi-hop communication between hosts. The hosts involved in our multi-hop routing scheme were those with best behavior and their behavior value was updated after every interaction. Performance parameters like throughput, end to end delay and packet delivery ratio were recorded for normal operation. We also generated black hole attack and compared the values of performance parameters with those under normal operation to create our data set for detection of black hole attack. We routed the packets along our multi-hop routing scheme and recoded throughput,the end-to-end delay and the packet delivery ratio (PDR). When the values of parameters exceed threshold, an alert is generated and the malicious node is removed from the routing path and a new routing path is constructed to route the packets between source and destination.

References

[1] Dabbagh, M., Hamdaoui, B., Guizani, M. and Rayes, A., 2015. Software-defined networking security: pros and cons. IEEE Communications Magazine, 53(6), pp.73-79.

[2] Dayal, N., Maity, P., Srivastava, S. and Khondoker, R., 2016. Research trends in security and DDoS in SDN. Security and Communication Networks, 9(18), pp.6386-6411.

[3] Zhang, B., Wang, X. and Huang, M., 2018. Dynamic controller assignment problem in software-defined networks. Transactions on Emerging Telecommunications Technologies, 29(8), p.e3460.

[4] Nagase, K., 2016. Software defined network application in hospital. InImpact: The Journal of Innovation Impact, 6(1), p.1.

[5] Open Networking Foundation (ONF). [Online]. Available: https://www. opennetworking.org/

[6] Software-defined networking: The new norm for networks," Palo Alto, CA, USA, White Paper, Apr. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdnnewnorm.pdf

[7] https://opennetworking.org/sdn-resources/whitepapers/software-defined-networking-the-new-norm-for-networks/

[8] Xia, W., Wen, Y., Foh, C.H., Niyato, D. and Xie, H., 2014. A survey on software-defined networking. IEEE Communications Surveys Tutorials, 17(1), pp.27-51.

[9] Smith, J.M., Farber, D.J., Gunter, C.A., Nettles, S.M., Feldmeier, D.C. and Sincoskie, W.D., 1996. SwitchWare: accelerating network evolution (White paper).

[10] Kohler, E., Morris, R., Chen, B., Jannotti, J. and Kaashoek, M.F., 2000. The Click modular router. ACM Transactions on Computer Systems (TOCS), 18(3), pp.263-297.

[11] M. Handley, O. Hodson, and E. Kohler, "XORP: An open platform for network research," ACM SIGCOMM Comput. Commun. Rev., vol. 33, no. 1, pp. 53–57, Jan. 2003

[12] Quagga Routing Software Suite. [Online]. Available: http://www.nongnu.org/quagga/

[13] The BIRD Internet Routing Daemon. [Online]. Available: http://bird. network.cz/

[14] j. Rexford et al., "Network-wide decision making: Toward a wafer-thin control plane," in Proc. HotNets, 2004, pp. 59–64.

[15] Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N. and Shenker, S., 2007. Ethane: Taking control of the enterprise. ACM SIGCOMM computer communication review, 37(4), pp.1-12.

[16] Gude N, Koponen T, Pettit J, Pfaff B, Casado M, McKeown N, Shenker S. NOX:towards an operating system for networks. Proceedings of ACM SIGCOMM Computer Communication Review 2008; 38(3): 105–110

[17] Nife, F.N., Kotulski, Z. Application-Aware Firewall Mechanism for Software Defined Networks. J Netw Syst Manage 28, 605–626 (2020). https://doi.org/10.1007/s10922-020-09518-z

[18] https://searchnetworking.techtargets/definition/data-plane-DP

[19] Mahmoodi, T., 2015. 5G and Software-defined Networking (SDN).

[20]] Big Switch. Floodlight. URL http://www.projectfloodlight.org/.

[21] NTT. Ryu. https://osrg.github.io/ryu/, 2016. Online available

[22]] OpenDaylight foundation. OpenDaylight: A Linux Foundation Collaborative Project, . URL https://www.opendaylight.org

[23] "Positive Technologies - vulnerability assessment, compliance management and threat analysis solutions", Ptsecurity.com. [Online]. Available: https://www.ptsecurity.com/ww-en/.

