ARCHITECTURAL IMPROVEMENTS FOR SECURE SDN TOPOLOGY DISCOVERY

Ph.D. Thesis

AJAY NEHRA

ID No. 2014RCP9553

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY JAIPUR

July 2019

Architectural Improvements for Secure SDN Topology Discovery

Submitted in

fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Ajay Nehra

ID: 2014RCP9553

Under the Supervision of

Dr. Meenakshi Tripathi

Associate Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY JAIPUR

July 2019

Declaration

I, **Ajay Nehra**, declare that this thesis titled, **'Architectural Improvements for Secure SDN Topology Discovery'** and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a degree of **Doctor of Philosophy** at **Malaviya National Institute of Technology (MNIT) Jaipur**.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at MNIT Jaipur or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this Dissertation is entirely my own work.
- I have acknowledged all main sources of help.

Date:

AJAY NEHRA
(2014RCP9553)

Malaviya National Institute of Technology Jaipur
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the thesis entitled "**Architectural Improvements for Secure SDN Topology Discovery**" is being submitted by **Mr. Ajay Nehra (2014RCP9553)** in partial fulfillment of the requirements for the award of the degree of **Doctor of Philosophy** in the Department of Computer Science & Engineering, Malaviya National Institute of Technology Jaipur, Rajasthan, India. It is a record of the original research work carried out by him under my supervision. The thesis has reached the standards fulfilling the requirements of the regulations relating to the degree. The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree or diploma.

Dr. Meenakshi Tripathi
Associate Professor, Supervisor
Department of Computer Science & Engineering
Malaviya National Institute of Technology Jaipur, India

Date:
Place: Jaipur
.

Abstract

Within the domain of SDN, in this thesis, we examine the security issues in topology discovery. It includes the discovery of the host, the network nodes and links between them. Most of the SDN controllers use a topology discovery mechanism which lacks main security components such as authenticity and integrity. This leads to the possibility of various attacks like poison, replay and flooding. In this thesis, we analyze the vulnerabilities in host and link discovery in SDN. We examine in detail, how an attacker can perform poison, replay or flooding attacks during link discovery. This detailed analysis provides the basis for designing our proposed prevention mechanism, TILAK.

In this thesis, we also propose an improved SDN Link Discovery Protocol (SLDP) which combines the security measures along with efficiency. The proposed method is lightweight considering SDN characteristics.

Topology discovery cannot be secured without securing data link layer based host discovery which is done by ARP protocol. Even if link discovery is secured, Man-in-the-Middle (MitM) attack is yet possible at the data link layer. In this thesis, we discuss the problem of ARP-based attacks in the context of SDN. We propose FICUR to detect and mitigate ARP-based attacks. FICUR leverages the programmability and centralized control of SDN. All proposed solutions are validated through extensive emulations. Experiments have shown that proposed methods are better than other existing methods in terms of computation overhead, packet construction time and attack detection time.

Dedication

This small piece of work is dedicated to this great nation i.e. India, भारत, 🇮🇳.

Acknowledgements

Ph.D. is a long journey that we have to walk alone. Without constant and unconditional support from my family, it is hard to survive this journey. My mother and father always make me strong enough to run on tough terrain. My brother helps me to improve in life with my own logic. My sister is a silent observer and her critical suggestions help me in improving a lot. I would like to thank my wife, whose emotional support is really appreciable. I really admire naughty 'लड्डू', i.e. 'Laddu', his consistent and energetic naughtiness refresh me a lot. I would like to thank all of them once again for their necessary and kind support.

I would like to pay my sincere gratitude towards my supervisor Dr. Meenakshi Tripathi. Without her constant demands for progress, it is not possible to complete it. Her kind gesture and understanding nature are highly appreciable. Her supporting behavior always helped me to keep moving. Her experiences always helped to improve research articles and thesis. Thank you.☺

The role of Prof. Vijay Laxmi and Prof. Manoj Singh Gaur in my Ph.D. journey is like a pole star enlightening the way of a stray passenger. Their constructive criticisms improved the thesis a lot. Without their crucial support, it is not possible to nurture ideas more clearly. Their critiques also helped me to expand ideas in various dimensions.

I would also like to thank Dr. Girdhari Singh, Dr. Santosh Kumar Vipparthi and Dr. C. Periasamy. As DREC members, their suggestions help me to precise the research direction. Their confidence in my work encourages me a lot. I truly appreciate unconditional support from Dr. Ramesh Babu Battula throughout the journey. Non-teaching staff was always kind to me and always ready to help within their capacity.

Ph.D. leads a way to prosperous life. It taught to be a critique. I would like to thank Mr. Gajendra Singh Shekhawat for being a discussion partner. His practical views always guides my ideal view toward many problems. Without his company, Ph.D. frustration cannot be overcome.

A good thesis is a completed thesis. Without Ms. Varsha Sharma, Mr. Avinash Avthu, Mr. Ankur Gupta, Ms. Ritu Sharma and Mr. Kurra Chaithanya, it is not possible to make thesis in the present form. Their unconditional and constant support as first reviewers is highly admired. I would also like to express gratitude to Dr. Chhagan Doot to share his experience to polish the research articles and thesis.

I would also appreciate Mr. Ashish Sharma, Mr. Pranjal Ranjan, Mr. Anugrah Jain, Mr. Jay Dave, Mr. Anurag Sharma and Mr. Ankit Pulkit for helping me in traveling this Ph.D. journey more smoothly. I consistently argue them on various topic, and they handle it gracefully. This discussion improves me a lot.

My two close friends from the far, i.e., Mr. Abhinav Srivastav and Dr. Gautam Chandra refreshed me a lot with long discussions and gave me always a unique dimension of thoughts. Such thoughts always help to build the character.

Thank, you all of you once again. It's truly a long and adventurous journey. Everyone plays a crucial role to complete the journey. I really appreciate your valuable time and efforts.

Acronyms

- **ARP** Address Resolution Protocol
- **BDDP** Broadcast Domain Discovery Protocol
- **CDP** Cisco Discovery Protocol
- **DHCP** Dynamic Host Configuration Protocol
- **DDoS** Distributed Denial of Service
- **DoS** Denial of Service
- **EIGRP** Enhanced Interior Gateway Routing Protocol
- **HMAC** Hash-based Message Authentication Code
- **IP** Internet Protocol
- **LAN** Local Area Network
- **LLDP** Link Layer Discovery Protocol
- **MAC** Media Access Control
- **MIB** Management Information Database
- **MitM** Man-in-the-Middle
- **NAT** Network address translation
- **OFDP** OpenFlow Discovery Protocol
- **OFPT** OpenFlow Packet Type
- **SDN** Software Defined Networking
- **SLDP** SDN Link Discovery Protocol
- **SNMP** Simple Network Management Protocol
- **TCAM** Ternary Content-Addressable Memory)
- **TCP** Transmission Control Protocol
- **TLS** Transport Layer Security
- **TLV** Time-Length-Value

This page is intentionally left blank.

Contents

List of Figures

List of Tables

List of Algorithms

This page is intentionally left blank.

Chapter 1

Introduction

Over the past decades, the world is experiencing a significant increase in the number of internet services such as e-commerce, Internet of Things (IoT) based services, social sites, chat applications search engines, etc. All these are hosted in the cloud/data centers, and end users are accessing them using their personal computer, phones or tablet through Ethernet, Wi-Fi or cellular network. New services create new markets. As Bain[1] also predicts that the IoT market is doubled in 2021 since 2017.

The assurance of fast, reliable and secure services in these complex networks, has raised several challenges in network management. Now there is a need to configure the network on the fly, which can respond to the updates, faults or loads. Traditional network is unable to satisfy these new requirements. SDN is a new paradigm which has come up as a solution to this. Statista[2] also predicts that the Software Defined Networking (SDN) market worldwide will be 30.8 billion USD in 2022.

One of the key concepts of SDN is the separation of control plane and data plane. A logical centralized controller introduces the logical concept of network programmability, which has simplified the possibility of innovation, network management, and configuration. With promises of flexibility, SDN also provides solutions to the traditional network security threats i.e. SYN Flooding, Address Resolution Protocol (ARP) Poison, rogue Dynamic Host Configuration Protocol (DHCP) server, etc. At the same time, SDN architecture also imposes a new class of vulnerabilities in network i.e. poisonous topology discovery, Controller Flooding, flow entry flooding and, etc. Among other threats, this thesis is restricted to vulnerabilities in the topology discovery mechanisms only. The topology discovery is done by the controller to get the information about the latest state of the network. It consists of the discovery of various components such as forwarding elements, links, and hosts.

1.1 Motivation

Software Defined Networking (SDN) introduces a sea of opportunities in the research domain. Its newly introduced architecture gives various challenges and new future directions to improve communication. The research areas in SDN include controller placement, programmable data plane, traffic classification, security, abstraction, verification[3][4]. Security is an essential factor to deliver basic functionalities efficiently. The security domain in SDN covers various aspects introduced by either its new architecture or assurance of new solution to traditional network threats. Few security related problems in SDN are Denial of Service (DoS) attacks at both the planes, topology poisoning, malicious app detection, and malicious link detection[5].

The controller gives ample flexibility to counter various security vulnerabilities. However, this flexibility can only obtain with the help of available information. This information makes the controller to take a decision and perform the required task. Information is available with data plane using OpenFlow protocol. Information about existence and arrangement of data plane is also important. For the security of any threat, detection, mitigation and prevention are different dimensions. Any attack detection strategy must require source data plane information. If attack probes are found on a particular switch, detection strategy for that switch will activate. For mitigation strategies, port-link information is needed. Few links must block/slowdown for a unique kind of traffic, for this some ports need to reconfigure. For prevention methods, entire topology must be known in advance. The controller must know, the possible paths by which attack probe may occur, all details for in-between forwarding elements, links, and ports. In a nutshell without accurate topology information, it seems hard to achieve optimal results. Let's start with two security problems and their unique solutions to examine whether topology information can help in designing an attractive solution? Problems are related to flow entry flooding and SYN flooding.

In SDN, every time a switch receives a packet, it checks a flow table to act accordingly. Flow table consists of flow entries, and flow entries are stored in Ternary Content-Addressable Memory (TCAM) based memory which is expensive and fast. Sometimes programmer uses the property from received packet rather than hard coding to generate flow entries. But if an attacker successfully passes packets to the controller with random attributes, and the controller generates a new flow entry for every such random attributed packet. It may be used for flooding the flow table of a switch. OpenFlow provides a mechanism for storage and retrieval of statistical counters associated with flows, ports, tables. The security vulnerability discussed here can be detected if we extract some of those counters. Flows are stored in one among many tables. If we can extract the number of flows and the number of packets that matched in the table, then we can detect an attack[6]. Packet Flow Ratio(`PFRatio`) is used to detect the aforementioned attack. `PFRatio` is defined as Number of Packet over the Number of Flows. This work is published in [C3].

Transmission Control Protocol (TCP) SYN Flooding is the most popular among all DoS attacks that exploit the TCP vulnerability at the server i.e. any type of service that uses TCP as the underlying transport layer protocol. The server keeps some half-open connections until the corresponding final ACK arrives from the interacting clients. Keeping such connections require a certain amount of memory reserved at the server. Attacker(s) can open a large number of such incomplete connections, thus depletes the SYN-Queue easily which will ultimately deny or delay the legitimate connections request. A proposal called *SAFETY*: an early detection and mitigation technique for TCP SYN Flooding attacks in SDN networks[7], is implemented. The Solution is based on entropy, which is used to measure the randomness or uncertainty associated with a random variable. SAFETY calculates the entropy of packets in a fixed time window to determine any abnormal behavior. In particular, entropy is calculated on few fields namely destination IP address, destination port, and TCP flags. In the case of non-attack scenarios, there are multiple hosts which communicates with each other and their packets identified with a pair consists of destination IP and port, are randomly distributed in the network. However, in presence of adversaries, a particular service irrespective of its location will generate packets with same destination IP and port address. To perform an attack with the highest possible strength, packets with the same destination IP and destination port have to be generated by a single malicious host or a group of malicious hosts. The attacker may also perform a low-intensity attack from the network and wait for other attack probes from outside the network. In such scenarios, the randomness is limited due to the nature of the attack, hence entropy decreases drastically. This work is published in [J3].

In both solutions i.e. PFRatio and SAFETY, some part of topology is used such as the existence of switch. Both are detection approaches. For mitigation in SAFETY, some ports information must know to the controller to block a specific type of traffic. Switch to Switch link will also play a significant role in preventive approaches. For example, if certain traffic seems to be malicious but needs more signature to conclude as malicious, a separate path can be provided so that legitimate traffic will not suffer. It seems our hypothesis that topology is crucial for optimal results, is validated.

1.2 Objectives

The goal of this research is to explore vulnerabilities of topology discovery in SDN and suggest the measures to strengthen the security. To meet this objective we have.

- Explored the vulnerability of topology discovery in SDN by performing state art of analysis.

- Developed a preventive solution for LLDP Poison, Flooding and Replay attacks.

- Proposed a lightweight and efficient protocol variant for the link discovery process.

- Developed a solution for ARP based threats.

1.3 Contributions

This section provides an overview of the contributions along with the corresponding publications. Most of the work has been published in reputed journals and conference proceedings.

- First of all, we accomplish an extensive literature survey on security issues in SDN topology discovery. This has been discussed in Chapter 2. Based on the review, we observe that most of the attacks in the topology discovery are launched by exploiting vulnerabilities in discovery protocols. We also notice that most of the proposed security solutions are based on cryptography or static binding. Such topology discovery methods which are costly in terms of computation. Our focus was on developing lightweight non-cryptographic solutions. Most of the work of this chapter is published in [J2, C1, C2].

- Secondly, we perform an in-depth investigation of threats during the link discovery mechanism. We investigate both theoretically and then through extensive implementation on various controllers such as POX[8], Ryu[9], OpenDayLight[10], Floodlight[11], Beacon[12], ONOS[13], and HPE-VAN[14], the impact of LLDP poison, LLDP Replay and LLDP Flooding attacks. We find that there are two major reasons for this kind of attacks in all the abovementioned controllers: *first*, lack of verification of source authentication and *second*, lack of the integrity of LLDP packets. By analyzing the characteristics of the attack, we propose TILAK, a prevention methodology for LLDP packet based attacks. TILAK generates random MAC for LLDP packet and uses this randomness to create a flow entry for LLDP packets[15]. This work has been published in [J2] and reproduced in Chapter 3.

- Thirdly, we also propose an SDN Link Discovery Protocol (SDLP) for efficient link discovery in SDN[16]. SLDP creates and maintains the global network topology at SDN controllers by using a small number of SLDP packets. Depending upon the scenario, it either prevents or detects and mitigates the LLDP based attacks. We prove both theoretically and with emulation that SLDP is lightweight, secure and efficient link discovery protocol for SDN. This work has been published in [J1] and discussed in Chapter 4.

- We notice that host discovery mechanism is also vulnerable to ARP spoofing attacks. We also observe that most of the existing solutions are based on either pre-stored MAC/IP binding or cryptographic mechanism. These methods are either static or require high computation, which makes them unsuitable for real-time implementations. Finally, we propose FICUR, a novel method for verification and detection of ARP based threats in SDN[17]. FICUR leverages the SDN programmability and analyzes the traffic patterns to detect the attacks at the centralized controller. The validation of FICUR has been done on both the emulated environment using Mininet and real-time environment using HP switch. Chapter 5 is devoted to this work and has been published in [C2].

1.4 Organization of the Thesis

The thesis is structured as follows. Topology discovery plays an important role in the operation of SDN. Chapter 2 provides a detailed description of topology discovery mechanisms used by different SDN controllers. This chapter also identifies the vulnerabilities of topology discovery mechanism and describes various attacks that exploit these vulnerabilities in SDN. Chapter 3 describes our proposed prevention for LLDP Poison, LLDP Flooding, and LLDP Replay attacks. Theoretical analysis and emulation results are also presented to increase confidence. Chapter 4 describes proposed link discovery protocol for SDN. SLDP packet and SLDP event sequence are discussed in detail. The chapter also provides details of emulation experiments conducted and result achieved. Chapter 5 describes our proposed, SDN solution for ARP threats. Emulation results for the proposed solution are also presented. Chapter 6 concludes our thesis along with suggestions for future work. The appendix lists the publication and the emulation details for our work with different controllers.

This page is intentionally left blank.

Chapter 2

State of the Art

In this chapter, literature is analyzed around our research objectives. Among various topology components, two components are focused mainly considering existing discovery process, possible threats, attackers view, attack's manifest, available orthogonal research, and observed gaps. This analysis is helpful to the next chapters where solutions will be provided for threats. A brief introduction to Software Defined Networking (SDN) and the comparison with traditional networking is also provided in the chapter to build abstract understanding regarding SDN.

2.1 Software Defined Networks: A new era for networks

Since unfolding the initial communication network in 1876 in the telephone, networks always served the humanity in best possible extent. Communication networks are still growing in tune with ever growing human needs. To serve the requirements, various modifications have been accepted in communication technology. Design of TCP/IP protocol suite in the 1970s by Vint Cerf and Bob Kahn for DARPA project is another milestone for current data networks[18]. Network research groups always try to provide a new solution over existing infrastructure. Applications are well known now so we can think of new architecture for the network paradigm. One such disruptive technology which will change the scenario from master complexity to extract simplicity is Software Defined Networking (SDN). Traditional network is unable to afford next-generation requirements. This section explains SDN features while comparing with traditional networking. The problems in traditional networking and how SDN resolves these are discussed as follows.

Heterogeneity in the traditional network is an operational essential. Traditional network is heterogeneous due to different devices, i.e., switches, routers, gateways, firewalls, Network Address Translators , etc. Heterogeneity in networks is also due to devices from different vendors, i.e., Cisco, hp, dell, Juniper, etc[19]. Different vendors use different proprietary protocols, i.e., Enhanced Interior Gateway Routing Protocol (EIGRP), Cisco Discovery Protocol (CDP), etc which also contribute to heterogeneity. Different devices are capable to perform computation on different layers of TCP/IP stack[20]. Generally, more the layers available for computation, more the cost of device. Different devices have different instruction sets for managing the devices[21]. This situation is even worse in case the devices available are from different vendors. SDN proposes a lesser heterogeneous environment with OpenFlow enabled forwarding elements. These forwarding elements have a simpler configuration than traditional router or switch. Simpler the design, cost-effective the solution. Therefore, each vendor is supposed to build forwarding devices with OpenFlow specifications. In this way, same devices are available in networking environment even if those are coming from different vendors. OpenFlow restricted protocols are introduced like OpenFlow Discovery Protocol (OFDP)[22]. To provide different behaviors from forwarding elements like routing or switching, a logically centralized controller is used to control the forwarding elements. The SDN controllers are written and can modify in generic purpose programming languages like java or python. Openness in SDN also improves the rate of innovation in networking[23].

Flexibility offers effectiveness in operation and managing the networking devices. In each networking device, two plane plays distinguish and significant role. Control plane computes logic while the Data plane forwards packets as per computation result. For example, the control plane performs all computation for shortest path calculation. Result for this computation is reflected in the form of a routing table[24]. The data plane forwards each incoming packet as per the routing table. In the traditional network, both planes reside on the same device. Each device has to manage separately and having a local view of available information i.e. neighbor devices. Managing each device in network separately is tedious and human resource intensive[25]. This process will be more complicated if the devices are from different vendors. Hence different instruction sets are needed to manage the network. Making decisions on a global view is always optimal than a decision on a local view. SDN gives the opportunity to separate the control and data plane. The control plane may be deployed with any other service for the infrastructure i.e. web service. And forwarding element is designed to hold data plane and an OpenFlow agent. Separation gives control plane a bigger visibility of topology hence helps to make efficient decisions. Because logically centralized controller(s) controls the entire network, a consistent policy can be installed in forwarding elements throughout the network[26]. It is also easy to manage a logically centralized controller(s) than a variety of devices.

In traditional networking devices, i.e. switch's or router's forwarding decision is made based on the destination address. A switch forwards packets based on the destination MAC address, while a router forwards the packet based on the destination's IP address. Traditional switches or router cannot be configured to make forwarding decision on the basis of other fields from the transport layer, network layer or data link layer. But SDN offers a unique opportunity to manage forwarding devices to make the decision from the various fields of different networking layers[19]. In SDN, OpenFlow is a widely accepted protocol for data and control plane communication. OpenFlow specification[27] provides forty-five match attributes to make forwarding decisions.

Traditionally a network administrator can configure logical isolation in Local Area Network (LAN) as either port based or MAC address based. Virtual LANs (VLAN) are used for such isolation for security, scalability and network management purposes. SDN comes with a new plan to logically isolate the network segments as per forty-five parameters[27]. These parameters are restricted with OpenFlow specifications[27–39].

Provisioning in the data center requires frequent configuration and reconfiguration of networking devices[24]. Provisioning is mandatory in case of faults and adaption to load changes. Traditional network devices support most of the manual provisioning, which is clearly an error-prone and time restricted approach. The manual process of uniform policy enforcement in a short period is also human resource intensive. If devices are from different vendors, the process will be more cumbersome. SDN gives a good spectrum of dynamic provisioning or automation in network management. All general purpose conditional rational can be built for provisioning. Another benefit of SDN is that all developing policies can be tested over production network without compromising the performance hence increase up-time for the production network. In traditional networking, devices need to program as per requirement but only can be programmed with the available instruction set[19]. Each device from a different vendor and different capability have different and limited instruction set. SDN offers to configure network rather than devices with any generic purpose programming languages.

Openness Traditional networking technology always suffers from slow innovation rate as compared to other fields of information technology. This is because of proprietary hardware govern with the dedicated instruction set. Traditionally custom Application-Specific Integrated Circuits (ASIC) are used for switches or routers. Fixed network interfaces are used to connect control and data plane. Different devices have different capabilities and must configure with a different instruction set. Anyone who buys them can configure them only. If she has to propose a new protocol variant which is suitable for her organization, has to go through enough large development cycle. SDN promises to be as open as possible[40]. General purpose CPUs are used for forwarding elements; and control plane is mostly placed along with other services i.e. web server. Like OpenFlow, OpenStack open standards are used to provide the communication in and management of the network. Vendor-neutral Application Program Interfaces (API) are used in the control plane to instruct the data plane[20]. Such openness also increases rapid development because of a possible reduction in expenses or generation of capital as entrepreneurship.

Cost plays a significant role in the traditional network. All devices are some proprietary hardware-software combinations. In any network, various devices with various capabilities are used. Higher the capability of devices higher the cost of devices. Different devices have different ASICs to boost the performance. But this approach does not seem very scalable. Only vendors are involved in developing new hardware and software[18]. The third party can not even try to give any software solutions for given hardware, which is definitely a hindrance in the innovation rate[41]. Traditional devices come with an instruction set to configure them to work accordingly. This configuration may followed by unlimited times reconfiguration, requires expensive human resources. In the era of Virtual Machines (VM) migrations in data centers, this configuration facility does not seems adequate. SDN comes with a trivial but yet effective solution which essentially consist of two parts. The control plane like any other server can be placed along with any other service. To write controller applications, generic programming languages are used so that anyone can write a new solution. The data plane takes instruction from control plane, is simpler as table look-up devices[42]. This approach makes data plane device simpler in logic hence cheaper in price. SDN networks are programmable which means network provisioning which might consist of a series of configuration and re-configuration, can be specified as network policy inside the controller applications. This approach will significantly reduces the human resources and helps inrunning the communication at a low operational cost.

Introduction of SDN improves the ***efficiency*** in several aspects. In the traditional network, each device has a local view because of a local control plane[42]. Local view always restrict the plane to make an optimal decision. Each device comes with a few configurable parameters. Sometimes configuration is needed beyond the provided configuration parameters. In the traditional network, this request takes a lot of time to serve from the vendor. If any device fails in the network, it should be removed/replaced from the topology. Failing devices may also result in a failed network. If devices fail, provisioning must be required. In the traditional network, provisioning is almost static[22]. Sometimes few redundant links are present in topology which need either to configure manually or be blocked with various data link layer protocol like Spanning Tree Protocol. Different devices have hardware of different complexities. Higher complexity is connected with higher cost. SDN can compute a larger view with the help of OFDP protocol. Controllers are programmable so that network administrator can specify almost any policy in the form of network applications. In SDN, same devices behave differently with different controller applications. The forwarding elements have to take instruction from the controller to perform the expected behavior. The hardware of forwarding elements only require fast table look-up facility to do any task; this needed hardware is much simpler than traditional switch or routers. Controller applications can also identify the failed device and logically removed it from the topology. In SDN topology, if redundant links are present, the controller has enough capability to manage/use these redundant links[43].

Table 2.1 summarizes all the above discussed properties. Each property has different aspects and effects which are shown with different colors and different shades. Different colors denote traditional networking and SDN while different shades represent a new aspect or effect in any of the networking. Red and Blue colors are used for Traditional network and SDN respectively.

Evolution of SDN also gives the opportunity to research in various aspects to make the network more robust, flexible, secure, performance efficient, etc. Its newly introduced architecture gives several challenges as well as new future directions to improve communication. Scalability due to a single controller with multiple forwarding elements must persist[44]. For a given/unknown topology, placement of the controller is itself a big problem[45]. A controller fails or is compromised then what next[46]? There must be consistency for multiple controllers in a given network segment[47]. Other research areas in SDN includes programmable data plane, traffic classification, security, abstraction, verification,

Property	Different Aspects	Effects
Efficiency	Local view	restricted opinions
	Limited configuration	new device for new capability
	Device dependency	Device fails network fails
	Less scalable	manual redundant path for new devices
	Complex hardware	higher capability, more specific integration
	Global view	better decisions
	Programmable	same device with new program
	Automation	failed device will logically be removed
	Scalable	controller can always find redundant paths
	Simplex solution	simple hardware for table look-up
Flexibility	United control & data plane	devices separately managed, local view
	Destination based forwarding	restricted forwarding decision
	Port and MAC based isolation	limited scope for isolation
	Static provisioning	error-prone and time restricted
	Separation(CP and DP)	bigger visibility, better manageability
	Many match attributes	large decision spectrum
	45 parameter for isolation	flexible isolation
	Dynamic provisioning	easy network management, increase up-time
Heterogeneity	Devices(router, switch,..)	higher cost for high capability
	Vendors(hp,cisco,..)	different instruction set to manage
	Protocol(EIGRP,CDP,..)	incompatible with other vendor's device
	Forwarding Element	cost effective solution for each capability
	Similar behaviour	simple management with each vendor
	OFDP,..	rate of innovation with openness
Cost	Proprietary hardware	higher capability, higher the price
	Dedicated Software	expensive branded solutions
	Configurable	need human resources to configure
	Open standard specification	cheaper hardware
	Generic programmes	write/buy as per need
	programmable	more automation less human resource
Openness	Proprietary hardware	configurable but not programmable
	Vendor specific instruction set	only way to configure devices
	Simpler data plane	efficiently programmable
	Vendor neutral APIs	same for all

Table 2.1: SDN features with various aspects

to name a few [3] [4]. SDN also helps to run the experiment on production network using network virtualization such as by creating different slices of network [48].

Among all orthogonal research, security is essential to efficiently deliver basic functionalities. The security domain in SDN covers both aspects, which is either introduced by its new architecture or to assuring a new solution to traditional network threats. Few security related problems in SDN are Denial of Service (DoS) attacks at both the planes, Topology Poisoning, malicious app detection, and malicious link detection [5, 49]. SDN may also looks for providing solutions for Address Resolution Protocol (ARP) poison, rogue Dynamic Host Configuration Protocol (DHCP) server, etc. traditional networking threats.

In SDN, any security threat can be handled by either prevention, detection or detection with mitigation. There is a hypothesis that for any of mentioned approach needs exact topology information to overcome. To validate the hypothesis, flow entry flooding[6], and SYN flooding attacks[7] are examined with SDN openness and programmability. A topology component is required for detection namely switch. Some other components such as links are also mandatory for mitigation. Next section discusses about topology in more detail.

2.2 Topology Discovery in SDN

Software Defined Networking (SDN) decouples the data plane from the control plane. In SDN, the control plane is separated from data plane so a logically centralized controller can control various data paths (switches) in a data plane. Thus, it provides visibility of the entire networking infrastructure to the controller. It enables the applications running on top of the control plane to innovate through network management and programmability. The visibility can be used in decision-making for some controller applications. Applications such as shortest path routing, link load balancer, etc., need precise topology information to provide optimal results. To envision the centralized control and visibility, the controller needs to discover the networking topology of the entire SDN infrastructure.

Topology is a fundamental part of the network. Various components are switches, links, and hosts as shown in Figure 2.1. Topology discovery performed by the controller is used to make topology informed solutions. In the beginning, switches are performing HELLO communication to the controller to show the existence. ECHO is periodically sent by both parties to show aliveness. Links are discovered with the help of Link Layer Discovery Protocol or Broadcast Domain Discovery Protocol (LLDP/BDDP) packet movement. These packets are generated with the controller. The same packets reach back to the controller with traveled link information. Hosts are also discovered with generated traffic. Hosts are also discovered differently at the data link level in which the Address Resolution Protocol (ARP) is used to discover the target machine's Media Access Control

(MAC) address.

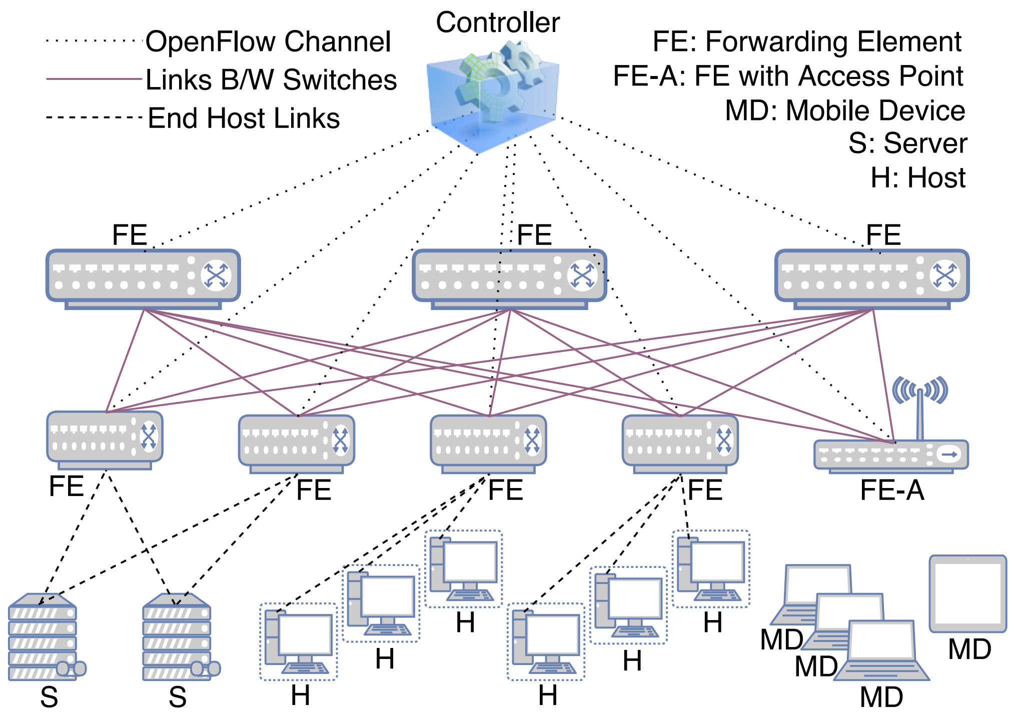


Figure 2.1: Global view construction in SDN

The switch discovery process is illustrated in Figure 2.2. In most of the scenarios, the controller starts earlier then switch boots. The switch establishes a Transmission Control Protocol (TCP) connection with the controller on pre-stored socket address. Optionally, Transport Layer Security (TLS) is also equipped along with TCP to provide communication security. After establishing successful TCP handshaking, OpenFlow Packet Type Hello i.e. OFPT-HELLO packets are exchanged to negotiate the highest common supported OpenFlow version. Once OpenFlow version is finalized, controller and switch further communicate on that negotiated version as shown in Figure 2.2. The controller asks the switch about supported ports and their capabilities in OFPT-FEATURES-REQUEST. A OFPT-FEATURES-REPLY packet holds response. Subsequently, OFPT-ECHO-REQUEST and OFPT-ECHO-REPLY messages are exchanged between switch and controller to confirm the liveliness status to each other. The link between switch and controller is either physical or dedicated virtual.

