Fingerprinting the Software Defined Networks

By
Bilal Ahmed
00000170506

Supervisor
Dr. Nadeem Ahmed
Department of Computing

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Information Technology (MS IT)

In
School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(May 2018)

Approval

It is certified that the contents and form of the thesis entitled "**Fingerprinting the Software Defined Networks**" submitted by **Bilal Ahmed** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Nadeem Ahmed**

Signature: ________________
Date: ________________

Committee Member 1: **Dr. Syed Taha Ali**

Signature: ________________
Date: ________________

Committee Member 2: **Dr. Muazzam Ali Khan**

Signature: ________________
Date: ________________

Committee Member 3: **Dr. Arsalan Ahmad**

Signature: ________________
Date: ________________

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS/MPhil thesis written by Mr. **Bilal Ahmed**, (Registration No. **00000170506**), of **SEECS** (School of Electrical Engineering and Computer Science) has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS/M Phil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

Signature: ______________________________

Name of Supervisor: ______________________

Date: __________________________________

Signature (HOD): ________________________

Date: __________________________________

Signature (Dean/Principal): ________________

Date: __________________________________

Abstract

Research in Software Defined Networks (SDN) has gained momentum in recent years due to unique features offered by it. The adaptation of the technology has resulted in many SDN enabled deployments. As it provides a centralized control of a whole network. However, the centralized nature of the SDN also makes it prone to many security threats such as denial of service attacks, especially if the policy parameters of SDN are known to adversaries. So that they can attack according to the discovered parameters of the network. In this research work, we present how to do fingerprinting of various SDN policy parameters such as hard and soft timeouts, OpenFlow match-fields deployed by the SDN controller, mitigating policy for over flowing of flow table entries and information about topology deployed in the targeted network. An adversary can launch a carefully planned attack, especially on the SDN data plane, if these policy parameters are discovered for the SDN enabled network. It has been assumed that adversary has got access to one of the end host within the SDN domain, from which is able to generate custom packets from the networking stack. Efficient algorithms are proposed to discover these aforementioned policy parameters and the impact of knowing these parameters has been discussed. The results of these fingerprinting algorithms are verified with SDN domain simulations in Mininet.

Dedication

To my parents,
Without whom this success would not be possible.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Bilal Ahmed**

Signature: ________________

Acknowledgment

I am absolutely grateful for my thesis advisor, Dr. Nadeem Ahmed for his leadership in ensuring my successful development through academic achievement and for polishing my skills. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing.

Finally, I must express my very profound gratitude to my family members and friends for providing me with unfailing support and continuous encouragement throughout my years of life.

Bilal Ahmed

Table of Contents

List of Figures ix

List of Symbols xii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Defination . . . 2
- 1.3 Objectives and Research Goals . . . 2
- 1.4 Thesis Organization . . . 3

2 Background Information 4
- 2.1 Architecture of SDN . . . 4
- 2.2 OpenFlow . . . 5
- 2.3 Forwarding & Match Fields in SDN . . . 6
- 2.4 Timeouts . . . 9
- 2.5 Limited Flow Entries . . . 9

3 Literature Review 11
- 3.1 Timeouts . . . 11
- 3.2 Match Fields . . . 12
- 3.3 Flow Table . . . 13
- 3.4 Network Security & SDN . . . 14

4 Methodology 15
- 4.1 Timeouts . . . 16
- 4.2 Match Fields . . . 27
  - 4.2.1 OpenFlow Version . . . 29
  - 4.2.2 Variation in Timeouts . . . 30
- 4.3 Mitigating Over Flowing Tables . . . 32
- 4.4 Topology Discovery . . . 36

5 Results & Discussion **38**
5.1 System Specification . . . 38
5.2 Granularity of Step Size . . . 39
5.3 Timeout Estimations . . . 42
5.4 Controller Reaction to Full Tables . . . 45
5.4.1 Packet Drop Policy . . . 45
5.4.2 Flow Replacement Policy . . . 47
5.5 Topology Discovery . . . 48

6 Conclusion & Future Work **51**
6.1 Future Work . . . 51
6.2 Conclusion . . . 52

References **53**

List of Figures

1.1 Taxnomoy of SDN fingerprinting 3

2.1 Architecture of SDN 6
2.2 Difference in first RTT (RTT_{FE}) and subsequent RTTs (RTT_{avg}) 7

4.1 One compromised host under adversary's access 15
4.2 Two compromised host under adversary's access 16
4.3 Branching between algorithms for finding timeouts 17
4.4 Flowchart - Finding first timeout and its nature 19
4.5 Flowchart - Finding hard timeout given hard timeout (T_{hard}) . 22
4.6 Flowchart - Finding the hard timeout given the soft timeout (T_{soft}) 25
4.7 Flowchart - Match fields fingerprinting 28
4.8 Flowchart - Detecting timeouts for multiple match fields . . . 31
4.9 Packet drops policy due to full table 34
4.10 Flow replacement policy to overcome full table 35
4.11 Fat tree topology 37

5.1 Plot - Step size and Error for T_{hard} 40
5.2 Plot - Step size and Reduced Error for T_{hard} 40
5.3 Plot - Step size and Error for T_{idle} 41
5.4 Plot - Step size and Number of loops in Algo 2 for T_{idle} 41
5.5 Plot - T_{hard} discovery 43
5.6 Plot - T_{idle} discovery 43
5.7 Plot - T_{idle} discovery using Algorithm 4.1 & 4.2 44
5.8 Plot - T_{hard} discovery using Algorithm 4.1 & 4.3 44
5.9 Plot - Packet drops at hard timeout = 60 seconds 46
5.10 Plot - RTTs of in-rack communication 47
5.11 Plot - RTTs of in-pod communication 47
5.12 Plot - RTTs of inter-pod communication 48
5.13 Plot - Variation in RTT_{FE} 49

5.14 Plot - Variation in $RTT_{reinstall}$ 49
5.15 Plot - Variation in RTT_{avg} 50

List of Tables

2.1 Flow entry timeouts 9

5.1 System specification 39

List of Symbols

Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
CPU	Central Processing Unit
DDoS	Distributed Denial-of-Service
GPU	Graphical Processing Unit
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MPLS	Multiprotocol Label Switching
ONF	Open Networking Foundation
OS	Operating System
RAM	Random Access Memory
RTO	Retransmittion Timeout
RTT	Round Trip Time
SDN	Software Defined Networking
SSH	Secure Shell
TCP	Transmission Control Protocol
ToR	Top of Rack
VM	Virtual Machine

Nomenclature

σ	Standard Deviation
RTT_{avg}	RTT of Data Plane
RTT_{FE}	RTT of Flow Entry Installation
$RTT_{reinstall}$	RTT of Flow Replacement Policy
RTT_{test}	RTT of Test Pings
T_{hard}	Hard Timeout
T_{sleep}	Sleep Time
T_{step}	Step Size
T_{soft}	Soft Timeout

Chapter 1

Introduction

1.1 Motivation

SDN (Software Defined Networks) provides the centralized control of whole network. Other than the basic networking features traffic re-scheduling, open-source and implementing QoS (Quality of Service) policies are also major features offered by the SDN [1]. Moreover, SDN enabled networks are getting significant research attention in the networking domain due to programmability of the networks and other numerous rich features. In a typical SDN, control plane is decoupled from forwarding plane, providing ability to network administrators (using control plane) to program the network features (at forwarding plane). OpenFlow, developed by Open Networking Foundation (ONF), has become a de facto standard for communication between control and forwarding planes. Every new version of OpenFlow (1.0 to 1.5) is bringing more options to manage the SNDs more intelligently. SDNs are getting mature, however, there are still many research challenges to consider [2]. One of the key challenge that seek immediate attention is its security vulnerabilities [3]. From a security perspective, there are many attack opportunities targeting both data plane [4], [5] and control plane, to degrade the performance of the SDN enabled network [6]. For example, flow entries table attacks target the data plane of the SDN enabled network. On the other hand, for instance, attacks on Link Layer Discovery Protocol (LLDP) [7] and topology poisoning attacks aim to degrade performance of the control plane [8].

1.2 Problem Defination

This research work is aimed to discover the parameters of a SDN enabled network. However, the purpose of gathering information is not from an attacker perspective but in general to present a more enhanced solution in the respective domain; fingerprinting the SDN enabled networks. Usually when a network is deployed, there is minimal probability that the network administrator or organization would use the default parameters. Moreover, due to flexibility offered by SDN, it is highly likely that network is to be configured according to the traffic density and traffic load of that particular network. Which is also the basic problem statement, to fingerprint those configurable parameters of the targeted SDN enabled network.

Moreover, to analyze the impact if that discovered information is known to attacker in advance i.e., before launching an attack. As, if an attacker has information about the targeted network, means he can find more vulnerabilities, ultimately resulting in paralyzing the SDN controller more efficiently. Furthermore, this research is not only focused on attacker's perspective, network administrators can also run the fingerprinting modules on their network in order to enhance the security of the network. The design logic of the proposed mechanism is limited to reality based condition, which includes, secure channel between SDN controller and the SDN enabled switches and no direct access to switches. It is assumed that attacker has access to only end host(s) and can communicate with other hosts. Furthermore, to analyze the difference in fingerprinting when attacker has one compromised host inside the targeted network and when he have more than one compromised end hosts inside the targeted network.

1.3 Objectives and Research Goals

The outcome of this research is multifold, finding out timeouts set by the controller, finding flow matching policy (header fields) while considering the network firewall, finding the flow entries overflowing policy at switches in the SDN network and finding information about the fat tree topology deployed in the network. As figure 1.1 shows the taxonomy of the research goals of this research work, to gather the information about the SDN enabled network while having access to just an end host. This research work indirectly presents concerns for network administrators to program the SDN network such that adversary/attacker may find it hard to predict the policy parameters. Our further contributions of this study are as follow:

- Using proposed mechanism/modules for fingerprinting the configurable

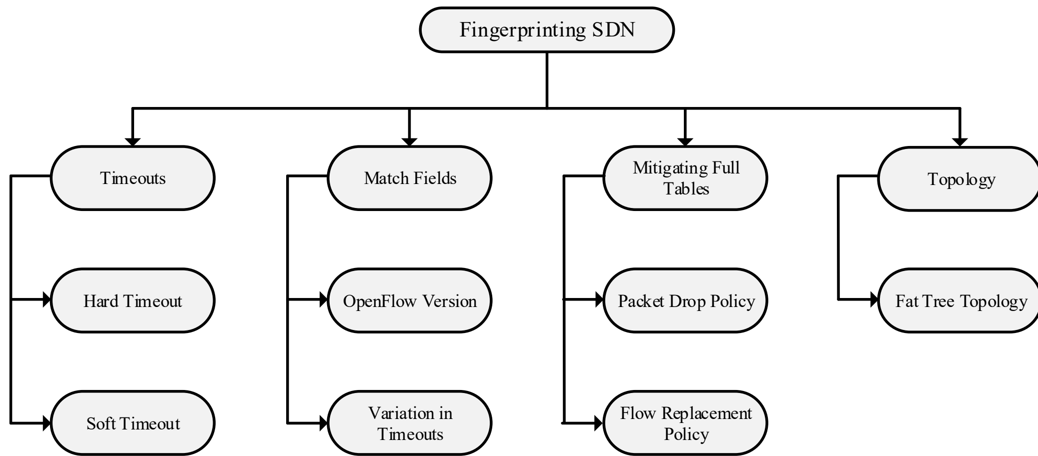

Figure 1.1: Taxnomoy of SDN fingerprinting

parameters of SDN and the empirical analysis of the proposed mechanism/modules

- Impact on the performance of SDN enabled network when fingerprinted configurable parameters are known to DDoS

1.4 Thesis Organization

The work has been organized as follows. Chapter 2 covers the basic background information about the SDN and policy parameters. Chapter 3 covers the literature review and limitation of existing SDN fingerprinting mechanisms in contrast to this research work. Chapter 4 explains the detailed methodology of proposed fingerprinting modules and their working. The results and evaluation details are discussed in the chapter 5. While the chapter 6 summarizes the work along with proposed future work.

Chapter 2

Background Information

This chapter covers a brief overview of the background knowledge related to this thesis. Almost all related theoretical and conceptual points have been covered. First of all, architecture of SDN has been briefly explained in section 2.1. Which is followed by an overview of OpenFlow in section 2.2. Next, concept of Match fields and packet forwarding in SDN has been elaborated in section 2.3. Then, section 2.4 covers the concept of timeout, associated with match fields in SDN. And lastly, section 2.5 contains the information about the flow tables and their limited size, in terms of accommodating flow table entries.

2.1 Architecture of SDN

Traditional networking architecture has some serious limitation such as scalability, vendor specific API, compatibility issues and much more. SDN provide solution all of these problems [9]. SDN decouples the control plane from the data plane. Which provide greater centralized control over the network. ALL SDN enabled networks have these basic components:

- **Controller**:
  Which is also referred as the "brain" of the SDN enabled network. Just like a brain, it handles all functions from a centralized point of the network. Network administrators manage the data plane functions such as forwarding of packets while having a statistical view of whole network.

- **Data plane infrastructure**:
  Which is also referred as the "body" of the SDN. Just like a human

body, it sends the signal to the brain (controller) and waits for the instructions to act accordingly.

SDN infrastructure [10] has been shown in the figure 2.1. There are API included for the network to work, which includes, Northbound APIs, Southbound APIs, Eastbound APIs and Westbound APIs. They have been mapped in the figure, and there brief functionality is:

- **Southbound APIs** is responsible for communication between controller and data plane infrastructure such as L2/L3 switches. All request and response messages for forwarding and statics follows this set of API. Some famous examples include OpenFlow [11], ForCES, PCEP, NetConf etc. Out of which OpenFlow is the most commonly used in academics as well as in industry because it's completely open source, scalable, vendor restricted free and properly documented.

- **Northbound APIs** is responsible for communication of application and services running over the network with controller. It can also be used in building cloud automation stacks, dynamically changing need of the applications and services without inferring with network. Some famous examples of northbound APIs include Restful, FML, frenetic etc.

- **Eastbound** and **Southbound APIs** is responsible for communication between multiple SDN controllers. The purpose of having multiple controller include fault tolerance, multiple applications running on multiple controller, controllers from different vendors etc. Some famous examples of eastbound and southbound API include ALTO, hyperflow etc.

2.2 OpenFlow

Many people confuses OpenFlow with SDN. They think OpenFlow is SDN and SDN is just OpenFlow, this is not true, OpenFlow is just the first standard in the SDN and it was initiated in 2008 at the Stanford University. In simpler words we can say that, OpenFlow is a subset of SDN domain and to be precise, OpenFlow is just a well-known southbound API. Since its first version, OpenFlow is continuously in development, not only new features are being introduced but existing infrastructure is also improved. It also supports reverse compatibility between different versions, whenever a switch

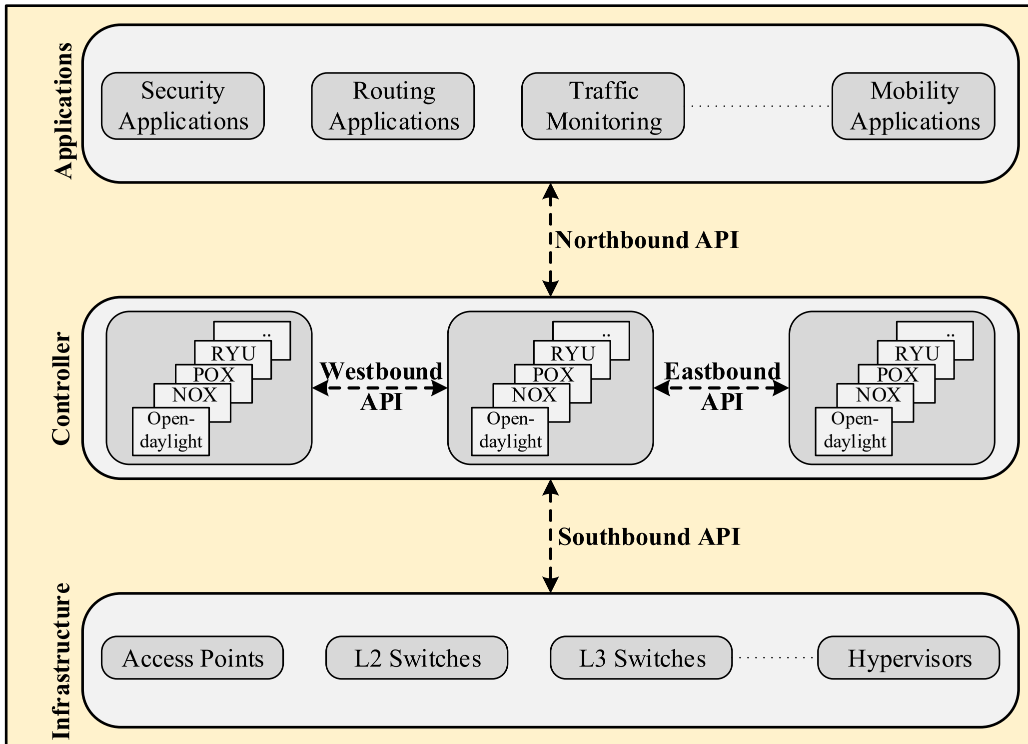

Figure 2.1: Architecture of SDN

and controller starts a communication, in the hand shake process they agree to use the supported version at both ends. For example if switch supports OpenFlow 1.0 to 1.5 but controller supports only OpenFlow 1.3, negotiations will take place initially only and both of them will agree to use OpenFlow 1.3.