[24] NOX Repo. NOX. URL http://www.noxrepo.org/.

[25] NOX Repo . POX . URL http://www.noxrepo.org/pox.

[26] KulCloud Inc Ltd. OpenMUL. http://www.openmul.org/, 2016. Online available

[27] https://www.sdxcentral.com/networking/sdn/definitions/southbound-interface-api/

[28] Yuchia Tseng. Securing network applications in software defined networking. Cryptography and Security [cs.CR]. Université Sorbonne Paris Cité, 2018. English. ffNNT : 2018USPCB036ff. fftel-02468016

[29] Jarschel, M., Zinner, T., Hoßfeld, T., Tran-Gia, P. and Kellerer, W., 2014. Interfaces, attributes, and use cases: A compass for SDN. IEEE Communications Magazine, 52(6), pp.210-217.

[30] Mininet software emulator to create virtual topology, [online] Available: http://mininet.org/.

[31] Dhawan, M., Poddar, R., Mahajan, K. and Mann, V., 2015, February. SPHINX: detecting security attacks in software-defined networks. In Ndss (Vol. 15, pp. 8-11).

[32] W.-Y. Huang, T.-Y. Chou, J.-W. Hu, and T.-L. Liu, "Automatical end to end topology discovery and flow viewer on SDN," in Proc. 28th Int. Conf. Adv. Inf. Netw. Appl. Workshops (WAINA), May 2014, pp. 910915.

[33] Shu, Z., Wan, J., Lin, J., Wang, S., Li, D., Rho, S. and Yang, C., 2016. Traffic engineering in software-defined networking: Measurement and management. IEEE access, 4, pp.3246-3256.

[34] T. Alharbi, M. Portmann and F. Pakzad, "The (in) security of topology discovery in software defined networks," in Proc. IEEE 40th Conf. Local Computer Network (LCN), 2015, pp. 502–505.

[35] A. Dawoud, S. Shahrestani, and C. Ruan, "Softwaredefined network controller security: Empirical study," in Proc. International Conference on Information Technology and Applications (ICITA), Sydney, Australia, 2017.

[36] Shaghaghi, A., Kaafar, M.A. and Jha, S., 2017, April. Wedgetail: An intrusion prevention system for the data plane of software defined networks. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (pp. 849-861).

[37] Mattos, D.M.F. and Duarte, O.C.M.B., 2016. AuthFlow: authentication and access control mechanism for software defined networking. annals of telecommunications, 71(11), pp.607-615.

[38] Hong, S., Xu, L., Wang, H. and Gu, G., 2015, February. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In Ndss (Vol. 15, pp. 8-11).

[39] M. F. Monir and S. Akhter, "Comparative Analysis of UDP Traffic With and Without SDN-Based Firewall," 2019 International Conference on Robotics,Electrical and Signal Processing Techniques (ICREST), 2019, pp. 85-90, doi: 10.1109/ICREST.2019.8644395.

[40] Shin, S., Xu, L., Hong, S. and Gu, G., 2016, August. Enhancing network security through software defined networking (SDN). In 2016 25th international conference on computer communication and networks (ICCCN) (pp. 1-9). IEEE.

[41] Lee, S., Yoon, C. and Shin, S., 2016, March. The smaller, the shrewder: A simple malicious application can kill an entire SDN environment. In Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks Network Function Virtualization (pp. 23-28).

[42] Spooner, J. and Zhu, S.Y., 2016. A review of solutions for SDN-exclusive security issues.

[43] Karmakar, K.K., Varadharajan, V., Tupakula, U. and Hitchens, M., 2020, April. Towards a Dynamic Policy Enhanced Integrated Security Architecture for SDN Infrastructure. In NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium (pp. 1-9). IEEE.

Appendix A

Source Codes