Ordinarily, switches are connected with other switches in any topology. This

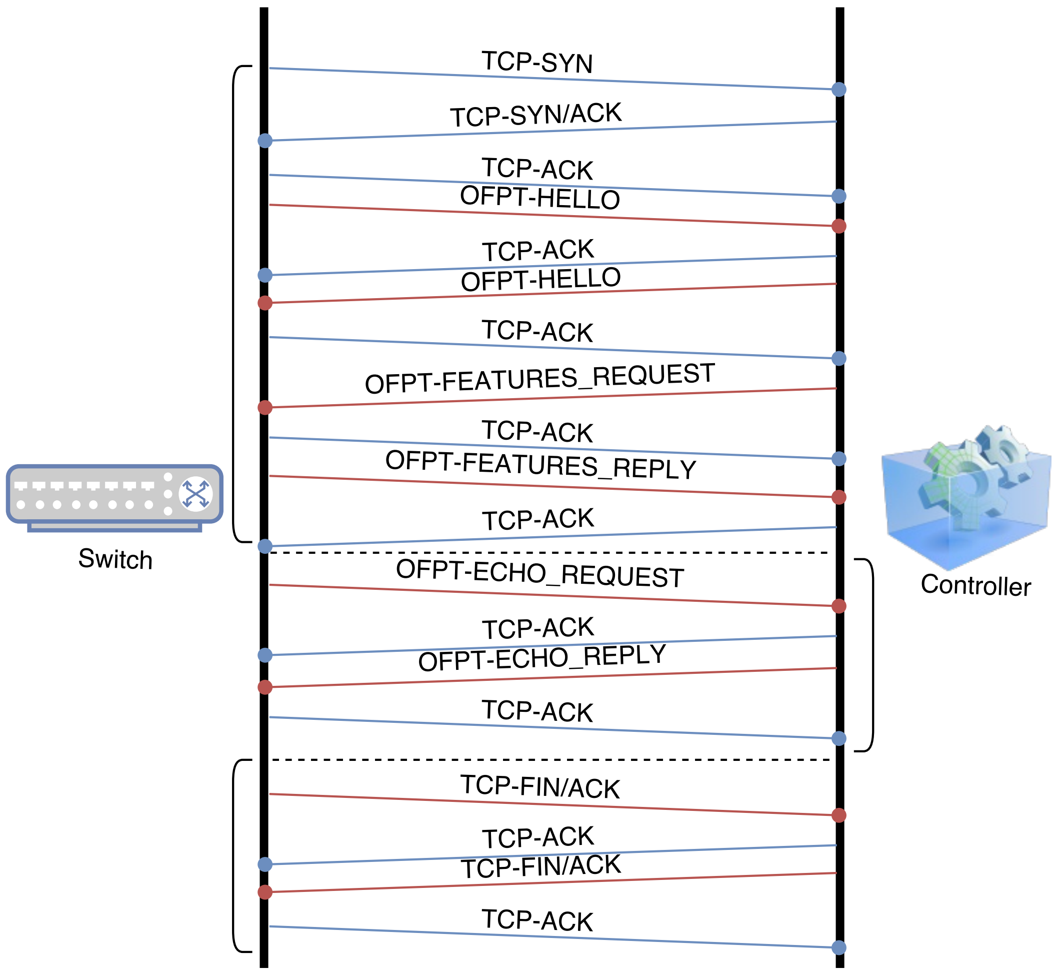


Figure 2.2: Switch discovery process

switch to switch link in SDN may be either configured as a control channel or left as a data channel. The control channel is to be manually configured which is used to carry OpenFlow packets. All other not-configured channels or links are used to carry data packets for the network. Hereafter data links are referred to as links. SDN provides programmable support to discover links. Two switches are connected with links using ports. In switch discovery, switch information and port information are made available at the controller. But still, the controller has no idea about the link of two switches. In the link discovery process, the controller gets the information about the link. In the link discovery process, the controller generates LLDP/BDDP packets for each port on every switch. Receiving switch pass them to the directed port. On the another end of the link, a switch receives and send it back to the controller. The controller uses this information for a link database. In the Section 2.3, a detailed discussion is made about link discovery[50–52].

Host discovery is also possible due to a separation of control plane and data plane. Hosts are attached to switch ports. Exact information of hosts can be tracked for better control over the network. Host tracking is also helps to monitor or maintains mobility to the hosts. Few network policy may be specified regarding hosts. For this scenario, the controller must track mobility of controller. Mobility monitoring also helps in virtual machine migrations. Host discovery is performed with traffic generated with hosts [50, 53].

At the data link layer, the host is again discovered. In the TCP/IP model, three types of address i.e. port, Internet Protocol (IP) and Media Access Control (MAC) are used to locate communication parties. An IP address is used to locate a logical network segment while MAC address is used to locate a host in the targeted network segment. ARP protocol is used to map IP address to the corresponding MAC address. Without this mapping, communication can‘t be completed. The ARP protocol is vulnerable to ARP Poison and ARP flooding attack due to lack of authentication and integrity. In the Section 2.4, ARP based communication and attacks are analyzed[17].

In topology discovery, although there are various components, most vulnerable are link discovery and data link layer host discovery. The switch discovery is performed over TCP communication and it can be secured using TLS. The host discovery is performed using traffic generated by hosts. It only affects mobility, which is rare in wired SDN. In next sections, a detailed analysis of the link discovery and host discovery(data link) is performed. Further threat model followed with attackers strategy is discussed.

2.3 Link Discovery

Link discovery is a process to identify links between OpenFlow switches. This section presents current deployments, the attack vector, and orthogonal research for link discovery. In Software Defined Networking (SDN), so far the link discovery is performed with a non-standard OpenFlow Discovery Protocol (OFDP) with Link Layer Discovery Protocol (LLDP) packets. Each controller, e.g., POX [8], Ryu [9], Floodlight [11], OpenDayLight [10], ONOS [13], and HPE-VAN [14] implement their own OFDP variant. However, all of these are using either LLDP packets or alike Broadcast Domain Discovery Protocol (BDDP) packets. As shown in Figure 2.4, LLDP/BDDP packets are initiated by the controller, pass through switches, and confirms a link between switches[50]. There are certain possible cases in which non-OF-Switches separates two OpenFlow enabled switches. The controller does not control a non-OpenFlow switch, hence, we assume that it works as per specifications. However, a link discovery process must identify the links.

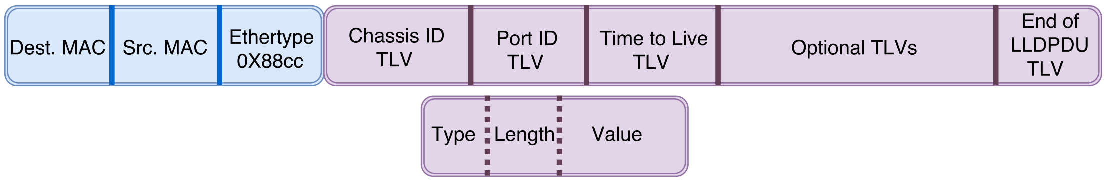


Figure 2.3: LLDP/BDDP packet format

Before understanding link discovery in SDN environment, let's look at LLDP packet format which is depicted in Figure 2.3. The LLDP packet is a collection of different set of Time-Length-Value (TLV)s. Different controllers maintain a different set of TLVs. The Chassis, Port, and TTL TLVs are generally used to store source data path id (*dpid*), source port number, and packet expiry time. It is because LLDP [54] is originally used in traditional networks, and SDN is just using the similar packet format. Thus, each combination of TLVs serve different purposes. In Section 2.3.1, we will show that few of TLVs are not required for the topology discovery process.

As per Figure 2.4, LLDP packets are generated for each port on each OF-switch (step 1). PACKET_OUT packet with LLDP packet as a payload is generated. After reaching at the switch, payload LLDP packet is transferred on a specified output port (step 3). If such a LLDP packet is received at the switch, its forwarding table is consulted for further action. After consultation, a PACKET_IN message is generated with received LLDP as payload (step 4). In step 5, the controller receives PACKET_IN message with event information or destination dpid and destination port. Because the LLDP packet holding source dpid and source port, the controller has complete source to destination information on which the LLDP packet is traveled. This information helps to create a unidirectional link between source and destination. The process will be completed for each unidirectional link to provide a complete set of unidirectional links.

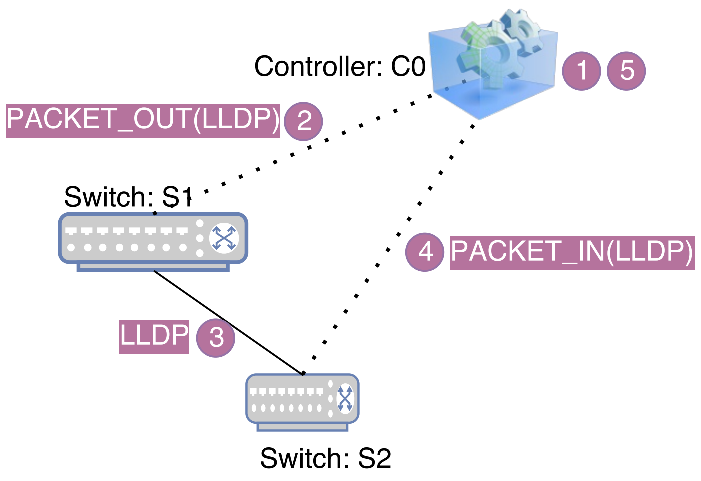


Figure 2.4: LLDP/BDDP movement for link discovery

2.3.1 Current Deployments

This section provides an overview of different implementations for the link discovery. We consider POX[8], Ryu[9], OpenDayLight[10], Floodlight[11], ONOS[13], Beacon[12] and HPE-VAN[14] controllers for link discovery implementation and vulnerabilities. The controllers with versions are specified in Table 2.2. Each controller generates different variant of LLDP packet. Some have security feature in it while other miss the opportunity. This section discusses different LLDP packets from different controllers and how LLDP packets are parsed at controller upon receiving.

POX generates LLDP packet with four TLVs as shown in Figure 2.5. ChassisId and SystemDiscription TLVs having same content i.e. string start with 'dpid:' follows with target switch's dpid. No security cryptic hash value in any of TLVs exists which indicates prone to LLDP poison. POX controller installs a flow entry with matching parameter dl_dst = 01:23:20:00:00:01, dl_type = 0x88cc to the switch which forwards LLDP packet to controller. The appendix can be use for directory tree for each controller to illustrate important class file and jar file. That demonstration helps the reader for further exploration.

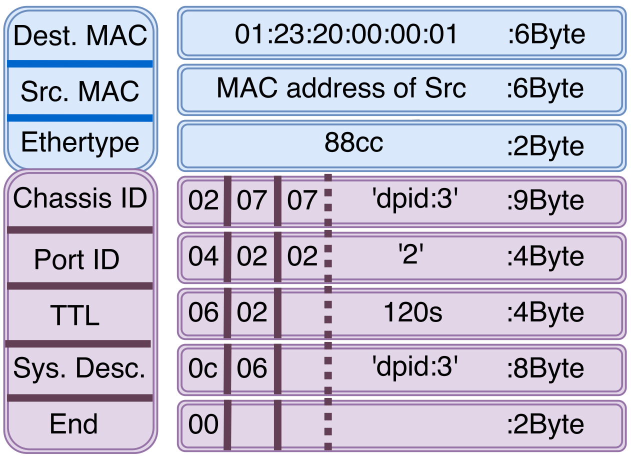


Figure 2.5: POX LLDP frame structure

Algorithm 1 gives an overall idea of LLDP packet parsing on the POX controller. We can observe here that different unique destination Media Access Control (MAC) addresses in LLDP packets from multiple controllers are present. Each LLDP payload is checked against some conditional TLVs which are specified in Table 2.2. For POX, conditional TLVs are ChassisId, Port, TTL and TLVs count. LLDP packet does not include any security TLV in POX. After all conditional processing, the controller extracts source datapath identifier(dpid) and source port to create link. For various controllers, this information is extracted from different TLVs, which can also observed in Table 2.2. The event.dpid and event.port information came with PACKET_IN, which is used as the destination of established link. If any switch receives LLDP packet over non OpenFlow link, it generates PACKET_IN with own id(event.dpid) and receiving port id(event.port).

Figure 2.6 shows Ryu LLDP packet structure which is quite similar to POX. Chassis Id is 21 Byte long string i.e. 'dpid:0000000000000003'. Before adding a link, the controller checks subtype of Chassis TLV and value, which should start with 'dpid:'. Algorithm 2 gives a basic idea about Ryu controller for parse a receiving LLDP packet. If few subtypes are matched against specified subtype then only dpid and port information values are extracted otherwise such packet is

Algorithm 1 POX LLDP packet parsing

```
Require: ethFrame, event.dpid, event.port

 1: procedure POX_LLDP_PARSE
 2:     if ethFrame.ethType = 0x88cc then
 3:         if ethFrame.dstMac = 01:23:20:00:00:01 then
 4:             ethlldp ← EXTRACT($ethFrame$)
 5:             if ethlldp.totalTvls $\geq$ 3 then
 6:                 if ethlldp.fstTvl = ChassisTlv then
 7:                     if ethlldp.sndTvl = PortTlv then
 8:                         if ethlldp.trdTvl = TtlTlv then
 9:                             sysDescPort ←
10:                         EXTRACT($ethlldp.PortTlv$)
11:                             sysDescDpid ←
12:                         EXTRACT($ethlldp.SysDesTvl$)
13:                             if sysDescDpid = NULL then
14:                                 sysDescDpid ←
15:                          EXTRACT($ethlldp.ChassisTlv$)
16:                             end if
17:                             if ethlldp.PortTlv.subType = 2 then
18:                                 if isExist(sysDescDpid) then
19:                                     ADDLINK($event.dpid$, $event.port$, $sysDescDpid$,
    $sysDescPort$)
20:                                 end if
21:                             end if
22:                         end if
23:                     end if
24:                 end if
25:             end if
26:         end if
27:     end if
28: end procedure
```

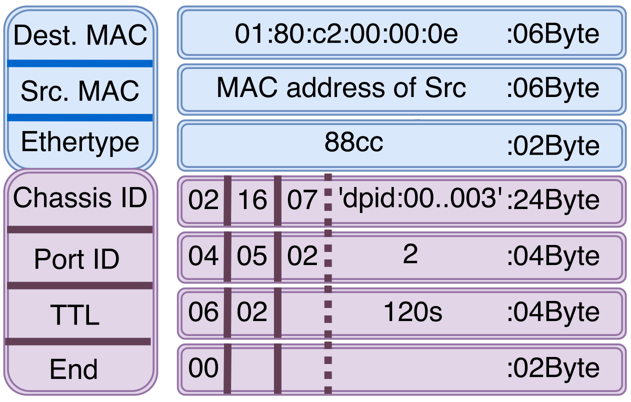


Figure 2.6: Ryu LLDP frame structure

simply discarded. In Ryu, Chassis TLV has subtype seven and value field is having a string starts with ‘dpid:’ then only source dpid will be extracted. For source port, Port TLV must have length equals to four. After successful extraction of source dpid and port, a link is added between source and destination. The destination is carried in event information i.e. ‘event.dpid’ and ‘event.port’.

Algorithm 2 Ryu LLDP packet parsing

Require: ethFrame, event.dpid, event.port

```
1: procedure Ryu_LLDP_Parse
2:     if ethFrame.ethType = 0x88cc then
3:         ethlldp ← Extract($ethFrame$)
4:         if ethlldp.ChassisTlv.subType = 7 then
5:             if startsWith(ethlldp.ChassisTlv) = ‘dpid:’ then
6:                 sysDescDpid ← Extract($ethlldp.ChassisTlv$)
7:                 if ethlldp.PortTlv.subType = 2 then
8:                     if ethlldp.PortTlv.length = 4 then
9:                         sysDescPort ←
10:                                Extract($ethlldp.PortTlv$)
11:                        addLink($event.dpid, event.port, sysDescDpid, sysDescPort$)
12:                    end if
13:                end if
14:            end if
15:        end if
16:    end if
17: end procedure
```

OpenDayLight introduces security hash in its LLDP payload as shown in Figure 2.7. This hash is a MD5 digest of unknown TLV with id 1(UK1) TLV content. UK1 TLV holds a string value i.e. ‘openflow:1:2’ which indicates that this packet is generated for a switch having dpid 1 and port number 2. If an attacker calculates md5 hash for similar string, then it can be used to poison LLDP.

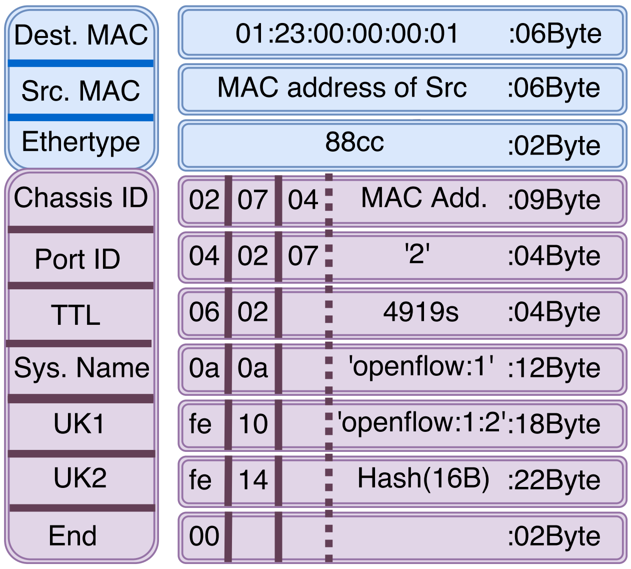


Figure 2.7: OpenDayLight LLDP frame structure

Figure 2.8 gives an idea about the calculation of hash value. A node connector id is added to a secure final key before calculating MD5 hash. Here a static security key is used, which is the cause of vulnerability. If the attacker can use that key to generate LLDP packet, it can deceive the controller. Section 2.3.3 presents a detailed explanation of attacker strategies to uses this information along with LLDP packet parsing algorithm to poison LLDP.

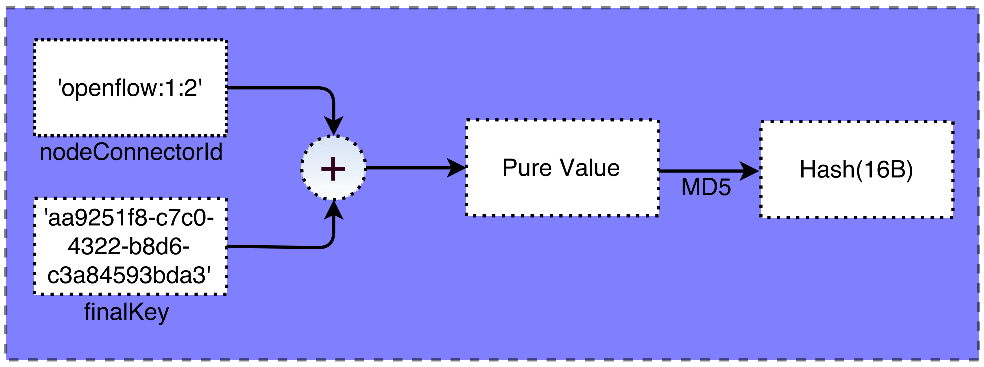


Figure 2.8: Hash calculation for OpenDayLight LLDP frame

Algorithm 3 can be used to understand LLDP packet parsing. After extracting LLDP payload from Mininet frame, system name TLV is checked for not null.

Along with dpid and port id one more field is extracted which is node connector id. Node connector id is added to a secure static key before calculation of the MD5 hash. Received and calculated hash are compared for equality to make sure of integrity. After assurance of integrity, algorithms add the link to link database.

Algorithm 3 OpenDayLight LLDP packet parsing

Require: ethFrame, event.dpid, event.port, finalkey

```
 1: procedure ODL_LLDP_PARSE
 2:     ethlldp ← EXTRACT($ethFrame$)
 3:     if ethlldp.sysNameTvl $\neq$ NULL then
 4:         sysDescPort ← EXTRACT($ethlldp.PortTvl$)
 5:         sysDescDpid ← EXTRACT($ethlldp.ChassisTvl$)
 6:         nodeConnectorId ← EXTRACT($ethlldp.sysNameTvl$)
 7:         if nodeConnectorId $\neq$ NULL then
 8:             if ethlldp.Uk1 $\neq$ NULL then
 9:                 receivedHash ← EXTRACT($ethlldp.uk2$)
10:                 calculateHash ← MD5($nodeConnectorId + finalkey$)
11:                 if receivedHash = calculateHash then
12:                     ADDLINK($event.dpid$, $event.port$, $sysDescDpid$, $sysDescPort$)
13:                 end if
14:             end if
15:         end if
16:     end if
17: end procedure
```

Floodlight controller uses two different fields, one is hash for security and another is time stamp for latency calculation. Figure 2.9 gives the detail of LLDP frame used in the Floodlight controller.

Hash calculation in LLDP packet generated by Floodlight Controller is shown in Figure 2.10. System time is multiplied by a prime number 7867 to generate a security number. This number further added to a hash of interface list. This operation generates a security hash of eight bytes long which is placed in second unknown TLV of LLDP packet. The attacker can not generate this hash on the local machine because of different network interface list i.e. Mininet, wireless. But security loophole exists, and this hash is calculated once during controller startup. So, same hash value put in each LLDP packet over the time again and again. Attacker can copy this LLDP packet content to perform the attack.

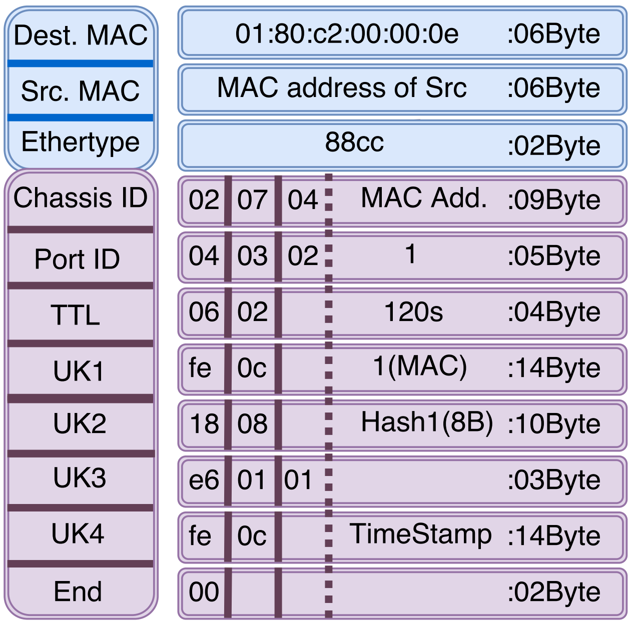


Figure 2.9: Floodlight LLDP frame structure

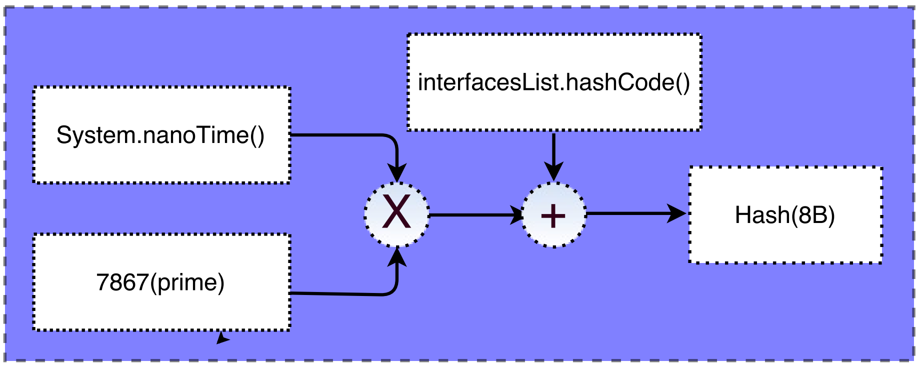


Figure 2.10: Hash calculation for Floodlight LLDP frame

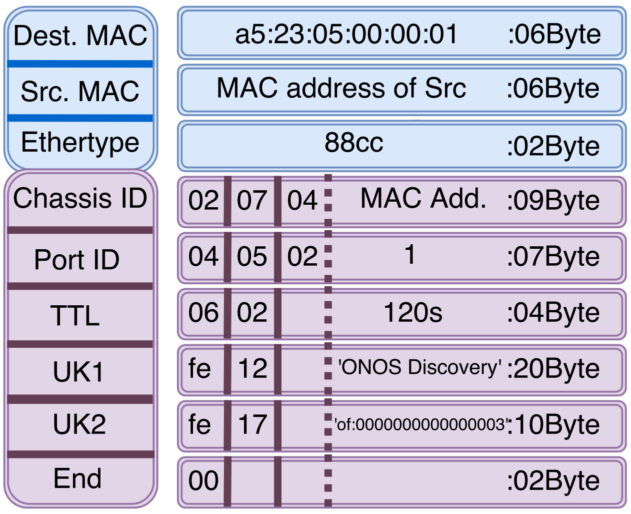
For a detailed understanding of LLDP packet parsing in Floodlight controller, Algorithm 4 can be referred. After extraction of LLDP payload, port length must be three to proceed. Port id and dpid are extracted from port and first unknown TLVs respectively. Time-stamp which is used for latency calculation is extracted from fourth unknown TLV. An order id(a security hash) comes in second unknown TLV is compared with a stored id(myId) to ensure the integrity. After assurance of the integrity of packet, a link is added.

Figure 2.11 shows the LLDP packet structure for ONOS controller. At the controller, ChassisId TLV is used for validation of LLDP packet. This packet does not contain any security hash, hence vulnerable to LLDP packet based threats.

Algorithm 4 Floodlight LLDP packet parsing

Require: ethFrame, event.dpid, event.port, myId

```
 1: procedure FDL_LLDP_PARSE
 2:     ethlldp ← EXTRACT($ethFrame$)
 3:     if ethlldp.PortTvl.length = 3 then
 4:         sysDescPort ← EXTRACT($ethlldp.PortTvl$)
 5:         sysDescDpid ← EXTRACT($ethlldp.Uk1Tvl$)
 6:         timeStamp ← EXTRACT($ethlldp.Uk4Tvl$)
 7:         linkLatency ← CALCULATELATENCY($timeStamp$)
 8:         otherId ← EXTRACT($ethlldp.Uk2Tlv$)
 9:         if myId = otherId then
10:             if isExist(sysDescDpid) then
11:                 ADDLINK($event.dpid$, $event.port$, $sysDescDpid$, $sysDescPort$, $linkLatency$)
12:             end if
13:         end if
14:     end if
15: end procedure
```



Dest. MAC
a5:23:05:00:00:01 :06Byte
Src. MAC
MAC address of Src :06Byte
Ethertype
88cc :02Byte
Chassis ID
02 07 04 MAC Add. :09Byte
Port ID
04 05 02 1 :07Byte
TTL
06 02 120s :04Byte
UK1
fe 12 'ONOS Discovery' :20Byte
UK2
fe 17 'of:0000000000000003' :10Byte
End
00 :02Byte

Figure 2.11: ONOS LLDP frame structure

Algorithm 5 gives insight for LLDP packet parsing in ONOS controller. Extracted LLDP payload is checked on no null value. Source port id and dpid values are extracted from port and chassis TLVs respectevely. And if dpid is present then link is added.

It is shown in Figure 2.12 that Beacon does not include any cryptographic value in its TLV, which clearly indicates that it is susceptible to LLDP packet based threats. Length field in Port TLV is used for validation of LLDP packet at

Algorithm 5 ONOS LLDP packet parsing

Require: ethFrame, event.dpid, event.port

```
1: procedure ONOS_LLDP_PARSE
2:     ethlldp ← EXTRACT($ethFrame$)
3:     if ethlldp ≠ NULL then
4:         sysDescPort ← EXTRACT($ethlldp.PortTvl$)
5:         sysDescDpid ← EXTRACT($ethlldp.ChassisTvl$)
6:         if isNotNull(sysDescDpid) then
7:             ADDLINK($event.dpid$, $event.port$, $sysDescDpid$, $sysDescPort$)
8:         end if
9:     end if
10: end procedure
```

the controller.

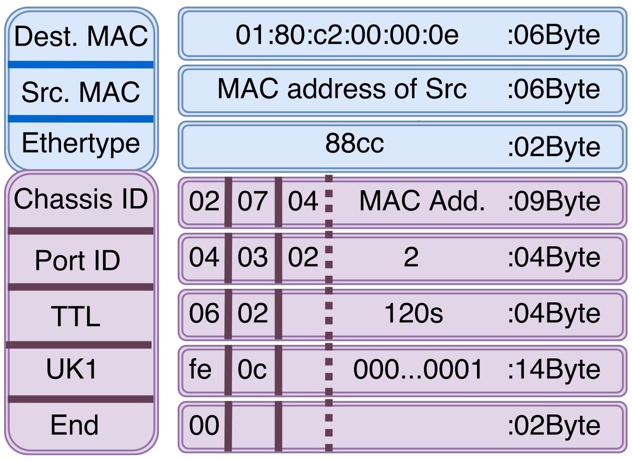


Figure 2.12: Beacon LLDP frame structure

In algorithm 6, LLDP packet processing on Beacon controller is discussed. After LLDP payload extraction, port value is extracted from port TLV. This value is used to validate valid LLDP packet. After extraction of source dpid, a link is added from source to destination.

In Figure 2.13, HPE-VAN controller generated LLDP (infect HP claims that it is BDDP) frame is represented. This frame use one of the two destinations MAC address. One is used for direct link while other denotes indirect link. Indirect link has one or more non-OpenFlow based switches. Ethertype field in frame is different from other controller implementation i.e. 0x8999. LLDP packet contains security hash. But that is fixed for controller's current cycle, so attacker put that directly in poisoned LLDP packet.

Algorithm 6 Beacon LLDP packet parsing

Require: ethFrame, event.dpid, event.port

```
 1: procedure BEACON_LLDP_PARSE
 2:     ethlldp ← EXTRACT($ethFrame$)
 3:     sysDescPort ← EXTRACT($ethlldp.PortTvl$)
 4:     if sysDescPort ≠ NULL then
 5:         if ethlldp.PortTlv.length = 4 then
 6:             sysDescDpid ← EXTRACT($ethlldp.Uk1Tvl$)
 7:             ADDLINK($event.dpid$, $event.port$, $sysDescDpid$, $sysDescPort$)
 8:         end if
 9:     end if
10: end procedure
```

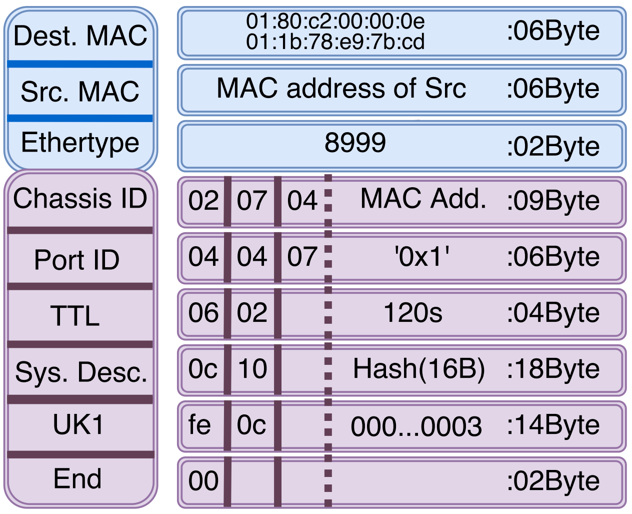


Figure 2.13: HPE-VAN LLDP frame structure

Figure 2.14 provides an insight of hash calculation in HPE-VAN. System time is multiplied with a prime number i.e. 7867 to get a secure key. This secure key is added to a hash of local IP address to generate a hash of 16 bytes. Due to its static nature, an attacker does not need to generate this hash locally.