2.3 Forwarding & Match Fields in SDN

SDN packet forwarding mechanism is different from the legacy packet forwarding mechanism. And in SDN, controller's policy plays a vital role in packet forwarding decisions. When a packet is transmitted in the SDN, switch does not know how to forward the packet, this is called a "Table miss" event. Switch forward it to the controller by encapsulating it in OpenFlow "Packet-in" message, for further processing [11], [12]. Pre-defined controller's policy then decides the forwarding path and sends the "flow-mod" or "packet-out" message, which contains the forwarding rule to be installed at the switch. This installed forwarding rule, is commonly known as the flow rule.

When the next packet arrives which matches the flow rules already installed at the switches will be forwarded directly rather than sending it to the controller. And RTT of such directly forwarded packet at data plane is far less as compare to the RTT when there is no flow rule and switches ask the controller for flow rule, as there is a processing delay at the controller and the RTT from switch to the control. Moreover, this difference in RTT will increase significantly if there are more than one switches installed between the end hosts. Because, usually each switch will ask the controller to install the flow entry. For the rest of this study, the RTT when there is no flow entry exists is donated by RTT_{FE} and when flow entries exists is donated by RTT_{avg} or RTT_{normal}. Figure 2.2 shows this difference in first RTT (RTT_{FE}) and subsequent RTTs (RTT_{avg}) along with all involved steps. Typically, flow rule consists of some matching criteria, life of flow rule, priority and actions to be performed by the switch. Moreover, flow rule entries also contains the statics information i.e. number of packets matched with the flow entry and total byte count, as controller can ask for statics.

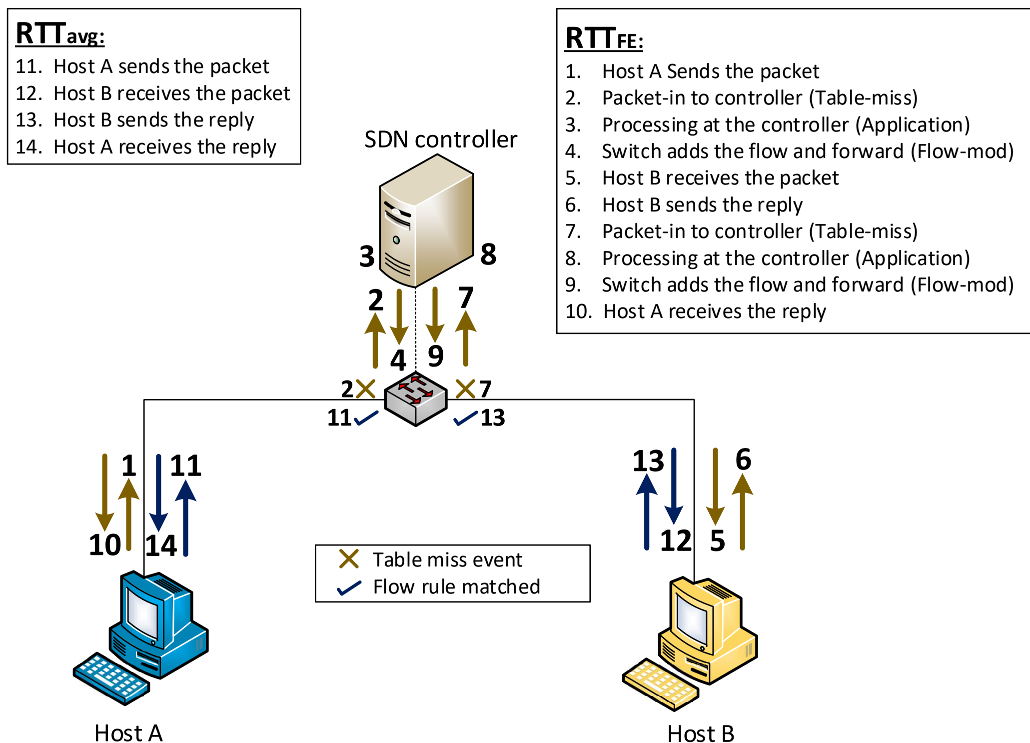

Figure 2.2: Difference in first RTT (RTT_{FE}) and subsequent RTTs (RTT_{avg})

Match fields are increased with each new version of OpenFlow. As it

can be seen in the table, in OpenFlow 1.0 there were 12 match fields but in OpenFlow 1.5 match fields are increased to 44. The count in the table also includes the experimenter match fields, typically which are not included in the documentation of the open flow but included in the implementation code. It also implies that OpenFlow header size has also been increased due to the increased match fields. Although it is also not compulsory that each OpenFlow version should increase match fields.

There are two types of match fields, pipelined match fields and header match fields. Pipelined match fields are not linked with packet headers, the solemnly purpose of these match fields is for pipeline processing at the switch. While the header match fields are extracted from the different layers of incoming packet. Some header fields have pre-requisites, for example declaring TCP port numbers as match fields requires declaration of IP layer protocol as match field as well.

Moreover, increased numbers of match fields also means that more flow entries will be installed for uniquely matching criteria for incoming packets. For example, if a SDN controller's policy is not considering the port numbers of TCP header as match field, M * N TCP SYN burst will require only two flow entries, where M is the number of unique destination ports and N is the number of unique source ports. On the other hand if SDN controller's policy is considering the port numbers of TCP as match fields, it will install 2 * M * N flow entries for M * N same TCP SYN burst. Similarly, if SDN policy is installing flow entries for source port of TCP only, than the M * N flow entries will be required for the same burst.

SDN also offers the features of a firewall at the controller's policy i.e. controller can block the internet traffic on this basis of any match field. If an adversary knows in advance which burst are useless, adversary can eliminate those values of match fields to attack efficiently with minimum utilization of resources. So, the match fields are of significant importance in launching an attack. Flow entries are stored in flow table at the switches. There are more than one flow tables at the switches, controller defines in which table flow should be installed, as pipelining between multiple tables is a regular matter for SDN enabled switches. And due to limited resources each flow table is restricted to certain number of flow entries, which a switch can accommodate. Which means if a table is full, switch cannot accommodate any new flow until the flow is removed, which provides a big opportunity to attack on the data plane of SDN enabled network. And this is of particular interest in the data centers environment with huge volume of traffic [13] resulting in drastic degradation of the networks performance.

2.4 Timeouts

Controller also allocate timeouts to each flow entry, by which switch knows when to remove the flow entry. In simple words, we can say that it is the life time of a flow entry. There are two types of timeouts, hard timeout and idle timeout, which is also known as soft timeout. Flow will be removed when inter arrival time between two consecutive packets equals or exceeds from the value of idle timeout configured in the network, as it names also suggests "idle" timeout. However, if a packet of the same flow arrives before the soft timeout is met, it will restart. While the hard timeout is not dependent upon the time gap between consecutive packets, it is the absolute time started since the first packet of the relevant flow and when it equals with the value installed by the controller, switch removes that particular flow entry. A brief working of timeout values and their combinations has been summarized in the table 2.1. The value of each timeout can be defined anywhere in the range of 0 to 65535 seconds and a value of '0' means, it is not set by the controller i.e., flow entry will not be removed due to that particular timeouts. Which means entry will not be deleted until control manually instruct the switch to remove the flow entry.

Table 2.1: Flow entry timeouts

Pair ID	**Idle/soft Timeout**	**Hard Timeout**	**Effect**
A	0	0	Infinite Idle and Hard timeouts. The entry will not be deleted until controller instructs to do so
B	0	Y	Entry will be only be deleted after Y seconds since its installation time
C	X	0	Entry will be only be deleted if inter packet gap equals X seconds
D	X	Y	Either of timeout is fulfilled irrespective of their priority

2.5 Limited Flow Entries

There is very limited space available for flow entries at the switches. And size of the flow table has a huge impact on the performance of the network [14]. Typically, SDN enabled switches can only accommodate few thousand flow

entries, an opportunity to attack on data plan. The default applications of SDN controllers such as POX, RYU etc. do not react to table full error messages, as they rely upon timeout mechanism for removal of flow entries tables at the switches. But there exist another mitigating strategy in which SDN controller's application instructs the switch to remove the flow(s) whenever it receives the table full error message.

The selection of flows to be removed varies, such as it can be based on network statics or on the bases of one or more match fields. In the first approach, ignoring the table full messages, there will be significant packet drops. Whereas in the latter approach, there will be increased load on the controller and RTT will increase. Because in normal case of table miss event, switch sends the packet-in and controller sends the flow rule for it and switch forwards the packet i.e., RTT_{FE}. But in the second case, when table full event occurs switch sends the error message against the flow rule message, then the controller sends the flow entries deletion message and then again sends the flow rule. So, the RTT in this case will be higher as compare to the RTT of normal forwarding.

Chapter 3

Literature Review

This chapter reviews the relevant state of the art researches and their comparison with the proposed schemes. Most of the literature focuses on the fingerprinting SDN controllers only, to the best of our knowledge this is the first research effort that focuses on the in-depth fingerprinting of SDN enabled network parameters rather than just fingerprinting the controller. Since, it is very easy to give wrong prediction of based on such techniques. As, SDN parameters are configurable as per requirements of the network administrators. For now, consider a simple example, default hard timeout of POX is 30 sec while RYU has default hard timeout 0 sec. But if a network administrator adjust POX hard timeout to 0 sec, and fingerprinting mechanisms may result in wrong prediction according to such techniques, so is the case with soft timeout. Similarly others parameters can also be manipulated and the fingerprinting schemes won't work or may result in wrong controller identification.

Section 3.1 presents some SDN fingerprinting techniques based on the timeouts. Section 3.2 enlists some famous researches on fingerprinting the match fields of a SDN enabled network. While, section 3.3 presents techniques for finding the size of flow entries table at the SDN enabled switches and controller reactions to the table full event. And some famous SDN enabled network security mechanisms have been presented in section 3.4.

3.1 Timeouts

Timeout values have been discovered by [15], but under very unrealistic assumption that flow entry will not be removed by hard timeout (if value exist) when discovering the value of soft timeout, and usually both values exists for same flow entries (Timeout values combination D in Table 2.1). Which is only

possible in one case, when hard timeout does not exist i.e., hard timeout = 0, otherwise soft timeout value will not be discovered. And proposed scheme also ignored the granularity of step size i.e., error in the discovered values of timeouts of SDN enabled network and impact of step size on the overall cost of the proposed scheme. Moreover, SDN controller has also been predicted on the basses of discovered timeout values, provided that the targeted network is using the default timeout values of SDN controllers.

Similarly, [16] also worked on fingerprinting controllers, they predicted the controller using the timeout values. But under their default parameters, as discussed earlier, there is very low probability that a SDN enabled network will be using the default values of timeouts and such techniques will results in wrong predictions. Moreover, they also ignored the hard timeout and soft timeout combination fact during finding the values of hard and soft timeouts. Granularity of step size in finding timeouts has also not been discussed. But our study, not only covers all timeout combinations but also analyzes the algorithm costs and errors associated with them. Moreover, authors in [15] and [16] did not mentioned that different match fields can have different timeout values and how to find them.

Authors in [17], presented the analysis and impact of soft timeout only in the DDoS attacks, hard timeout has been neglected by the authors. And they neither include how to find timeouts nor they cover all the combinations of timeouts and their impact and what types of packets can consume network resources at max depending upon the timeout values being deployed in the SDN enabled network.

3.2 Match Fields

Network administrator can choose match fields according to the traffic of that particular network. But as discussed in section 2.3, match fields has a huge impact in launching a successful flow table attack on SDN enabled network. Concept of fingerprinting match fields was initiated by S. Shin et al. [18], but they ignored the life span of flow entries generated by those match fields at the SDN enabled switches i.e., timeout values. Although they presented a state of the art SDN scanner, almost all SDN fingerprinting techniques are based on the same logic. But their technique is for basic match fields only and now, in the latest version of OpenFlow, match fields have been significantly increased. Furthermore, we have also fingerprinted the basic firewall; omitting the blocked values of match fields when generating forged packets in DDoS attacks to make attacks much more strong.

Zhang et al. [19], identified the default match fields of SDN controllers

but the match fields varies policy to policy within the same controller. For example, in the default policy of RYU, it considers ethernet addresses of hosts only as the match fields but a network administrator can easily deploy any other header field as a Match field such as TCP port numbers, TCP flags etc. Moreover, they have used the UDP packets for evaluation of their work and ignored the time synchronization error in analyzing the results. Firewall feature of SDN controller was completely ignored by the aforementioned analyzed studies. T. Alharbi et al. [20] analyzed the impact of variations in DoS attacks targeted on SDN enabled network. But they have also assumed that the target SDN enabled network is running the default application of match fields (simple learning switch). Moreover, relation between timeouts of match fields and their impact under DoS conditions has not been discussed.

3.3 Flow Table

Using the information about timeouts and match fields of a SDN enabled network, Zhou. et al [21] predicted the numbers of flow entries at the switches. They have assumed that, flow generated by the hosts can completely take hold of flow entries table of SDN enabled switches. But in a commercial network there will be already thousands of entries installed for on-going communication. It's highly likely that on the bases of soft timeout, those flow entries will never be removed. As, each packet of the same flow refreshes the life of the flow entries. Let's assume the switches have the capacity of 2000 flow entries, after generating 650 packets (1300 flow entries) host may assume that switch capacity for handling flow entries is 1300 only. They assumed that on the table full event, old entries will be deleted and new entries will be installed for their new flows. But, flow entries to be removed (in-order to free the space for new packets) can also varies. For example, what if SDN controller is removing the newly generated flow entries only or dumping the whole table at the table full message received by the switch?

Consider an example, when host has generated 650 packets and due table full event occurs due to any other host and controller dumps the whole flow table at switch in such cases host will predict the switch capacity for handling flow entries more than 2000. But our study is focused only on the reaction of controller to the table full event and how it will affect the SDN enabled network under DDoS condition. Moreover, default applications dont have flow replacement policy i.e., removing already installed flow entries for entertaining the new arriving packets. The default applications of SDN controllers simply ignores the request until the flow is removed automatically (on the bases of timeouts). Authors in [17], presented the table full under

the default behavior of controllers, in which packet drop occurs. And to the best of our knowledge, our research work is the first empirical analysis study that considers both, default behavior as well as flow replacement policy at table full event.

3.4 Network Security & SDN

SDN enabled networks are prone to DDoS attacks. Objective of such attacks is to destabilize the SDN controller [22] by consuming the networks resources such as processing at the controller, blown flow tables at the switches and throughput of the network. However, this is not the only case. SDN can also be used for network security purposes. Although such discussion is beyond the scope of this research work. But as brief future direction has been proposed, which is the future direction of this research work.

Chapter 4

Methodology

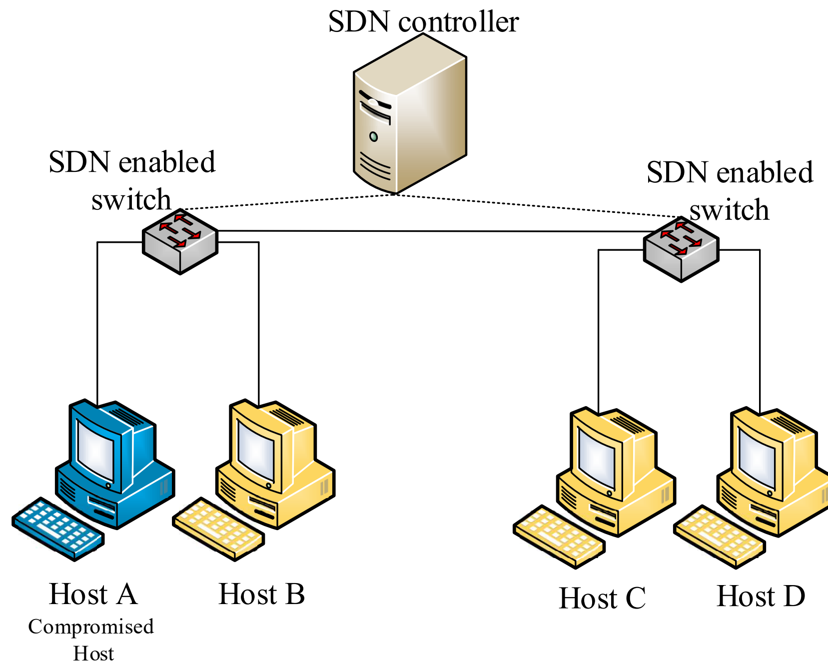

Figure 4.1: One compromised host under adversary's access

The objective of this research is to fingerprint the SDN controller's policy parameters deployed in a network. This chapter covers thorough explanation and design logic of algorithms to achieve the desired goals. Design logic considers two scenarios when we have only one compromised host inside the network and when we have two compromised hosts. In section 4.1, timeout fingerprinting schemes have been proposed and explained in detail. While, section 4.2 discuss the design logic of finding match fields modules. Moreover, detecting OpenFlow version and verifying timeouts for all match fields

are also discussed in the same section. Next, fingerprinting of controller's reaction to table full event has been discussed in section 4.3. And lastly, section 4.4 discuss the design logic of end to end topology discovery; from an attacker perspective.