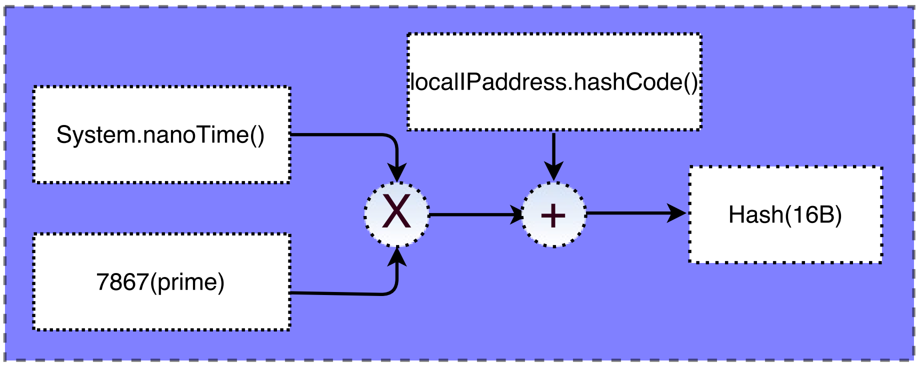


Figure 2.14: Hash calculation for HPE-VAN LLDP frame

Algorithm 7 HPE-VAN LLDP packet parsing

Require: ethFrame, event.dpid, event.port, calculateHash

```
1:  procedure HPVAN_LLDP_PARSE
2:      if ethFrame.ethType = 0x8999 then
3:          if ethFrame.dstMac = 01:80:C2:00:00:0e or 01:1b:78:e9:7b:cd  then
4:              ethlldp ← EXTRACT($ethFrame$)
5:              sysDescPort ← EXTRACT($ethlldp.PortTvl$)
6:              if sysDescPort $\neq$ NULL then
7:                  sysDescDpid ← EXTRACT($ethlldp.Uk1Tvl$)
8:                  receivedHash ← EXTRACT($ethlldp.sysDesTvl$)
9:                  if receivedHash = calculateHash  then
10:                     ADDLINK($event.dpid, event.port, sysDescDpid, sysDescPort$)
11:                 end if
12:             end if
13:         end if
14:     end if
15: end procedure
```

The reader can use Algorithm 7 to understand BDDP packet parsing at the HPE-VAN controller. Two different MAC addresses are used to identify different type of packets. One for direct links and other for indirect link. After extraction of Mininet payload, port value is extracted. For a valid port number dpid and security hash are extracted. If recevied hash and calculated hash are equal then a

Controller(Ver.)	Conditional Tlvs					Information Source	
	TlvCnt	CTlv	PTlv	TTlv	SDTlv	Dpid	Port
POX(0.2.0)	✓	✓	✓	✓		*CTlv*	*PTlv*
Ryu(4.12)		✓	✓			*CTlv*	*PTlv*
OpenDayLight(3.0.7)					✓	*CTlv*	*PTlv*
Floodlight(1.2)			✓			*Uk1Tlv*	*PTlv*
Beacon(1.0.4)			✓		✓	*Uk1Tlv*	*PTlv*
ONOS(1.9.0)		✓				*CTlv*	*PTlv*
HP VAN(2.7.18)			✓			*Uk1Tlv*	*PTlv*

Table 2.2: Summary table for different controllers

link is added.

Table 2.2 presents a summary of considered controllers. Each controller is summarized for conditional TLVs and information sources. The conditional TLVs are examined for validation of LLDP packet. Source information for the link is extracted from LLDP packet. Destination information is extracted with event information. Conditional TLVs are TLV count, ChassisId TLV, Port TLV, TTL TLV, and SystemDescription TLV.

2.3.2 Threat Model

Suppose an LLDP packet with false information is injected to an OpenFlow-enabled switch, then it will be forwarded to the controller in the same way as any other genuine LLDP packet. The poisoned LLDP packet creates a false view to the controller which may lead to packet drop or eavesdropping, and such packets are also used to perform DoS attack. Few formal notations as it is seen below are discussed to define the threats formally.

Formal Description

- Controller C = $\{Vs, \delta\}$
- View states $Vs = \{\phi,\ GV',\ GV\}$
- Transition_function $\delta :< LLDP > \ \times Vs \ \rightarrow \ Vs$
- Global view GV = $< S, L >$
- Partial global view GV' = $< S, L' >$
- $< LLDP > = < CTlv, PTlv, TTlv, UK1Tlv, UK2Tlv...UKnTlv,$ $ELLDP >$
- Switch set S = $\{s_1, s_2, s_3....s_n\}$
- Link set L = $\{l_{12}, l_{21}, l_{23}, l_{32}....l_{mn}, l_{nm}\}$
- Partial link set $L' : L' \subset L$
- $Tlv = < Type, Length, Value >$

Here, we consider the controller with LLDP packets and global view only. Hence the controller (C) can be defined as the view states (Vs) and the transition functions (δ). Different view states can be empty state (ϕ) or partial global view (GV') or global view (GV). At any moment, the controller has a view state, and it consumes a $LLDP$ packet to create another view state. l_{mn} specifies a unidirectional link between switch m and n, and $\{,,\}$ and $<,,>$ are unordered and ordered sets respectively.

Transition_function

- $\delta(< LLDP >,\ \phi) \rightarrow GV'$
- $\delta(< LLDP >,\ GV') \rightarrow GV'$
- $\delta(< LLDP >,\ GV') \rightarrow GV$

A controller with view state ϕ or GV' consumes $LLDP$ packet and may transit to another GV'. Among various GV', at one GV' the controller consumes $LLDP$

and creates the global view GV. In link discovery process, three kinds of threats are possible called *LLDP poison*, *LLDP flooding* and *LLDP Replay*.

The LLDP packets are generated for each port on each switch. These packets are reached to target switch, the switch forwards them to the controller with the help of a dedicated flow entry or default miss entry. The major threats for the link discovery are Replay, Poison, and Flooding [55] attacks. Let's investigate these attacks with the help of Figure 4.1, which consists of three OpenFlow switches, switches $S1$ and $S2$ have three hosts attached to each of them. The two hosts are malicious, e.g., either a person with malicious intentions is operating them or malicious application are installed on them. The switches $S1$ and $S2$ are connected with $S3$, and the dash line between $S1$ and $S2$ represents a fake link.

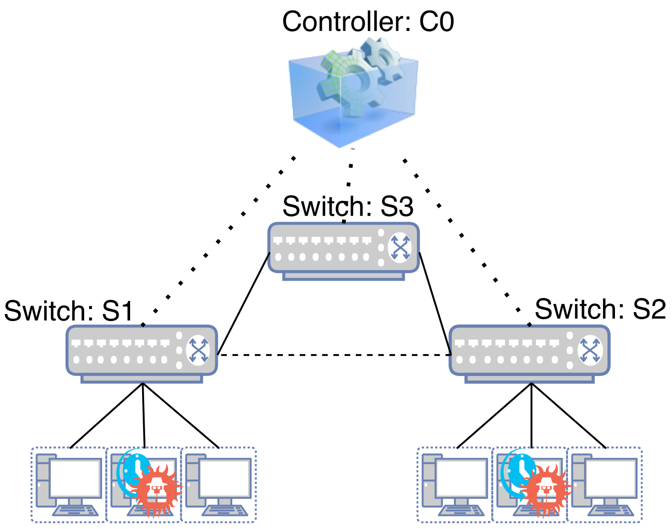


Figure 2.15: Topology for the link discovery attack vector

Poison Attack (PA): The attacker creates fake LLDP packets and send it to the attached switch. The switch is unable to differentiate genuine and fake packets, thus the packets are sent to the controller. The controller also has no way to find the source of the packet and it cannot evaluate the integrity of the packet. In this way, a generated false link creates topology poison. For instance in Figure 4.1, if an attacker at $S1$ creates an LLDP packet containing information like Chassis id is 2 and port is 3, then the switch would forward it to the controller. The controller makes a unidirectional link between $S2$ to $S1$.

LLDP Poison

- $\delta(< LLDP^P >, GV') \rightarrow GV^P$
- $\delta(< LLDP^P >, GV) \rightarrow GV^P$ where $GV^P \not\subset GV$
- $GV^P = < S, L^P > \bigcup < S^P, L^P >$ where $S^P \not\subset S$ and $L^P \not\subset L$

If a poisoned LLDP packet $LLDP^P$ is consumed by the controller (C) with GV' or GV, it creates poisoned global view (GV^P). Each switch uses a flow entry which forwards LLDP packets to the controller, and the entry can be used to flood LLDP packets towards controller. Here, m_1, m_2 are match entries, and a_1, a_2 are actions specified on OpenFlow protocol.

Flooding Attack (FA): When a controller receives a fake crafted LLDP packet, it computes logic. Hence when an attacker sends a flood of LLDP packets, e.g., 50,000 packets per second, the resource consumption at controller increases rapidly, and it negatively effects the benign packets service rate. Additionally, these large number of fake LLDP packets waste switch to controller bandwidth and controller CPU cycles.

LLDP Flooding

- $< LLDP >$ moves from $DataPlane \rightarrow ControlPlane$ iff matching flow entry Fe exists
- $Fe = < M, A >$ where M $= \{m_1, m_2, ..m_n\}$ and A $= \{a_1, a_2, ..a_n\}$
- If Fe exists then all packet with M moves from $DataPlane \rightarrow ControlPlane$

Replay Attack (RA): Due to LLDP packet propagation to each port on a switch, the attached hosts also receive LLDP packets. However, if one of the attached host send a received LLDP packet from some other host that is attached to another switch, then the receiving switch and the controller has no way to identify the source of that LLDP packet. For instance, if a malicious host at switch $S1$ receives LLDP and share it with malicious host attached to $S2$, then the host at $S2$ send the LLDP packet on local port to $S2$, and again $S2$ sends the received LLDP packet to $S1$ with its malicious host. Finally, $S1$ and $S2$ will send these packets to the controller, and the controller confirms that $S1$ and $S2$ have a direct link.

LLDP Replay

- $< LLDP >$ packet generated at the time ‘t’ is $< LLDP_t >$
- $[\delta(< LLDP_t >, GV') \rightarrow GV^P]_{t'}$
- where $t' > t + latency$

Sometimes an attacker captures the LLDP packet generated at time t, and inject the same packet through the same or any attached node, and if the controller consumes it at time t', then it will generate a poisoned view. It is assumed that the packet’s content are kept same otherwise it will become the LLDP Poison instead of LLDP Replay attack.

In SDN, there is no robust mechanism to verify the authenticity and integrity of the LLDP packets. It is the responsibility of controller to secure the LLDP communications. The LLDP communication cannot be secured with Transport Layer Security (TLS) because TLS secures only the switch to controller communication and vice verse. At the edge switches, the end hosts are attached. If a host with malicious intention sends an LLDP packet to switch, the switch has no clue to differentiate. It forwards the poisoned LLDP packet to controller. The controller has to identify genuineness of received LLDP packet to obtain a correct view.

Manifestation of Attacks

Effects of described attacks are ranging from a fabricated link to controller fingerprinting, which leads to wastage of network resources.

Fabricated Link: Replay and Poison attacks create fabricated links. Any such fabricated link poison the topology, which effects topology aware applications, e.g., load balancer.

Controller Fingerprinting: If a controller generates and sent LLDP packets to each port, the host also receives these LLDP packet. Each controller have different LLDP packets [56] which make the easy guess for controller identification.

Resource Wastage: In Flooding attack, the controller receives large number of LLDP packets, and the processing of these packets waste controller CPU cycles. These packets traverse from switch to the controller over TCP/TSL layer, hence it causes bandwidth wastage. When OF-switch is configured with TSL to ensure security, a large amount of computation has to be done to encrypt the fake LLDP packets, this further increases the resource consumption at switches and controller due to encryption/decryption process.

2.3.3 Attacker's Approach

An attacker can proceed with the knowledge of LLDP packet format and parsing algorithm employed by the controller. Each controller generates unique LLDP packets. Some LLDP packets do not include any security hash, therefore an attacker can place a poisoned LLDP packet directly to the local network interface. For the LLDP packets that include hash, to break security for such controllers an attacker can takes help of local instance of the same controller's library. The overall structure of the attack mechanism is shown in Figure 2.16. The attacker takes help from packet parsing algorithm and local library to build poisoned LLDP and send it to the nearest switch. The switch to controller communication follows OpenFlow standard. In a nutshell, two kinds of attack are possible Rcraft and Lcraft.

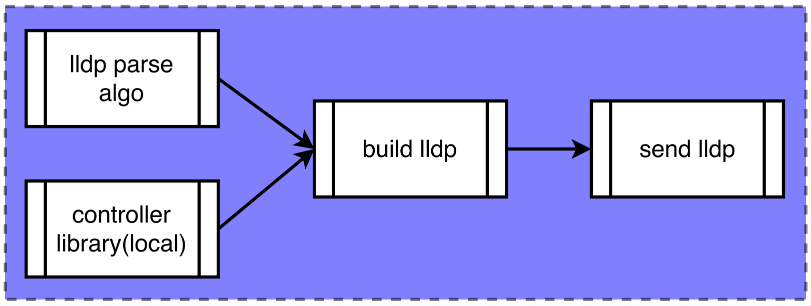


Figure 2.16: Basic setup to perform link discovery attacks

***Rcraft*:** **(row craft)** In this attack, attacker craft poisoned LLDP packet with the help of LLDP packet format and LLDP packet parsing algorithm of a given controller. This crafting is possible only if the controller does not include any library generated content which is completely unpredictable. After generation of such poisoned LLDP packet, the attacker places it on an interface which is directly connected to an OpenFlow-enabled switch. An attack is created in the same way for POX, Ryu, ONOS, Beacon, and HPE-VAN. HPE-VAN includes a library generated security hash but it is static, so its security could be broke.

***Lcraft*: (craft using a local library instance)** Some controllers, e.g., OpenDayLight and Floodlight, uses more secure way by generating dynamic content in LLDP packet. To poison these controllers, the attacker needs the knowledge of library which is used to build LLDP packet. We performed reverse engineering to obtain important details. We use the same library on the local machine to create a poisoned packet with desired changes. This packet is kept on a local interface to forward it to switch.

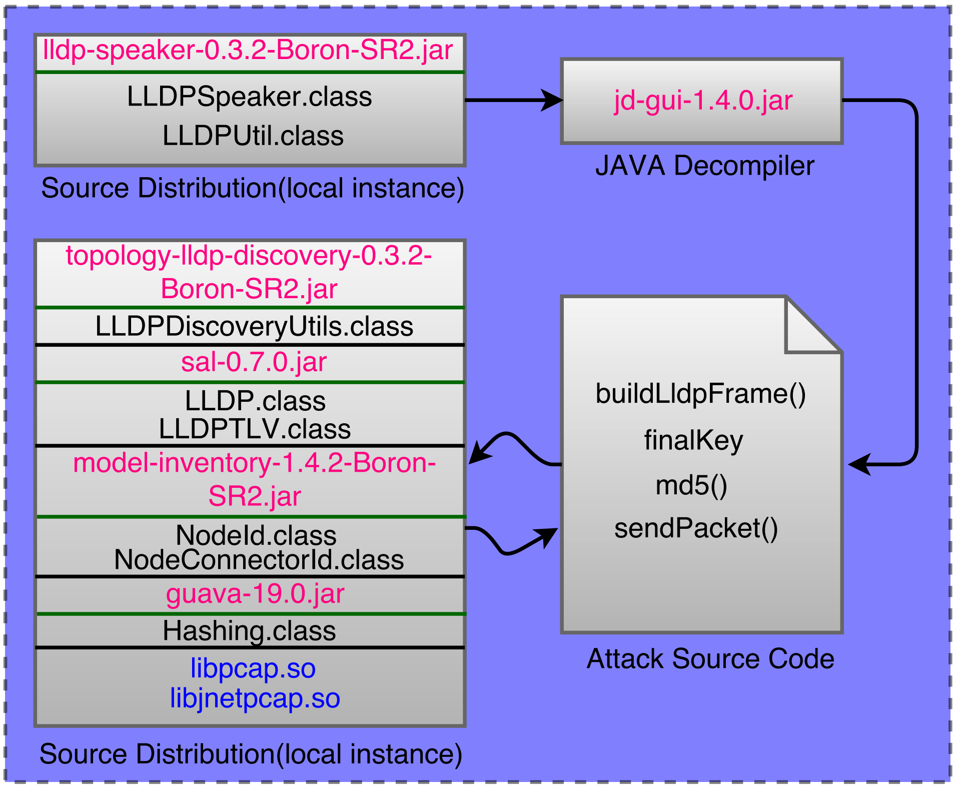


Figure 2.17: Attack details for OpenDayLight controller

Figure 2.17 shows the attack strategy on the OpenDayLight controller, and JAVA decompiler is used to obtain the source of building function for LLDP packet.

Secure key is extracted, and a MD5 function from the same controller library is used to build poisoned packets. Some shared object files, i.e., libpcap.so and libjnetpcap.so, are used to send Mininet frame containing LLDP directly on a network interface.

Attack details for Floodlight controller is shown in Figure 2.18. As the source code is available, some code is taken directly to build LLDP packet and use network library to send a packet. The Floodlight controller includes some dynamic content, but it is static.

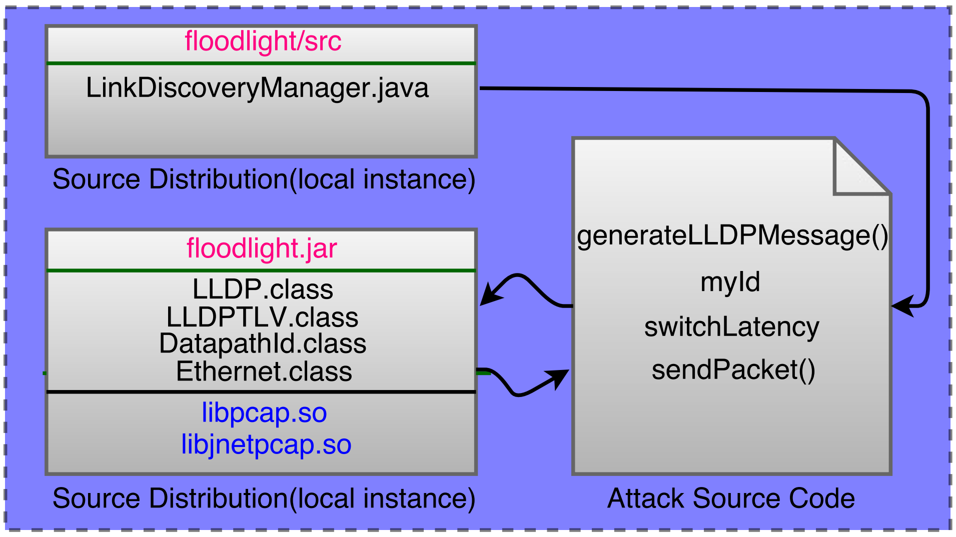


Figure 2.18: Attack details for Floodlight controller

We performed LLDP Poison, LLDP flooding, LLDP Replay attacks on the topology shown in Figure 2.19. All three attacks are performed on each type of controller. It is also observed the effect of these attack on the Mininet [57] emulator as well as a physical test bed. Table 2.3 briefs about configuration parameters used while performing the attacks. Table 2.4 shows the complete information about different types of attacks carried out on various controllers and required attack type. $S1$, $S2$, $S3$, and $S4$ are the OpenFlow-enabled switches, and $C0$ is an SDN controller. We explore seven specified controllers to find vulnerabilities. The attacker operates using the host which is attached to switch $S3$.

To perform LLDP Poison, the attacker crafts an LLDP packet with information such as ChassisId and PortId. In other words, the packet asserts that it has been created for switch $S4$ and forwarded to the port 3. The attacker has sent this

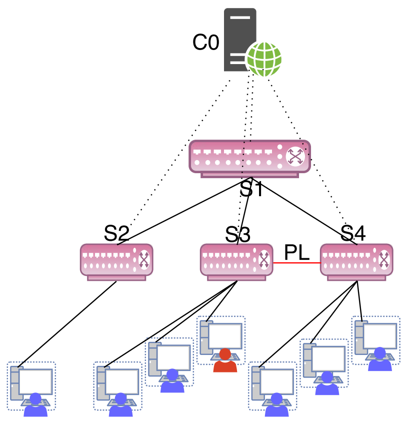


Figure 2.19: Attack topology

packet to switch via its local interface that is directly connected to the switch. The switch $S3$ (OpenFlow-enabled) reads the packet and send it to the controller in PACKET_IN message. The controller parses the message to get event source like port two on switch $S3$. It also explores the payload and gets LLDP packet. Now the controller creates a link between port three on switch $S4$ and port two on switch $S3$.

Resource	Configuration
Victim/Attacker OS	UBUNTU 16.04$LTS(64bit)^*$
Victim configuration	4CPUs and 8 GB
Attacker configuration	4CPUs and 8 GB
Attack traffic	50000 Packet/Sec
Software switch	OpenVSwitch(2.5.0)
Hardware switch	HP4506R
Network	1 Gbps

Table 2.3: Attack setup

An attacker also sends a large volume of LLDP packet, i.e., 50,000 packets to switch S3. The switch forwards these packets to the controller with ready flow entry. Although we perform attacks on each controller, Figure 2.20 presents experimental evidence for CPU utilization during standard load, LLDP flooding, and regular, again at the POX controller only. In our scenario, the POX controller is hosted at Intel Core i5 CPU 650 @ 3.20GHz machine with 4GB RAM and the operating system used is UBUNTU 16.04LTS 64bit.

To perform LLDP Replay attack, the attacker captures LLDP packet at one

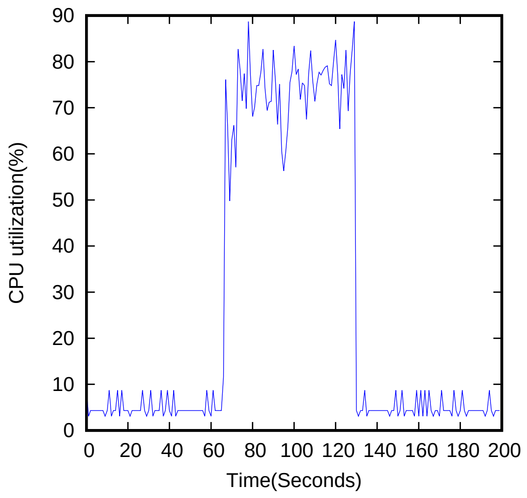


Figure 2.20: CPU utilization during LLDP flooding

of the hosts. The captured LLDP packet is transmitted after some time to the controller via an attached switch. Table 2.4 shows that apart from OpenDayLight and Floodlight controllers, all other controllers consume a delayed LLDP packet and produce a poisoned view.

Table 2.4 illustrates security strength of controllers against LLDP Poison (LP) , LLDP Replay (LR), LLDP flooding (LF) attacks. RC (Row Craft) and LC (Library Craft) are two classes of attacks which are used to perform the security attacks on topology in SDN. In RC, the attacker uses LLDP packet information and parsing algorithm to perform the attack, while in LC the attacker uses a library of the controller to perform the attack. Few controllers (e.g., Floodlight, OpenDayLight, and HPE-VAN) attach a hash to the LLDP packets. To perform the attack on these controllers, the library information is needed, which may have the information about the hash generation algorithm (or a secure static key) that is required for the successful attack. To understand more about how attacks can happen on different controllers, please refer to [56].

POX, Ryu, and ONOS has no security content in their generated LLDP packets, hence these controllers are prone to attacks. The OpenDayLight controller generates an MD5 hash of a string, i.e., ‘openflow:1:2’, which an attacker can also create if she knows the MD5 library and a secret key. Both the information can be achieved, if an attacker performs reverse engineering to the controller code.

Hence, having static hash key won't help to protect from such attacks. In Floodlight controller, local machine interfaces are required to calculate the hash, hence the attacker is not able to compute the same hash on the local machine. But once the hash is calculated, it is kept same for further LLDP packets. The attacker has an opportunity to put the same hash in fake crafted LLDP packets. In case of HPE-VAN, same mistake is repeated for all the packets, and same hash is used. Hence, an attacker can copy and use the same fake LLDP packets. A point to note in OpenDayLight is that hashes for each switch port are separate, but in case of Floodlight and HPE-VAN hash for each LLDP packet is same. In a nutshell, most of the industry and academic grade controller are insecure, specifically against the LLDP-based threats.

Controller(Ver.)	Vulnerability	LLDP Attack			Attack Type
		LP	LF	LR	
POX(0.2.0)[8]	No hash	✓	✓	✓	*RC*
Ryu(4.12)[9]	No hash	✓	✓	✓	*RC*
OpenDayLight(3.0.7)[10]	Static hash	✓	✓		*LC*
Floodlight(1.2)[11]	Static hash	✓	✓		*LC*
ONOS(1.9.0)[13]	No hash	✓	✓	✓	*RC*
HPE-VAN(2.7.18)[14]	Static hash	✓	✓	✓	*RC*
Beacon(1.0.4)[12]	No hash	✓	✓	✓	*RC*

Table 2.4: Different controllers with attack vector

The possible cause of these attacks are failing to check source authentication of the received packet, and failing to verify the integrity of packet, static content and constant flow entry. For instance, if a controller is unable to identify the source of LLDP packets, and the LLDP packet holds static content then the controller is prone to Replay attack. If the controller is unable to identify a source of LLDP packets, fail to check the integrity of LLDP packet, and the LLDP packet have static content, then Poison attack may happen. If the switch has a constant flow entry to pass LLDP packets to the controller then flooding attack may happen [56].

2.3.4 Orthogonal Research

Due to the prominent features of SDN such as dynamic resource management and centralized control, SDN can significantly improve the data communication and security in the underlying network. Hence, the use of SDN paradigm is being envisioned in various next generation networks such as Internet of Things (IoT)[58, 59], cellular networks (e.g., 5G)[60, 61], and intelligent transportation system (ITS)[62]. Authors propose an integrated framework consisting of SDN and edge computing application, which can enable dynamic orchestration of networking, caching, and computing resources to improve the performance of applications for smart cities. The topology creation and topology management at controller is one of the key function in SDN for efficient networking. However, the decoupling of control and data planes also opens SDN to various security attacks such as Distributed Denial of Service (DDoS), packet injection, and topology discovery attacks [50, 63], thus limits the applications of SDN. Topology discovery attack affects the visibility of the network by exploiting different core functionalities of the SDN controller. To target LLDP based threats, few attempts in history were also made. In rest of this section, we provide a brief overview of relevant and substantial state-of-the-art efforts that are done for the mitigation of topology discovery attacks. Mainly, these approaches are based on the inclusion of hash or static binding.

In existing deployments[8–14], the link discovery is performed with LLDP/BDDP packet movements. The controller generates LLDP/BDDP packets for each switch on each port. These packets are wrapped in OpenFlow packets. Each receiving switch unwraps packet and sends to the described port. A dedicated link is connected to the port to another port on another switch. Upon receiving an LLDP/BDDP packet, a switch sends to the controller after wrapping in OpenFlow message. Controller discovers the link with information within the packet and information coming with the packet. This entire process is commonly known as OpenFlow Discovery Protocol (OFDP). OFDP is a non-standard and vendor dependent protocol as we already seen in Section 2.3.1.

Authors in [64] suggest a unique variant, which is an extension to OFDP, i.e., OFDPv2. In OFDPv2 LLDP packets are only generated as one for each switch instead of each port on each switch. It is quite simple but reduces the number of LLDP packet from the controller to switch drastically.

TopoGuard [53] identifies the cause of LLDP packet based threats as a check failure in integrity and origin of the LLDP packet. It also refers that we have to stop any host to participate in LLDP packet propagation to stop the attack. To prove integrity and origin, an Hash-based Message Authentication Code (HMAC) TLV is added into the LLDP packet. But HMAC is calculated once, and it is used forever. Hence it suffers from LLDP Replay attack. This technique uses default host traffic, i.e., DNS and ARP, to find hosts, but if malicious host stops all such traffic then this process fails. Furthermore, the LLDP flooding attack is not considered at all in this solution.

SPHINX [65] uses an abstraction of flow graphs generated by PACKET_IN and FEATURES_REPLY messages. Flow graphs are used to validate all the network updates and the given constraints. The LLDP Poison detection technique uses static switch-port bindings to prevent LLDP Poison attack. Thus it does not support the SDN dynamic evolution.

OFDP_HMAC [66] uses HMAC authentication to provide both integrity and authentication. The HMAC is added to each LLDP packet. The dynamic value of the key is used to prevent the LLDP Replay attack. This detection technique is not addressing LLDP flooding attack, and it exhibits resource penalty to produce HMAC for each LLDP packet.

ESLD [67] only generates LLDP packet for non-host ports on each switch. But the approach is based on host traffic, and an attacker can forge its behavior. Also, a key-based hash is sent to each packet, which is time-consuming.

sOFTDP[68] suggests a technique to secure the link discovery process. If any switch is added to topology, LLDP packet is generated and sent to switch. The LLDP packet containing a hash value instead of clear MAC address. After initial discovery, no LLDP packets are sent to switches. To prevent from LLDP flooding attack ‘no flood’ rule is installed in switch after an initial constant time. This approach requires a change in OpenFlow switch design to support Bidirectional

Forwarding Detection (BFD). The race condition between initial flow entry and LLDP packet is unresolved. The LLDP containing a hash value for MAC address which is a time-consuming process.

Recently, authors in [69] propose a design of a self-healing protocol for automatic topology discovery and maintenance in SDN. The proposed approach integrates two enhanced features (i.e., layer two topology discovery, and autonomic fault recovery) in a unified mechanism.

Similarly, authors in [70] propose an approach for rapid recovery from link failures in SDN by using the local immediate (LIm) and the immediate controller dependent (ICoD) recovery techniques, which addresses the limitations of OF-based link recovery approaches. The results show that the proposed approach reduces the alternate path flow rules by aggregating the disrupted flows using VLAN tagging.

TEDP[71] or Tree Exploration Discovery Protocol is sending a single TEDP packet to selected switch for each port. Receiving switch sends the packet to the controller and all ports. Controller notes the source of the TEDP packet, which is present in the packet. Topology can be constructed with a series of such packets received from various switches. This approach is vulnerable to poison, Replay and flooding attack.

Apart from present deployment of OFDP and LLDP packet in the various controllers, the research community is also putting hard efforts to make efficient and secure link discovery. Authors in [51] propose ForCES based link discovery which provides additional computational capabilities to run LLDP at the switch level. The controller queries periodically to gather the topology information. Authors in [72] propose a switch agent based topology discovery mechanism. Initially, the controller generates and sends a multicast message, called TDP-Request. Upon receiving, the switches change from Standby to either Father nodes or Active node. Each node collect neighbor information but only Father nodes send the information asynchronously to the controller. Authors in [73] propose SDN-RDP, a distributed resource discovery protocol. More than one controllers manage various switches, hence the proposed protocol works in two phases, one for the controllers announce and another for joining the switches. For each event, packets are moved in various network entities to form the topology. SHTD [74] is a layer two topology discovery with autonomic fault recovery protocol. Topology discovery is performed with controller sending a topoRequest message. The propagation of this multicast message with nodes in four roles, i.e., non-discovered, leaf, v-leaf or core and each port in four states, i.e., standby, parent, child or pruned discover the topology. Autonomic fault recovery is performed with the help of managed components and autonomic manager. The autonomic manager detects the port status to make the update for managed components.

We can classify existing approaches into two categories, one which is accepting current OpenFlow specifications[53, 65–67]. In another classification, a certain change is required in the specifications[68, 69, 72–74]. Table 2.5 compares LLDP based current specification obeyed solutions to secure the link discovery. We assumes change in OpenFlow specification is adequate.

Approach	Authentication	Integrity	LLDP broadcast	Poison	Flood	Replay
TopoGuard[53]		y	y	y		
SPHINX[65]			y	y		
OFDP_HMAC[66]	y	y	y	y		y
ESLD[67]		y		y		y

Table 2.5: Comparison of different research proposals for security

2.3.5 Inferences

Based on deployed solutions and available literature on the link discovery, few inferences are made as follows.