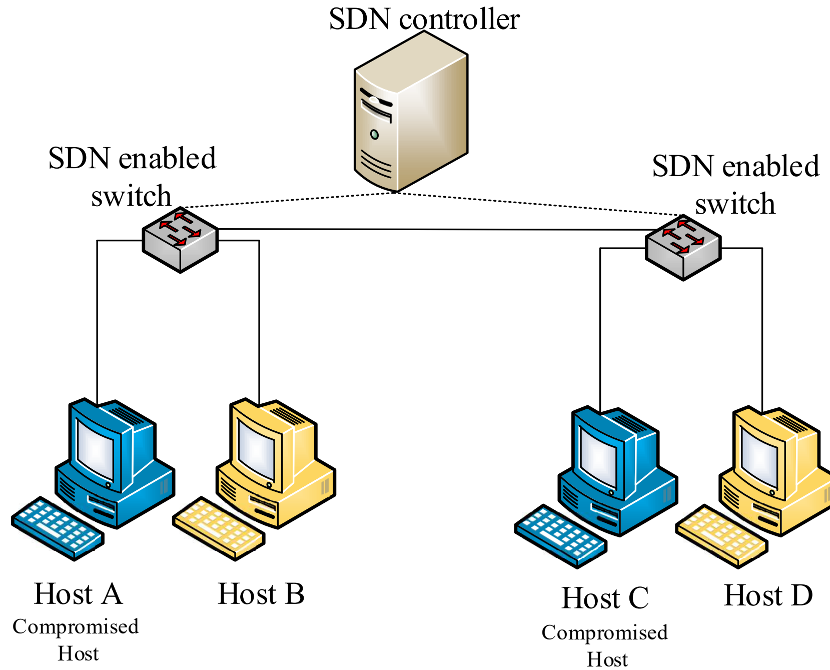

Figure 4.2: Two compromised host under adversary's access

4.1 Timeouts

Timeouts is the life of a flow entry and has been explained in section 2.4. A series of algorithms has been designed and implemented, to investigate the timeouts values deployed in a SDN enabled network. The design logic of finding timeout values is based on ICMP (Internet Control Message Protocol) pings. Algorithm 4.1 has two main functions, finding the value of first timeout and nature of first algorithm i.e., hard timeout or soft timeout. Algorithm 4.1 will return value of one type of timeout only, the next objective is to find the value of remaining timeout.

Depending upon the output of Algorithm 4.1, the control is branched out to two algorithms. If first timeout found was hard timeout, then control is

transferred to Algorithm 4.2 for finding soft timeout. Similarly, control is transferred to Algorithm 4.3 for finding value hard timeout, if Algorithm 4.1 finds the soft timeout. This transfer of control has been highlighted in figure 4.3. It is worth mentioning that, ICMP ping with same header fields will be transmitted in this module. As changing values of header fields of ICMP might enforce the new flow installation, such as ICMP code and ICMP type can also be a match field in a SDN enabled network. Initial pings in this mechanism also confirms the SDN functionality, if it is enabled or not, as the RTT of first ping will be significantly higher than the RTT of subsequent pings, which has been thoroughly discussed in section 2.3.

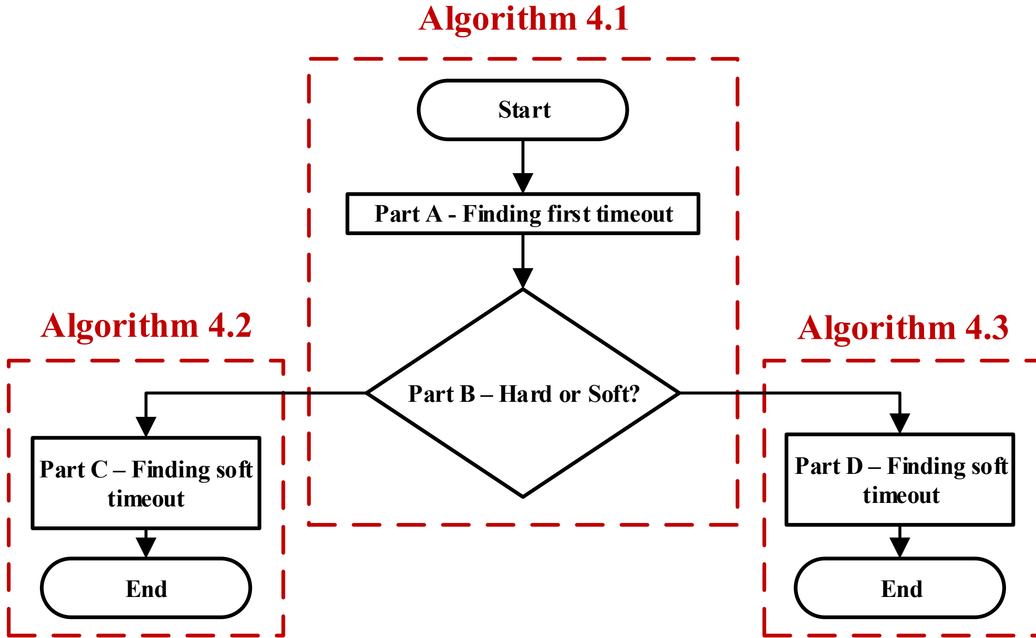

Figure 4.3: Branching between algorithms for finding timeouts

Now design logic of each algorithm is explained. Let's assume that an adversary has a compromised host (Host A) under his control in SDN domain. It is further assumed that there exist no pre-installed flow entry for communication, host A sends a ping to host B and notes the current time as T_0. This first ping will install flow entries at all hops between host A and host B. The RTT of this first ping is noted as T_{FE}. Now host A sends 'n' consecutive pings, to compute the standard deviation (σ) and average RTT (T_{avg}), for the implementation purpose number of pings in this function has been limited to 10 but choosing the number of pings for this function can vary. Using the value of standard deviation and mean, threshold $T_{thresh} = RTT_{avg} + 4 * \sigma$ (Line

5 in Algorithm 4.1) has been computed. Design logic assumes that whenever any RTT value crosses T_{thresh}, a new flow entry has been installed and RTT of the respective ping is increased due to inclusion of switch-controller communication for flow installation. Other than that all pings are traversing through data plane only as flow entry already exists between the end hosts.

$$n^{th} ping = (n-1) \times T_{step} \quad (4.1)$$

Now value of first timeout will be explored. From now on all pings will be gapped at T_{sleep} and T_{step} will be added in it at each ping ($T_{sleep} = T_{sleep} + T_{step}$). Initially T_{sleep} and T_{step} have value of 0 ms and 100 ms respectively. This process of adding T_{step} in T_{sleep} will continue unless until RTT_{ping} exceeds T_{thresh} i.e., when a new flow entry will be installed. So, as the number of pings increases the time gap between consecutive pings will also increase (Lines 7 11, Algorithm 4.1). As T_{step} is being added to T_{sleep} on each ping. For example, host will send a ping, then sleeps for 100 ms before sending the second ping and then sleep for 200 ms before sending the third ping. The time gap between n^{th} and n^{th-1} ping can be calculated by the equation 4.1, this equation is valid only if T_{sleep} is set initially to 0 ms. If T_{sleep} is intially set to any other value, then it should also be incorporated with it i.e., it should also be added to the answer of equation 4.1.

Total elapsed time since the first ping, equals the sum of all gaps between the subsequent pings. Whenever a new flow entry is installed i.e., RTT_{ping} exceeds T_{thresh}, its infer that the previous flow entry has been expunged due to timeout. Now host will note the current system time T_1 and current value of sleep time T_{sleep}. As the first timeout has been occurred but it is still unknown that the flow entry was removed due to soft timeout or hard timeout. If the flow entry is expunged due to hard timeout, then hard timeout equals the total elapsed time ($T_1 - T_0$). Or if the flow is removed due to soft timeout, it is inferred that the T_{sleep} is the soft timeout of the network since it is the sleep time (waiting) between the last two consecutive pings. Now the host needs to find the nature of the first timeout, host will sleep for T_{sleep} and then sends a ping again. If the RTT_{ping} of this test ping exceeds the T_{thresh}, it means that flow expiry was due to soft timeout and it declares T_{soft} equal to the T_{sleep} i.e., value of last sleep time. If the value of RTT_{ping} found to be less than the T_{thresh}, this infers that previous flow entry was removed not because of soft timeout but it was removed due to the hard timeout and it declares $T_{hard} = T_1 - T_0$, as by the definition of hard timeout it equals the total elapsed time. Moreover, if the flow was removed due to soft timeout, the value of timeout has confirmation level of 2, as it

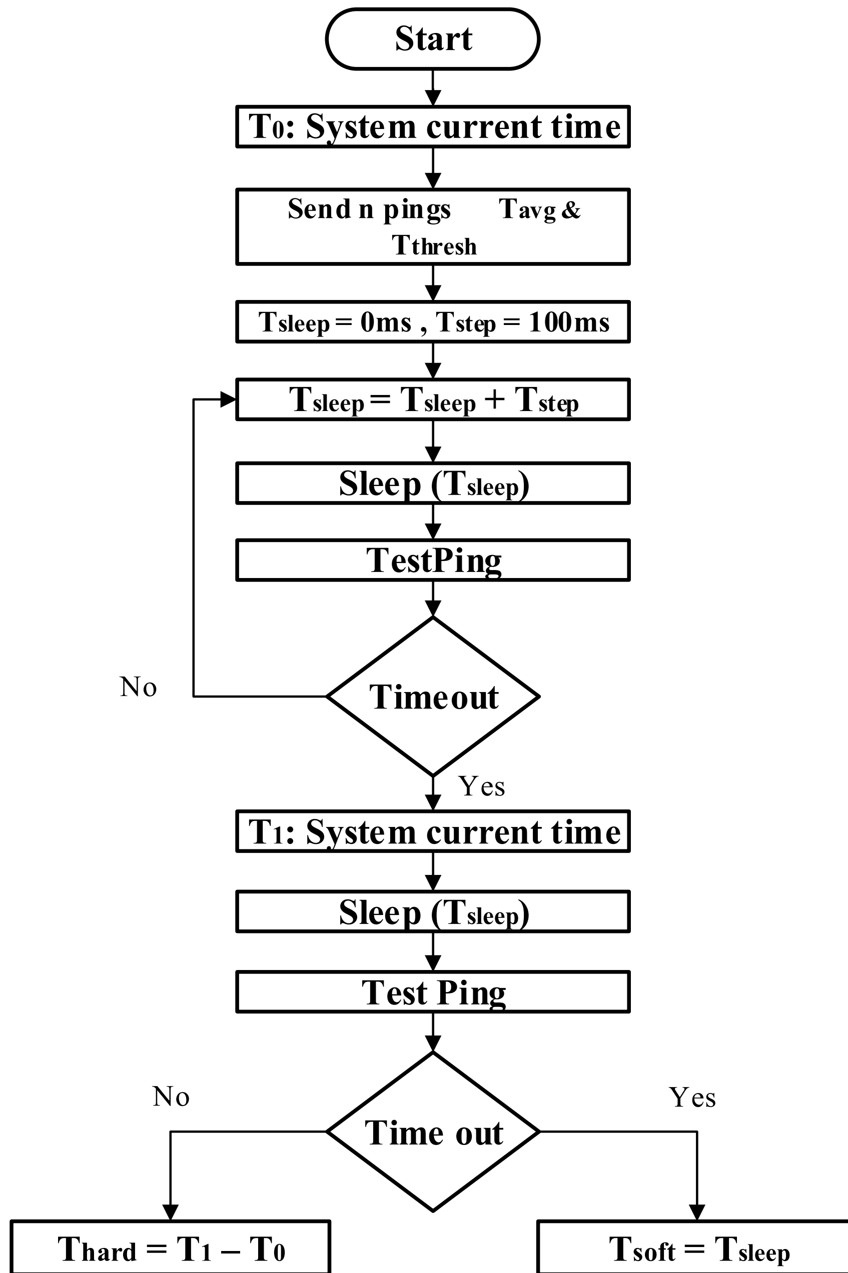

Figure 4.4: Flowchart - Finding first timeout and its nature

was expired in the initial stage and then in the verification stage as well. Algorithm 4.1 thus returns the value of first timeout and disclose its nature as well, as shown in flowchart (Figure 4.4).

Note that for hard timeout, the discovered value would be not accurate

Algorithm 4.1 Find first timeout and its nature

1: Note current system time as T_0
2: Send the first ping to install the flow entry
3: Calculate RTT as RTT_{FE}
4: Send n pings to calculate average RTT (RTT_{avg}) and the standard deviation σ
5: Calculate $T_{thresh} = RTT_{avg} + 4 * \sigma$
6: Set $T_{sleep} = 0$ and $T_{step} = 100ms$
7: $T_{sleep} = T_{sleep} + T_{step}$
8: Sleep for T_{sleep}
9: Send a ping and calculate the RTT (RTT_{ping})
10: **if** ($RTT_{ping} < T_{thresh}$) **then**
11:     Go to 7
12: **else**
13:     Note the current system time as T_1 and last sleep time T_{sleep}
14: **end if**
15: Sleep for T_{sleep}
16: Send a ping and calculate the RTT (RTT_{ping})
17: **if** ($RTT_{ping} > T_{thresh}$) **then**
18:     Idle timeout $T_{soft} = T_{sleep}$
19: **else**
20:     Hard timeout $T_{hard} = T_1 - T_0$
21:     Note current system time as T_2
22:     **while** $(T_2 - T_1) < (0.8 * T_{hard})$ **do**
23:         Note current system time as T_2
24:         Sleep for T_{sleep}
25:         Send a ping
26:     **end while**
27:     **while** ($RTT_{ping} < T_{thresh}$) **do**
28:         Send a ping and calculate RTT_{ping}
29:         Sleep for T_{step}
30:     **end while**
31:     Hard timeout T_{hard} = Current time $-T_1$
32: **end if**

because of error in estimation ranging in the difference of the last two consecutive pings (T_{sleep}). The hard time out may have occurred anywhere in the duration of the last T_{sleep}. We now try to reduce the error in estimation by pinging at a granular rate than T_{sleep}. The host knows the value of hard timeout with error (T_{hard}) and that T_{sleep} is not the soft time out as flow

entry was not expunged after sleeping for T_{sleep} (Line 17, Algorithm 4.1). Host will send the pings gapped at T_{sleep} for the 80% of T_{hard} (Lines 22 - 25, Algorithm 4.1) and after that it will start sending continuous pings after every T_{step} (Lines 27 - 28, Algorithm 4.1) i.e., very small value as compared to the T_{sleep}, in order to reduce the estimation error. Pings will be sent until a flow entry expiry event occurs (Line 27, Algorithm 4.1). The hard timeout is now calculated as $T_{hard} = Currenttime - T_1$ that is not dependent on the ratio between sums of all pings and actual value deployed in the network. Algorithm 4.1 thus returns the value of first timeout and ascertains its nature as well.

The proposed technique is trailed by two other algorithms. If hard timeout T_{hard} has been discovered in Algorithm 4.1 then control is transferred to Algorithm 4.2, as it is intended to discover the estimation of soft timeout T_{soft}. If soft timeout T_{soft} has been found in Algorithm 4.1 then control is transferred to Algorithm 4.3 as it is intended to discover the estimation of hard timeout T_{hard}. After Algorithm 4.1, control is transferred to only one of these algorithms.