- Link discovery is vulnerable to poison, flooding and Replay attacks. So a new solution is needed to reduce the effects of such attacks.
- Link discovery can be improved on current OpenFlow specifications.
- Current process of link discovery is inefficient. A new variant may increase efficiency.

2.4 Host Discovery at Data Link Layer

Address Resolution Protocol (ARP) is used to get the physical address or Media Access Control (MAC) address of an available Internet Protocol (IP) address. Unavailability of authentication and integrity in ARP communication may lead to exploitation of it for various kind of attacks. For example, a malicious machine can craft an ARP reply for which there was no ARP request or it can reply to a request which was originated for some other machine, or it can generate a false ARP request etc. By doing so an attacker can perform ARP poison, ARP flooding attacks which lead to Denial of Service attacks, Man-in-the-Middle (MitM) attack or redirection attacks. In this section, we examine current version of ARP working, the possibility of these attacks and try to assess their effects on security of Software Defined Networking (SDN). This will help to utilize the SDN programmbility to detect and mitigate such kind of attacks.

2.4.1 Current Deployments

Traditional computer network uses TCP/IP model (few layers) for communication. In this, whenever a computer (source) wishes to communicate with another computer (destination), the application layer of source computer generates a byte stream and transport layer converts it into segments using TCP or UDP. Next, IP layer attaches the IP header which includes source and destination IP address. IP address is used to uniquely identify a network segment. Based on the IP address of destination, source computer will figure out if the destination is a part of local network segment or not. If the destination is on a local network segment, source computer will look into its ARP table (a table where the responses to previous ARP requests are cached) to find the MAC address [75–77]. If it's not there, then it will broadcast an ARP request to find out the MAC address for the destination IP. If the destination is on a different network segment then the gateway will respond. Although the behavior and services of ARP is same in SDN as in traditional network but due the basic working principle of SDN, the mechanism is different. Figure 2.21 illustrates the basic steps of ARP communication in SDN.

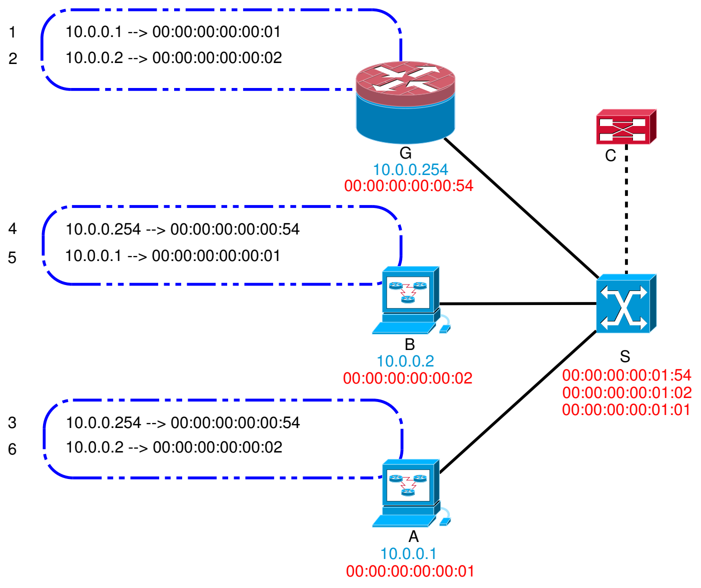


Figure 2.21: IP address to MAC address translation using ARP

As shown in Figure 2.21, each node has its own IP and MAC address while switch has three MAC addresses. In SDN, the switch forwards a packet to the destination only if it has a matching flow entry. If a packet is unable to match with any stored entries, the packet is directed to the controller. So sometimes ARP packet may visit controller C. For given topology, we are assuming switch will forward data to a relevent port with the consultation of the controller. Initially, each host if configured with Dynamic Host Configuration Protocol (DHCP), will obtain the own IP address from gateway G, so each host has gateway IP-MAC pair in own ARP table. Gateway G also has the pair for both the hosts because it is serving as DHCP server in the network. DHCP process is out of context, so we are omitting the details here. When host A needs to communicate with host B, it generates ARP request such that "who-has 10.0.0.2 tell 10.0.0.1". Switch S will receive this broadcast message and forward it to all ports except receiving port. Host B updates its ARP table with 10.0.0.1 —> 00: 00: 00: 00: 00: 01 entry assuming this entry will be needed in near future. Host B will respond with a unicast message such as "10.0.0.2 is-at 00: 00: 00: 00: 00: 02" to host A via switch S. After receiving ARP reply host A updates its own ARP table with entry 10.0.0.2 —> 00:00:00:00:00:02. In this figure, order for each entry is assigned a positive number, where larger number denotes later in sequence. MAC entry at

G depends on DHCP request arrival at G. Gratuitous ARP is a mean which is sometimes used to inform the receivers regarding information like change of L2 address or duplicate L3 address detection.

ARP frame must be considered to understand things more precisely in ARP communication. Figure 2.22 illustrates an ARP frame structure. Given illustration is the total of forty-two bytes, which contains fourteen bytes Ethernet header and twenty eight bytes long ARP payload. The Ethernet frame containing destination MAC Address, source MAC Address, and frame type. In the case of ARP communication, frame type is set to 0806. ARP payload starts with hardware type which is 0001 for Ethernet. Subsequently, the protocol type is set to 0800 for IPv4. Next, hardware address length is specified which is 06 for Ethernet MAC addresses. Protocol address length is set to 04 for IPv4 address. If the payload is ARP request packet then 0001 is used, and 0002 is used for ARP reply payloads as an operation code. Next field is dedicated to source MAC Address following with source IP address. A source is a machine which initiates ARP communication. It follows with destination MAC and IP address.

2.4.2 Threat Model

Figure 2.22 of ARP frame infers that packet is not containing any field to ensure integrity and confidentiality. It also means that there is a chance to exploit ARP based communication. ARP packet based threats are ARP Poison and ARP flooding. In this section, we go to deep understanding of these threats and manifestation of the attack.

Communication parties maintain MAC table as ARP list. Whenever communication happens, either MAC entry is available in ARP list, or ARP request is generated. In response, ARP reply carries desired MAC entry. As a formal description, ARP system S can be described as a collection of ARP list ARP_L and a transition function δ. ARP list ARP_L is a set of ARP entries. The transition function δ is a mapping from ARP_L and ARP_E to ARP_L.

```
-------------------------------------------------------------------------------
|<--Byte-->|<--Byte-->|<--Byte-->|<--Byte-->|<--Byte-->|<--Byte-->|
-------------------------------------------------------------------------------
|ETHERNET FRAME HEADER:   ---Destination MAC Address--- |
-------------------------------------------------------------------------------
|ETHERNET FRAME HEADER:   -----Source MAC Address-----   |
-------------------------------------------------------------------------------
|EFH: -Frame Type-  | 0806 for arp req/rep  and 0800 for ip(tcp/udp)
---------------------
|AH:    --H/W Type--  | 0001 for ethernet
---------------------
|AH: -Protocol Type- | 0800 for IPv4
---------------------
|AH: -HAL-| hardware address length 06 for ethernet nic address
----------
|AH: -PAL-| protocol address length 04 for IPv4 address
---------------------
|AH: Operation Code | 0001 for ARP request and 0002 for ARP reply
--------------------------------------------------------------
|ARP HEADER:          --------Source MAC Address--------          |
--------------------------------------------------------------
|ARP HEADER:     --Source IP Address--    |
--------------------------------------------------------------
|ARP HEADER:        -------Destination MAC Address--------        |
--------------------------------------------------------------
|ARP HEADER:   -Destination IP Address-  |
-----------------------------------------
```

Figure 2.22: An ARP Frame

Formal Description

- System S = $\{ARP_L,\ \delta\}$
- ARP list $ARP_L = \{ARP_{E1},\ ARP_{E2} \ldots ARP_{En}\}$
- $ARP_{Ei} = i^{th}$ ARP entry in ARP_L
- Transition_function $\delta : ARP_L \ \times \ ARP_E \ \rightarrow \ ARP_L$

Transition function can be interpreted as a union of new ARP entry ARP_E with existing ARP list ARP_L.

Transition_function

- $\delta(ARP_L,\ ARP_E) \rightarrow ARP_L \ \cup \ ARP_E$

ARP Poison Attack:

ARP entries are not validated against authentication and integrity hence vulnerable to Poison attack. A poisonous ARP packet is treated the same as a benign ARP packet. A poisonous ARP entry $ARP_{E(p)}$ is also added to ARP list ARP_L. Afterward, any communication uses false IP to MAC binding.

ARP Poison

- $\delta(ARP_L, ARP_{E(p)}) \rightarrow ARP_L \cup ARP_{E(p)}$

ARP Flooding Attack: If a single poisonous ARP packet can be added as ARP entry in ARP list, then multiple ARP entries can also be added. An attacker may send a massive number of fake ARP packets to a targeted host. This kind of attack commonly known as ARP flooding attack. Or we can say like if it is true that every ARP entry ARP_E is added to ARP list ARP_L then it must also true that there will not be a single case for which a poisonous ARP entry $ARP_{E(p)}$ is denied to added in the list. The speed of the attack probe is restricted by underlying network capabilities, target host process capabilities, and the attacker's programming skill.

ARP Flood

- If it is true that $\forall[\delta(ARP_L, ARP_E) \rightarrow ARP_L \cup ARP_E]$

- Then it must be true that $\nexists[\delta(ARP_L, ARP_{E(p)}) \nrightarrow ARP_L \cup ARP_{E(p)}]$

Manifestation of Attacks

ARP packet based attacks are simple to perform but have significant effects. Most prominent effects are Man-in-the-middle (MitM) attack and resource wastage. ARP Poison for a single machine drop packets.

Man-in-the-middle attack: Two machines poisoned at the same time to eavesdrop the traffic. An attacker set his machine as the middle of two targeted machine. In the Section 2.4.3, exact implementation for the same is demonstrated. With this attack, unencrypted data can be looked by the attacker.

Resource Wastage: ARP flooding attack installs a huge number of ARP entries in the target machine. Huge ARP entry needs more time to be looked. This also overruns the capacity of a MAC table in the target machine. After a hard limit, the operating system removes certain entries. So next time any benign ARP entry is uninstalled due to the attack, infers to new ARP request from the machine or a reply from another machine. Extra processing, extra packets certainly have a cost associated in terms of CPU cycle and network bandwidth.

Hinder Communication: An attacker can poison the victim to stop the communication with pear. Suppose A and B need to communicate. The attacker M poison A's MAC table to indicate a random MAC address as of B's. So whenever A generate traffic for B, traffic will move toward random MAC not towards B. Because B will not accept traffic generated for other than it.

2.4.3 Attacker's Approach

ARP table is used by each host to fill destination MAC address in the frame. Figure 2.22 specifies details of an ARP frame. If someone corrupts ARP table, it leads to a problem like a Man in the Middle(MitM) or packet drop. ARP Poison is usually used to perform MitM because if packets are dropped using poisoned ARP table, then upper layer will re-initiate ARP request. In this section, we will explore the feasibility to perform ARP poison. To create any ARP request or reply, we have to follow the details given in Figure 2.22. By creating a frame with required specification and putting it on a network interface helps the attacker to perform the attack.

Attacker who wants to perform an ARP Poison attack can think of generating a false ARP reply to poison the ARP table but will not be allowed to do so because whenever an ARP request is generated, a neighbor entry is also created at the host. A neighbor entry is a data structure which holds target IP address of sent ARP request. If a host receives an ARP reply will for which no neighbor entry is created, ARP reply will be dropped silently[78]. However, the attacker can wait for some ARP requests and can send crafted ARP reply to it. Now there will two ARP replies, one from the attacker and other from genuine node, ARP reply received later will be fixed. Now in this case attacker have to wait to poison the ARP table.

The attacker performs MitM only if some ARP requests are generated. Like if malicious host M have to be in the middle of host A and host B, as shown in Figure 2.23. It has to poison both A's and B's ARP table in displayed manner. For this, host M have to wait for A's ARP request for B and B's ARP request for A.

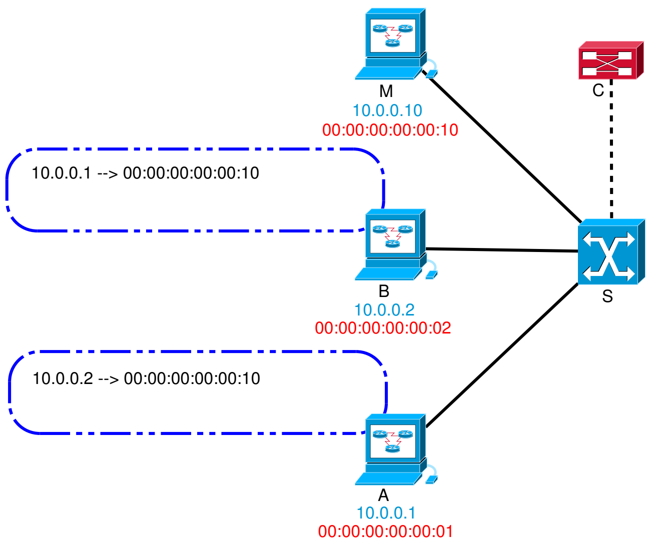


Figure 2.23: ARP tables in Man-in-the-Middle attack

Instead focusing on ARP reply, an attacker concentrates on ARP request. If any host receives an ARP request for its own IP address, its ARP table gets updated with the details received like source IP, source MAC, with no authentication. Suppose in Figure 2.23 attacker M have to poison A's and B's ARP table. The attacker will craft two ARP request. In the first request, all fields are arranged like "who-has 10.0.0.2 tell 10.0.0.1" with setting source MAC address to 00: 00: 00: 00: 00: 10. Second ARP request looks like "who-has 10.0.0.1 tell 10.0.0.2" with source MAC address to 00: 00: 00: 00: 00: 10.

First ARP Request:

ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28

```
0x0000 | ffff ffff ffff 0000 0000 0010 0806 0001
0x0010 | 0800 0604 0001 0000 0000 0010 0a00 0001
0x0020 | 0000 0000 0000 0a00 0002
```

Second ARP Request:

ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28

```
0x0000 | ffff ffff ffff 0000 0000 0010 0806 0001
0x0010 | 0800 0604 0001 0000 0000 0010 0a00 0002
0x0020 | 0000 0000 0000 0a00 0001
```

The first column of above representation specifies the offset. ARP request is represented in hexadecimal format. ARP request is broadcasted hence first six bytes are all 'f' followed by source MAC address which is 00:00:00:00:00:10 (MAC of attacker machine). More details or each byte can deduced from Figure 2.22. By putting first and second ARP request byte streams on network interface of machine M, performs ARP poison for host A and B.

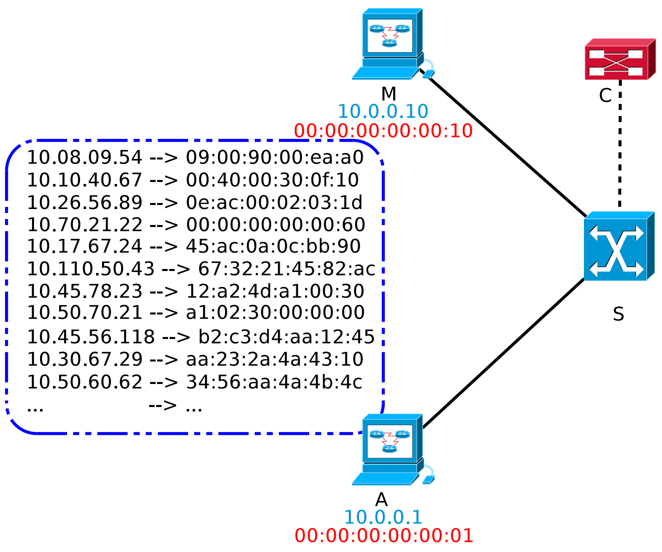
We have already discussed how ARP table could be poisoned. We will look further in this section how attacker can perform ARP flooding using aforementioned technique. ARP table is always referred whenever a new packet is sent to the target. ARP flooding is a situation in which ARP table is flooded with some random information. Same is illustrated in Figure 2.24.

To perform this attack, attacker uses same trick as discussed for ARP poison. But the only difference is that Now he will write a crafted packet with the desired randomness and burst such packets on interface. Randomness can be inserted from 6^{th} byte to 11^{th} byte (0 is first) and same should be repeated from 22^{nd} byte to 27^{th} byte. Next bytes used for randomness are 28-31.

ARP Request for ARP flood:

ARP, Request who-has 10.0.0.1 tell 10.45.78.23, length 28

```
0x0000 | ffff ffff ffff 12a2 4da1 0030 0806 0001
0x0010 | 0800 0604 0001 12a2 4da1 0030 0a2d 4e17
0x0020 | 0000 0000 0000 0a00 0001
```



```
10.08.09.54 --> 09:00:90:00:ea:a0
10.10.40.67 --> 00:40:00:30:0f:10
10.26.56.89 --> 0e:ac:00:02:03:1d
10.70.21.22 --> 00:00:00:00:00:60
10.17.67.24 --> 45:ac:0a:0c:bb:90
10.110.50.43 --> 67:32:21:45:82:ac
10.45.78.23 --> 12:a2:4d:a1:00:30
10.50.70.21 --> a1:02:30:00:00:00
10.45.56.118 --> b2:c3:d4:aa:12:45
10.30.67.29 --> aa:23:2a:4a:43:10
10.50.60.62 --> 34:56:aa:4a:4b:4c
...             --> ...
```

Figure 2.24: ARP table in ARP flooding attack

Whenever a host's ARP table is filled with random information, it needs more time to process a genuine request and resources will be busy in fulfilling unwanted ARP requests. There is size imposed on ARP table same as gc_thresh1 , gc_thresh2 , gc_thresh3 on linux and then garbage collector will work upon[78]. The garbage collection is a process to free unused data structures.

2.4.4 Orthogonal Research

Traditional solutions for ARP based security threats start with putting static entries in ARP cache. In some approaches, an IP to MAC pairing will be stored using Simple Network Management Protocol (SNMP) or DHCP. ARP request/reply is verified against those stored ones. Cryptographic solutions are also proposed to address the issue. In this section, we discusses few of these approaches.

Arpwatch[79] tool keeps the log for IP/MAC pairings and generates email alert if pairing is changed along with time stamp. Email analysis is totally dependent on the ability of network administrator to identify which pairing change will use to perform an attack. Carnut et. al.[80] proposed Request-Reply Imbalance Algorithm for SNMP capable switches. The article uses statistics on each port to calculate imbalances in arriving ARP requests and replies. To calculate and process each port statistics becomes complex computation. Other SNMP based model is proposed by Hsaio et. al.[81] using data mining techniques to detect ARP based security threats. Goyal et. al.[82] proposed a solution based on digital signatures and one-time password in the ARP reply. Secure Address Resolution Protocol[83] and Ticket-based ARP[84] are other cryptography-based approaches. Involving digital signature in the process increases overhead. Cisco Dynamic ARP Inspection uses valid IP to MAC address binding. Each ARP packet will be forwarded if follows accurate mapping. Trusted database provides this mapping built during DHCP service[85]. Process rely on DHCP service, so what if static ip addresses are provided. Nam et. al.[86] proposed MitM-resistant address protocol. In this protocol, long-term IP/MAC mapping table is introduced along with original ARP cache. In long-term table, timer is associated with each entry. A new ARP request will send whenever those timers expire.

Existing solutions of ARP-based threats are either use pre-stored IP/MAC bindings or cryptographic approaches. In first approach, IP/MAC bindings are created using DHCP or SNMP protocols. Therefore, in large networks the number of these bindings are enormous hence, the lookup time increases, and also leads to resources wastage. The cryptographic approaches require additional computational overhead in complex cryptographic algorithms. Apart from this, the traditional methods of security are directly applicable but not taking the benefits from the separation of control plane and data plane. In SDN, control plane (controller) is a programmable plane having a global view of the network.

SDN offers separation of control plane and data plane, so resource intensive computation is responsibility of controller. Controller runs along with its control module on any high capacity general purpose computer. Dynamic programming helps to take countermeasure automatically, if any of such attacks is detected. Some recent SDN based solution are also discussed. Ding et. al.[87] calculates the probability of a host being an attacker. ARP packet is processed for feature extraction. These features are used to calculate some probability to match against statistically achieved threshold. Mohammad et. al.[88] creates the database using DHCP reply and use this database to verify any ARP Reply. ARP Poison can happen due to ARP request as well. SPHINX[65] store MAC/IP bindings using

PACKET_IN event. This stored pair will use to validate any incoming connection. If any deviation from these permissible bindings are observed, this controller extension will generate alarm.

Alharbi et. al.[89] suggests a NAT alike SARP_NAT to detect ARP Poison attack. In SARP_NAT each ARP request and reply visit to the controller to be sanitized. The source protocol address and source hardware address are changed to fixed one in ARP request. Once ARP request reaches to end host, end host store safe addresses in MAC table. In response, the end host generates ARP reply for safe addresses. The controller again intervenes to translate the addresses. The solution is assuming that ARP Poison can be done using ARP reply. A malicious ARP request also poisons the ARP proxy table at the controller.

2.4.5 Inferences

After evaluating the existing literature for ARP based threats in both traditional networking and software-defined networking, the inferences are as follows:

- Current deployed solutions for ARP based threat are insecure, costly.
- Pre-stored IP/MAC binding cannot be a preferable solution.
- Cryptography based solution hinder the performance.

2.5 Summary

SDN uses programmability and openness to produce an optimal result not only in the new scenario but for traditional threats as well. Topology discovery is an essential part of the SDN. Various components such as the switch, the link, and the host must discover accurately to provide beautiful solutions in SDN. This chapter focuses on the link discovery and data link layer based host discovery. Current deployments, possible threats, and state of the art research are discussed to improve the understanding. In the view of listed inferences, solutions for LLDP threats and ARP threats are needed.

As this chapter provides adequate evidence that present deployed and proposed link discovery is not secure against LLDP poison, Replay and flooding attack. In the next chapter, a preventive solution is discussed for link discovery.

Chapter 3

A Preventive Solution to Secure Link Discovery

The previous chapter figures out that possible causes for Link Layer Discovery Protocol (LLDP) threats (LLDP Poison, LLDP Flooding, and LLDP Replay) are the lack of authentication and integrity mechanism with a static payload. This chapter introduces a token based preventive approach TILAK to secure the link discovery process[15]. In the discussed approach, a periodic token is provided by the controller to each LLDP packet. The same token is used to validate any receiving LLDP packet. The implementation results for TILAK confirm that it covers targeted threats with lower resource penalty.

The data plane switches receive OpenFlow messages and update its forwarding table or respond to the queries. A forwarding table consists of a set of match attributes, a set of actions, and some statistical data. Every packet arrived at the switch is compared with a set of matching attributes of every flow entry. A match satisfying flow entry provides an action to perform, and switch follows the instruction to drop or pass it on a port. A packet with unmatched attributes makes the switch to conduct a consultation with the controller or drop the packet as per some default rules. Links in topology are identified using LLDP messages [50]. A LLDP message consists of few Type-Length-Values (TLVs). The controller initiates the LLDP communication by generating LLDP packets for every port on each switch. The switch forwards these packets to the designated port(s). If any switch receives an LLDP packet on one of its port, it forwards the packet to controller. Once a controller has LLDP packet content which confirms the source and event information from a switch, it extracts the destination information. This process establishes a link between source and destination switches, which helps topology aware applications. To achieve the view in the existing implementation, LLDP packets created by different controllers are either not using any security scheme for ensuring integrity or authentication or it carries a hash field which can not be considered secure as discussed in previous chapter.

To detect LLDP attacks, several attempts were made in the state-of-the-art survey [50]. TopoGuard [53] checks integrity and origin of LLDP packets with a separate Hash-based Message Authentication Code (HMAC) TLV, but it suffers from LLDP Replay attack. This technique assumes that the malicious host has to send host traffic, i.e., Domain Name System (DNS), and Address Resolution Protocol (ARP). SPHINX [65] generates flow graphs to validate with static switch-port bindings to detect LLDP Poison attack, and OFDP_HMAC [66] uses HMAC authentication to ensure both integrity and authentication. These approaches address LLDP Poison and LLDP Replay attacks, but not the LLDP Flooding attack. Additionally, all methods which are using HMAC always has a significant resource penalty in both generation and verification processes. TILAK prevents all attacks with negative resource penalty.

3.1 Foundation

This section explains the overall direction to build a solution. An attacker may exploit vulnerabilities in Link Layer Discovery Protocol (LLDP) packet processing to launch LLDP Poison, LLDP Replay, and LLDP Flooding attacks. In LLDP Poison attack, an attacker crafts false LLDP packets to poison controller's global view. The LLDP packets are also received to attached hosts, some of these recorded LLDP packets may be used to poison controller view later. This kind of attack is termed as LLDP Replay attack. In LLDP Flooding attack, a large number of LLDP packets are sent to the controller to waste its resources. Identification of possible causes which let the attack happen, will help to build a robust solution. Possible causes are authentication check failure, integrity check failure, LLDP broadcast and reuse of static LLDP packets for the entire period of link discovery.

Authentication Failure(AF): Authentication is a process to confirm the origin of the communication. In link discovery, the controller and switches don't have adequate knowledge to judge the origin of received LLDP packet. The received packet also don't carry any information to prove the source. Hence, when a switch receives LLDP packets in any non-OpenFlow port, it has to accept and forward to the controller. Once the controller receives LLDP packet, it also has to accept for further processing without ensuring the origin.

Integrity Failure(IF): Integrity is a process to examine the tempering in message since origin. In link discovery, an LLDP packet is generated at the controller pass through two switches and consumed at controller again. The controller has no way to ensure the integrity of the received LLDP packet. In the solution to this problem, various hash-based approaches are used. Few solutions encourage a static hash or a universal hash for each LLDP packet. In other solution, a unique hash for different LLDP packet is used.

LLDP Broadcast(LB): In the link discovery, LLDP packets are sent to every port on each switch. Few ports on any switch may be attached to hosts. For those ports, LLDP packets are waste of computation resource, bandwidth resource and vulnerable to LLDP packet attack. If any host receives an LLDP packet, it gets more information about packet payload. And the same can be used as payload for LLDP Poison, Flooding and Replay attacks.

Static Packets(SP): Mostly the controllers generate LLDP payload once and use it for the future link discoveries. It may lead to an attack, if anyhow attacker guesses or brute-force an LLDP packet, for a future attack such information may be utilized. But if packets are constantly changing then it is less probable to get attacked.

To target these, this chapter introduces *TILAK*. The word 'TILAK' is a re-

Proposal	Authentication	Integrity	Limit LLDP	Dynamic Content
TILAK	**y**		**y**	**y**

Table 3.1: TILAK with different threat causes

production from the Hindi language i.e. "तिलक" which generally is used for authentication. Table 3.1 gives brief of working domain for TILAK. TILAK ensures authentication with limited dynamic LLDP packets. TILAK uses randomness in LLDP frame for source authentication. For each iteration, a flow entry is installed to allow LLDP packets with randomness. TILAK sends LLDP packets only at eligible ports which reduces the overall overhead, even making it negative. The negative overhead means that after applying security extension to original implementation, the original overhead is reduced. To reduce the probability of a successful attack, TILAK further removes unidirectional links which are possibly created by attackers. Emulation results were shown that TILAK successfully prevents LLDP Poison, LLDP Flooding, and LLDP Replay attacks. Additionally, we present the proof of concept to gain the confidence for TILAK's methodology.

Assumptions:

Designing a universal solution for any problem is always challenging. The same applies for TILAK also. Few assumptions are identified which ensure proper working of solutions.

- The controller with running applications is malware free.
- The switches work based on OpenFlow specifications.
- The switches are malware free.

3.2 TILAK Design

Packets are generated by the malicious hosts which carry the required specifications to generate the poisoned view. Threats related to the OpenFlow discovery protocol

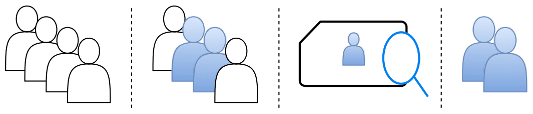


Figure 3.1: Token-based authentication

(e.g., LLDP Poison, LLDP Replay, and LLDP Flooding) are due to the source authentication and integrity failures. In source authentication problem, the process is unable to authenticate the source of a received packet. In integrity problem, the process is unable to verify the integrity of the received packet. Some controllers like OpenDayLight, Floodlight, and HPE-VAN use hash value to provide integrity but they fail to do so as we discussed in the previous chapter. In this section, we discuss a solution which is based on source authentication. Figure 3.1 gives an overall idea of our proposed token-based authentication technique. The tokens are source identifiers which are generated and assigned to packets. The packets are verified with token upon receiving. The valid token holding packets are considered as genuine. It is because the token is only generated by a particular process which confirms the source of the packet. The source authentication solution also relaxes the integrity problem, which we will discuss in Section 3.3.4. TILAK[15] is a approach in which random token is generated, assigned, and verified to solve source authentication problem.

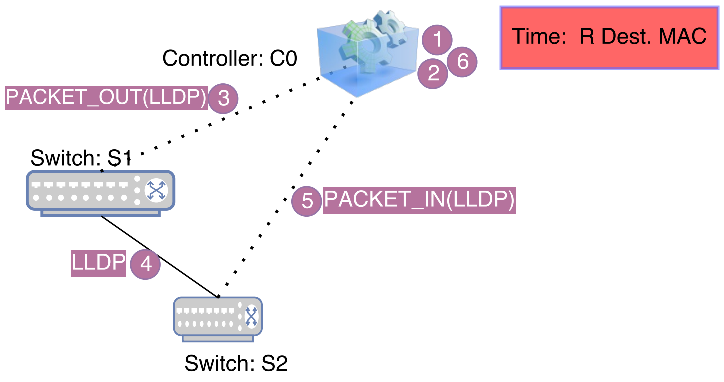


Figure 3.2: LLDP packets life-cycle in TILAK

In Figure 3.2, a detailed overview of link discovery process while using TILAK is illustrated. Initially, random destination Media Access Control (MAC) address is generated, which is used in Link Layer Discovery Protocol (LLDP) Ethernet frame as shown in Figure 3.3. In each switch, a flow entry is installed with same random MAC address so that LLDP packets can be redirected to the controller. These packets are generated by the controller, traverse a link and reach back to the controller.

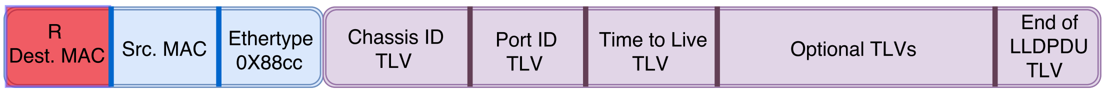


Figure 3.3: LLDP packet format for TILAK

TILAK link discovery process

Step 1: Controller (say C0) generates a random destination MAC.
Step 2: C0 installs a flow entry on each switch to allow LLDP packet with same random destination MAC.
Step 3: C0 generates LLDP packet for each allowed port on each switch.
Step 4: Switch S1 receive one such LLDP packet to transmit on a port.
Step 5: Switch S2 receive this packet and pass it to controller due to already installed flow entry.
Step 6: C0 creates a link between S1 and S2.

Available Software Defined Networking (SDN) controllers follow nearly the same sequence to discover the topology. Any destination MAC address could be chosen which is used for building LLDP packet and installing flow entry. Building LLDP packet is done once for ever and the LLDP packets are sent subsequently as shown in Figure 3.4. In this process, the problem lies in the creation of static packets. The attacker could quickly discover that packet, as each host also receives LLDP packets. Hence, the attacker generates similar packet as per need and can use for attack as discussed in the previous chapter. Thus, TILAK uses two separate threads for sending the packets and creating the links.

TILAK change the original discovery process slightly to solve source authentication problem. The LLDP packets build periodically, so after each fixed time interval the LLDP packets with new random MAC are generated. To forward newly created packets, the flow entry installation is done as per random MAC address. We can refer to Figure 3.5 for exact event sequence in TILAK. The TILAK assumes that if at a specified period, the LLDP packet flow entry is changed, then less chance for an attacker to poison the topology will exist. Topology discovery could always be poisoned if there is a flow entry in the switch, however, if switch

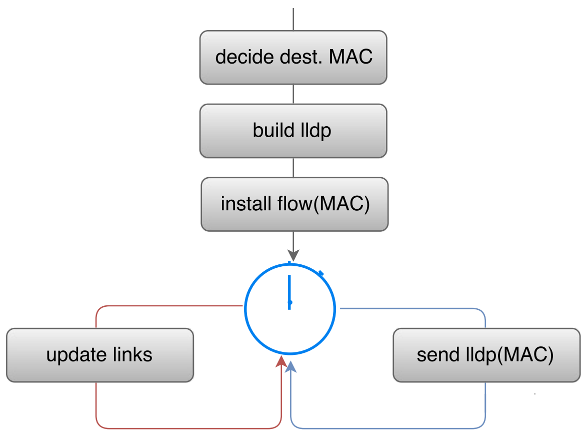


Figure 3.4: Original event sequence in topology discovery

is instructed to change its LLDP flow entry after each period, then there are lesser chances for an attacker to guess the flow entry and build LLDP packet accordingly.

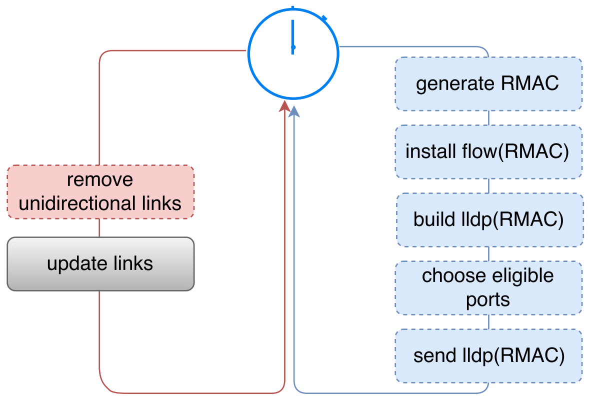


Figure 3.5: TILAK event sequence in topology discovery

TILAK also addresses the issue from the problem of generating a LLDP packet for each port on each switch. The problem also means that the host attached to a switch will also receive the LLDP packet. Hence if an attacker remains on any of such host, it will know the LLDP packet and also about random MAC. This is because the discovery cycle is periodic and usually for five seconds, hence the same flow entry can be used to forward LLDP packet to the controller. The attacker gets enough time to use the received LLDP packet to build malicious LLDP packet and then launch the attack. Therefore, instead of sending LLDP packet to each port, TILAK choose only the ports on which the switches are connected to other switches. If ports are restricted, then no LLDP packet will reach to any of host, and it will prevent an attacker to get the random destination MAC address.

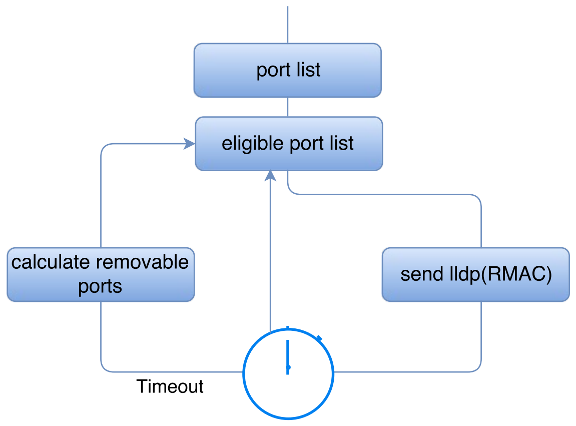


Figure 3.6: TILAK: Marking of removable ports

TILAK creates an eligible port list as shown in Figure 3.6. Initially, all ports are eligible, so the LLDP packet is sent on all ports. After some period, it is checked that if some ports which are not receiving any LLDP packet are liable to removal from the eligible port list. On next, in each discovery cycle, the LLDP packets are sent to only eligible ports. In a nutshell, the steps for creating the eligible port list are as follows.

Step sequence in TILAK for reducing number of ports
Step 3a: At beginning, controller creates a table for eligible port list $< switch, port1, port2.. >$ ***Step 3b:*** Controller generates the LLDP packet for each port on each switch. ***Step 3c:*** Controller removes ports from the table which are not receiving the LLDP packet or no other switch receives LLDP packet sent on those port. ***Step 3d:*** goto Step 4

What if any malicious host is activated even before the controller starts, will TILAK fail ? It is because the early awake host will also receive the initial LLDP packet which can be used to launch the attack. If it has to vary with less probability, TILAK target few of the suspicious links, so even if an attacker awakes at the start of the process, a false link is created with the help of a known random destination MAC address. But if the link from reverse direction is not established in a same fixed time interval, then the forward link is liable to get removed. For instance, in Figure 2.19, if a malicious host attached to Switch $S3$ creates a false link with $S4$. However, if the link in reverse direction is not formed in the fixed interval of time then the forward link will also be removed. After removing the link, the port on which such information is received will be removed from the eligible port list.

Even after all the above improvements in TILAK, one problem still exists. For instance, if two malicious hosts are booted before the controller. These will get the LLDP packets from the starting, and both of them can involve in a collaborative attack. In this case, both the links, i.e., forward and reverse, can be established, hence no removal will follow. However, doing so, the probability is very low that two hosts have the idea when the controller will start. In such scenario, we believe that further investigation is needed.

Any packet can only move to controller from a switch, if there is a matching flow entry installed in the switch. In case the TILAK installs a new flow entry after a fixed time interval, the attacker must know about the applied randomness to get success. Section 3.3.4 demonstrates that the guessing randomness will be not so easy task. Hence, TILAK prevents LLDP Poison and LLDP Replay attacks. The same is also true for LLDP Flooding attack. However, a reader may argue that any unmatched packet will always be sent to the controller. But, in any production network, a network administrator has full knowledge for all possible flows. Thus, in the production network, usually "Drop all others" rule can be applied for unknown traffic.

3.3 Implementation

TILAK prevents Link Layer Discovery Protocol(LLDP) packets based threats to secure link discovery process. This section discusses the implementation details and used resources. Further, obtained results are examined and discussion on correctness analysis is made.

3.3.1 Experimental Setup

To validate the implementation, three different topologies are used. Figure 3.7, 3.8, and 3.9 depicts the different topologies, where Table 3.2 presents related statistics. In different topologies, some switches are fairly large to validate the proposal. The number of links also include links between the switch and the host apart from a switch to switch.

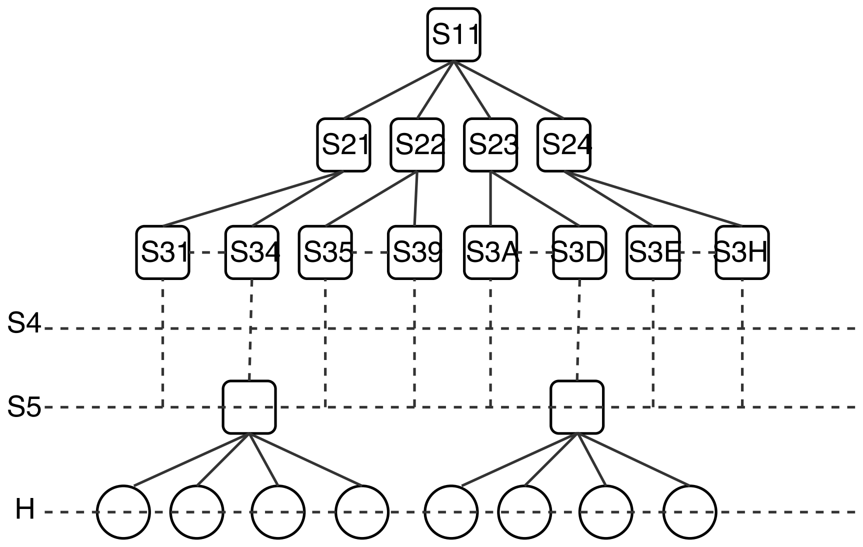


Figure 3.7: Topology 1: Tree topology with 4 level of depth and fan-out 4

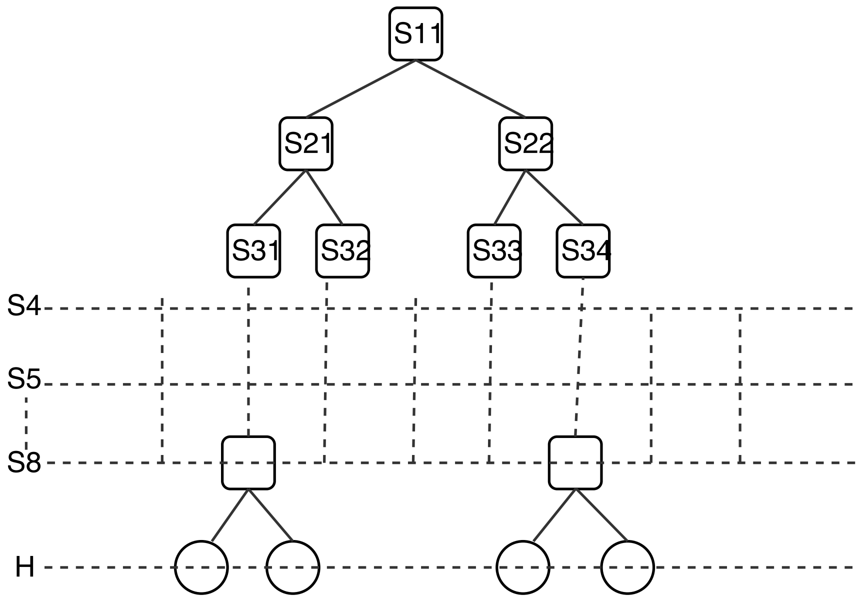


Figure 3.8: Topology 2: Tree topology with 7 level of depth and fan-out 2

In Figure 3.7, five layers of switches are shown. Each switch is connected to five switches except root switch. For example, S31 - S34 represents four switches in layer 3 and so on. S4 and S5 switch in layer four and five respectively. H represents hosts connected to layer five switches. In Figure 3.8, each switch connected with three switches except root switch. Here eight layers of switches(total 127 switches) form a tree topology. Figure 3.9 is a Fat tree topology which is typically used in data centers. Here switches are arranged in three-layer arrangement namely core, aggregate, and edge switch. In the experiments, two hosts per edge switch are chosen.

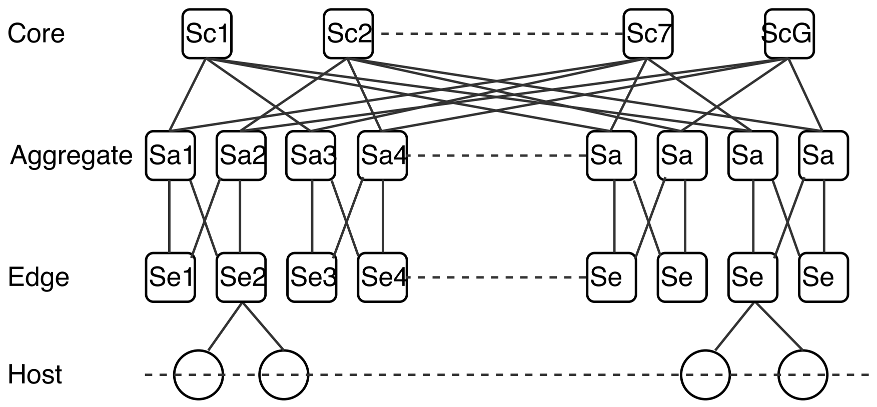


Figure 3.9: Topology 3: Fat tree

Topology	Switch	Link	Port	Host
Tree,4,4	85	340	424	256
Tree,7,2	127	254	380	128
Fat tree	80	384	705	64

Table 3.2: Number of Switches, Links, Ports and Hosts

Table 3.3 provides the details of the execution environment for TILAK. It includes Operating system and testbed details. We tested the implementation of TILAK in the Mininet [57]. We have access to the private data-center to set up the victim and attacker.

3.3.2 Performance Metrics

The performance metric is decided to verify implementation for correctness. In this section, various metrics, the importance, and obtained methodology is discussed.

Resource	Configuration
Testbed	Mininet(emulation)
Victim/Attacker OS	UBUNTU 16.04$LTS(64bit)^*$
Victim configuration	7 CPUs and 16GB
Attacker configuration	7 CPUs and 16GB(private datacenter)
Controller	Ryu
Attack traffic	150000 Packet/Sec
Software switch	OpenVSwitch(2.5.0)
Network	1 Gbps

Table 3.3: TILAK execution environment

These metrics are further used to understand the obtained results.

Resource Consumption: To run any algorithm computational and bandwidth resources are used. In TILAK, the computational resources are used to compare with original deployed implementation. The computational resources are the time taken and the memory required to run the algorithm. In the following subsection, time is used for comparison. Algorithms for the same task can be compared with the time taken. Lesser the time, better the algorithm. In Ryu controller, dedicated processes are used to LLDP packet sending and link discovery. In a continuous process, measurement of resource consumption is always dependent on implementation. Here the original switches.py have two separate threads to maintain the LLDP sending and the link discovery. Some additional code segment is also executed, but it is restricted to once. Hence, two threads are targeted, and each thread is invoking an infinite loop which runs periodically. Average time taken by these loops in one iteration is used as the measurement. After that, a sum of averaged values is used to generate final resource consumption.

LLDP Packet Construction Time: LLDP packets are generated and used in the link discovery. In TILAK, an LLDP packet contains a random Media Access Control (MAC) address. Random MAC generation also takes time. By LLDP packet construction time, variance in delay can be obtained. It also helps to identify that modification in LLDP packet is performed in allowable time.

LLDP Packet Verification Time: Once an LLDP packet received at the controller, it must be verified before adding to the link database. This parameter helps to identify the effect due to modification performed in an LLDP packet.

Once the packet is received, the total time taken is used as this metric.

Initial overhead: In current deployment, flow entry installation and LLDP packet creation are performed once and at the initial stage. Once a packet is created, the same packet is used for further iteration. Total time used in such activities is used as the metric. In TILAK no such activities are performed.

3.3.3 Results and Discussion

To validate TILAK proposal, we present the result analysis in this section. The SDN network could consist of multiple controllers. On each controller, different link discovery implementations exist. We choose Ryu, a python based controller. In this controller, switches.py file is dedicated to the link and host discovery process. We compare TILAK (switchesTilak.py) with the original implementation (switches.py). It will give a resource penalty analysis. Experiments are conducted with three topologies with and without the solution. Experiments on Mininet [57] are emulation only, and these are widely accepted in SDN research community.

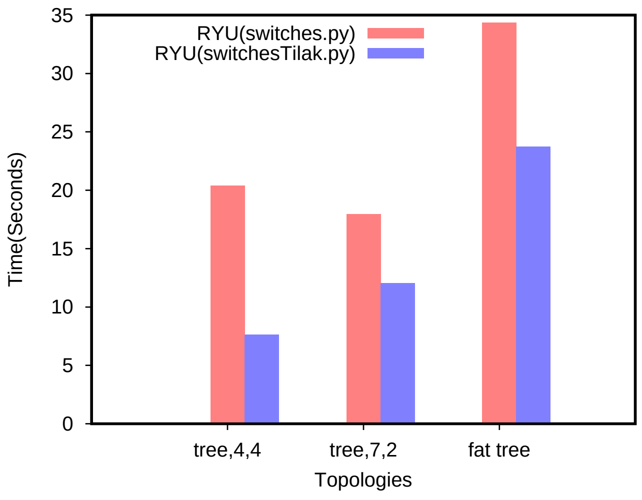


Figure 3.10: Performance comparison

Figure 3.10 shows the resource consumption in different situations. In Figure 3.10, it is visible that resource consumption by the original implementation is higher then TILAK irrespective of the topology.

Table 3.4 shows average resource consumption in different situations. Here one may argue that the shown time is relatively large while few controllers have

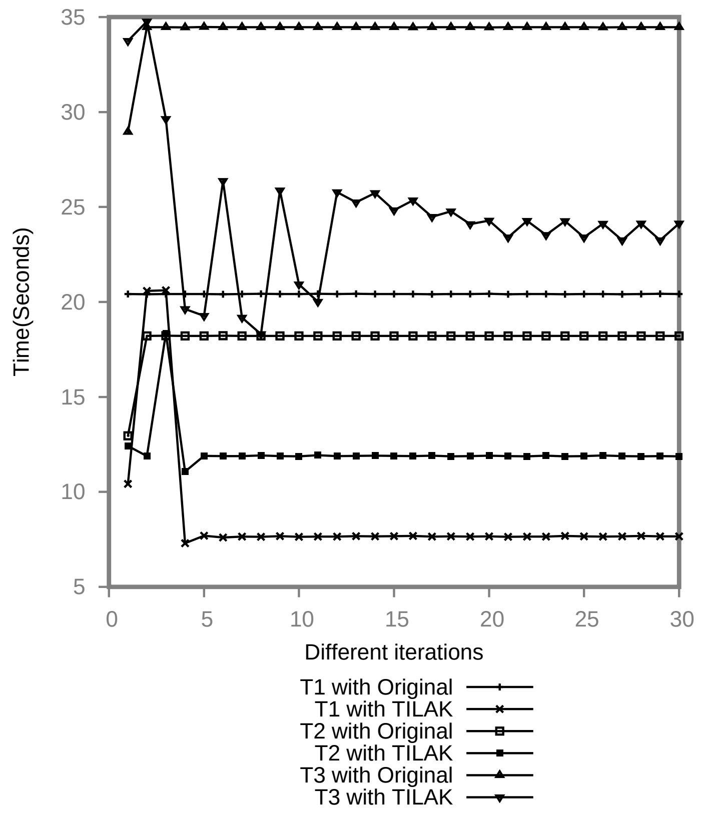


Figure 3.11: Performance stabilization graph in the different iteration

Topology	Ryu (switches.py)	Ryu (switchesTilak.py)
"tree,4,4"	20.42	7.63
"tree,7,2"	17.96	12.05
"fat tree"	34.37	23.74

Table 3.4: Averaged resource consumption in second(s)

LLDP cycle time of five seconds. Our work claims that even without a solution, i.e., switches.py, it took 34.3 sec in a Fat tree. These are emulation result, and we strongly believe that the original implementations will take decidedly less in the actual hardware testbed. One thing to notice here is that in every topology, TILAK is taking less time when comparing with switches.py.

Figure 3.11 gives the idea that how such lower resource consumption is achieved with TILAK. It illustrates resource consumption in first few iterations. In this, three different information pairs are shown, one for each topology. We have to compare the results in each pair which has resource consumption for both with and without the solution. Figure 3.10 is drawn as per these average values. Initially, the solution in each topology is taking the same time as the time taken in without solution, but after few iterations, it gets lower as compared to without solution scenario.

The total time taken to complete an iteration is the sum of the time to install flow entry in each switch, time to send LLDP packets to all eligible ports, and time to create link after receiving the LLDP packets. In TILAK, we reduce the number of eligible ports after T_{eport} time interval. So as per Table 3.2 in Fat tree topology, initially, 705 ports are discovered for which LLDP packets are created and sent. But after T_{eport}, 64 ports are declared not eligible. Thus average time taken in each subsequent iteration is reduced accordingly. In Figure 3.11, we can see that in Fat tree implementations, the switches.py is taking near to thirty five seconds constantly. But in switchesTilak.py, initially the time taken to complete the iteration reaches to thirty five seconds and then it drops to twenty five seconds.

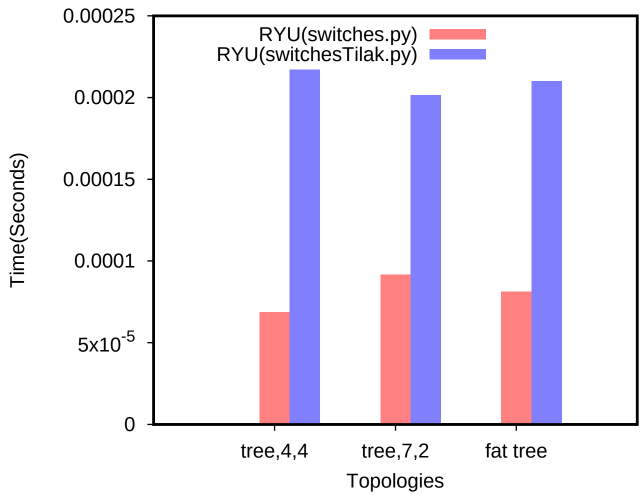


Figure 3.12: LLDP packet construction time comparison

Figure 3.12 shows the time taken in the LLDP packet construction. In this, we can easily identify that in both, switches.py and switchesTilak.py, the time taken is almost constant. For instance, the Tree 7,2 topology has taken 0.0001517 seconds in switches.py, while 0.0001778 seconds in switchesTilak.py. Here the given times are entirely dependent on the executing environment, but the critical thing to note is that the time taken by switchesTilak.py is higher then switches.py, and it is due

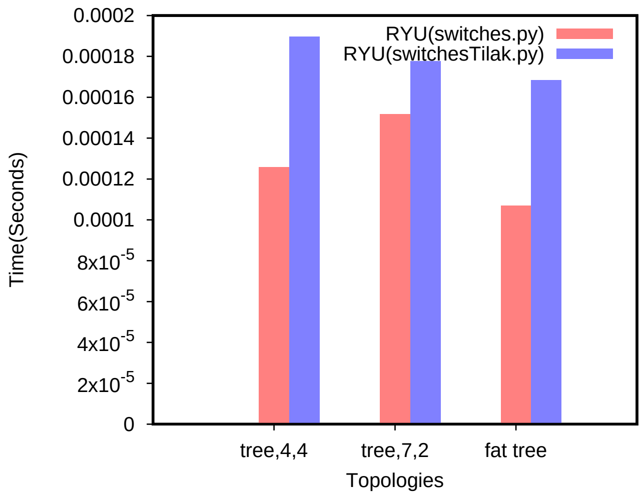


Figure 3.13: LLDP packet verification time comparison

to LLDP packet construction and verification event sequence.

Figure 3.13 represents the time taken in the LLDP packet verification. A controller verifies LLDP packet upon receiving. As we can see that time taken by TILAK is higher then the original discovery process, i.e., switches.py. The reasons for this is LLDP packet verification and construction sequence. As the TILAK does not introduce any complexity in both, the LLDP packet construction and the verification of original link discovery process, then how the time is taken by TILAK is higher than the original solution. The answer lies in the event sequence of LLDP packet construction and verification as it is shown in Figure 3.14. Originally, switches.py construct LLDP packet once, and after that only sending and verification is performed. We know if packets are static then the situation is prone to attack as discussed in Section 2.3. In TILAK, after every fixed interval T_{flow}, a new flow is inserted, and new packets are generated for all the eligible ports. If packet construction is done before any valid load on the controller, then the time taken will be fairly shorter. This is because, after T_{flow}, the newly generated LLDP packets will surely affect the verification process in the form of a time penalty. In Figure 3.14, we can see that in TILAK, both LLDP packet verification and construction is a continuous process.

The TILAK introduces a periodic flow entry installation and LLDP packet creation. In the switches.py, the sum of these two times are considered as initial

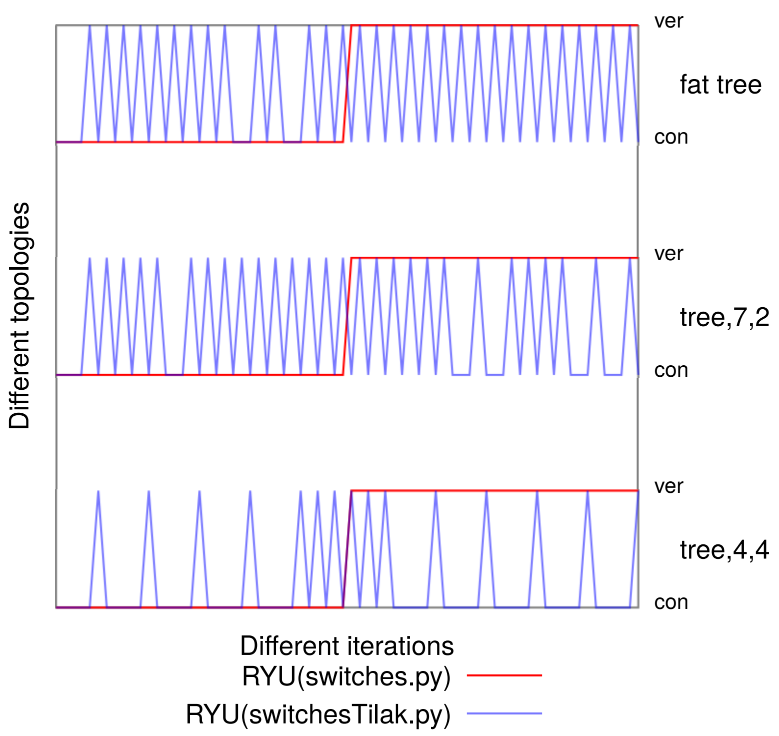


Figure 3.14: LLDP packet construction and verification event sequence

overhead because the LLDP packet creation and flow entry installation are non-periodic processes, and we removed it in TILAK. Figure 3.16 shows such initial overheads across different topologies. The total initial overhead is directly proportional to the number of switches in which a flow entry will be installed and the number of ports for which LLDP packets will be created.

One important aspect is that the average time considered in Figure 3.10 accumulates both, the LLDP packet construction time in Figure 3.12 and the LLDP packet verification time shown in Figure 3.13. As it is depicted in Figure 3.15, the initial overload is only with switches.py, so it counts zero in TILAK overhead. Thus in a nutshell, the Figure 3.10 claims that TILAK performs better than the original link discovery implementation.

Figure 3.16 gives the evidence of preventing LLDP packets flooding. Here, three different packet streams were generated. The first stream is genuine LLDP packets on switches.py, the second stream of packets is on the same implementation but it is a flooded stream. Within a second, 5000 LLDP packets will be reached to the controller. The third stream is again the LLDP packets flood stream but with

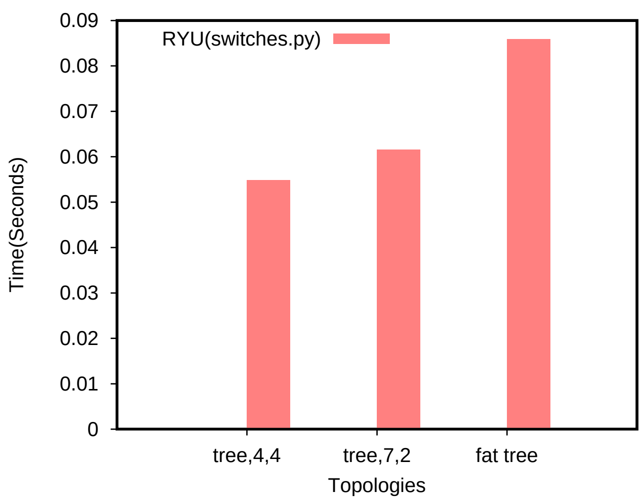


Figure 3.15: Initial overhead in the discovery protocol

switchesTilak.py. In this case, we can see in the Figure 3.16 that only genuine LLDP packets are reached to a controller, and all the forged LLDP packets are blocked itself on the port due to unmatched flow entry for those packets. Discussed examples are for the prevention of LLDP Flooding attack, but other attacks like LLDP Poison and LLDP Replay are also prevented in the same way. It is because the basic prevention approach is common for all. In the graph, we can see that the number of packets from different streams and points at which sudden growth in packets occurs.

The results compared with other controller are always tricky. Specifically, in SDN environment where a number of controllers are available, each implementation may have a different topology. Hence, chosen parameter is controller independent and available to compare, i.e., computational overheads. TopoGuard is generating 4.56% overhead while OFDP_HMAC introduces 8%. TILAK successfully makes a negative overhead, i.e. (-)40.32%. This negative overhead is only possible due to limiting the ports for which LLDP packets are generated and verified. Let's understand this negative computational overhead. Suppose, P program is completed in T CPU-seconds in a given environment. A security extension to P is P_s, which is completed in T_s CPU-seconds. Now we have two scenarios, in first, where T_s is greater than T, i.e., ($T_s > T$). It reflects that implementation of security is added as a resource penalty. In another scenario, where T is greater or

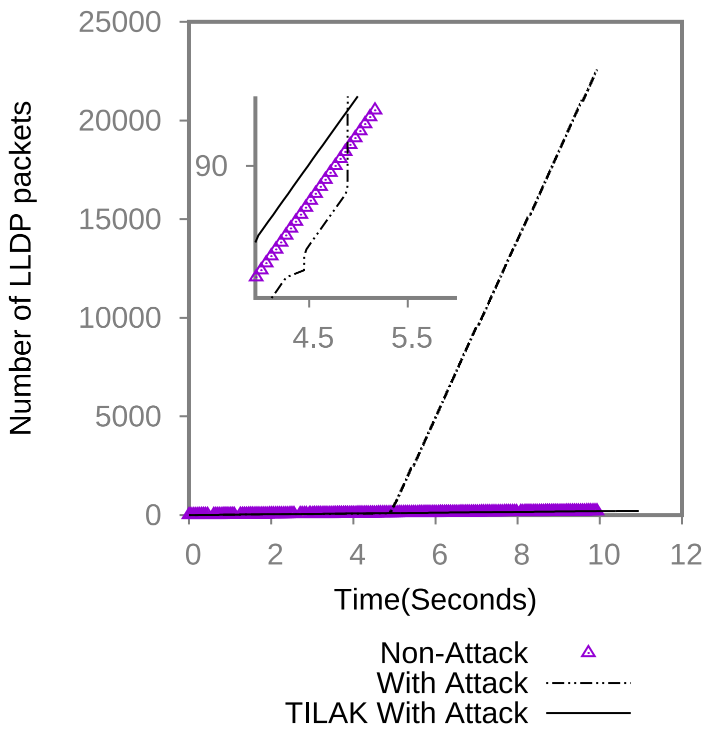


Figure 3.16: Evidence for LLDP Flooding attack prevention

equals to T_s, i.e., $(T \geq T_s)$. It means, in this implementation, improvement of the algorithm is also happened along with security extension and the total required time is less than the time required for original implementations without security.

Table 3.5 gives a comparison of all other available approaches with TILAK. Here, TILAK is the only approach which is preventing all types of LLDP attacks, i.e., LLDP Poison, LLDP Flooding, and LLDP Replay attack, and it also exhibits lowest overheads. This is because TILAK does not broadcast the LLDP packets to all ports after T_{eport} times the new eligible port list is maintained to forwarding LLDP packet. The resource penalty is negative as it is shown in Table 3.4, due to limited LLDP packet generation and sent to the selected ports.

Approach	Authentication	Integrity	LLDP broadcast	Detection	Prevention	Poison	Flooding	Replay
TopoGuard [53]		y	y	y		y		
SPHINX [65]			y	y		y		
OFDP_HMAC [66]	y	y	y	y		y		y
TILAK [15]	y				y	y	y	y

Table 3.5: Comparison in different solutions of LLDP packet based threats

3.3.4 Correctness Analysis

In this section, we establish the correctness of TILAK. The correctness is a measure that system works correctly or produce expected output within pre-specified and erroneous inputs. Let's define the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) for the system to check its correctness. TP is a rate at which genuine LLDP packets successfully allowed to reach to the controller, and TN is the rate at which system successfully stop false LLDP packets. In case of FP, the rate at which false LLDP packets are reached to the controller, and FN is the rate at which genuine LLDP packets are blocked to reach to the controller. Here malicious LLDP packets are LLDP packets generated by an attacker and genuine LLDP packets are generated by the controller.

Definition 1 (Correctness): TILAK is correct if holding following conditions:

- It must discover all the links (i.e., maximize TP and minimize FN),
- It must prevent the LLDP packet based threats (i.e., maximize TN and minimize FP).

For instance, if the topology have links as a set L = $\{l_{12}, l_{21}, l_{23}, l_{32} ....l_{mn}, l_{nm}\}$ and discovered links are represented as a set K = $\{k_{12}, k_{21}, k_{23}, k_{32} ....k_{mn}, k_{nm}\}$, then an essential property must satisfy to prove correctness as $\forall x[x \in L \iff x \in K]$, i.e., all elements which belong to L must also belong to K. The attack only happens if an attacker have the same randomness which is present in the flow entry. If random MAC addresses can be represented as A = $\{a_1, a_2, a_3 ....a_x\}$ and possible packet generated by attackers as B = $\{b_1, b_2, b_3 ....b_y\}$. Then the probability that attacker crafts a packet with identical randomness is $P = \frac{|B|}{|A|}$, and the desired condition for prevention of the attacks is $P \approx 0$.