Let's assume Algorithm 4.1 has found the value of hard timeout and now the objective is to discover the soft timeout, which is the design goal of Algorithm 4.2. Now the design logic of Algorithm 4.2 is explained, as shown in flowchart (Figure 4.5). First of all, host will remove the previously installed flows i.e., by sleeping for T_{hard}. Now, host sends a ping to install a fresh flow entry and notes the current system time as T_0. In this Algorithm, 50 ms has been used as the value of T_{step}. Now, host begins figuring of RTT_{ping} utilizing T_{step} additions in sleep time to build the time gap between progressive pings as in Algorithm 4.1. Initial value of T_{sleep} is imported from the algorithm 4.1 rather than starting it from the value of 0, because soft timeout has not been found till that value, so to make the design efficient, previous value of T_{sleep} is used. Whereas, the purpose of changing the value of step size is to obtain the more fine grained results. But, the criteria for checking for flow entry has been expunged or not remains the same i.e., T_{thresh} is also imported from algorithm 4.1. Now, RTT of test pings (RTT_{ping}) are compared with T_{thresh} to detect whether the flow entry has been expunged or not. If the flow has not been expunged i.e., RTT_{ping} remains less than the T_{thresh}, process of increasing the value of T_{sleep} will repeat (Line 5 in Algorithm 4.2), until the value of T_{sleep} exceeds from the value of T_{hard} and then Algorithm 4.2 terminates by declaring that no value of T_{soft} has been set (Lines 6-9 of Algorithm 4.2) i.e., the value of soft timeout is set as 0 (Combination B). In case RTT_{ping} exceeds the T_{thresh} indicating that the flow has been expunged, the current system time as T_1 and calculate the total elapsed time ($T_1 - T_0$) since the flow was originally installed in line No 4 of Algorithm 4.2 and we

noted the system time in variable T_0. If this total elapsed time is equal or more than the T_{hard}, the flow has actually been expired due to the hard time out not due to the soft timeout.

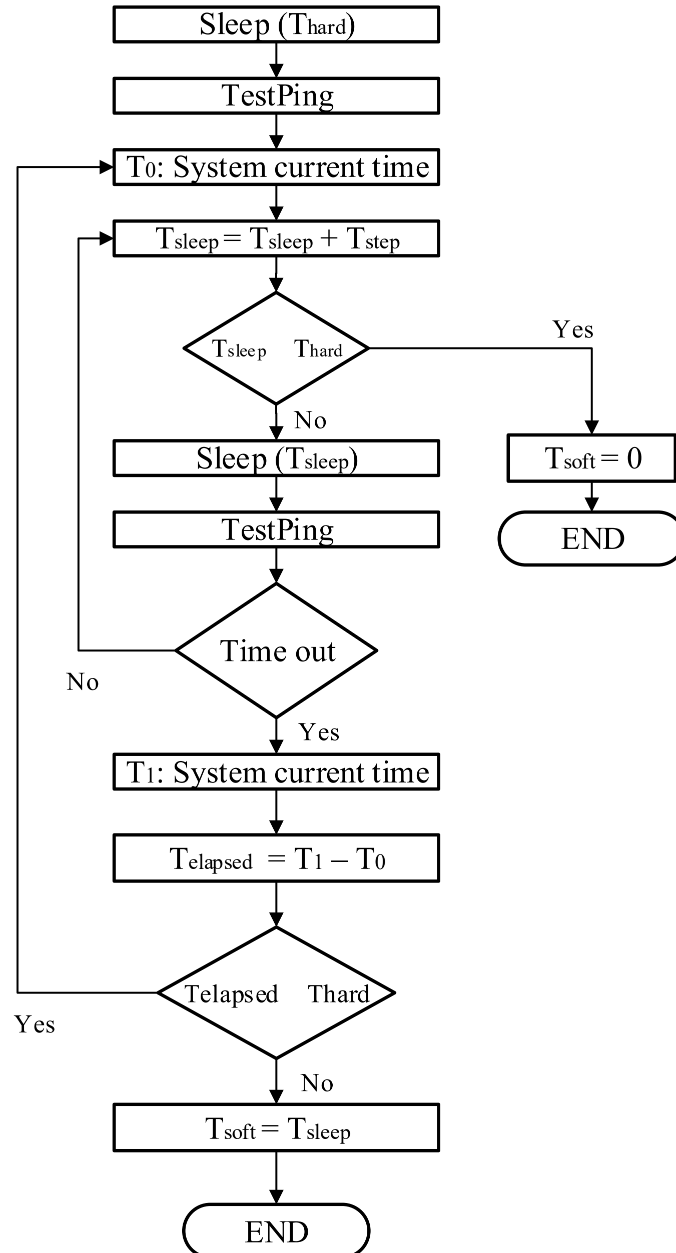

Figure 4.5: Flowchart - Finding hard timeout given hard timeout (T_{hard})

Flow entry expiry due to hard timeout, such event is occurred because design logic is incrementally testing for soft timeout. Consider a network

Algorithm 4.2 Find idle timeout given hard timeout (T_{hard})

1: Sleep for T_{hard}
2: Initialize T_0 = Current system time and $T_{step} = 50ms$
3: Use T_{sleep} and T_{thresh} from Algorithm 4.1
4: Send a ping to install the flow
5: $T_{sleep} = T_{sleep} + T_{step}$
6: **if** $T_{sleep} \geq T_{hard}$ **then**
7: Idle/soft timeout $T_{soft} = 0$
8: Terminate
9: **end if**
10: Sleep for T_{sleep}
11: Send a ping and calculate RTT (RTT_{ping})
12: **if** ($RTT_{ping} < T_{thresh}$) **then**
13: Go to 5
14: **else**
15: T_1 = Current system time
16: **if** $(T_1 - T_0) \geq T_{hard}$ **then**
17: $T_0 = T_1$
18: Go to 5
19: **else**
20: Idle timeout $T_{soft} = T_{sleep}$
21: **end if**
22: **end if**

where hard timeout is set as 30 seconds, soft timeout is set as 15 seconds and value of T_{sleep} imported from Algorithm 4.1 is 3 seconds. In Algorithm 4.2, using 50 ms as value of T_{step}, host sends the first ping and then sleep for 3.05 seconds ($T_{sleep} = T_{sleep} + T_{step}$) before sending the next ping and then sleep for 3.10 seconds and then wait for 3.15 seconds before sending the third ping and so on. After sending the second ping (testing for soft timeout for 3.10 seconds), total elapsed time would be 6.15 seconds since T_0. In the same fashion, after sending the third ping (testing for soft timeout for 3.15 seconds), total elapsed time would be 9.2 seconds since T_0. This relationship is expressed in the form of Equation 4.2, which returns the total elapsed time at n_{th} ping where $T_{sleepinit}$ represents the initial sleep time, imported from Algorithm 4.1. Within 10 pings, the total elapsed time $T_{elapsed}$ will be approximately 32.75 seconds. At the 11th ping T_{sleep} is 3.55 seconds that is still less than the actual value of T_{soft} i.e., 15 seconds.

$$T_{Elapsed} = \sum_{i=1}^{n} (T_{sleep_{init}} + (i) \times T_{step}) \tag{4.2}$$

However, a new flow will be installed by this 11th ping as the previous flow entry was removed due to total elapsed time being greater than T_{hard} (32.75 seconds > 30 seconds). In such a case, the reference time of flow installation is changed and proceed with the last probed value of soft time out (lines 16-18 in Algorithm 4.2). In the next round, value of T_{sleep} is used from the last round, so at each new round sleep time (gap between consecutive packets) is higher i.e., in the second round initial value of T_{sleep} will be 3.6 seconds. Otherwise it will become an infinity loop if initial value is same in each round. Moreover, as the number of rounds increases T_{sleep} will approach to the value of soft timeout or declare it as zero if it exceeds from hard timeout. Finally, the algorithm will terminate when the T_{soft}, if set is found. Number of rounds i.e., how many times hard timeout will occur before soft timeout is found, can be calculated by the dividing the total elapsed time since the first ping by the actual value of hard timeout i.e., Equation 4.3.

$$Numberofrounds = \frac{T_{Elapsed}}{T_{hard}} \tag{4.3}$$

Now consider the second case where Algorithm 4.1 has discovered the value of soft timeout and control will be transferred to Algorithm 4.3. In Algorithm 4.3 (Flowchart in figure 4.6), first of all, previous existing flow entry should expire i.e., host A will sleep for T_{soft}. There is no need of T_{step} in this part, as the pings are equally gapped from each other. Although value of T_{sleep} is imported from Algorithm 4.1, but as it equals the value of soft timeout so a new value of sleep time is required, 75% of T_{soft} has been proposed to avoid the removal of flow entry during probing because of soft timeout. This number is selected because marginally it is a good ratio to avoid uninstallation of flow entry without sending too many packets at lower value. Host A sends test pings and calculate RTT_{ping} to check for flow entry expiry.

Value of T_{thresh} is imported from Algorithm 4.1 for comparing of RTT_{ping} of these test pings. After each ping, current system time T_1 is noted and total elapsed time $(T_1 - T_0)$ is checked. Lines 9 - 11 in Algorithm 4.3 specifies the threshold of the range we are interested to probe (10 times the value of T_{soft}) for hard timeout. A ratio of 10 covers almost all default combination of SDN controller's timeouts and practically implementing ration higher than this is also not feasible. If hard timeout is found within this range, total

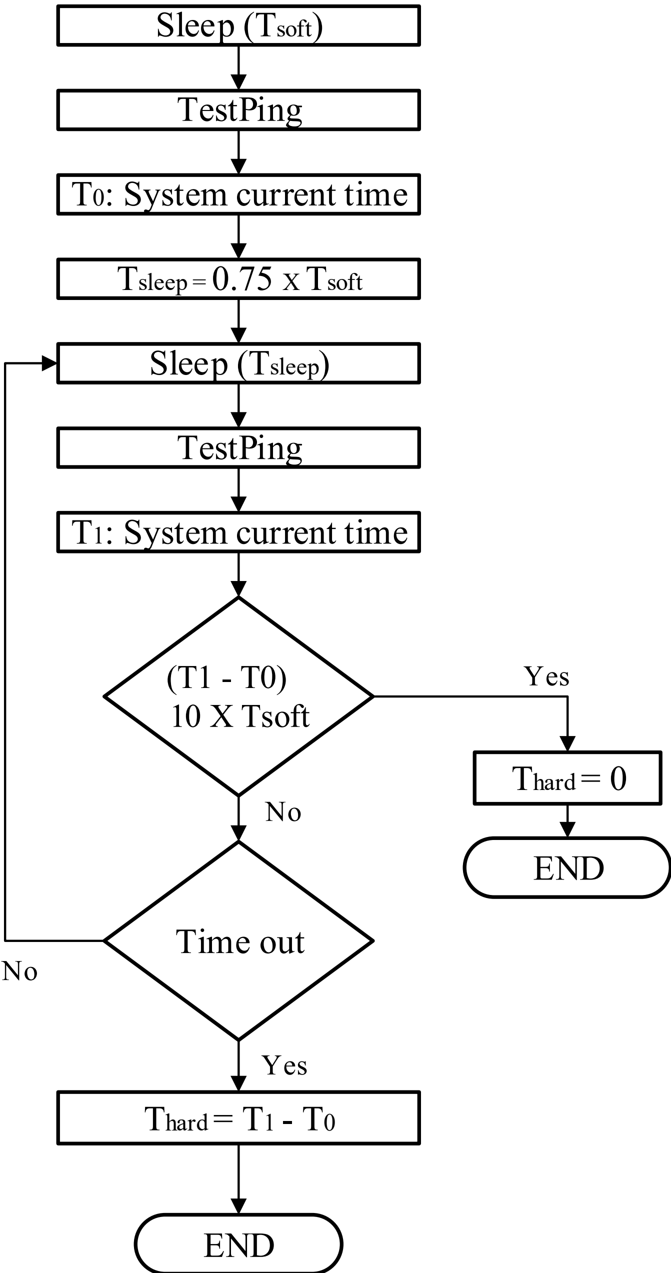

Figure 4.6: Flowchart - Finding the hard timeout given the soft timeout (T_{soft})

elapsed time (current time - T_0) since the first ping will be the value of hard timeout, otherwise it declares the hard timeout equal to 0. In the worst case, when hard timeout is 0, the total run time will be almost 14 times of the

Algorithm 4.3 Find the hard timeout given the idle timeout (T_{soft})

```
1:  Sleep for T_soft
2:  T_0 = Current system time
3:  Use T_thresh from Algorithm 4.1
4:  T_sleep = 0.75 * T_soft
5:  Sleep for T_sleep
6:  Send a ping and calculate RTT_ping
7:  if (RTT_ping < T_thresh) then
8:      T_1 = Current system time
9:      if (T_1 − T_0) ≥ (10 ∗ T_soft) then
10:         Hard timeout T_hard = 0
11:         Terminate
12:     else
13:         Go to 5
14:     end if
15: else
16:     Hard timeout T_hard = Current system time −T_0
17:     Note current system time as T_2
18:     while (T_2 − T_1) ¡ (0.8 * T_hard) do
19:         Note current system time as T_2
20:         Sleep for T_sleep
21:         Send a ping
22:     end while
23:     while (RTT_ping < T_thresh ) do
24:         Send a ping and calculate RTT_ping
25:         Sleep for T_step
26:     end while
27:     Hard timeout T_hard = Current time −T_2
28: end if
```

T_{soft}. Otherwise, it depends on the actual configured value of hard timeout. The impact of timeouts has been analyzed in Chapter 5, along with the granularity of step size and its impact on the error and total cost. Moreover, error in the discovered value of hard timeout has been reduced (Lines 18 - 26, Algorithm 4.3) by the similar mechanism as in Algorithm 4.1 i.e., sending pings with gap of T_{sleep} for 80% of detected T_{hard} and then sending pings gapped equally by the T_{step}.

4.2 Match Fields

The goal of this module is to detect the match fields implemented by the SDN controller's policy and firewall elements associated with the detected match fields. The controller allows/blocks installation of a flow based on certain match fields or even particular values of those match fields. The concept of match fields has been explained in section 2.3. The design goal of this module is to find out the match fields for flow entries and which of legal values of those detected match fields are blocked for transmission by the SDN controller. In simpler words, firstly, proposed scheme detects the match fields, whose manipulation yields new flow entries at the switches in the network. And if a certain header field is a match field then checking for declared legal values under firewall policy of the controller deployed in the network. For now, consider a simple example, it might be possible that a SDN controllers policy is considering TCP port number as a match field but controller has blocked all incoming traffic of HTTP port (TCP port # 80) or SSH port (TCP port # 22), then sending forged packets using such value of match fields or launching DDoS attacks of those values of match fields is totally useless. This information would come very handy when an adversary is trying to consume the maximum resources of the SDN network by installing as much flows as possible for producing DDoS scenario. The adversary can use compromised host(s) to fingerprint the SDN enabled network to determine the allowed/blocked match fields to update information about the implemented firewall policy. For efficiency and effectiveness reasons, adversary can ignore the blocked match field values while launching an attack.

It is worth mentioning that, limited information can be gathered about fingerprinting SDN when there is only one compromised host under the adversary's access (Figure. 4.1) in the network. For example, it is hard to determine if a particular flow or packet is blocked due to the controller policy (Firewall feature) or by the receiving host firewall. However, this situation improves when an adversary has access to two compromised hosts under control (Figure. 4.2), in such a scenario it can be assured that receiving host has not blocked any port or any flow at its firewall. So, whatsoever is dropped is due to the controller firewall functionality.

The actual process of fingerprinting the match fields is quite straight forward. The host will pick a match field and forge a packet using a random legal value of that match field. Now, host will wait and check weather a new flow has been installed for this forged packet or not. Checking for new flow installation is based on comparison of difference in RTT i.e., RTT of test packets RTT_{test} greater than T_{thresh} as explained in timeout (Section.