TP and FN Analysis

The true positive and false negative are constrained with a valid flow entry at each switch. In TILAK, for each cycle, a flow entry with random destination

MAC address is installed. The installed flow entry in each switch allows LLDP traffic towards controller. After flow entry installation, the LLDP packets are generated and sent to each switch to be sent to the directed port.

Race condition: A race condition can occur in an installation of a flow entry and LLDP packets. If flow entry is installed successfully, then only LLDP packets are sent to the controller. Otherwise, LLDP packet may drop at the switch because of a flow entry which specifies that *drop all packet from unintended flows.* It will help to prevent LLDP Flooding attack. The flow entry and LLDP packet ordering can be assured with barrier messages provided by OpenFlow [31]. After proper uses of barrier messages, the race condition will not happen.

Time relations: In TILAK, few time constraints are important to make sure its correct working. The T_{cycle} is total time taken to complete LLDP packet sending cycle. In this cycle, random destination MAC addresses are generated, and LLDP packet for each eligible port are created and sent. T_{flow} is the time for which an LLDP flow entry remains valid. After T_{flow}, new LLDP flow entry with new random MAC is installed. T_{eport} is time for the controller to wait before removing ports from the eligible port list. T_{link} is the time for which controller waits before eliminating unidirectional links and update the eligible port list. T_{fhard} is *hard_timeout* for LLDP flow entry. After T_{fhard}, the switch will remove the corresponding flow entry. Few relations must be maintained to achieve desired results in TILAK.

If T_{cycle} is more than T_{flow}, it causes LLDP packet drop. Because maybe some ports receive LLDP packet after removal of the required LLDP flow entry. So T_{flow} must be greater than T_{cycle} with some minimum threshold T_{thsld1}.

$$T_{flow} > T_{cycle} + T_{thsld1} \tag{3.1}$$

If T_{fhard} is set to be less than T_{flow}, then removal of an LLDP flow entry happens early. A new flow entry is installed after a while, and in meanwhile, LLDP packets may drop. The T_{fhard} will always be greater than T_{flow} because a flow entry will remain in the switch until the new flow is generated and installed.

Topology	T_{cycle}	T_{flow}	T_{fhard}	T_{eport}	T_{link}
"tree,4,4"	20.42	25	30	35	30
"tree,7,2"	17.96	25	30	35	30
"fat tree"	34.37	40	45	50	45

Table 3.6: Choosen values for different time parameters in second(s)

However, if T_{fhard} is set to too large, then there are chances that multiple LLDP flow entries will exist in the switch, which weaken the defense. We can also use barrier messages to ensure that an old LLDP flow entry remains up to the new flow entry. Additionally, T_{link} must be greater than T_{flow} to ensure that before removal of any unidirectional link all flow entries must be installed, which can only provide with T_{flow}. In our experiment, we use different time parameters as it is shown in Table 3.6. All threshold are observationally chosen, and it gives the desired results.

$$T_{fhard} > T_{flow} + T_{thsld2} \tag{3.2}$$

After successful link discovery, we can think of removal of few ports from the eligible port list. So T_{eport} must be kept such that before the removal event all valid link should be discovered.

$$T_{eport} > T_{flow} + T_{thsld3} \tag{3.3}$$

$$T_{link} > T_{flow} + T_{thsld4} \tag{3.4}$$

TN and FP Analysis

The TN and FP are dependent on the strength of an adversary. We phrase it with the following query "what is the probability that the false LLDP packets will move to the controller?". If the provided bit string matches with the flow entry bit string, then only the packet will move towards the controller. The different packets that an attacker can generate to the switch in a second at 1GBps network with 100 bytes for an LLDP packet is Total_packet.

$$Total\_packet = 10^9/100 = 10^7 \tag{3.5}$$

$$Total\_packet\ in\ 5\ seconds = 10^7 * 5 \tag{3.6}$$

Suppose the controller install a flow with random MAC address in every 5 seconds. So total possible combination of installed flow entry is Total_combi.

$$Total\_combi = 2^{48} \approx 2.8 * 10^{14} \tag{3.7}$$

The probability of generation of LLDP packet with same random MAC address is Pr.

$$Pr \approx Total\_packet/Total\_combi \tag{3.8}$$

$$Pr \approx (10^7 * 5)/(2.8 * 10^{14}) \approx 2.5 * 10^{-7} \tag{3.9}$$

From Equation 3.9, we can observe that to match a single packet, the probability is very low. Hence we can also conclude that LLDP Flooding is hard to perform in TILAK.

3.4 Summary

SDN performs link discovery using a nonstandard OFDP and LLDP packet. The discovery process is prone to LLDP packet based attacks as shown in the previous chapter. This chapter discusses a novel countermeasure to prevent LLDP based threats. In a nutshell, this chapter can be summarized in the following points.

- This chapter discusses the design of *TILAK*, which is a token-based prevention technique for topology discovery threats in SDN. TILAK provides solutions to effectively perform source authentication for LLDP packets to countermeasure a set of LLDP-based security threats.

- The implementation of TILAK on Mininet network emulator shows its correctness i.e. resource penalty, LLDP packet construction and verification time. A True Positive and False Negative analysis give more confidence to prevent LLDP attacks, on various SDN scenarios.

This chapter presents TILAK as a security solution to link discovery. Each current solution including TILAK is using a borrowed LLDP packet. Some fields of the packet are not in use for link discovery in SDN. In the next chapter, a new protocol is demonstrated which use lightweight packets to discover links more efficiently and robustly.

Chapter 4

A Lightweight Protocol for Efficient and Secure Link Discovery

This chapter analyzes the need for an alternative in link discovery considering existing deployed mechanisms and provided solutions from the research community. This chapter also discusses an SDN Link Discovery Protocol (SLDP) for efficient discovery and extraction of topology information in Software Defined Networking (SDN) networks. SLDP aims to prevent, detect, and mitigate various security threats such as Poison, Replay, and Flooding attacks. SLDP is implemented on Mininet emulator, and the results show the effectiveness and correctness of SLDP concerning topology discovery time, CPU computational time, and bandwidth overheads when compared with the traditional OFDP.

In SDN, the global view is generated by performing the switch discovery, the link discovery, and sometimes the host discovery. Once an OpenFlow-enabled switch connects to the network, it performs a Transmission Control Protocol (TCP) three-way handshake with a pre-stored remote socket residing at the SDN controller. After successful handshaking, both negotiate on the OpenFlow version. Subsequently, the switch is asked for its capabilities and ports status. These steps help controller to discover the switch with available ports. To perform various topology-aware activity, link discovery is mandatory. Most of the SDN controllers use OFDP and LLDP for the discovery process. An LLDP packet is generated at the controller, and sent to a switch with the forwarding instruction [52] [50]. The switch receives such a packet, it consults with flow entry and forward the LLDP packet to the controller. The controller puts source information in the LLDP packet, and it also receives destination information on successful reception of the same packet. This information is used to create a unidirectional link. The same process is repeated in the discovery of each unidirectional link. To discover the hosts in topology, the controller uses traffic from hosts. Such host discovery is controller dependent. The controller independent host discovery is also performed with Address Resolution Protocol (ARP).

In this chapter, we propose an SLDP for efficient discovery and extraction of topology information in SDN networks[16]. The design of SLDP is motivated from the need of a secure, lightweight, and efficient link discovery protocol in SDN. SLDP aims to prevent, detect, and mitigate various security threats such as Poison, Replay, and Flooding attacks, which are due to lack of source authentication, lack of packet integrity checks, and reuse of static packets. SLDP creates and maintains the global network topology at SDN controller by using smaller size and lower number of SLDP packets during the topology discovery process. Thus, it significantly minimizes the topology discovery overhead in the network. We implemented SLDP on Mininet emulator, and the results show the effectiveness and the correctness of SLDP concerning topology discovery time, CPU computational time, and bandwidth overheads, when compared with the traditional OFDP. Additionally, SLDP successfully prevent, detect, and mitigate various attacks (e.g., Poison, Replay, and Flooding) in different SDN scenarios.

4.1 Foundation

A link discovery protocol can be considered good if it discovers link as quickly as possible, and it is secure against known threats while consuming less bandwidth and CPU cycles. Further in this section, we provide adequate evidence that motivates us to design a new link discovery protocol in Software Defined Networking (SDN).

4.1.1 Security

A secure link discovery in SDN ensures accurate topology discovery performing prevention, detection, and mitigation from the known threats. It must be adaptive to zero-day threats also. This subsection starts a quick discussion about security threats even though already discussed in Chapter 2 in detail. The security threats are possible due to the controller's inability to perform source authentication, packet integrity check, and static packet creation.

Attack Vector

The Link Layer Discovery Protocol (LLDP) packets are generated for each port on each switch. These packets are reached to target switch, the switch forwards them to the controller with the help of a dedicated flow entry or default miss entry. The major threats for the link discovery are Replay, Poison, and Flooding [55] attacks. Let's investigate these attacks with the help of Figure 4.1, which consists of three OpenFlow switches, switches $S1$ and $S2$ have three hosts attached to each of them. The two hosts are malicious, e.g., either a person with malicious intentions is operating them or malicious application are installed on them. The switches $S1$ and $S2$ are connected with $S3$, and the dash line between $S1$ and $S2$ represents a fake link.

Replay Attack (RA): Due to LLDP packet propagation to each port on a switch, the attached hosts also receive LLDP packets. However, if one of the attached host sends a received LLDP packet from some other host that is attached to another switch, than the receiving switch and the controller has no way to identify the source of that LLDP packet. For instance, if a malicious host at switch $S1$ receives LLDP and share it with malicious host attached to $S2$, then the host at $S2$ send the LLDP packet on local port to $S2$,, and again the $S2$ sends the received LLDP

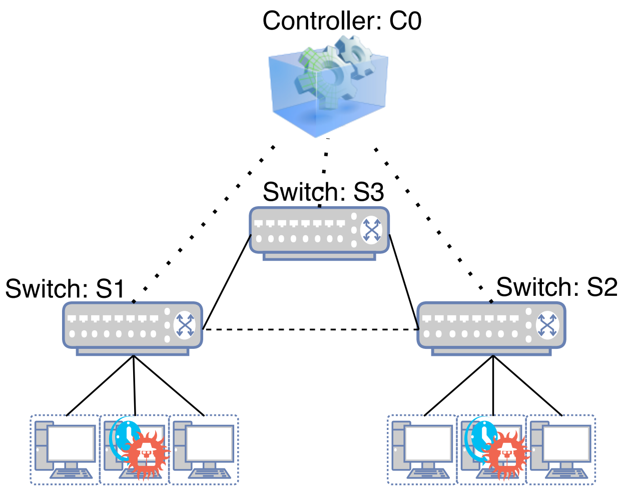


Figure 4.1: Topology for attack vector

packet to $S1$ with its malicious host. Finally, $S1$ and $S2$ will send these packets to the controller, and the controller confirms that $S1$ and $S2$ have a direct link.

Poison Attack (PA): If the attacker creates fake LLDP packets and sends it to the attached switch. The switch is unable to differentiate between genuine and fake packets, thus the packets are sent to the controller. The controller also has no way to find the source of the packet and it cannot evaluate the integrity of the packet. In this way, a generated false link creates topology poison. For instance in Figure 4.1, if an attacker at $S1$ creates an LLDP packet containing information like Chassis id is 2 and port is 3, then the switch would forward it to the controller. The controller makes a unidirectional link between $S2$ to $S1$.

Controller(Ver.)	Vulnerability	LLDP Attack			Attack Type
		LP	LF	LR	
POX(0.2.0)[8]	No hash	✓	✓	✓	*RC*
Ryu(4.12)[9]	No hash	✓	✓	✓	*RC*
OpenDayLight(3.0.7)[10]	Static hash	✓	✓		*LC*
Floodlight(1.2)[11]	Static hash	✓	✓		*LC*
ONOS(1.9.0)[13]	No hash	✓	✓	✓	*RC*
HPE-VAN(2.7.18)[14]	Static hash	✓	✓	✓	*RC*

Table 4.1: Different controllers with attack vector

Flooding Attack (FA): If a controller receives a fake crafted LLDP packet, it computes logic. Hence, if an attacker sends a flood of LLDP packets, e.g., 50,000 packets per second, the resource consumption at controller increases rapidly, and it negatively effects the benign packets service rate. Additionally, these large number of fake LLDP packets waste switch-to-controller bandwidth and controller CPU cycles.

The possible cause of these attacks are failing to check source authentication of the received packet, and failing to verify the integrity of packet, static content and constant flow entry. For instance, if a controller is unable to identify the source of LLDP packets and the LLDP packet holds static content, then the controller is prone to Replay attack. If the controller is unable to identify the source of LLDP packets, fail to check the integrity of LLDP packet, and the LLDP packet have static content, then Poison attack may happen. If the switch has a constant flow entry to pass LLDP packets to the controller then Flooding attack may happen [56].

Table 4.1 illustrates the security strength of controllers against LLDP Poison (LP) , LLDP Replay (LR), LLDP Flooding (LF) attacks. RC (Row Craft) and LC (Library Craft) are the two classes of attacks which are used to perform the security attacks on topology in SDN. In RC, the attacker uses LLDP packet information and parsing algorithm to perform the attack, while in LC the attacker uses a library of the controller to perform the attack. Few controllers (e.g., Floodlight, OpenDayLight, and HPE-VAN) attach a hash to the LLDP packets. To perform the attack on these controllers, the library information is needed, which may have the information about the hash generation algorithm (or a secure static key) that is required for the successful attack. To understand more about how attacks can happen on different controllers, please refer to [56].

POX, Ryu, and ONOS has no security content in their generated LLDP packets, hence these controllers are prone to attacks. The OpenDayLight controller generates an MD5 hash of a string, i.e., ‘openflow:1:2’, which an attacker can also create if she knows the MD5 library and a secret key. Both the information can be achieved, if an attacker performs reverses engineering to the controller code. Hence, having static hash key won’t help to protect from such attacks. In Floodlight controller, local machine interfaces are required to calculate the hash, hence the attacker is not able to compute the same hash on the local machine. But once the hash is calculated, it is kept same for further LLDP packets. The attacker has an opportunity to put the same hash value in fake crafted LLDP packets. In case of HPE-VAN, same mistake is repeated for all the packets, and same hash is used. Hence, an attacker can copy and use the same fake LLDP packets. A point to note in OpenDayLight is that hashes for each switch port are separate, but in case of Floodlight and HPE-VAN hash for each LLDP packet is same. In a nutshell, most of the industry and academic grade controller are insecure, specifically against the LLDP-based threats.

Approach	Authentication	Integrity	LLDP broadcast	Poison	Flooding	Replay
TopoGuard[53]		y	y	y		
SPHINX[65]			y	y		
OFDP_HMAC[66]	y	y	y	y		y
ESLD[67]		y		y		y

Table 4.2: Comparison of different research proposals for security

The research community is also trying to secure the link discovery process. Table 4.2 provides the summary of such efforts. TopoGuard [53] identifies source authentication and integrity check failures as possible reasons for the described threats. Their proposed approach only considers LLDP based poison attack but not the remaining attacks. SPHINX [65] detects the Poison attack with static mapping of ports with hosts. OFDP_HMAC [66] identifies correct reasons, but still, packets are broadcast to each port. LLDP Flooding is not detected or prevented. All approaches are detection and mitigation based, hence leaves scope to look for some preventive measures which can also reduces resource consumption during the attacks. In conclusion, some security improvements are possible. For instance, ESLD [67] only generates LLDP packet for non-host ports on each switch. But the approach is based on host traffic, and an attacker can forge its behavior. Also key-based hash is sent to each packet, which is time-consuming..

4.1.2 Lightweight

Currently, link discovery in SDN is performed with controller specific OpenFlow Discovery Protocol (OFDP) implementation and LLDP packets. In traditional networks, LLDP packet is designed to be a vendor-neutral link layer protocol for Local Area Networks (LAN). LLDP is used to advertise network elements' identity, capabilities, and neighbors. Different fields in LLDP packet are Chassis ID, Port ID, Time To Live (TTL), Port description, System name, System description, System capabilities, Custom Time-Length-Value (TLV), and End of LLDPDU. Chassis ID, Port ID, TTL, and End of LLDPDU are compulsory, and the rest are optional. Each networking device is supposed to run an LLDP agent. The LLDP agent gathers remote device information and advertise local information to remote devices. The collected data is stored in management information database (MIB) and it can be queried with the Simple Network Management Protocol (SNMP).

Currently, SDN community is using a borrowed packet format to perform the link discovery. LLDP packet structure has some extra features which can not fit anywhere in the picture with SDN's topology discovery phase. Firstly, we need chassis id and port id for source information. TTL field in LLDP protocol is used to instruct remote LLDP agent for validness of the information. But in SDN, no LLDP agent is installed in any of the forwarding elements. Hence, the TTL field is not required. For validness of link in SDN, the controller sends periodic packets. In SDN, no forwarding element is interested in the names or capabilities of neighboring elements because the controller takes care of this information and receives it in the first few packets of communication with the forwarding elements. Also, the end of LLDP packet is not required if we can provide a fix length packet for discovery.

In TLV structure, the LLDP has to insert various types and subtypes along with the length. If we need fixed length chassis id (dpid) and port id, then no need to put all type and subtype values. The OpenFlow specification specifies the length of dpid and port id fields as eight and four bytes respectively. Because type, subtype, and length field also consumes extra storage, it is essential to remove this structure if not required for the topology discovery process.

Table 4.3 shows the length of each LLDP frame generated by different SDN controllers. We can see that if we need only 26 bytes then why to go for 40 bytes in Ryu and 85 bytes in the OpenDayLight controller. A reader may argue that

Deployments	Size of link discovery frames(bytes)
POX(0.2.0)	41
Ryu(4.12)	40
OpenDayLight(3.0.7)	85
Floodlight(1.2)	75
ONOS(1.9.0)	66
HPE-VAN(2.7.18)	67
Target	14+8+4=26

Table 4.3: Length of different LLDP packets

controller might be storing some hash value in it. Yes, we agree but not ready to recommend it because we can see in Table 4.1 that all controllers are vulnerable then what is the use of storing the hash. Secondly, computation of hash is a costly process. It will be more worse, if the controller has to calculate a different hash for each LLDP packet.

4.1.3 Efficient

In the link discovery process the controller generates LLDP packets and sends them to all switches. The receiving switch forwards these packets to the instructed ports. If any switch receives a LLDP packet from its neighboring switch, it forwards the packet to the controller. The controller parses the received LLDP packet and creates a link. In present deployments, LLDP packets are generated for each port on each switch. If some of the ports are attached to hosts, LLDP packets for those ports are of no use. If the controller generates less number of LLDP packet, fewer resources will be consumed at the controller. Same arrangement will also prevent a malicious host to perform controller fingerprinting.

Table 4.4 shows the statistics about the number of switches, number of hosts, number of ports, and number of links in few SDN topologies. For the structure of topologies, consider Section 4.4. In Tree,4,4 topology, the controller has to generate 424 LLDP packets to discover 340 links. In the same topology 256 host are there, which means 256 generated LLDP packets are of no use. The last column in the table specifies non-eligible ports for LLDP packet. In particular, a controller can reduce resource consumption by reducing the number of LLDP packets.

Topology	**Switch**	**Link**	**Port**	**Host**	**eligible ports**
Tree,4,4	85	340	424	256	168
Tree,7,2	127	254	380	128	252
Fat tree	80	384	705	64	641

Table 4.4: Eligible ports in different topologies

In this section, we demonstrated with various examples that a lightweight, efficient, and secure link discovery is still needed to achieve optimal results with lesser resources in SDN.

4.2 The SLDP Protocol

In Section 2.3, we have discussed various security vulnerabilities in the existing controllers. In this section, first we provide motivations for a new design of a link discovery protocol. In particular, we propose an SDN Link Discovery Protocol (SLDP). The design of SLDP consists of SLDP packet format, system architecture, and event sequence. We also defines SLDP characteristics along with SLDP packet structure in this section.

SLDP discovers a unidirectional link between two OpenFlow enabled forwarding elements, which may also be separated by a non-OpenFlow switch. The protocol is a lightweight, efficient, and secure solution for discovering links in SDN physical topology. The discovered links along with the switch information is used to create a global view at the controller.

Assumptions:

This section introduces a novel lightweight, efficient and secure link discovery protocol for SDN. Correct and desired working of the protocol also depends on few assumptions as follows.

- The controller with running applications is malware free.
- The switches work based on OpenFlow specifications.

- The switches are malware free.

4.2.1 Desired Characteristics

SLDP aims to discover links in a more secure, efficient, and lightweight way. More precisely, SLDP will work correctly if holding the following characteristics:

Definition 1 (Correctness):

- SLDP must discover the link between two OpenFlow enabled switches.
- SLDP must discover the link between two OpenFlow enabled switches separated with a non-OpenFlow switch.
- SLDP must provide latency for each discovered link.
- SLDP should secure against Replay, Poison, and Flooding attacks. **(Secure)**
- SLDP packet size must be kept to minimum. **(Lightweight)**
- SLDP must perform link discovery with less number of packets. **(Efficient)**

If the discovery process is secure, the controller will work as intended, hence, better controller resource utilization. If the link discovery is done with lightweight packets (i.e., lower packet size and less number of packets to discover the topology), then the discovery process uses less bandwidth and light traffic on network interfaces. If the discovery process is efficient, system requires less bandwidth and CPU resources to generate the discovery packets. In particular, the SDLP's theoretical and practical analysis gives confidence towards its correctness. We theoretically examine some of the stated correctness one by one. Security-related correctness is analyzed with test cases that we will discuss in Section 4.3. Furthermore, the experimental setup and evidence are demonstrated in Section 4.4.

SLDP must discover the link between two OF-switches. For instance, all unidirectional links in a topology belongs to a set L = $\{l_{12}, l_{21}, l_{23}, l_{32}....l_{mn}, l_{nm}\}$, and SLDP's discovered links belong to set D = $\{d_{12}, d_{21}, d_{23}, d_{32}....d_{mn}, d_{nm}\}$. The following condition must be followed $\forall x[x \in L \iff x \in D]$, i.e., all elements which belong to L must also belong to D. SLDP packets generated at the controller are sent to a switch with OpenFlow TCP/TLS channel with forwarding instructions. The switch obeys instruction and a linked switch receives the SLDP packet. The receiving switch consults flow table to forward it to the controller. SLDP uses the randomized information to create both flow entry and the SLDP packet. Here one possible problem is to be considered which is called `race condition`. If an SLDP packet reaches to the second switch before the flow entry packet, it must be dropped. To prevent the race condition, SLDP uses Barrier message [31]. The barrier message ensures the order of flow entry installation and SLDP packet sent instructions. It confirms clarification of a single doubt while SLDP discovers the link.

SLDP must discover the link between two OF-switches that are separated with a non-OpenFlow switch. Layer two or layer three switch forwards layer two broadcasts. SLDP Ethernet frame uses broadcast destination address. Hence, once a broadcast packet is received by the layer two or three device, it must be forwarded to all of its ports. An OF-switch is connected to one of its port so it will receive the packet. Packets are received over non-OpenFlow switches also, however the remaining process for link discovery stays the same.

SLDP must provide latency for each discovered link. Whenever a packet is sent for discovery, the time stamp is noted. After traversing two switches, the same packet comes to the controller. Upon receiving this packet, SLDP records the second time stamp. The latency is calculated with both these time stamps. The SLDP ensures that the discovery packet size stays to minimum. As Table 4.3 and Figure 4.2 suggests, SLDP takes the minimum bits for a link discovery packet. SLDP uses 26 bytes, while others discovery protocols are taking in a range from 40 bytes to 85 bytes.

The SLDP must perform link discovery with less number of packets. As Table 4.5 suggests, eligible ports are less than the number of total ports irrespective of the topologies. The SLDP finds out non-eligible ports after few iterations and removes them form eligible port list. For instance, a controller's OFDP implementation is generating 'o' bytes link discovery packet. SLDP implementation for the same controller is taking 's' bytes. If $p_1, p_2, p_3 ... p_n$ are ports of switches in a topology, and $h_1, h_2, h_3..h_m$ are the hosts attached to switches on some of the ports. Here 'n' and 'm' are the numbers of ports and hosts attached to switches respectively. Hence, the total profit in terms of bytes can be represented as Equation 4.1 as per the iteration, i.e., every five seconds.

$$\left\{\sum_{i=1}^{n} p_i - \sum_{i=1}^{m} h_i\right\}\left\{o - s\right\} \tag{4.1}$$

4.2.2 SLDP Packet Format

SLDP is designed to identify links between two OF-switches, which may be separated with a non-OpenFlow switch. For each discovered link, the latency is also provided. SLDP is secure against Replay, Poison, and Flooding attacks. Lightweight and efficient link discovery is desirable. For all these features, SLDP uses a simple yet effective frame format, which can be seen in Figure 4.2.

To understand SLDP working methodology, understanding of the SLDP packet structure is mandatory. Two partitions are shown in Figure 4.2, first is Ethernet header and second is SLDP payload. SLDP utilizes both the partition to work correctly. In Ethernet header destination, a broadcast address is used. In source address field, a random MAC address is used. In EType a custom type is introduced. The dpid and port are the source information of the link. In a nutshell following information is used in SLDP.

DMAC	$\rightarrow$ ff:ff:ff:ff:ff:ff
SMAC	$\rightarrow$ Random MAC address
EType	$\rightarrow$ 0xabcd
dpid	$\rightarrow$ Source dpid
port	$\rightarrow$ Source port

A reader may argue that how security and latency will be provided in SLDP with such a basic packet structure. In Section 4.4.3, detail of security and latency is discussed.

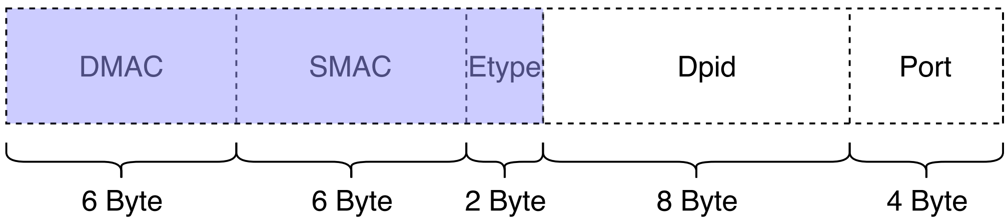


Figure 4.2: SLDP packet format

4.2.3 SLDP System Architecture

SLDP system architecture explains about basic function blocks, which are designated for dedicated work. In Figure 4.3, system architecture for SLDP is illustrated. Link discovery is a periodic activity, in each cycle, few operations needs to be performed, i.e., random MAC address generation, flow entry installation, SLDP packet generation and transmission. Randomness generator block will provide random MAC address, which is used in SLDP packets and flow entries. Randomness ensures all restrictions, which are applicable to a MAC address. A flow entry installer module installs flow entries in each OF-switches. Packet generator takes random source MAC and creates SLDP packet. To generate random MAC addresses, a python based cryptographically secure pseudo-random number generator i.e. os.urandom() is used. Once flow entry is installed, the SLDP packet with the same randomness is allowed to pass to the controller. The packet sender node sends an SLDP packet to the source switch. The Unique selling proposition (USP) of this approach is an eligible port identifier, which identifies eligibility for each iteration. Initially, all ports are considered eligible, but after each iteration, the list is updated. In SLDP whenever a switch awakes, its every port is added in the *ePorts* or eligible port list. Therefore, in the next cycle the SLDP packets are generated for each eligible port including the latecomers of the previous cycle, and the rest of the process for these latecomer ports remains the same.

An OF-switch receives an SLDP packet with instructions to forward it on a particular port. On the other end of the tunnel, when an OF-switch receives the SLDP packet, it consults the flow entry table. Due to flow entry installer, there will be a flow entry, which forwards the packet to the controller. The packet receiver receives the packet and validates it for SLDP packet. SLDP packet comes

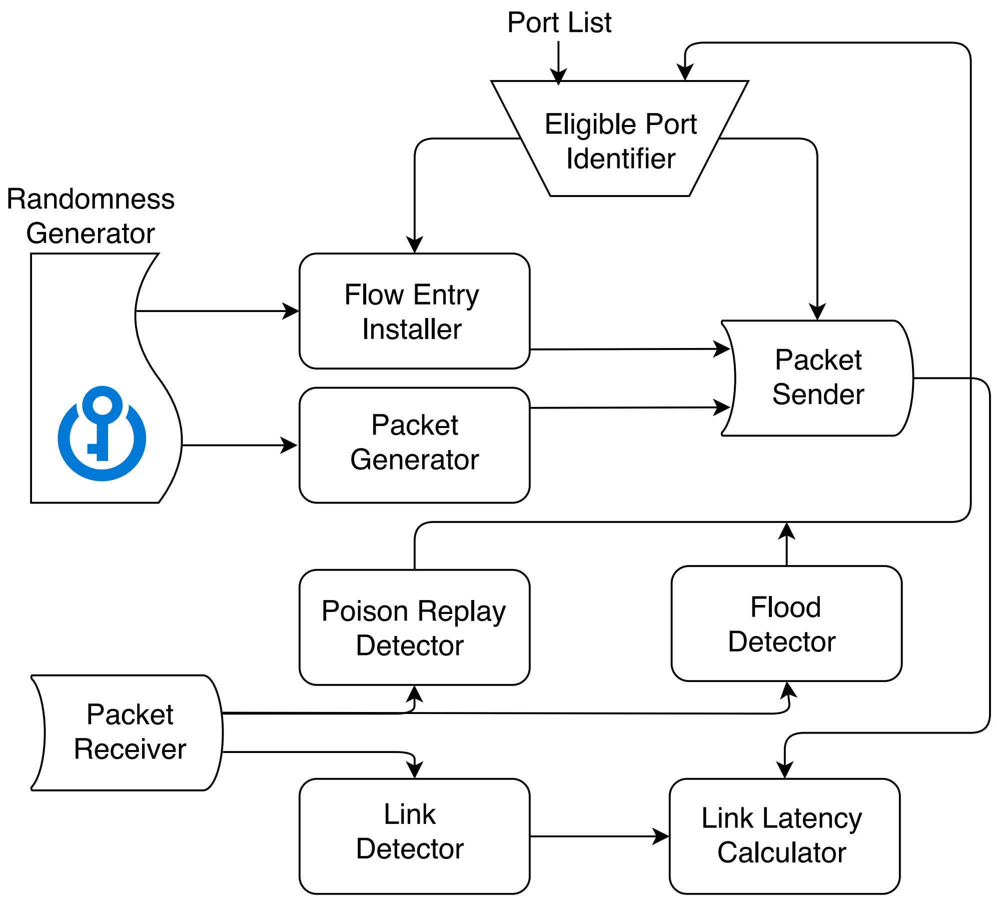


Figure 4.3: SLDP system architecture

in wrapped in PACKET_IN packet. The header contains destination information for the link. Both source and destination information are used to form the link. Each packet is sent on a time stamp and receives on another time stamp, such information is used to calculate latency in link latency calculator. If receiving packet alerts Poison, Replay, and Flooding detector, the eligible port list will be updated.

Flow Entry Structure: In each SLDP packet cycle, a new flow entry with provided randomness is installed in each participating switch. Here an example of that flow entry is shown.

```
*** s1 ---------------------------------------------------
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1.687s, table=0, n_pa ckets=2, n_byte
```

```
s=52, hard_timeout=5, idle_age=0,  priority=65535, dl_src=
96:66:e8:27:9f:94, dl_dst=ff:ff:ff:ff:ff:ff, dl_type=0 xab
cd actions=CONTROLLER:65535
```

In the above example, source MAC address is randomly generated, while destination MAC address is set as a layer two broadcast address. Ethernet type is fixed to 0xabcd and action is set to the controller, i.e., matching packet will move to the controller. Each time a port is declared as non-eligible, SLDP removes it from eligible port list. Algorithm 8 gives the abstract idea of maintaining the eligible port list. The updateEports procedure requires three arguments. One for eligible port list, i.e., *ePorts*, and other two for the operation selected, i.e., *opCode* and *portId* for a targeted port to addition or removal. Possible operations are addition (ADD) and removal (REMOVE).

Algorithm 8 Update eligible port list

```
Require: ePorts, opCode, portId

1: procedure UPDATEEPORTS
2:     if opCode = ADD then
3:         ePorts = ePorts + portId
4:     else if opCode = REMOVE then
5:         ePorts = ePorts - portId
6:     else
7:         return
8:     end if
9: end procedure
```

If a port is not receiving an SLDP packet for a long time or a packet is not reached back to the controller which is designated for a port, then the port is removed from the eligible port list. $tflow$ is taken as 10 seconds as an observational threshold. The optimal $tflow$ calculation is one of the future works. Algorithm 9 illustrates that such ports are removed after $tflow$ time period.

To detect Poison and Replay attack, Algorithm 10 is used. First part of the algorithm suggests the case where an attacker is sitting on a non-OpenFlow switch and sends fake SLDP packet to switch. Because of packet broadcast destination address, this packet travels for two directions to the controller. At the controller, if a packet (i.e., *sldpPkt*) is already available to SLDP packet set (i.e. *sldpPktSet*),

Algorithm 9 Calculate port eligibility

```
Require: ePorts, port

 1: procedure portEligible
 2:     On each *tflow* seconds
 3:     portId, portTstamp ← Extract(*port*)
 4:     if portTstamp + $teport \leq now$ then
 5:         updateEports(ePorts, REMOVE, portId)
 6:     else
 7:         return
 8:     end if
 9: end procedure
```

then an alarm is generated. In later part of the algorithm if a unidirectional link (i.e., *ulink*) is not accompanying with the reverse direction within the report time, then the destination for that link is suspected, hence it will be removed from the eligible list (i.e., $ePorts$).

Algorithm 10 Detect Poison and Replay attacks

```
Require: ePorts, ulink, linkSet, sldpPkt, sldpPktSet
 1: procedure PRDetect

 2:     On each received *sldpPkt*
 3:     if sldpPkt $\in$ sldpPktSet then
 4:         dpId, portId ← Extract(*sldpPkt*)
 5:         generateAlert(dpId, portId)
 6:     end if
 7:
 8:     On each *teport* seconds
 9:     if rev(ulink) $\notin$ linkSet then
10:         srcPort, dstPort ← Extract(*ulink*)
11:         portId ← Extract(*dstPort*)
12:         updateEports(ePorts, REMOVE, portId)
13:     end if
14: end procedure
```

Algorithm 11 is used to detect the Flooding attack. If at any switch port (i.e., portId), the number of SLDP packets are received more than the maximum number of ports on available switch (i.e., maxPort), it generates suspicion. SLDP generates packet periodically, one for each port on each switch (not always), hence if any port receiving more than $maxPort$ is eligible for removal.

In SLDP, the following are the periodic tasks: flow entry installation, packet

Algorithm 11 Detect Flooding attacks

```
Require: ePorts, sldpPkt, maxPorts
1: procedure FLOODDETECT

2:     On each received sldpPkt
3:     dpId, portId ← EXTRACT(sldpPkt)
4:     if COUNTFOR(dpId, portId) > maxPorts then
5:         UPDATEEPORTS(ePorts, REMOVE, portId)
6:     end if
7: end procedure
```

generation, sent, reception, and link discovery. The packet generation is restricted with flow entry installation. Lets suppose T_f is a time interval for flow entry installation. After T_f, a new set of flow entries are installed, thus new randomness is inserted in flow entries. Let $l_1, l_2, l_3..l_n$ are the latencies or round-trip-time in delivering the SLDP packets back to the controller, i.e., the time between the SLDP packet generation by the controller and the same SLDP packet received by the controller. Here, l_i is the latency for i^{th} link discovery in topology, therefore, $T_f > Max\{l_1, l_2, l_3..l_n\}$ will ensure that the SLDP packets are delivered back to controller on time. If not, i.e., few SLDP packets not route back to controller due to latency in the network, then the controller falsely calculates that few links does not exist in the constructed global topology. If T_f is kept sufficiently large, then the SLDP will become more vulnerable to described attacks. It is because the attacker gets more time to craft a packet with the desired randomness. Test case #1 in Section 4.3 gives a probability analysis of being attacked while T_f is five seconds. Advancing T_f will increase the probability of being attacked. Also, Figure 4.15 suggests that the computational overhead for link discovery in SLDP is almost constant, i.e., ≈ 1 second, irrespective of the examined topologies. We chooses the time interval (T_f) to 5 seconds, which is based on the available literature [50] [67], different implementations, and observations. Our observations indicate that for any communication, the controller processes symmetric and asymmetric messages, which are very low as compared to the entire packets. Therefore, there was not identified any significant difference in the controller overhead with the varying time interval.

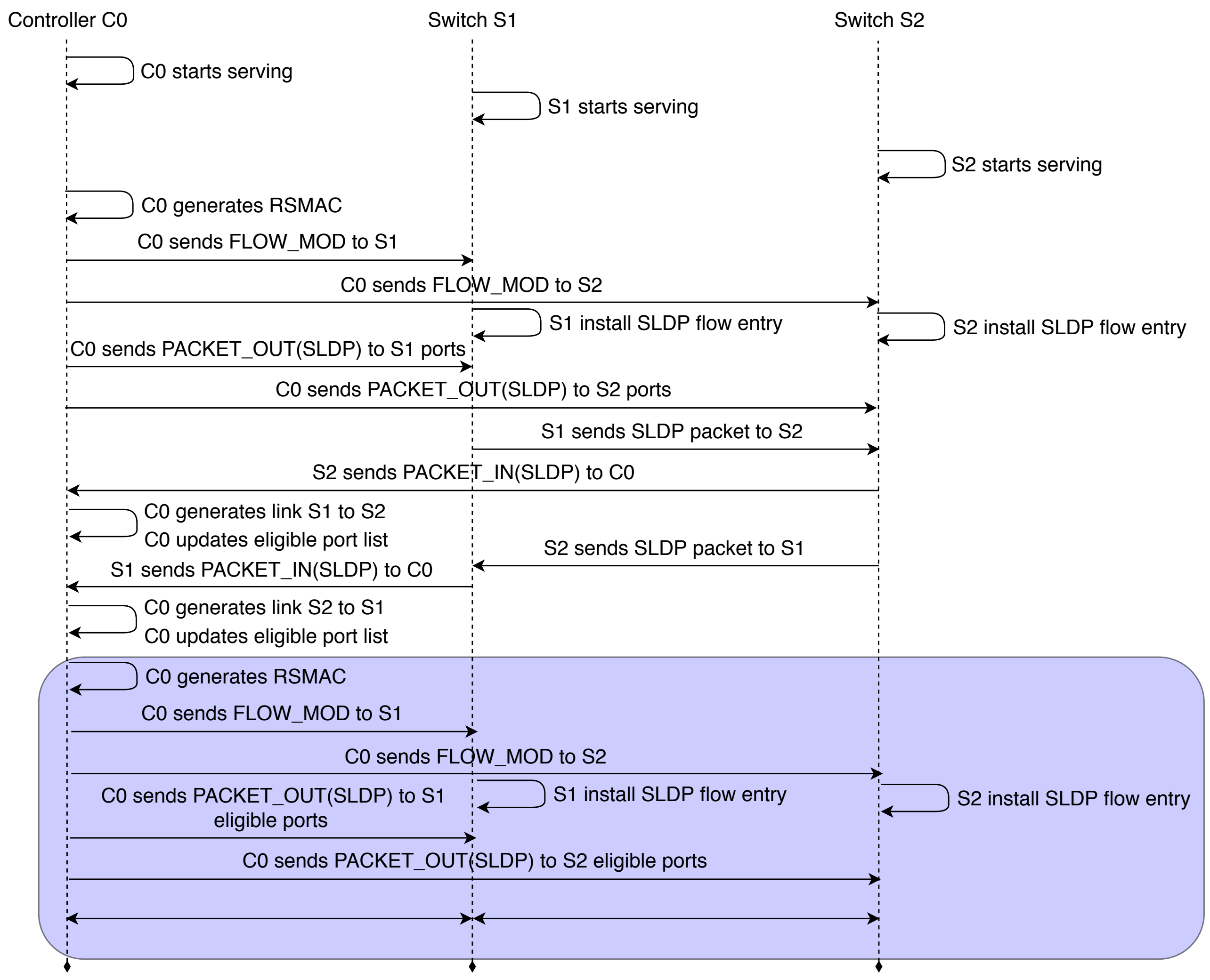

Controller C0
Switch S1
Switch S2
C0 starts serving
S1 starts serving
S2 starts serving
C0 generates RSMAC
C0 sends FLOW_MOD to S1
C0 sends FLOW_MOD to S2
S1 install SLDP flow entry
S2 install SLDP flow entry
C0 sends PACKET_OUT(SLDP) to S1 ports
C0 sends PACKET_OUT(SLDP) to S2 ports
S1 sends SLDP packet to S2
S2 sends PACKET_IN(SLDP) to C0
C0 generates link S1 to S2
C0 updates eligible port list
S2 sends SLDP packet to S1
S1 sends PACKET_IN(SLDP) to C0
C0 generates link S2 to S1
C0 updates eligible port list
C0 generates RSMAC
C0 sends FLOW_MOD to S1
C0 sends FLOW_MOD to S2
C0 sends PACKET_OUT(SLDP) to S1 eligible ports
S1 install SLDP flow entry
S2 install SLDP flow entry
C0 sends PACKET_OUT(SLDP) to S2 eligible ports

Figure 4.4: Event sequence in SLDP

Event sequence in SLDP: Figure 4.4 illustrates events sequence to understand SLDP in more detail. Here, three vertical time lines are shown for each participating entity, i.e., the controller $C0$ and two switches $S1$ and $S2$. Initially, the controller $C0$ starts serving after the switch $S1$ starts serving. $C0$ updates the eligible port list with information it received from $S1$ i.e., ports. Later switch $S2$ starts and $C0$ updates the eligible port list again. Hence, each participating entity is ready to participate. $C0$ generates random source MAC address, which is used in flow entries and SLDP packets. $C0$ sends flow entry in OpenFlow message FLOW_MOD to $S1$ and $S2$ with generated randomness. After both $S1$ and $S2$ receives FLOW_MOD from $C0$, both installs SLDP flow entry. Only after installation of flow entries, $C0$ generates and sends SLDP packet in PACKET_OUT for the eligible port list for $S1$ and $S2$. $S1$ receives SLDP frame in PACKET_OUT from $C0$ and unwraps SLDP frame from the received packet. $S1$ sends SLDP frame to the designated port. $S2$ receives SLDP frame and use installed SLDP flow entry to generate PACKET_IN. The controller $C0$ receives PACKET_IN from $S2$ and extracts eventDpId and eventPort information from PACKET_IN header. The dpId and portId information is stored in SLDP packet. $C0$ creates a link with source and destination information. Here source information is content of SLDP packet, i.e., dpId and portId. Destination information is eventDpId and eventPort information. Later $C0$ updates the eligible port list and makes it ready for next iterations. The same process is repeated when $S2$ sends SLDP frame to $S1$. Link discovery is a periodic process, i.e., entire process is repeated after a fixed time interval. The highlighted area in the event sequence diagram shows the repetitively executed instructions.

4.3 Test Case Analysis

In this section, we show the assurance of the correctness of SDN Link Discovery Protocol (SLDP) with few test cases. SLDP promises to provide lightweight, efficient, and secure link discovery protocol. SLDP works with less number of bits in a packet and less number of packets for discovery process. The protocol provides a probable security at three different levels, which are as follows.

```
A: Poison, Replay, and Flooding prevention
B: Poison, Replay, and Flooding detection  and mitigation
C: Flooding attacks detection and mitigation
```

As the name suggests, best defense strategy is Poison, Replay, and Flooding prevention. A controller creates an environment in which no such attack will happen. Second best line of defense is Poison, Replay, Flooding detection and mitigation. While detection is performed, still the system resources, i.e., CPU and bandwidth are wasted. To fix further, attack mitigation helps. SLDP removes few ports from the eligible port list to prevent further attacks. The last line of defense is Flooding detection and mitigation. In some cases, SLDP can prevent all, while in others, SLDP detect and mitigate the attacks. We ensure and prove that at least SLDP detects flooding and mitigates for future incidents.

Now we explain the SLDP working by using few test case scenarios. In each test case, a controller is attached with two or three switches. Switch to host are attached as shown in Figures 4.5 to 4.9. Some of the host are infected with the malicious application or controlled with a person with malicious intention. A sidebar in each test case figures is showing the strength of the security defense in that test case.

Test case # 1 In this test case, an attacker host exists which is attached to an OF-switch as shown it is shown in Figure 4.5. If the malicious host starts serving after the switches, and try to perform Replay, Poison, and Flooding attack. The SLDP prevents the network from all the stated attacks. In case of SLDP, no SLDP packet comes to any of attached hosts, hence protecting against the reply attack. Poison uses crafted packet. The working of SLDP suggests that a packet can only move to the controller if it has exact randomness as flow entry does. If the host is unable to pass one packet to the controller then performing the flooding is too hard.

Attacker generated packets for replay or poisoning will only reach to the con-

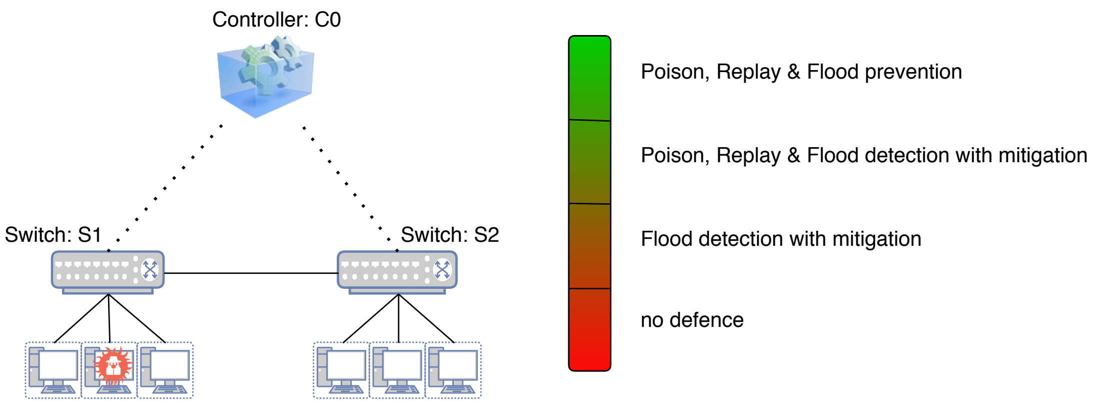

Controller: C0
Switch: S1
Switch: S2
Poison, Replay & Flood prevention
Poison, Replay & Flood detection with mitigation
Flood detection with mitigation
no defence

Figure 4.5: Test case 1: Attacker host starts work after switch

troller if the packets have the same randomness as the flow entries. In each cycle, SLDP generates a new random number, which is used in both flow entry and link discovery packets. For instance, random MAC address is denoted as a set $M = \{m_1, m_2, m_3...m_n\}$, and the packet generated by an attacker is denoted by a set $A = \{a_1, a_2, a_3...a_x\}$. Then the desired condition for successful prevention is $\frac{|A|}{|M|} = 0$. Let's examine the chances that a fake packet have the same randomness.

MAC address is 48 bits long, hence $|M| = 2^{48} \approx 2.81 * 10^{14}$. Size of SLDP packet is 26 bytes. Total number of such packets on 10 GBps in one second is equal to $(10 * 10^9)/26 \approx 3.85 * 10^8$. If the flow entry remains constant for five seconds, then in the same time period the number of total packets are $|A| \approx 1.92 * 10^9$. Now $\frac{|A|}{|M|} = 0.00000683214 \approx 0$ confirms that even in theory the full speed packets are generated, which makes the probability of a successful attack nearly to zero. If one fake packet is hard to reach the controller, then the Flooding attack is hard to believe.

Test case # 2 As shown in Figure 4.6, an attacker attached to an OF-switch starts working before the switch starts. In this case, because the SLDP chose the entire port list as an eligible port list in the beginning, the malicious host will also receive the SLDP packet. As the host has the packet or the randomness stored in a packet, it can perform attacks. Even though attacker successfully makes a unidirectional link, it will still be unable to make a reverse link. The attacker only sits on one end of the fake link; to make it in reverse direction, attacker's control is required on other side of the fake link. The above information helps the SLDP to detect such attacks. To mitigate attacks, the SLDP removes packet receiving switch port from the eligible port list.

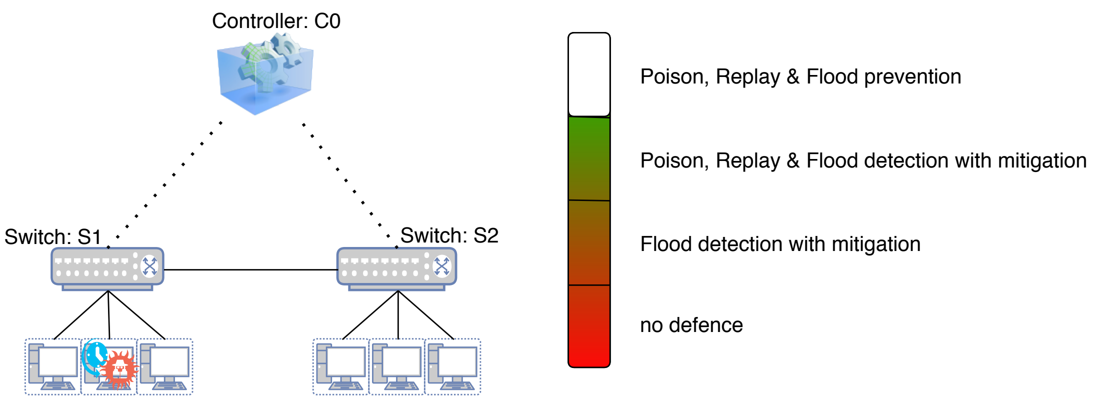

Controller: C0
Switch: S1
Switch: S2
Poison, Replay & Flood prevention
Poison, Replay & Flood detection with mitigation
Flood detection with mitigation
no defence

Figure 4.6: Test case 2: Attacker host starts work before switch

Test case # 3 The Figure 4.7 shows two attacker hosts attached to different OF-switches. Now both the switch receives SLDP packets, extracts the randomness and craft the fake SLDP packets with the same randomness. Both hosts can create the fake bidirectional link at the controller (shown with dash lines). In this case, the detection of Replay and Poison is not possible, however the SLDP will be able to detect and mitigate the Flooding attacks.

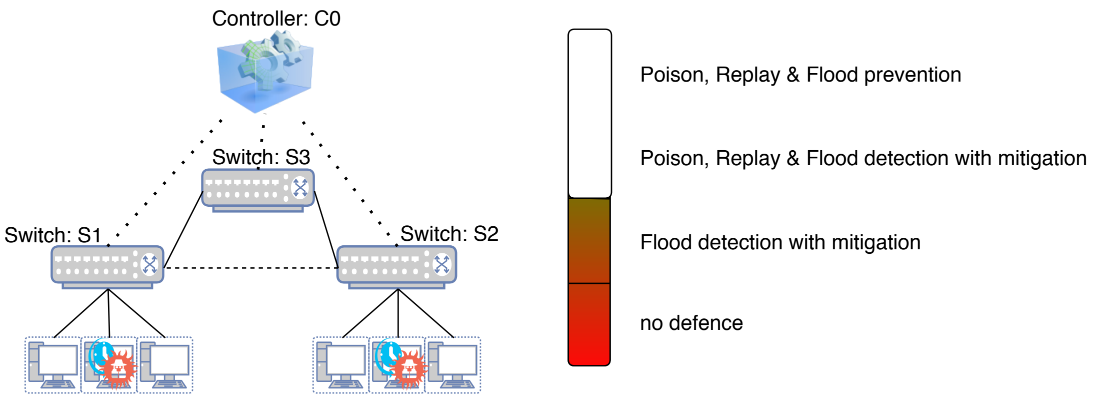

Controller: C0
Switch: S3
Switch: S1
Switch: S2
Poison, Replay & Flood prevention
Poison, Replay & Flood detection with mitigation
Flood detection with mitigation
no defence

Figure 4.7: Test case 3: Two attacker hosts starts work before switch

Test case # 4 As shown in Figure 4.8, a non-OpenFlow switch separates a link between two OF-switches. An attacker or host can be attached to an OF-switch, but it will not receive any randomness information to perform an attack. Hence, SLDP prevents Poison, Replay, and Flooding attacks.

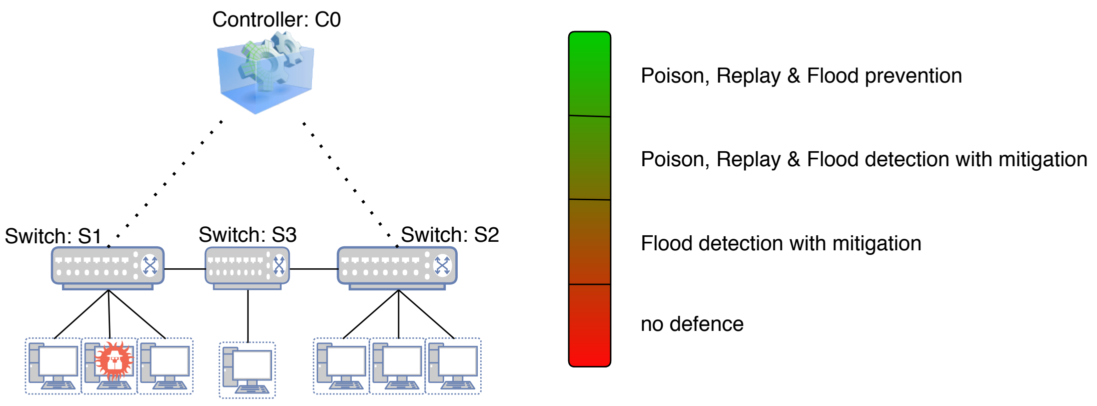

Controller: C0
Switch: S1
Switch: S3
Switch: S2
Poison, Replay & Flood prevention
Poison, Replay & Flood detection with mitigation
Flood detection with mitigation
no defence

Figure 4.8: Test case 4: A non-OpenFlow switch as separator

Test case # 5 Figure 4.9 represents an attacker host attached to a non-OpenFlow enabled switch. The attacker always gets the information of same randomness due to the broadcast destination MAC address. If the attacker crafts an SLDP with the spoofed randomness, the SLDP packet reaches via both $S1$ and $S2$ switches. Hence, SLDP detects Poison, Replay and Flooding attacks. However, because a non-OpenFlow enabled switch can not be controlled by the controller, mitigation is not possible.

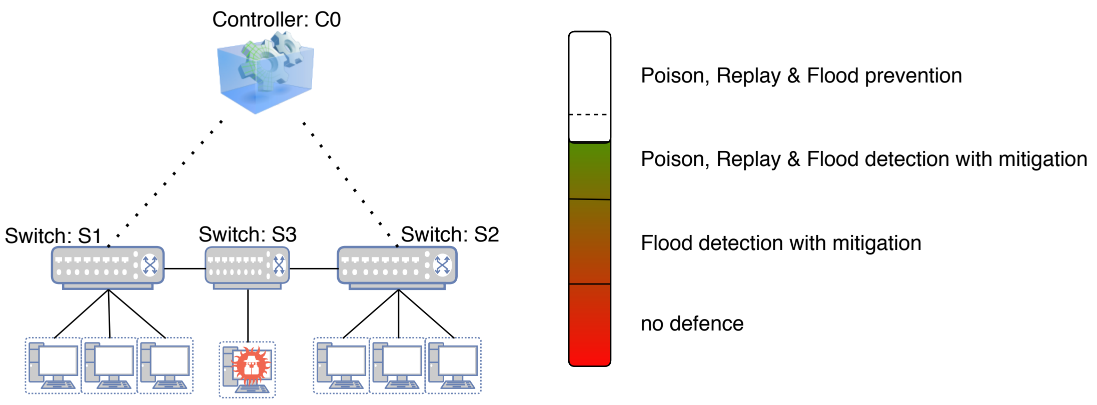

Controller: C0
Switch: S1
Switch: S3
Switch: S2
Poison, Replay & Flood prevention
Poison, Replay & Flood detection with mitigation
Flood detection with mitigation
no defence

Figure 4.9: Test case 5: Attacker host starts work before switch

4.4 Implementation

To prove any given proposal's correctness, theoretical and practical analysis is necessary. In this section, we focus on the experimental evidence for correctness of SDN Link Discovery Protocol (SLDP). Typologies and experimental environment are discussed to justify the experiments.

4.4.1 Experimental Setup

All the experiments are performed on Mininet[57] network emulator. To perform experiments three different topologies are used as shown in Figure 4.10, 4.11 and 4.12. Table 4.4 is shows the relative statistics, i.e. number of switches, links, hosts and ports. We consider the topologies that has 80, 85 and 127 number of switches, which is fairly large to justify our experiments. In the Table 4.5, the links are between switch to switch and switch to host. In link discovery process only switch to switch links are considered, which can be obtained from the number of hosts subtracted from the number of links in the table. ‘tree,4,4’ and ‘tree,7,2’ are topologies from Mininet environment. ‘fat tree’ is tree topologies to simulate a data center arrangement of switches.

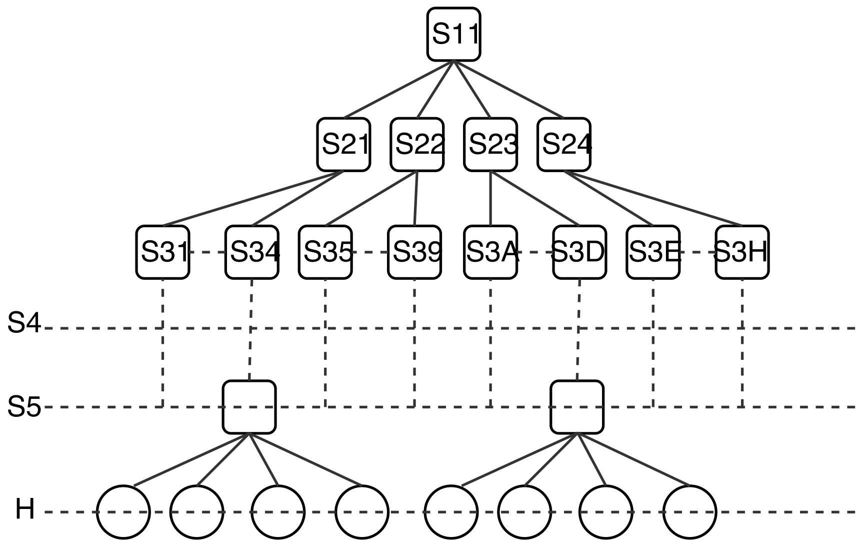


Figure 4.10: Topology 1: Tree topology with 4 level of depth and fan-out 4

In Figure 4.10, five layers of switches are used. Except for root and leaf switches, each switch is connected to five switches. For example, $S35 - S39$ represents four connected switches in layer three. $S4$ and $S5$ are switches in layer four and five respectively. Hosts connected to layer five switches are represented with symbol H.

In Figure 4.11, switches are connected with three switches except for root and

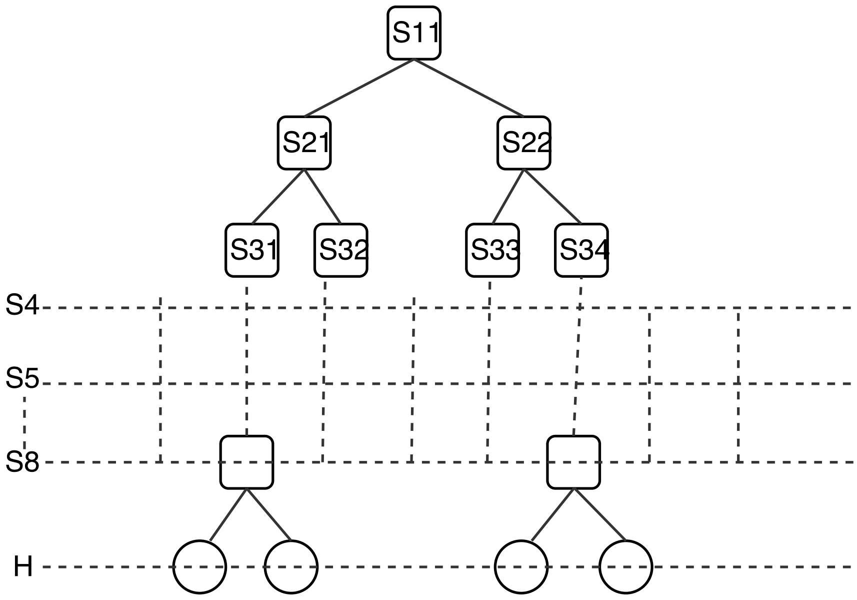


Figure 4.11: Topology 2: Tree topology with 7 level of depth and fan-out 2

leaf switches. The eight layers of switches(total 127 switches and 254 links) forms a tree topology.

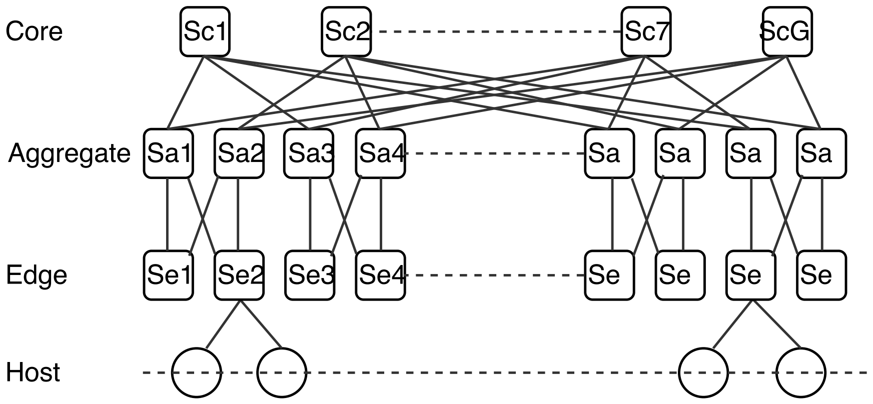


Figure 4.12: Topology 3: Fat tree

As in Figure 4.12, the Fat tree topology is shown and alike topologies are used typically in a data center. Here the switches are arranged in a three-layer arrangement, namely core, aggregate, and edge. Core switches are connected to alternate aggregate switches for redundancy. The aggregate switches are also connected to more than one edge switches. In our experiments, we choose two hosts per edge switch.

Table 4.6 gives the details of an execution environment for validation of SLDP, which includes operating system and test-bed details. The SLDP implementation

Topology	**Switch**	**Link**	**Port**	**Host**
Tree,4,4	85	340	424	256
Tree,7,2	127	254	380	128
Fat tree	80	384	705	64

Table 4.5: Number of Switches, Links, Ports and Hosts

is tested in the Mininet[57].

Resource	Configuration
Test-bed	Mininet(emulation)
Victim/Attacker OS	UBUNTU 16.04$LTS(64bit)^*$
Victim configuration	4CPUs and 4 GB
Attacker configuration	4CPUs and 4 GB
Controller	Ryu
Attack traffic	150000 Packet/Sec
Software switch	OpenVSwitch(2.5.0)
Network	1 Gbps

Table 4.6: Experimental environment for SLDP

Generating results in self created environment is always questionable. Author hooks ‘print’ statement over several code locations to generate the results. Even this ‘print’ statement is performing I/O causes CPU resources. The results can be viewed in relative terms rather than absolute. For comparison, other implementations are considered which varies from two implementations of the same controller, i.e., RYU-OFDP, RYU-SLDP or various controllers, e.g., POX, RYU, and ONOS. RYU-OFDP is OFDP implementation in Ryu, while RYU-SLDP is SLDP implementation in Ryu controller.