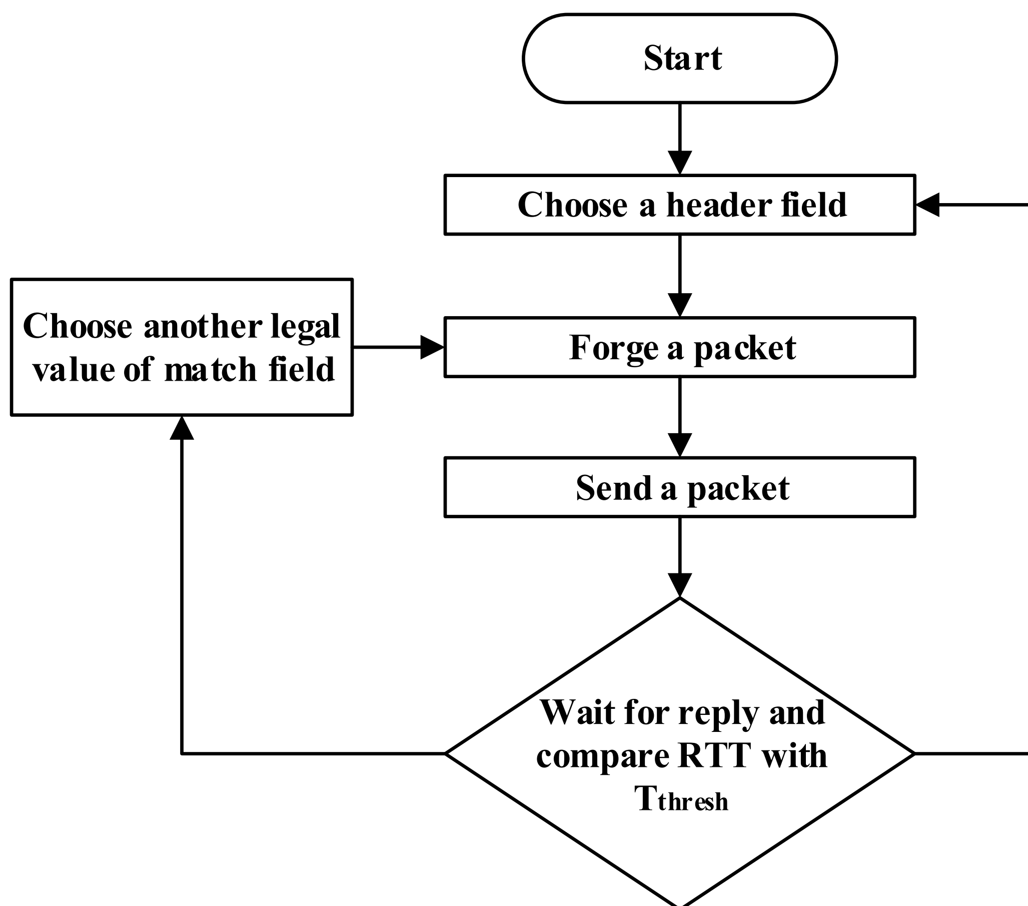

Figure 4.7: Flowchart - Match fields fingerprinting

4.1) module. Based on the difference in RTT, it is inferred that a new flow has been installed for this match field or not. If the packet is dropped for a value of match field, the algorithm will check for another legal value of that particular match field and note that value of match field for dropped packet. This is followed by choosing another legal value for the same match field (keeping all other fields as unmodified) for further analysis. For example, at transport layer, if first match field used is TCP port # 88, then in the next packet it can try TCP port # 80 (Flow chart in Fig. 4.7). The adversary may check for all legal values of a particular match field against which he wants to launch a DDoS attack. Now the question arises for how many values, an adversary should check firewall policy? And the answer is as many as he desires, the design logic of this module will check firewall policy for all match fields and all declared values in which the adversary is interested. Checking for firewall policy is also not compulsory, he may just check for

match fields only using one packet. RTT for each match field depend upon that particular header field communication behavior. For example RTT of ICMP is calculated on the bases of reply sent by the host. Similarly, in case of TCP, if the targeted host is listening on a port than SYN + ACK will be received in the response of a SYN packet, RST + ACK will be received in the response of a SYN if the other host is not listening on that particular port and RST will be received in the response of a ACK packet. However, no response will be received if a TCP is blocked either by the host firewall or by the network policy and this situation improves when attacker has two hosts under control inside the network. Note that nested matching is also possible. TCP flag field is a common example for such a scenario where TCP flag is treated as a separate match field along with TCP port numbers. Scapy [23] has been used for the testing and implementation of this module. Moreover, this module has been tested on the commonly used match fields such as IP addresses, Protocol types, MAC addresses, transport layer protocols (TCP & UDP), Transport layer port numbers (TCP port numbers and UDP port numbers), TCP flags, ICMP, ICMP types and ToS (Type of Service).

Moreover, some other useful information can also be obtained about the network using the detected match fields, such as OpenFlow version deployed in the SDN enabled network. Moreover, an algorithm has also been proposed to verify timeouts values efficiently for each detcted match fields. Now, both of these feature are briefly explained below:

4.2.1 OpenFlow Version

OpenFlow has become a de facto standard southbound API which provides access to administrators for managing network devices. In its initial version i.e., Openflow 1.0 there were only 12 match fields available. However, with the emergence of SDN and network applications, the latest available version (Openflow 1.5) has almost 44 match fields. Further development of its next version is in process and is yet available to ONF (Open Networking Foundation) members only. There are match fields which exist in all versions of Openflow e.g. TCP ports, Ethernet addresses, IP addresses, to name a few. However, existence of enhanced fields varies from one version to another. For instance, MPLS to provide QoS was not part of Openflow1.0, but has been incorporated in later versions. Though existence of protocol specific fields eases network application developers and administrators, however, this incurs redundancy and increases size of header of the Openflow packet, thereby increased demand of network bandwidth. With these considerations, a single space in match fields has been reserved for mutually exclusive protocols. For example, considering transport layer, from TCP, UDP and SCTP, only one

is used at a time for a flow. In this case, same match fields are used, without considering the protocol being used. Where evolved 44 match fields provide a lot of flexibility for network applications, however, these also impose a threat to a SDN. For example, generally speaking, If maximum number of 44 match fields are being used, one might need least effort to overflow the switch flow table entries by launching an attack with various protocols and there unique header values (ports, addresses etc.). Moreover, on the basis match fields, OpenFlow version can also be predicted but not fully ensured of exact version, provided if a unique match field is being found. For example MPLS BoS was introduced as a Match field in OpenFlow 1.3, but it was also supported by all latter versions. So if a network has implemented MPLS BoS means network has implemented OpenFlow 1.3 or latter version. If an adversary has information about the deployed OpenFlow version, he may also predict features of that network which have been introduced for first time such as security features, unique messages (between switch and controller), etc.

4.2.2 Variation in Timeouts

A SDN controller might allocate different timeouts to different match fields, which means different types of flow entries (match fields) may have different life in flow tables of switches. This can be dependent upon the traffic characteristic of that particular SDN enabled network. As SDN offers dynamicity and flexibility. An adversary may be interested in a particular match field such as TCP port numbers only. But, as it is not ensured that a network administrator has deployed same timeouts for all types of traffic. The design goal of this module is to investigate timeouts of a particular match field efficiently, using the values of initial found values of timeouts, rather than running whole module of timeouts again.

As the basic timeouts, investigated on the base of ICMP pings are known. The design logic of this module (Flowchart shown in figure 4.8) first check for those values, if they are applicable on the match field under consideration or not. First of all, host will choose a match field, forge the packet using legal value of match field and sends toward the targeted host to install the flow entry by noting the current time of system. Firstly, host will check for hard timeout value i.e., by sending equally gapped packets and checking for flow entry expiry by comparing RTT of test packets (RTT_{test}) with T_{thresh} at the known timeout value from ICMP pings. If hard timeout has not been found for ICMP pings then this step will simply get ignored and control will be transferred to next step. In either case, the next step is to match the soft timeout, host will investigate that if soft timeout of ICMP is also applicable on the target match field or not. Next, host decides if the goal of this module

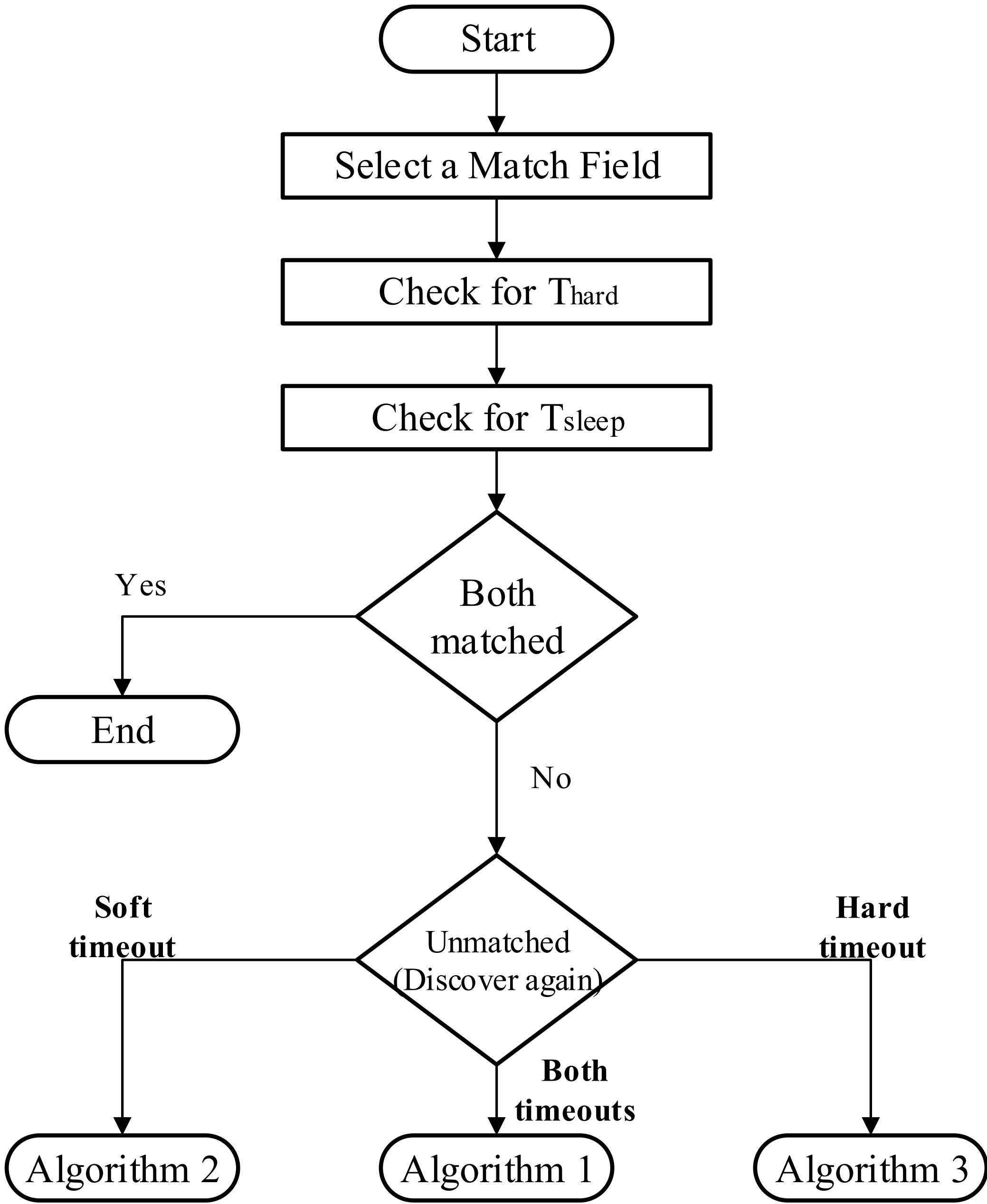

Figure 4.8: Flowchart - Detecting timeouts for multiple match fields

has been achieved or not i.e., if both timeout values of the targeted match field are same as of known timeout values of not. If the values are same, the module terminates. But if any of value is not matched than the respective algorithm from the timeout section will be called for the targeted match field. Means, if hard timeout is matched but soft timeout is different than

the control transfers to Algorithm 4.2 and if soft timeout is matched but hard timeout is different than the control transfers to Algorithm 4.3. And if both of timeouts dont match with the known values than the control is transferred to Algorithm 4.1, which will compute the timeout for the targeted match field from scratch. It is worth mentioning that matching means, initial found values of timeouts should also holds their respective actions on the targeted match field. But if flow expiry has not been occurred at the respective time or no initial value is found for timeout i.e., value of 0, than it is considered as a matched value, hosts needs to investigate it. For example, consider a network where ICMP is configured at 30 seconds and 0 seconds for soft and hard timeout respectively. But TCP, in the same network is configured at 30 seconds and 60 seconds respectively. Host will not check for hard timeout, as it is set 0; value returned by the ICMP timeout module. Host matches the value of soft timeout and then computes the hard timeout by calling Algorithm 4.3.

Running this module on each match field is not compulsory at all. If an attacker wants to launch an attack using less match fields, it will be helpful to him. Or he may choose wrong packet generation rates or number of hosts for an attack based on wrong timeouts. So he may need to verify the timeouts for the desired match field(s), if required.

4.3 Mitigating Over Flowing Tables

SDN enabled switches can support only few thousand flow entries at maximum, as elaborated in section 2.5. Typically, SDN enabled switches supports both software flow tables as well as hardware flow tables. Hardware flow tables process at higher speed but on the other hand, they can accommodate only limited numbers of flow entries. Whereas processing speed of software flow table is much lower; degraded by up to two orders. Software flow entries tables are managed in SDRAM (Static Dynamic Random Access Memory) whereas hardware flow entries tables are maintained in either BCAM (Binary Content-Addressable Memory) or TCAM (Ternary Content-Addressable Memory). As switches can accommodate only limited number of flow entries, which further reduces the performance [24] of the network because additional messages exchange are involved between the switches and the SDN controller to remove the already installed flow entries and then installing the new entries. The design objective of this module is to fingerprint the controller mitigation policy for full table event in SDN enabled network. And how the implemented mitigation policy effects the network under DDoS scenario.

The design logic assumes that host has already fingerprinted the values of timeouts and valid match fields implemented in the SDN enabled network. Fingerprinting the match fields helps the host to forge an effective burst of packets, from which each packet will enforce the controller to install a new flow entry. And fingerprinting the timeouts of the SDN enabled network, helps the host to determine the type of burst needed in order to overflow the flow entries tables efficiently.

The first step of the proposed mechanism requires to fill the flow entries at the bottleneck switch toward the target host in the network. And to full the flow table, host needs a burst of packets which enforce the controller to install the distinct flow entries for each packet, as host has the information of the deployed match fields. It is worth mentioning that if a host has checked the firewall module of the values of match fields, those values of match fields should be ignored while generating the traffic burst. But host should also consider the impact of deployed timeouts on the packet burst.

If T_{soft} is not defined in the network, host does not have to repeat the values of match fields in forged packets in order to hold the flow entries table. On the other hand, if T_{soft} is defined in the network, than host needs to forge packets at a high rate and needs to repeat the values of match fields in order to generate as much flow entries as possible before the switch start removing flows through soft timeout mechanism. Means, auto flow expiry due to soft timeout should be avoided, especially when the value of soft timeout is too much low. However, if T_{hard} is defined in the network then host needs not to repeat the values of match fields at all, as by the definition of hard timeout flow entries will be removed at T_{hard}. But the most ideal situation for an attacker will be the network where both of timeouts are not defined (Case A in Table 2.1). At this stage, when flow table is full, host begins fingerprinting of mitigating policy for full flow table event.

There are two mitigating strategies for overflowing tables at the switches. The default applications of SDN controllers do not react to table full error messages sent by the switches, as they rely upon timeouts mechanism for auto expiry of flow entries. But there exist another mitigating strategy for overflowing tables in which switch removes the flow entries when instructed by the SDN controller, upon receiving the table full error message by the respective switches. But the question arises, on what bases controller selects the victim flow entries i.e., which have to be removed? Well, the selection of flow entries to be removed varies, such as it can be based on traffic statics gathered by the SDN controller or on the bases of one or more match fields.

In the first approach (no response on receiving the table full error message), there will be significant packet drops, as shown in figure 4.9. Whereas in the latter approach (flow replacement policy on receiving the table full

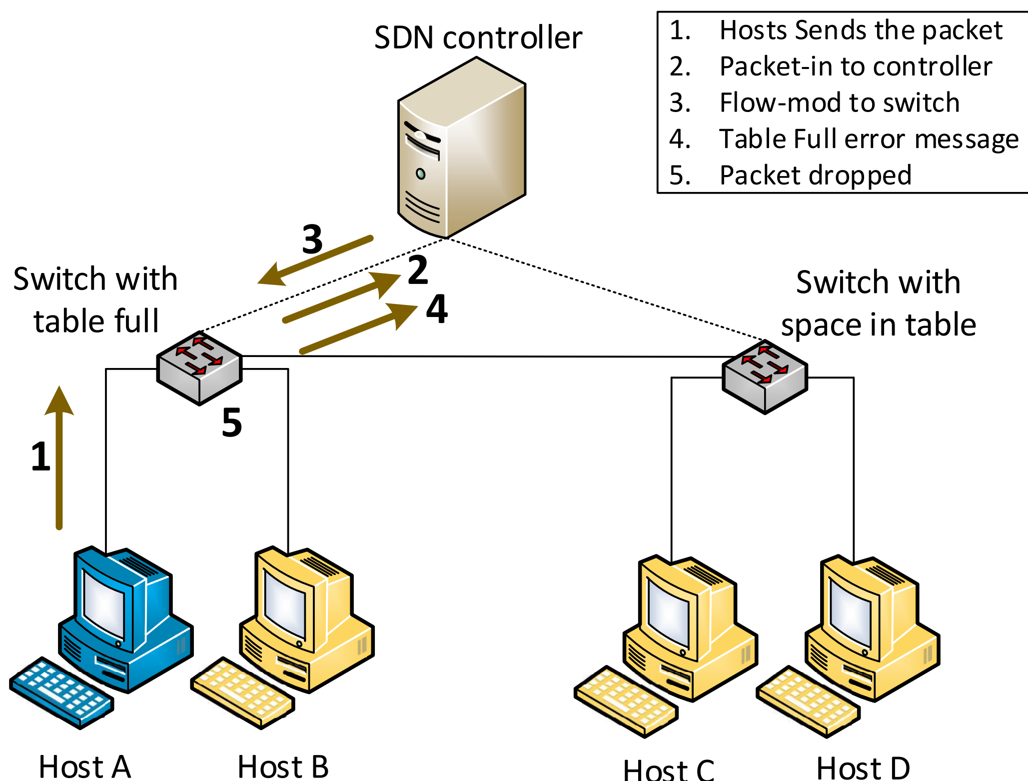

Figure 4.9: Packet drops policy due to full table

error message), there will be increased load on the SDN controller which results in increased RTT of forged packets sent by the host, but there will be no packet drops. Normally in case of table miss event, switch sends the packet-in to the controller and then controller sends the flow rule in flow-mod message for the respective packet and switch forwards the packet i.e., RTT_{FE}. On the other hand, when table full event occurs at the switch, it sends the error message (table full error, flow entry not installed) against the flow rule message rather than installing the flow entry at the switch, then the controller sends the flow entries deletion message to the switch and then resends the flow rule. Which means there are three extra messages involves in this event which also results in increased RTT, represented by $RTT_{reinstall}$. This has been elaborated briefly in the figure 4.10 with all involved steps in it. So, in normal mode (flow table not full), RTT will be RTT_{FE} for every packet which generates a new flow entry. Where as in case of full flow table, either the packet gets dropped or RTT increases to $RTT_{reinstall}$.

When the host detects the table full event has been occurred i.e., first packet is dropped or the RTT increases to $RTT_{reinstall}$. The host notes the

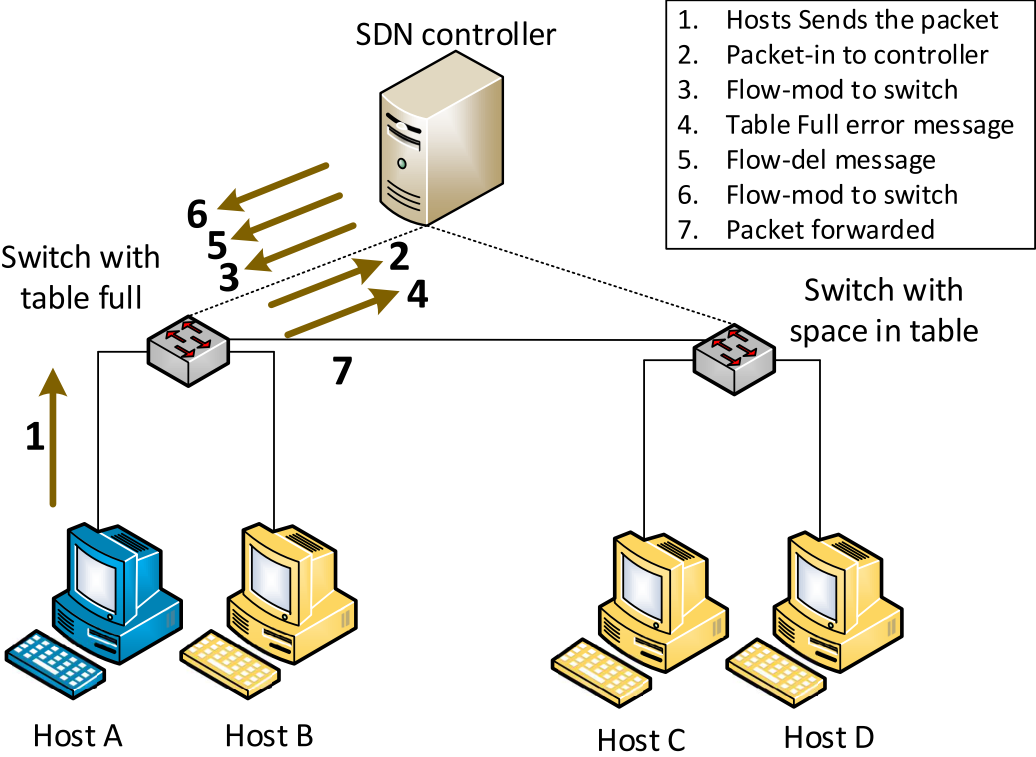

Figure 4.10: Flow replacement policy to overcome full table

elapsed time ($T_{elapsed}$) since the first forged packet of the burst and multiplication of it with packet generation rate $Rate_{packet}$ (burst rate) can be used to estimate the number of ow entries i.e., how many distinct flow entries have been generated by the burst in the flow entries tables. But this number cannot be inferred as the total number of flow entries accommodatable at the switch. In fact, exact number of flow entries accommodatable at switches are unpredictable because in the commercial SDN network there will be already hundreds of flow entries installed at the switches for ongoing packets transmissions between the multiple hosts. It gets worst, if the SDN network has hard timeout set as 0, which means already installed flow entries will not be expunged as the soft timeout renews the flow entry life at each packet arrival of the same flow. But still using the proposed scheme, it can be observed how SDN enabled network will behave under DDoS attack at flow table entries. However, this module focuses on analyzing the impact of full tables under the DDoS condition only, criteria for deletion of already installed flows varies from policy to policy such as on the base of a match field or any other predefined policy is beyond the scope of this research.

4.4 Topology Discovery

If an attacker have the information about the topology of the network, it helps him to launch much strong and mature attacks on the targeted network. In case of SDN enabled network, best case will be targeting a host with maximum hops in the network, so that SDN controller will have to install flow entries again and respond to table full error messages when flow entries table full event occurs at each hop in the network. Even in the normal mood, there is significant difference in RTT, when a flow entry already exists (RTT_{avg}) as compared to when a controller installs a flow entry (RTT_{FE}). This will also happens at each hop between the end hosts inside the network, which further increases the difference in RTTs. The design goal of this module is to predict the network topology, by using the trend in difference of RTTs. But there are multiple topologies, used by the commercial networks but design logic of this module considers fat-tree topology topology only. Basic fat tree topology has been shown in figure 4.11. Fat tree topology is the most commonly used topology in data center networks, as it provides many redundant links [25]. It consists of three layers, edge layer, aggregation layer and core layer. Switches in the edge layer are known as edge layer switches or ToR (Top of Rack) switches. ToR switch allows the communication between the hosts of a same rack and connects them to upper layer of the topology i.e., aggregation layer switches. No two ToR switches can communicate directly with each other, there should be at least one aggregation layer switch, which connects two ToR switches and also connects to the upper layer i.e., core layer switch. Similarly, no two aggregation layer switches can communicate directly with each other, there should be a core layer switch, which connects at least two aggregation layer switches. Interconnected ToR switches using one aggregation layer switch only i.e., without involving core layer forms a pod, as highlighted in dotted squares in figure. 4.11

The design logic of this module requires that an adversary should know the IP addresses of hosts inside the SDN enabled network in advance or can find them by using network scanning applications i.e., ARP scan function of Scapy or any other network scanning application. Then host can ping them and using the retrieved information, hosts inside the target network can be sorted into in-rack, in-pod and inter pod categories. In-rack communication involves only one switch whereas in-pod communication involves three switches and inter pod communication involves five switches. Similarly, their respective RTTs will have an increasing trend as the packets are traversing one, three and five switches respectively. Although this trend is not linear in nature but still enough to categorize the hosts. Especially when a new flow entry has been installed for each packet, as RTT_{avg} is a smaller number

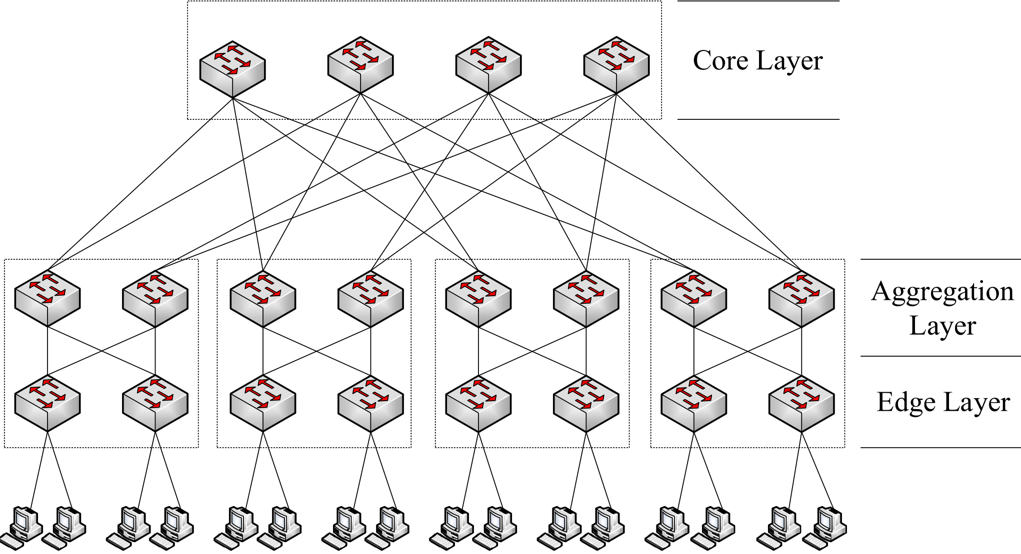

Figure 4.11: Fat tree topology

there is less trend in variation whereas RTT_{FE} is a bigger number, which makes it easier to decide. Moreover, incase if flow replacement policy (table full scenario) is being implemented by the controller, probability of categorizing hosts in the target network further increases, as $RTT_{reinstall}$ can help to retrieve further confirmation about the hosts. However, if controller is not using flow replacement policy than RTT_{FE} is the only best option.

Chapter 5

Results & Discussion

This chapter contains the discussion on the results of proposed techniques in chapter 4. Respective plots of the designed modules and respective simulation environment has been explained briefly. Section 5.1 summarizes the system specification on which experiments had been performed for the evaluation. Section 5.2 discusses the granularity of the step size for investigating the timeout values. Next, section 5.3 elaborates the impact of different values of timeouts under the DDoS scenarios and section 5.4 discusses the controller reaction to the table full event and when it can easily degrade the performance of the SDN enabled network. Lastly, discussion on the topology discovery experiments have been summarized in the section 5.5.

5.1 System Specification

The experiments for the evaluation of the proposed techniques are performed on system with the specification given in Table 5.1.

The machine was running Linux as operating system, with Mininet [26] as an emulator installed on it. Mininet was used to run emulated network where each node (Controller, switches and end hosts) of the network is provided as a VM (Virtual Machine) with individual Linux kernel. Moreover, it also allows configurable link parameters such as bandwidth. Python, interpreted high-level programming language, was used to write simple ICMP ping script using the default libraries available in Linux kernel, which returns the RTT for the pings. RYU [28] was used as SDN controller. Scapy API is used to write python scripts for packet forging (for determining match fields and mitigating policy for overflowing tables) purpose such as forging packets with different TCP port numbers, TCP flags, ToS (IP layer header) etc. These python scripts also return the respective RTT of forged packets and response

Table 5.1: System specification

Name	Specification
CPU	Intel Core (TM) i5-7200 CPU 2.7 GHz
RAM	8 GB - DDR4 2133 MHz
GPU	2 GB - GeForce 920 MX
OS	Ubuntu 16.04.03 LTS
Emulator	Mininet 2.3.0d1 [26]
Packet Forger	Scapy [23]
Packet Analyzer	Wireshark [27]

sent by the targeted host to analyze the flow installation time and firewall function i.e., if any value of match field is blocked. The output of the scripts has also been cross checked by the Wireshark [27], packet analyzer tool.

5.2 Granularity of Step Size

In Algorithm 4.1 and 4.2, proposed in section 4.1 for finding timeout values of the SDN enabled network, selection of step size T_{step} has a huge impact on the performance of the proposed timeout's fingerprinting mechanism, as it defines the fashion for time gap between the continuous probing packets. For example, in algorithm 4.1, if T_{step} is chosen as 50 msec than the time gap between 10th and 11th ping will be 500 msec, calculated using equation 4.1 and total elapsed time will be 2750 msec (using equation 4.2). Whereas, if T_{step} is chosen as 200 msec than the time gap between 10th and 11th ping will be 2000 msec, then total elapsed time will be 11 seconds. So, choosing T_{step} effects the time gap between n^{th} and n^{th-1} ping as well as total elapsed time (total time for fingerprinting the timeouts of targeted network).

Conclusively, the step size (T_{step}) is directly proportional to overall running cost (number of rounds) of timeout algorithms and inversely proportional to the error in the values of timeouts. But this is not the case in this mechanism, error in timeout is also dependent upon actual values of timeouts. For example, consider a network is using T_{idle} as 10 seconds and T_{hard} 30 seconds with Algorithm 4.1. If we use step size as 100 msec in Algorithm 4.1, within 25 increments the total elapsed time will be 32.5 seconds and the last sleep will be 2.5 seconds. Now the flow entry will be expunged by the controller due to hard timeout and host will detect an error of approximately 2.5 seconds (lines 1-20 in Algorithm1). Comparing this with the case when

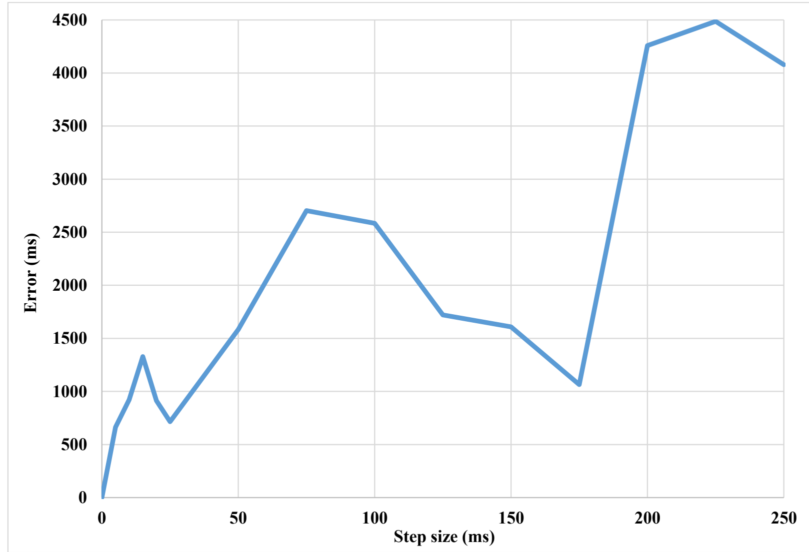

Figure 5.1: Plot - Step size and Error for T_{hard}

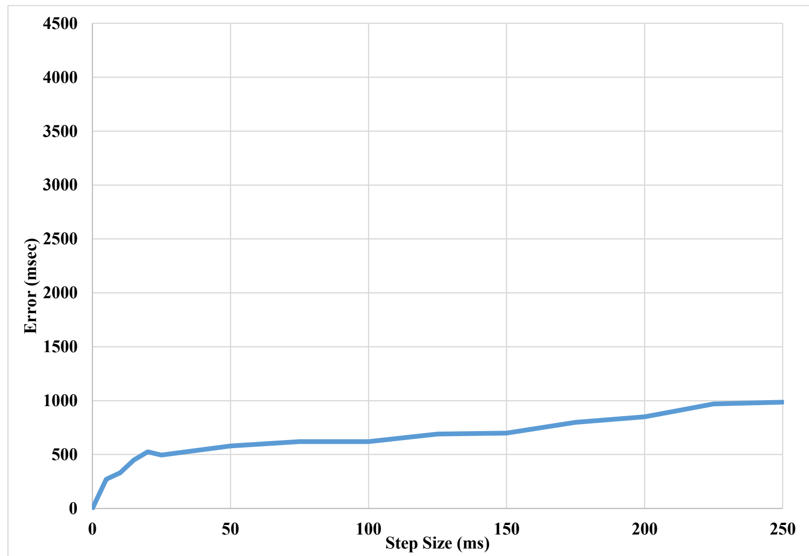

Figure 5.2: Plot - Step size and Reduced Error for T_{hard}

we choose step size of 125 msec in Algorithm 4.1, within 22 increments the total elapsed time will be 31.625 seconds and the last sleep will be 2.75 seconds resulting in an error of approximately 1.625 seconds. Figure 5.1 show this relationship between step size and the error in estimation where the error is fluctuating depending on the values of hard timeout and step size. We thus have to introduce additional processing (lines 21-31 in Algorithm 4.1) to

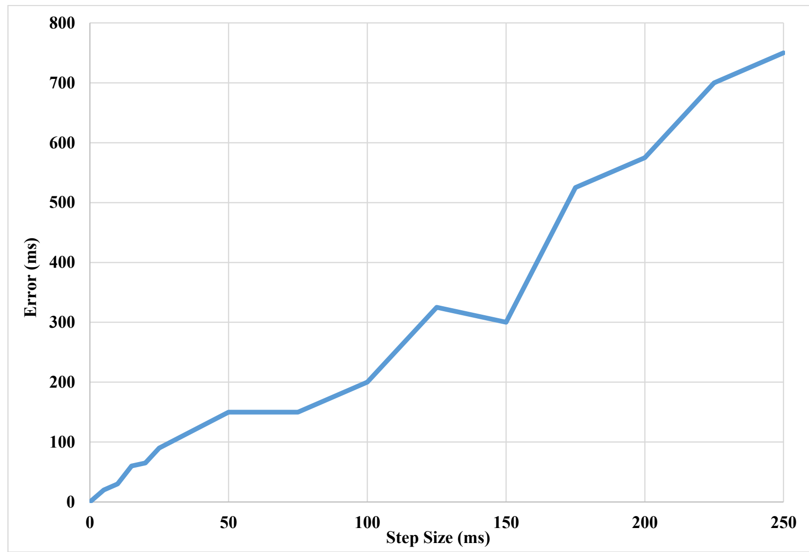

Figure 5.3: Plot - Step size and Error for T_{idle}

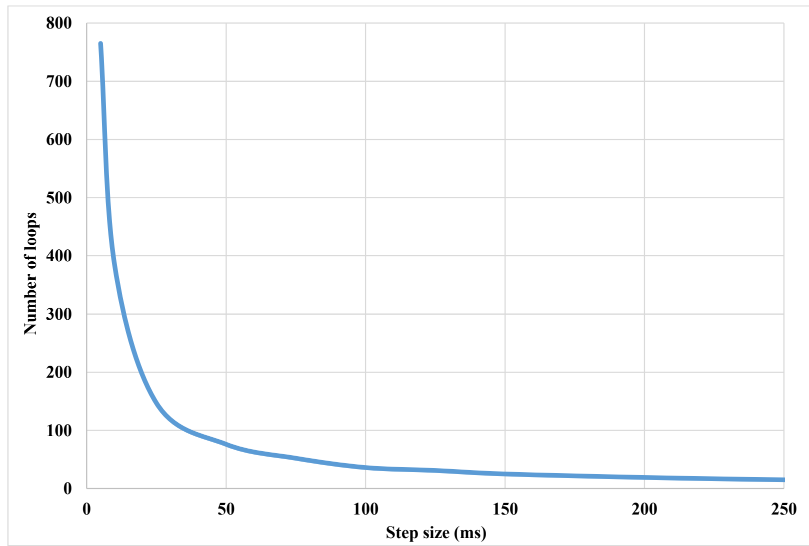

Figure 5.4: Plot - Step size and Number of loops in Algo 2 for T_{idle}

reduce this estimation error resulting in the error shown in Figure 5.2 where the step size of 250 ms is producing error within 1 sec. For the evaluation, topology with one compromised host (Figure 4.1) under the adversary's control has been considered, and also cross verified when the adversary have two compromised hosts (Figure 4.2) under his control. In this module, design logic is based on RTT (Sender side), so the number of compromised hosts in the SDN enabled network does not matter. Choosing T_{step} depends on the

purpose of the fingerprinting i.e., for DDoS attacks or security analyzing by the network administrator.

Once the value of hard time out is found through Algorithm 4.1, this value is used again in Algorithm 4.2 to find the T_{idle}. The fingerprinting mechanism also induces error in estimation of idle timeout as shown in figure 5.3 where the estimation error for idle time out values using T_{step} of 100 msec is about 200 msec (2%) that increases to 750 msec for step size of 250 msec (7.5%). Similarly, T_{step} also effects the number of rounds in Algorithm 4.2, if initial value of T_{sleep} is 3 seconds and T_{step} is 50 msec then there will be almost 72 rounds i.e., Flow removal due to hard timeout before soft timeout is discovered. Similarly, T_{step} is 200 msec then there will be almost 18 rounds. These evaluations are based on the associated equation 4.3, which have been verified by the simulations as shown in graph 5.4, which shows that the reduction in number of loops becomes almost at beyond step size of 100 ms. That's why, the value of 100 msec for T_{step} is selected for Algorithm 4.1.

5.3 Timeout Estimations

Proposed algorithms for discovery of timeouts have been checked for estimation error. In these experiments, each instance is the average of five simulations with the step size set as 100ms. Figure 5.5 shows the difference between actual and discovered timeout values using the Algorithm 4.1, for the case when only T_{hard} is set in the targeted network. X-axis of the plot represents the different trials numbers whereas different timeout values and Y-axis shows their respective discovered values along with the percenatge age difference. Algorithm 4.1 show low estimation error when discovering the hard time out values with the error getting amortized with higher values of hard timeout. Figure 5.6 shows the error in estimation of soft timeout when this timeout value is discovered by Algorithm 4.1 (Combination C in Table 1, hard time out is set as 0). The experiment was conducted with soft timeout values ranging between 10 sec to 40 sec. The results show the maximum average estimation error of about 3.4 percent for the case when idle time out is being estimated by Algorithm 4.1 for actual timeout value of 10 seconds.

The combination D listed in Table 1, where both hard and idle timeout values are being used in the policy can be discovered using Algorithm 4.1 followed by Algorithm 4.2 or Algorithm 4.3. The actual sequence of algorithms depends on the ratio between the two timeout values and the T_{step}. For example, let's consider a scenario where the idle timeout is set as 6 seconds and hard timeout is set as 40 seconds and the step size is 100 ms. In

this case, a flow entry will be expunged within 28 pings because total elapsed time would be 40.6 seconds, resulting in the discovery of the hard time out value first.

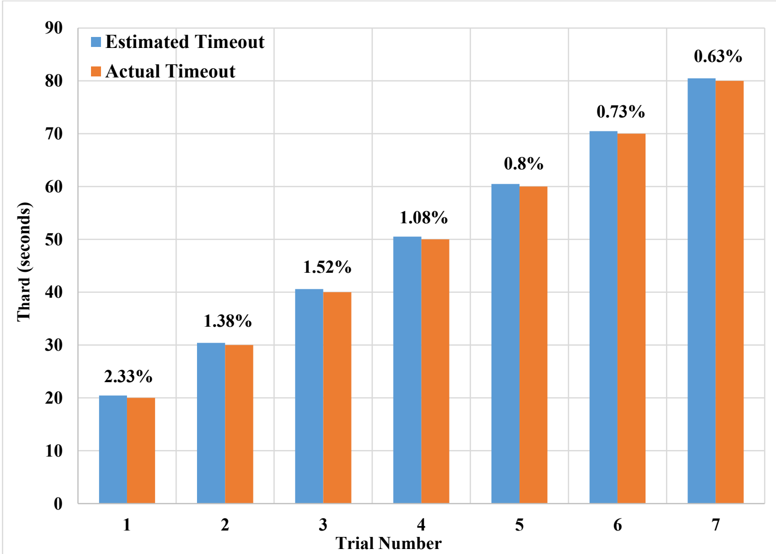

Figure 5.5: Plot - T_{hard} discovery

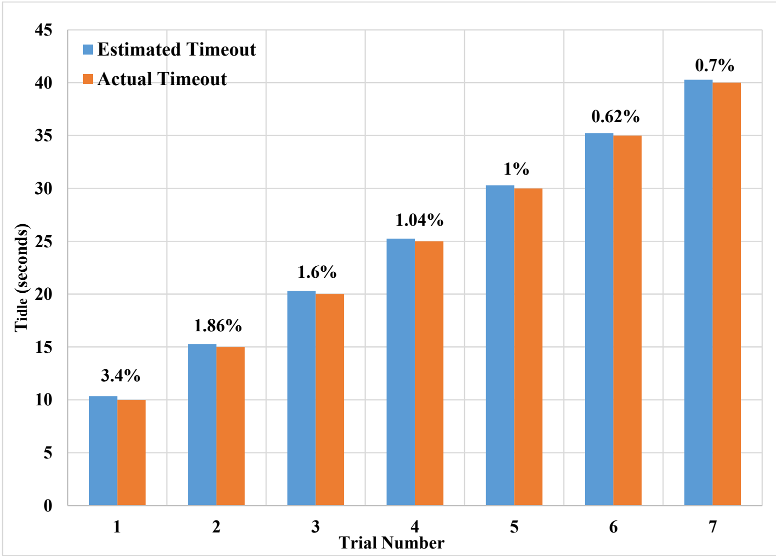

Figure 5.6: Plot - T_{idle} discovery

Algorithm 4.1 will be followed by Algorithm 4.2 for discovery of soft timeout. Whereas, if we change the step size to 750ms than after 8 pings,

a flow entry will be expunged because of idle timeout and at that instance total elapsed time would be 27 seconds. So, Algorithm 4.1 will be followed by Algorithm 4.3 for discovery of soft timeout, if that value exists.

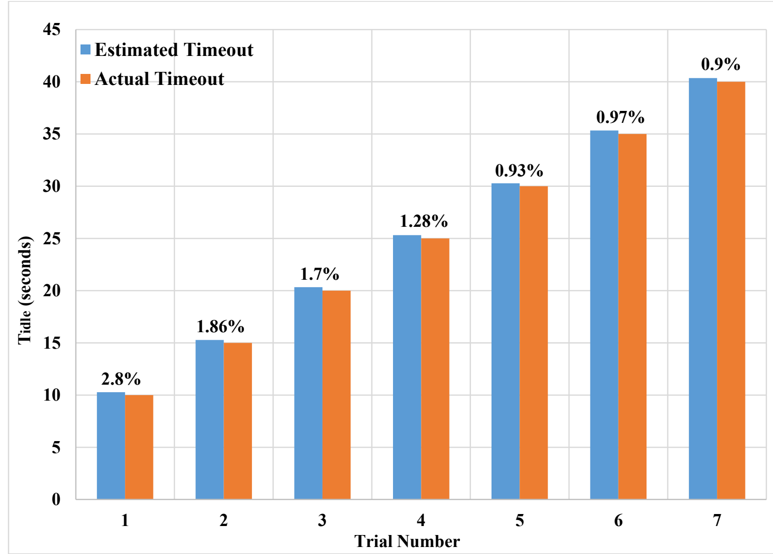

Figure 5.7: Plot - T_{idle} discovery using Algorithm 4.1 & 4.2

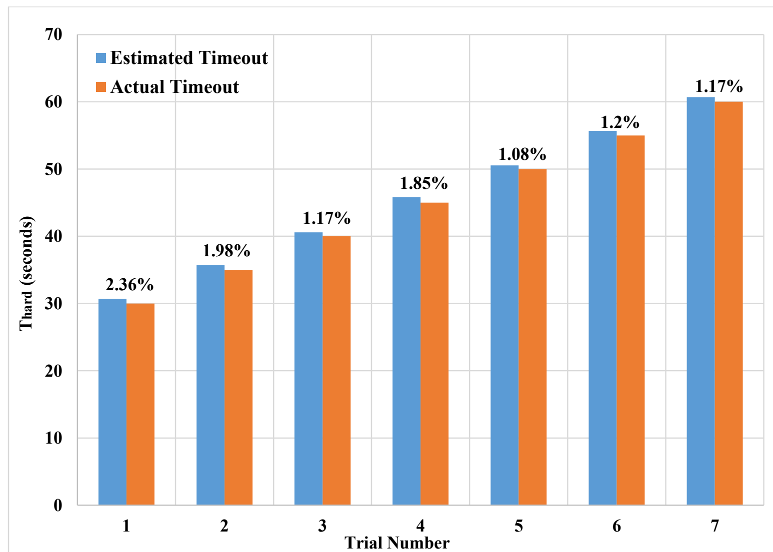

Figure 5.8: Plot - T_{hard} discovery using Algorithm 4.1 & 4.3

Given the normal ranges for the step size, idle and hard timeouts, it is thus more likely that hard timeout will discovered first by Algorithm 4.1

rather than the idle timeout. We have considered both combinations of these algorithms and have plotted the estimation error percentage in Figure 5.7 and 5.8 .When Algorithm1 is followed by the Algorithm2, the value of hard timeout was set as 60 seconds and the idle timeout values were changed between 10 to 40 seconds. For the case when Algorithm 4.1 is followed by the Algorithm 4.3, the value of idle timeout was 6 seconds and in this case only, T_{step} was set at 750 ms whereas in all remaining experiments T_{step} was set at 100 ms. For these set of experiments, the maximum average estimation error was 2.8 percent for the case when actual idle timeout is 10 seconds, hard timeout is 60 seconds with step size of 100ms, resulting in the combination Algorithm 4.1 followed by Algorithm 4.2.

5.4 Controller Reaction to Full Tables

Each of mitigating policies for table full event in SDN enabled network have their own pros and cons. In the case where controller is not taking any action only one part of the network will be compromised. Because controller is not taking any action on the table miss entry and that packet will not be forwarded to next switch while the other switches will be working fine. Whereas in flow reinstallation policy, controller's resources will be highly utilized although there will be no packet drops but it affects the performance of the whole network as the RTT of all messages will increase, due to extra messages involved in each new flow installation, especially if it involves multiple switches between the end hosts. Moreover, different timeout combination have their own impact on the network because of difference in the working of hard and soft timeouts. So, adversary should choose packet burst according to match fields and timeout values deployed in the targeted network. As the adversary has fingerprinted the timeout values, so the adversary knows better when to repeat the match field values in the attack burst. Same burst will have different impact under different combinations of timeouts. Even same burst will have different percentage of dropped packets and total completion time under the same timeout combination but different timeout values.

5.4.1 Packet Drop Policy

As discussed earlier, when controller is not taking any action on the table full message generated by the switch, all incoming packet will be dropped unless some of the flow entries are expired by timeout mechanism. Which means, packet drops will increase if flows has greater life, as shown in figure **??** and overall burst completion time will also increase because as in our

reference implementation, Scapy waits for timeout of a packet before generating the next one, this trend was shown in figure **??**. One instance of these experiments has been explained in detail, when hard timeout of controller was discovered as 60 seconds. Scapy packet generation rate in normal mode, when switch was able to accommodate all incoming flow entries, was around 75 flow entries per second. As switch table size was reduced to accommodate 1024 flow entries only, within 13.65 seconds of the experiment simulation time, flow table was full and table full messages were generated in response to the new incoming packets. Now, packet drops is observed until 60 seconds,

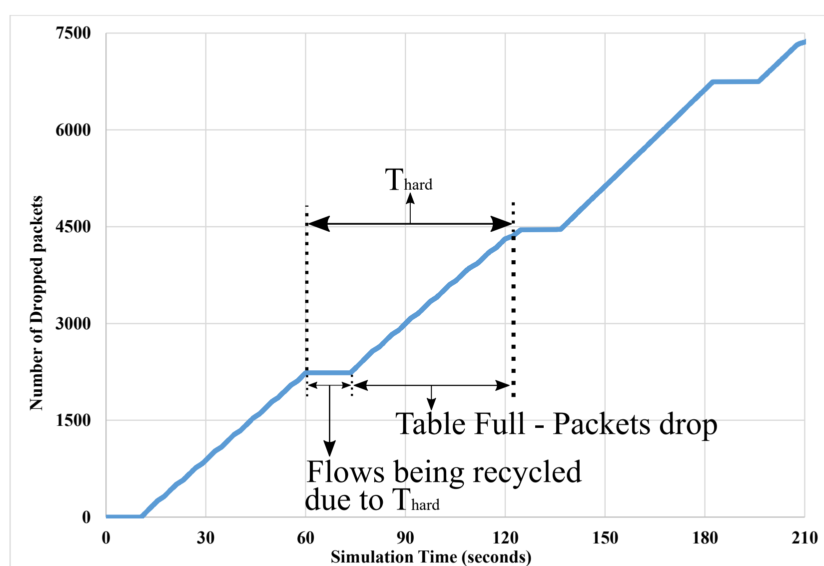

Figure 5.9: Plot - Packet drops at hard timeout = 60 seconds

after that switch started expunging old flow entries and new flow entries were entertained again for almost 14 seconds i.e., until the table was full again and at 74.49 seconds again packet drop started. First packet drop interval lasted for almost 46 seconds and 183 packets were dropped, which was expected to be around this number because RTO in Scapy was set at 250 ms. This repeated fashion of packet drop in relation to hard timeout of controller (Hard timeout = Drop interval + Time taken to full the table) i.e., Drop interval of 45 seconds, was observed for the whole experiment and almost 25% of the packets from the total burst of 10000 packets were dropped during this experiment, which took 866.689 seconds in total to complete (Figure 5.9). However, when RTO was reduced to 25 ms, around 70% of packets results in dropped because during the drop interval host is not waiting so long for the reply, but in this scenario overall burst completion time remains the same as of a normal flow due to reduced RTO.

5.4.2 Flow Replacement Policy

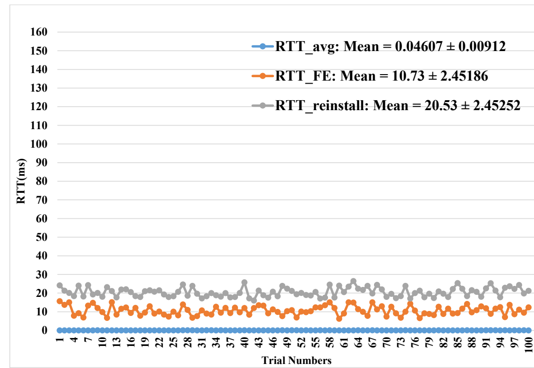

Figure 5.10: Plot - RTTs of in-rack communication

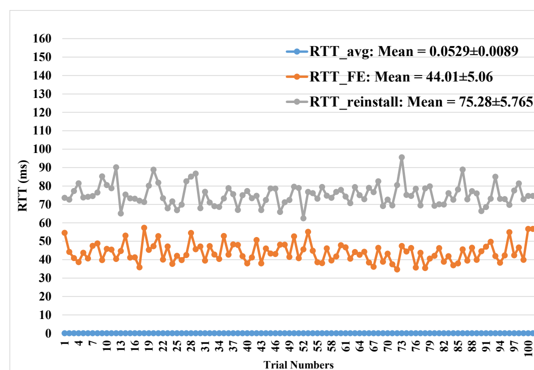

Figure 5.11: Plot - RTTs of in-pod communication

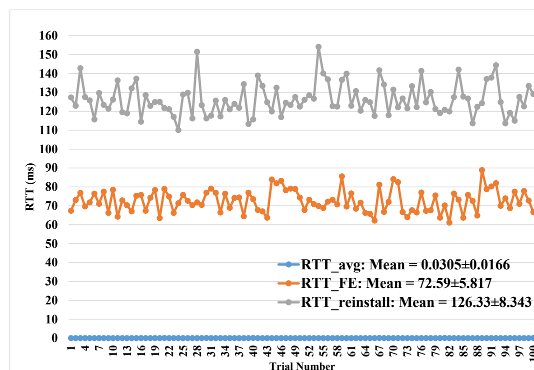

Figure 5.12: Plot - RTTs of inter-pod communication

For the evaluation of this module i.e., controller reaction to table full event, fat tree topology has been deployed in the Mininet, one switch for in-rack communication, three switches for in-pod communication and five switches for the inter-pod communication. And flow table at the switches in the network was reduced, to observe the flow replacement policy. If SDN controller is using the default action on table full event then flow table size is of no use. Because such networks will not return the $RTT_{reinstall}$ and adversary has to relay on RTT_{FE}.

Moreover, in these experiments, TCP SYN burst is generated in which two consecutive packets contains a new value of TCP port, first packet enforce the SDN controller to install a new flow entry due to table miss event and second packet is used to observe the RTT when packet passess through the data plane directly. The normal RTT (RTT_{avg}), flow installation RTT (RTT_{FE}) and flow replacement RTT ($RTT_{reinstall}$) is shown in figures 5.10, 5.11 and 5.12 for one, three and five switches respectively. Moreover, in these graphs RTT_{FE} and RTT_{avg} also reflects the match fields testing i.e., when a new flow entry is being installed for each forged packet (RTT_{FE}) and when copy of the same packet passes through the data plane (RTT_{avg}). These experiments have been carried out using hundred packets of each type (RTT_{avg}, RTT_{FE} and $RTT_{reinstall}$) in each category (in-rack, in-pod and inter-pod).

5.5 Topology Discovery

As discussed earlier, if an adversary fingerprints the information about the topology of the targeted network, he can launched a more mature attack. A fat tree topology has been implemented in the Mininet and it has been observed that sending forged packets to inter-pod targeted host results in seven times increased RTT_{FE} as compared to in-rack targeted host (Figure 5.13).

Best case scenario for an attacker will be the over flowed tables at the switches in the targeted network and sending forged packets to inter-pod targeted host, then $RTT_{reinstalled}$ will be almost twelve times of RTT_{FE} (Figure 5.14). Conclusively, adversary should target the host, which involves maximum number of hops. And generating packets with unique values of match fields, further increases the probability of a successful attack on SDN enabled network. The behavior discussed above is observed when tables are not full i.e., RTT_{avg} and RTT_{FE} and $RTT_{reinstall}$ when tables were full, provided that controller has implemented the flow replacement policy.

However, situation will be different if controller has implemented the

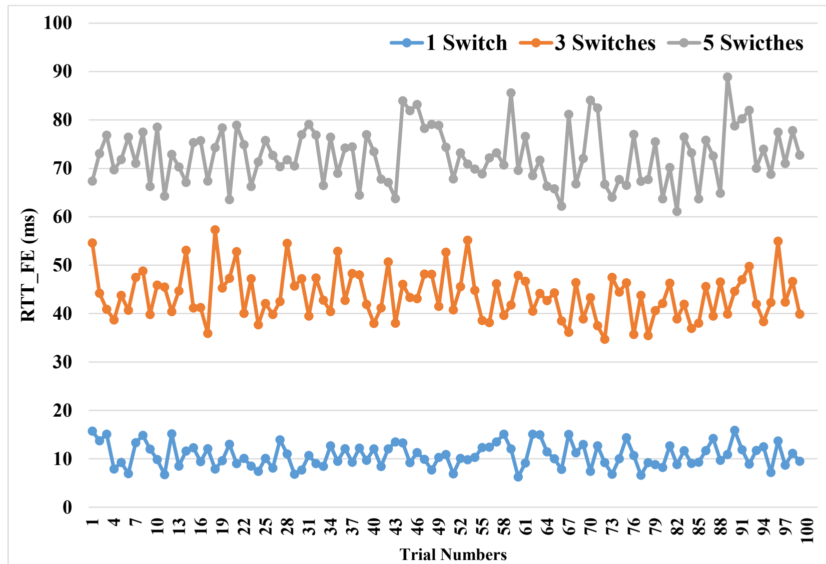

Figure 5.13: Plot - Variation in RTT_{FE}

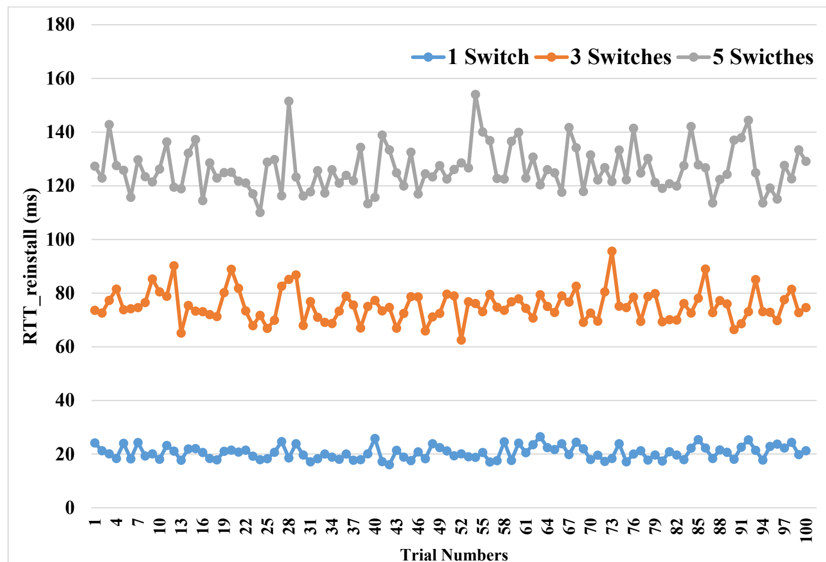

Figure 5.14: Plot - Variation in $RTT_{reinstall}$

packet drop policy in response to table full event. In that case, hosts can be sorted on the base of difference in RTT_{FE} only. Moreover, if table is flow entries table is full at any switch between the hosts, more experiments needs to be performed in order to achieve the design goal of this module.

Moreover, the latency at data plane (RTT_{avg}) is ambiguous to be used as sorting crietria of hosts into in-rack, in-pod and inter-pod groups. Although

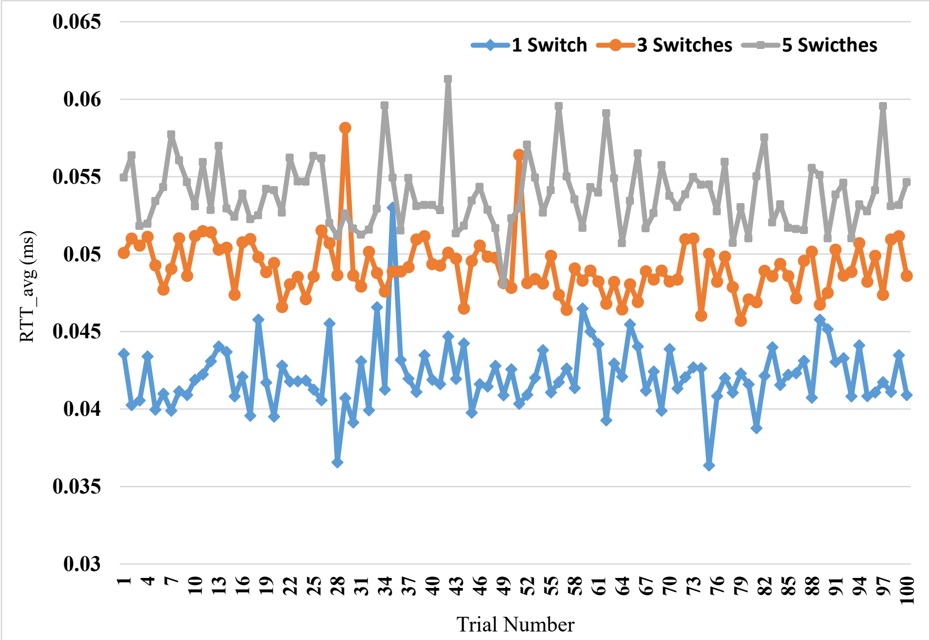

Figure 5.15: Plot - Variation in RTT_{avg}

there is difference in RTTs when a packet is transmitted through one switch, three switches and five switches respectively. But almost 8% randomness has been observed which do not follow the trend and might yield wrong answers, as shown in figure 5.15. A long experiment i.e., hundreds of packets should be transmitted to different hosts and using probabilistic distribution, hosts can be categorized.

Chapter 6

Conclusion & Future Work

Lastly, this chapter concludes the presented research work. In which, section 6.1 covers the future directions and some other research challenges that need to be addressed and section 6.2 presents the conclusion of this research work.

6.1 Future Work

From security perspective of SDN, there are many things which need to be taken care of. More enhanced policies are required, for example implementation of dynamic parameters rather than fixed parameters. Means, a network should change timeout values with respect to the traffic load and vacant space in the flow entries table. Similarly, if the flow entries table is full, controller should reduce the number of implemented match fields. So that more and more incoming packets match with a single flow entry.

From fingerprinting perspective, there are some research challenges that need to be addressed. For instance, predicting the victim selection of flow entries in flow replacement policy, can be interesting topic i.e., which flow entries are being expunged by the controller (oldest or on the bases of a particular match field or protocol). Furthermore, machine learning is an emerging domain, implementation of machine learning for security analyzing of SDN enabled network can be a good future direction.

This research work is assuming that a SDN enabled network is using a single SDN controller. However, multiple SDN controller can be deployed as per requirement and design of the network. For example, a separate controller/agent can be synchronized with the main SDN controller for traffic engineering or security applications (firewall).

6.2 Conclusion

In this research work, various techniques have been proposed for fingerprinting the SDN enabled networks. This work revolves around SDN fingerprint techniques like parameters (timeout values, flow rule match fields), controller's reaction on event when switch's flow table is full and lastly estimating the number of hops involved in host-to-host communication, the topology discovery in the targeted network. It is worth mentioning that all proposed techniques merely use end host(s) only, thereby relaxing assumption that network devices (switches, routers) can be in control of an attacker. With the help of different simulation experiments, it has been showed that how parameters like timeouts can be measured and match fields can be guessed and eventually exploited for DDoS alike attacks. Moreover, we also revealed that, if an adversary using proposed techniques, knows SDN parameters (timeouts, match fields, topology) in advance, network is more prone to attacks; thereby creating alarming situation for the network administrators and obviously, fortunate situation from an attacker's perspective. Furthermore, possible future direction has been given for both directions i.e., the need of enhanced SDN policies, to improve the security and performance of the SDN enabled networks and for more enhanced fingerprinting using machine learning based techniques.

Bibliography

[1] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[2] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[3] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "Sdn security: A survey," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For.* IEEE, 2013, pp. 1–7.

[4] A. Shaghaghi, M. A. Kaafar, R. Buyya, and S. Jha, "Software-defined network (sdn) data plane security: Issues, solutions and future directions," *arXiv preprint arXiv:1804.00262*, 2018.

[5] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti, "A survey on the security of stateful sdn data planes," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1701–1725, 2017.

[6] Q. Yan, F. R. Yu, Q. Gong, and J. Li, "Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 602–622, 2016.

[7] T.-H. Nguyen and M. Yoo, "Analysis of link discovery service attacks in sdn controller," in *Information Networking (ICOIN), 2017 International Conference on.* IEEE, 2017, pp. 259–261.

[8] S. Khan, A. Gani, A. W. A. Wahab, M. Guizani, and M. K. Khan, "Topology discovery in software defined networks: Threats, taxon-

omy, and state-of-the-art," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 303–324, 2017.

[9] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[10] M.-K. Shin, K.-H. Nam, and H.-J. Kim, "Software-defined networking (sdn): A reference architecture and open apis," in *ICT Convergence (ICTC), 2012 International Conference on*. IEEE, 2012, pp. 360–361.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[12] V. Shukla, *Introduction to Software Defined Networking: Openflow & VxLAN*. CreateSpace Independent Publishing Platform, 2013.

[13] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 123–137.

[14] G. Zhao, L. Huang, Z. Yu, H. Xu, and P. Wang, "On the effect of flow table size and controller capacity on sdn network throughput," in *Communications (ICC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–6.

[15] Z. J. Zeitlin, "Fingerprinting software defined networks and controllers," AIR FORCE INSTITUTE OF TECHNOLOGY WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF ENGINEERING AND MANAGEMENT, Tech. Rep., 2015.

[16] A. Azzouni, O. Braham, T. M. T. Nguyen, G. Pujolle, and R. Boutaba, "Fingerprinting openflow controllers: The first step to attack an sdn control plane," in *Global Communications Conference (GLOBECOM), 2016 IEEE*. IEEE, 2016, pp. 1–6.

[17] R. Kandoi and M. Antikainen, "Denial-of-service attacks in openflow sdn networks," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE, 2015, pp. 1322–1326.

[18] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 165–166.

[19] M. Zhang, J. Hou, Z. Zhang, W. Shi, B. Qin, and B. Liang, "Fine-grained fingerprinting threats to software-defined networks," in *Trustcom/BigDataSE/ICESS, 2017 IEEE*. IEEE, 2017, pp. 128–135.

[20] T. Alharbi, S. Layeghy, and M. Portmann, "Experimental evaluation of the impact of dos attacks in sdn," in *Telecommunication Networks and Applications Conference (ITNAC), 2017 27th International*. IEEE, 2017, pp. 1–6.

[21] Y. Zhou, K. Chen, J. Zhang, J. Leng, and Y. Tang, "Exploiting the vulnerability of flow table overflow in software-defined network: Attack model, evaluation, and defense," *Security and Communication Networks*, vol. 2018, 2018.

[22] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "Flooddefender: protecting data and control plane resources under sdn-aimed dos attacks," in *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*. IEEE, 2017, pp. 1–9.

[23] "Scapy traffic generator," http://www.secdev.org/projects/scapy/, accessed: 2018-04-14.

[24] P. Rygielski, M. Seliuchenko, S. Kounev, and M. Klymash, "Performance analysis of sdn switches with hardware and software flow tables," in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2016)*, 2016.

[25] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[26] "Mininet network emulator," http://www.mininet.org, accessed: 2018-04-14.

[27] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[28] "Ryu sdn framework," https://osrg.github.io/ryu/, accessed: 2018-04-14.