The controller selection in any experimental setup is dependent on aspects such as programming language, documentation, and controller updations. Most controllers are written in either JAVA or python. For instance, the POX and Ryu controllers are written in python while OpenDayLight, Floodlight, ONOS, and HPE-VAN are written in JAVA. Few controllers are developed for academic purpose, i.e., POX, Floodlight, and Ryu, while others are industry grade controllers, i.e., OpenDayLight, ONOS, HPE-VAN. For our experimental setup we use Ryu because it is being developed as open source using python and it is a well-documented controller.

4.4.2 Performance Metrics

Some metrics mentioned below are used to understand the obtained results. ***Packet Length:*** In link discovery, a dedicated packet is used. Different controllers use different packets. Lighter packet uses less bandwidth in communication. However, packet content may carry security information. Hence, a light packet which also secures the link discovery will be a good choice.

Number of packets: In each cycle, discovery packets are generated. Few modifications also reduce the number of packets. With less number of packets, bandwidth can be saved. Hence an algorithm can be examined for the number of discovery packets used in an iteration.

Computational Overhead: In Ryu controller, two threads controls discovery packet generation and transmission. Each thread is infinite loop runs at a fixed time interval. Total time taken in completion of one iteration is averaged over several observations. An addition of averages for both threads gives computational overhead.

Packet Construction Overhead: Time taken for discovery packet creation can be used for comparing result. Based on the content, different controllers use different times for packet generation. Packet construction and verification sequence also affect the time taken.

Packet Verification Overhead: Whenever a controller receives a discovery packet, the packet is examined for validness. Different deployments use different validation checks which confer in the time taken in verification.

Topology Discovery Time: A received and validated discovery packet used to create a link. The link is added to the topology database. Time taken to build entire topology is measured as the time difference between a first link and last link discovery time stamps.

Initial Overhead: The Ryu controller installs flow entries and generates discovery at the initial phase. Further, the controller only transmits stored discovery packet. Static installation and packet creation costs overhead.

4.4.3 Results and Discussion

SLDP is a lightweight link discovery protocol. The number of bits required in SLDP is least among other implementations. As Figure 4.13 shows, the SLDP requires 26 byte long Ethernet frame to accomplish link discovery along with efficiency and security. SLDP removes some unnecessary fields and restructures the packet to reduce its size. SLDP uses position based data separation to save some additional space.

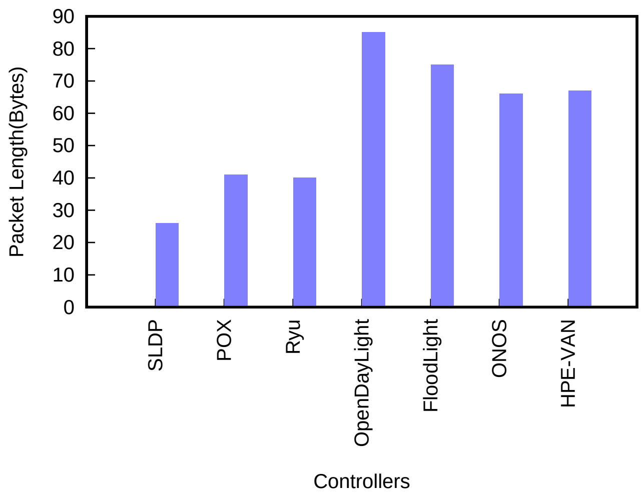


Figure 4.13: Link discovery packet length among all controllers

In each link discovery, few discovery packets are generated and sent over OpenFlow channel to switch. The traditional implementation of link discovery generates LLDP/BDDP packets for each port on each switch. Edge switches are connected to host, which infers no need to generate packet reaching to end host. The SLDP approach also helps to prevent the controller fingerprinting. In SLDP, few non-eligible ports are identified and SLDP packet is sent to each port except ineligibles. Fewer generation of SLDP packet also consumes lesser CPU and bandwidth resources. Figure 4.14 gives the idea of number of discovery packet required with or without the solution in different topologies. RYU-SLDP always require lesser packet irrespective of topology.

CPU resources are required for sending and parsing link discovery packets. Figure 4.15 shows resource consumption in RYU-OFDP and RYU-SLDP with three topologies. It is clearly identified that SLDP is doing so well because of

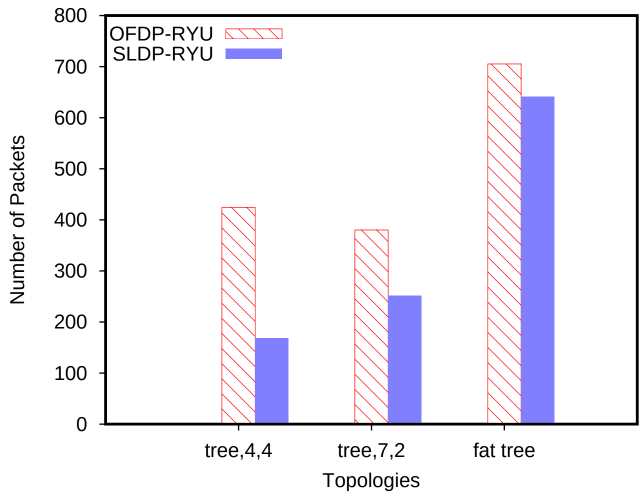


Figure 4.14: Number of packets in link discovery with three topologies

its lightweight and an efficient number of the packet. As shown in Table 4.5, the number of ports in topology Tree,7,2 is lower than other topologies (i.e., Tree,4,4 and Fat tree). In OFDP number of ports is equal to the number of LLDP packets for any cycle. Hence, in the Tree,7,2 topology, less number of packets are being sent and parsed during the link discovery. The same is reflected in Figure 4.15, which shows the lowest overhead for OFDP.

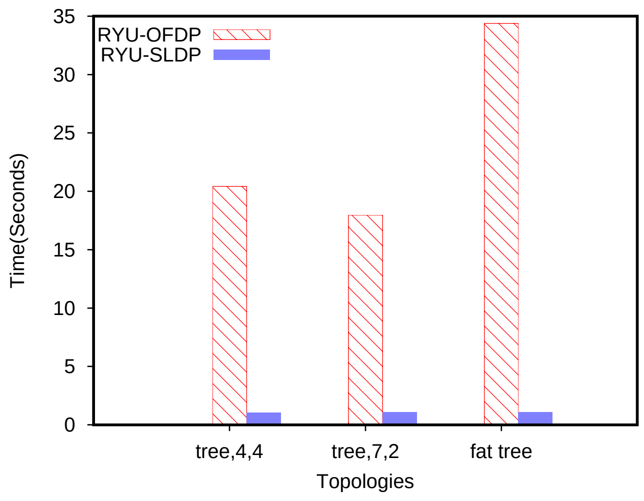


Figure 4.15: Computation overhead for link discovery

Figure 4.16 demonstrates, the time taken in link discovery packet construction. As lighter packet takes lesser time to be constructed. In this figure, there is a clear difference between RYU-SLDP and RYU-OFDP. RYU-SLDP is taking lesser time to construct.

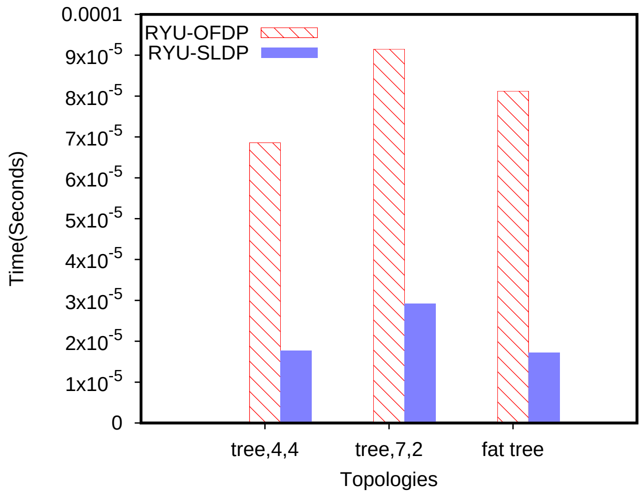


Figure 4.16: Link discovery packet construction overhead

Upon receiving any link discovery packet, it is verified and parsed. Figure 4.17 shows that RYU-SLDP is parsed in smaller time than RYU-OFDP. SLDP use lighter and simple packet which is parsed on a less complex algorithm.

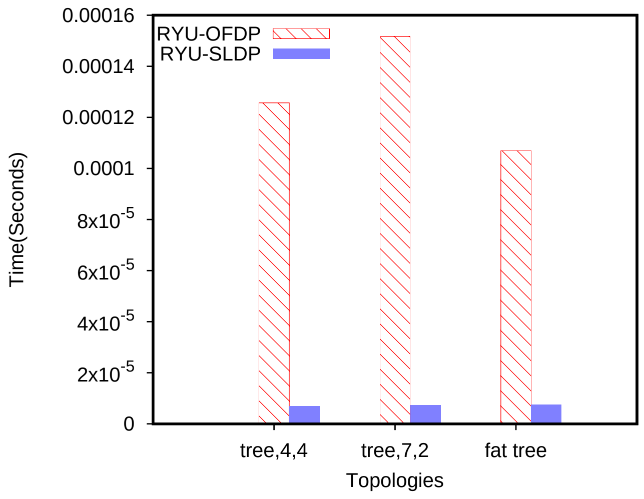


Figure 4.17: Link discovery packet verification overhead

Most of the link discovery implementations in SDN are creating static LLDP or BDDP packets. Later these packets are sent periodically. This kind of propagation has a vulnerability that can be exploited by adversaries[56]. SLDP uses a new packet for each iteration for a fixed port. Figure 4.18 shows event sequence for discovery packet construction and verification. RYU-OFDP create packet once and later use them, while RYU-SLDP generates and verifies SLDP packet alternatively.

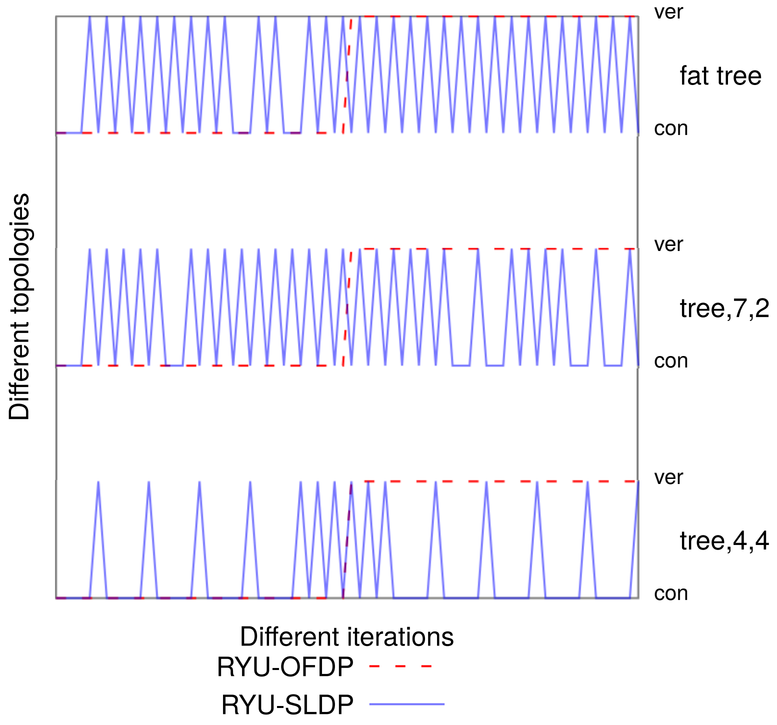


Figure 4.18: Link discovery packet construction and verification sequence

Topology discovery time is the time taken by the controller to build entire topology. It is measured as the time taken from first LLDP or SLDP packet generation to complete topology discovery. Figure 4.19 illustrates RYU-SLDP is performing much better then RYU-OFDP irrespective of the topology. The reason behind it is small packet size and the efficient way of processing them.

One of the key to success for SLDP is the calculation of eligible ports. SLDP only sends SLDP packet to eligible ports. Initially, all ports are eligible ports but later some ports are declared as non-eligible. Figure 4.20 demonstrates eligible ports over time for SLDP with three topologies.

SLDP packets are generated and sent periodically, it also includes installation of flow entry, which allows SLDP packet to reach back to the controller. In traditional implementation of link discovery, a unique flow entry is installed once and it remains forever. It is because link discovery packets are created once in OFDP, and thus the strategy works. With the SLDP, it is not the case, and it is considered as initial overhead for RYU-OFDP only. Figure 4.21 shows the initial

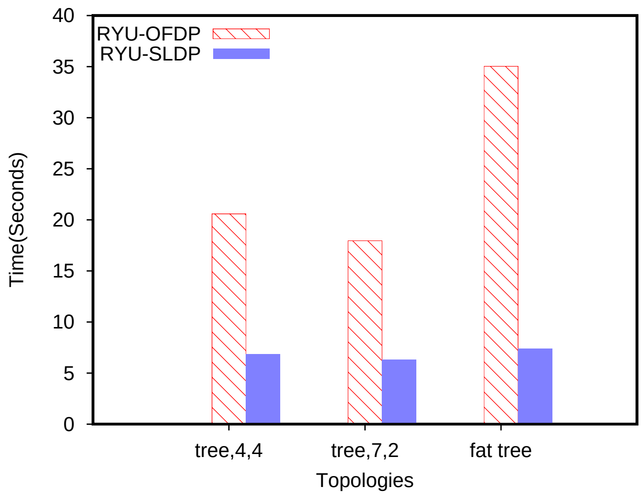


Figure 4.19: Topology discovery time

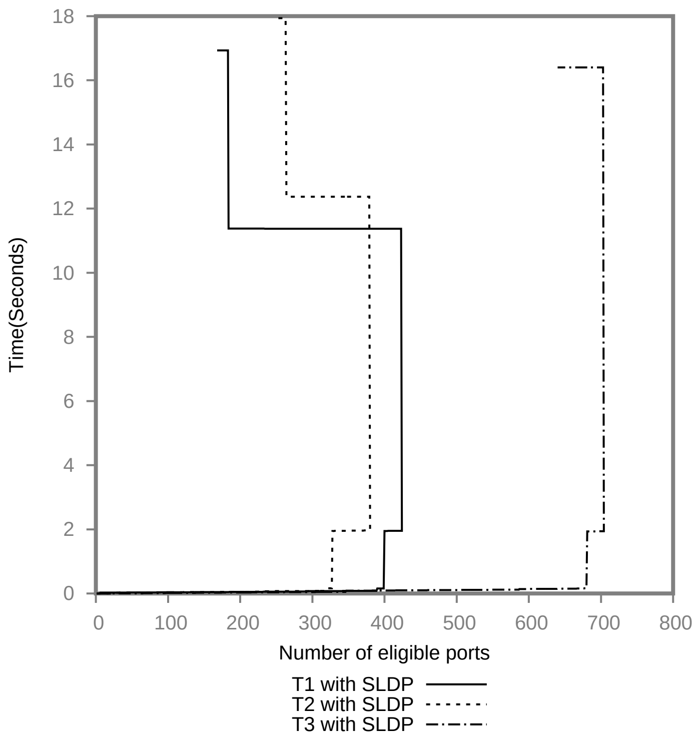


Figure 4.20: Eligible ports over the time

overhead, which is the sum of the initially generated LLDP packets and flow entry installation which is done once for RYU-OFDP implementations. However, in RYU-SLDP, the same is a periodic task. Hence, the initial overburden is limited to RYU-OFDP implementations only. Additionally, Figure 4.15 shows the overhead in link discovery for both RYU-OFDP and RYU-SLDP. RYU-OFDP have initial overhead apart from overhead to link discovery.

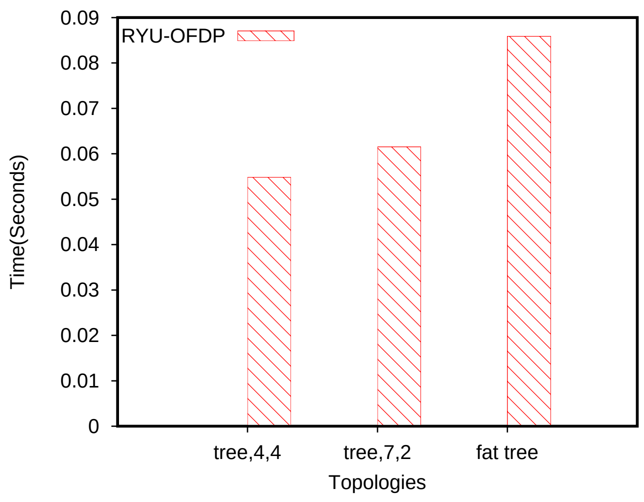


Figure 4.21: Initial overhead in RYU-OFDP

SLDP demonstrates link detection ability with lighter weight packets. Results demonstrate that low number of discovery packets are generated implying a lower time for SLDP packet construction and verification.

4.5 Summary

Link discovery in SDN is crucial for topology-aware applications. This chapter present the motivations i.e. security, lightweight, and efficient, for the design of a new link discovery protocol in SDN. Design and implementation is also discussed in detail. We can summarize as follows.

- A secure, lightweight, and efficient link discovery protocol (SLDP) is discussed for SDN networks. The design of the major components of SLDP which includes a new packet format, system architecture, flow entry structure, and event sequence are explained. To ensure the security against various attacks, SLDP uses a token-based technique that generates random source MAC addresses for SLDP packets, and it uses the randomness to create a flow entry for the SLDP packets.

- A fully implemented SLDP is examined on Mininet emulator with Ryu controller. The performance analysis done on different SDN scenarios show the effectiveness of SLDP regarding prevention and detection of various attacks (e.g., Replay, Flooding, and Poison attacks), computational overhead, topology discovery time, and bandwidth consumption. A comparison of SLDP with OFDP protocol shows the effectiveness of SLDP over the state-of-the-art.

Even if link discovery is secured, a man in the middle attack is still possible. Strange but true. At data link layer, the host is located using the ARP protocol. ARP Poison can be utilized to perform a man in the middle attack. Hence without securing the data link layer host discovery, topology discovery cannot be secured. In the next chapter, an SDN solution is illustrated to detect and mitigate ARP based threats.

Chapter 5

Securing Data Link Layer Host Discovery

This chapter introduces a novel mechanism to detect and mitigate attacks in data link layer based host discovery. In Internet Protocol (IP) based communication, Address Resolution Protocol (ARP) based threats are so prominent that without dealing, secure topology discovery is incomplete. In this chapter, a signature-based detection and mitigation for ARP related threats are discussed. The validation of solution has been done on both the emulated environment using Mininet and real time environment, i.e. production Local Area Network (LAN) with OpenFlow enabled HP5406R switch.

Software Defined Networking (SDN) is a new paradigm of networking which has transformed the traditional way of network management. In SDN, the protocol stack is the same as traditional networking. ARP is used to locate and get the physical address or MAC address of an available IP address. Unavailability of authentication and integrity in ARP communication may lead to exploitation of it for various kind of attacks such as ARP spoofing and ARP Flooding. For example, a malicious machine crafts an ARP reply for which there was no ARP request or it can reply to a request which was originated for some other machine, or it can generate a false ARP request, etc. By doing so, an attacker can perform ARP Poison, ARP Flooding attacks which leads to Denial of Service attacks, Man-in-the-Middle (MitM) attacks or redirection attacks. Existing solutions of ARP spoofing attacks for traditional networks may not give the best result because either solution is based on pre-stored MAC/IP binding or use cryptographic solutions. MAC/IP bindings are created using Dynamic Host Configuration Protocol (DHCP) or Simple Network Management Protocol (SNMP) protocols. In large networks, the number of these bindings are enormous therefore, the look-up time increases. The cryptographic approaches require additional computational overhead in complex cryptographic algorithms. Apart from this, the traditional methods of security are directly applicable but not taking the benefits from the separation of control plane and data plane. In the SDN, the control plane is a programmable plane having a global view of the network. These feathers of controller supported in defending the ARP-based security threats.

This chapter presents *Traf**fic** Pattern Based Sol**u**tion to ARP **R**elated Threats* (**FICUR**), a novel method for verification and detection of ARP-based attacks[17]. The word ‘FICUR’ is a reproduction from the Hindi language i.e. "फिक्र" which generally used for worry or concern. In FICUR, the SDN controller has been extended by a module which gathers the required network parameters. This module also analyzes these parameters to verify and detect ARP based attacks. The key aspect of the proposed approach is that it does not require any authentication check, cryptographic keys, network topology changes or network operator intervention. Instead, FICUR analyzes the current traffic pattern to detect anomaly in ARP packets. The validation of FICUR has been done on both the simulated environment using Mininet and real-time environment using HP switch. It was observed that the method is fast and does adds a limited overhead to the network.

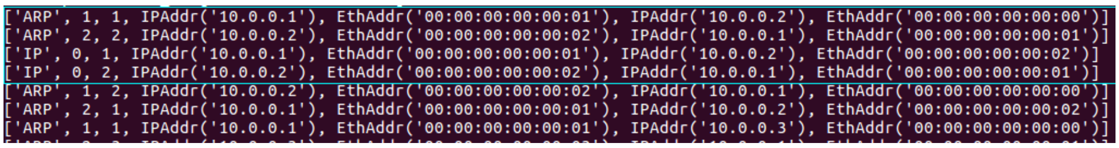
5.1 Foundation

This section explores the directions in which the solution have to focus. Existing techniques to solve Address Resolution Protocol (ARP) based attacks can be broadly classified into two classes. In first class, authors store MAC/IP binding and generate an alert if a mismatch occurs. If detection solution uses Dynamic Host Configuration Protocol (DHCP) database which is very large then lookup process becomes very time consuming and sometimes, DHCP is not even used. The cryptographic solution is provided in other class. In traditional network intermediate devices like switch and router have limited resources so resource intensive solutions are not preferable. In most of these solutions, a host/server other than router or switch takes care of such detection strategies. Suppose if detection happens, network administrator has to perform countermeasure independently.

ARP based threats are prevalent due to lack of authentication and integrity in ARP packets. ARP packet sequence in entire communication can also be considered for detection for ARP based threats. SDN gives the opportunity for detection in a novel way due to two reasons. The first reason is separation of control plane and data plane. The second reason is generic computation capabilities for the control plane. In traditional networking, these reasons are missing. In any genuine communication, ARP request followed with ARP reply. The ARP packet pair is followed with valid Internet Protocol (IP) packets. We can rephrase the statement as hosts are discovered only if needed. This can be used as a detection signature for ARP based threats.

Let us observe some interesting facts about traffic pattern with ARP packets. Figures 5.1, 5.2 and 5.3 are snapshots from the output of customized POX controller module. Column one specifies packet type. The controller module only interested in two types of packets i.e. ARP and IP packet. Column two represents the sub-type of packet i.e. if the type is ARP then sub-type may be '1' for ARP request or '2' for ARP reply. In case of IP packet, sub-type is '0' only. Column three specifies forwarding element event port i.e. port number on which packet has-been arrived. Column four & six are IP addresses while five & seven are Media Access Control (MAC) addresses. For IP packets, these addresses are network layer and data link address while in a case of ARP, these addresses are ARP header parts.

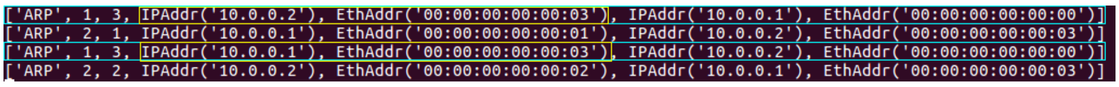
Highlighted box in Figure 5.1 depicts a normal communication. The communication starts with a host with 10.0.0.1 asking what is MAC address of 10.0.0.2.



```
['ARP', 1, 1, IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:01'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:01')]
['IP', 0, 1, IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:01'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02')]
['IP', 0, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:01')]
['ARP', 1, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 1, IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:01'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02')]
['ARP', 1, 1, IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:01'), IPAddr('10.0.0.3'), EthAddr('00:00:00:00:00:00')]
```

Figure 5.1: Normal temporal sequence of IP and ARP packets

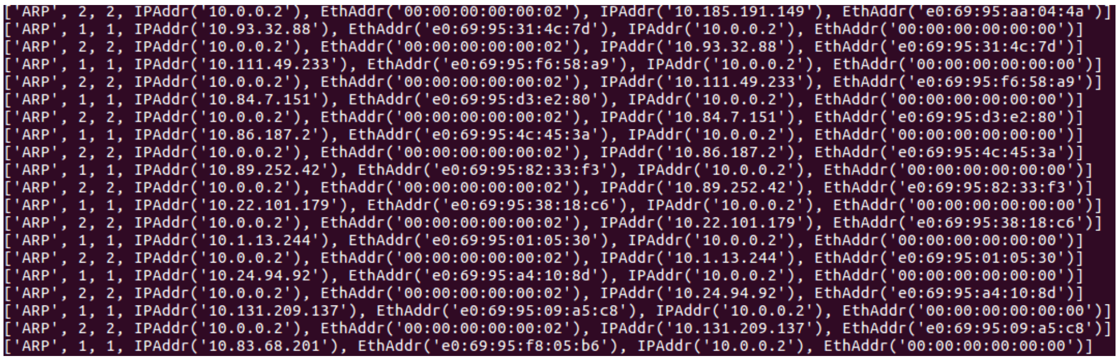
Host at 10.0.0.2 confirms that it is 00:00:00:00:00:02. After obtaining the MAC address, actual IP packet travels from 10.0.0.1 to 10.0.0.2 and subsequently from 10.0.0.2 to 10.0.0.1.



```
['ARP', 1, 3, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:03'), IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 1, IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:01'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:03')]
['ARP', 1, 3, IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:03'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.0.0.1'), EthAddr('00:00:00:00:00:03')]
```

Figure 5.2: ARP packets sequence with Man-in-the-Middle attack

Observation #1: In Figure 5.2, two lines are highlighted, both are ARP requests. The attacker can perform ARP Poison using ARP request only. The first ARP request can be a phrase like “A host with IP address 10.0.0.2 and MAC address 00:00:00:00:00:03 want to know MAC address of a machine having IP 10.0.0.1”. The second ARP request can be a phrase like ”A host with IP address 10.0.0.1 and MAC address 00:00:00:00:00:03 want to know MAC address of a machine having IP 10.0.0.2”. Now there is a catch; both ARP requests claiming different source IP address but with same MAC address. This case is not possible in any situation. Suppose a machine has two different interfaces then, there should be two different IP address. This observation suggests a malicious host trying to perform Man in the Middle attack.



```
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.185.191.149'), EthAddr('e0:69:95:aa:04:4a')]
['ARP', 1, 1, IPAddr('10.93.32.88'), EthAddr('e0:69:95:31:4c:7d'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.93.32.88'), EthAddr('e0:69:95:31:4c:7d')]
['ARP', 1, 1, IPAddr('10.111.49.233'), EthAddr('e0:69:95:f6:58:a9'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.111.49.233'), EthAddr('e0:69:95:f6:58:a9')]
['ARP', 1, 1, IPAddr('10.84.7.151'), EthAddr('e0:69:95:d3:e2:80'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.84.7.151'), EthAddr('e0:69:95:d3:e2:80')]
['ARP', 1, 1, IPAddr('10.86.187.2'), EthAddr('e0:69:95:4c:45:3a'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.86.187.2'), EthAddr('e0:69:95:4c:45:3a')]
['ARP', 1, 1, IPAddr('10.89.252.42'), EthAddr('e0:69:95:82:33:f3'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.89.252.42'), EthAddr('e0:69:95:82:33:f3')]
['ARP', 1, 1, IPAddr('10.22.101.179'), EthAddr('e0:69:95:38:18:c6'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.22.101.179'), EthAddr('e0:69:95:38:18:c6')]
['ARP', 1, 1, IPAddr('10.1.13.244'), EthAddr('e0:69:95:01:05:30'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.1.13.244'), EthAddr('e0:69:95:01:05:30')]
['ARP', 1, 1, IPAddr('10.24.94.92'), EthAddr('e0:69:95:a4:10:8d'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.24.94.92'), EthAddr('e0:69:95:a4:10:8d')]
['ARP', 1, 1, IPAddr('10.131.209.137'), EthAddr('e0:69:95:09:a5:c8'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
['ARP', 2, 2, IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:02'), IPAddr('10.131.209.137'), EthAddr('e0:69:95:09:a5:c8')]
['ARP', 1, 1, IPAddr('10.83.68.201'), EthAddr('e0:69:95:f8:05:b6'), IPAddr('10.0.0.2'), EthAddr('00:00:00:00:00:00')]
```

Figure 5.3: ARP packets during ARP Flooding

Observation #2: Figure 5.3 describes the scenario for ARP Flooding attack in which only ARP requests/replies are shown. This is important to note if any genuine communication happens, it requires ARP to resolve the physical address. That Communication then followed by valid IP packets. We can say, no genuine communication completes generally with ARP packets only. So if a history of packets in network suggests that only ARP communication is happening, it will a possible alert for ARP Flooding.

Observation #3: In Figure 5.2 and 5.3, It's worth noting that many ARP requests are coming but at same event port on the switch. In any network environment, hosts are connected with the switch on different ports. Above observation also gives an idea that something went wrong in the network. After observing packets, a reader can conclude that attacker is attached to port number two of the forwarding element.

Two ARP requests holding common source MAC address with different source IP addresses creates suspicion. This doubt helps to detect ARP Poison for Man in the Middle attack. The number of ARP request/reply packets without following related IP packets leads to another doubt. This doubt helps to detect ARP Flooding attack. Forwarding elements' port information helps to locate and quarantine malicious host.

Assumptions:

In next section, detailed design modules of FICUR are introduced. The desired working of this solution depends on few assumptions as follows.

- The controller and running applications are malware free.
- The switches work on OpenFlow specifications.
- The switches are malware free.

5.2 FICUR Design

In the previous section, some important observations related to Address Resolution Protocol (ARP) traffic patterns are discussed. These observations help to design the desired solution. Design modules for FICUR are examined in this section. Figure 5.4 illustrates a block diagram for FICUR. FICUR is divided into three parts i.e. attack detection module, attack source localization module, and mitigation module. Attack detection module takes ARP and IP packets into inspection and separates ARP Poison and ARP Flooding associated packets. Attack source localization module extracts and calculates the source of attack information. In mitigation module, the same information is used to quarantine the malicious host.

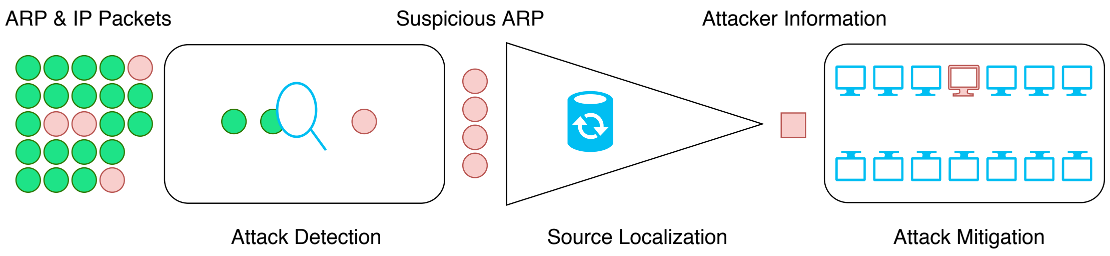


Figure 5.4: FICUR block diagram

The Software Defined Networking (SDN) can provide a global view of the network so the network administrator can gather the required data and analyze it to reveal the possibility of attacks. Once the attack has been identified, security policies are reinforced using the SDN programmbility.

5.2.1 Attack Detector

Figure 5.5 represents flow diagram of FICUR attack detection module. The detection module is only interested in ARP request packets or IP packets. On receiving such packets, a _logDetail file is updated, which is used to identify any suspicious activity. If any two ARP packets with same source IP address and different MAC addresses arrived at the controller, this leads to Man in the Middle attack detection. In another scenario, if multiple ARP packets are received without any IP packet, this leads to detection of ARP Flooding attack. Detection of either ARP Poison or ARP Flooding attack, deletes _logDetail file.

Algorithms can be considered for more detail to detect ARP Poison and ARP Flooding attacks. Algorithm 12 helps to create a data structure which is needed in the detection of both attacks. Algorithms 13 and 14 detect ARP Poison and

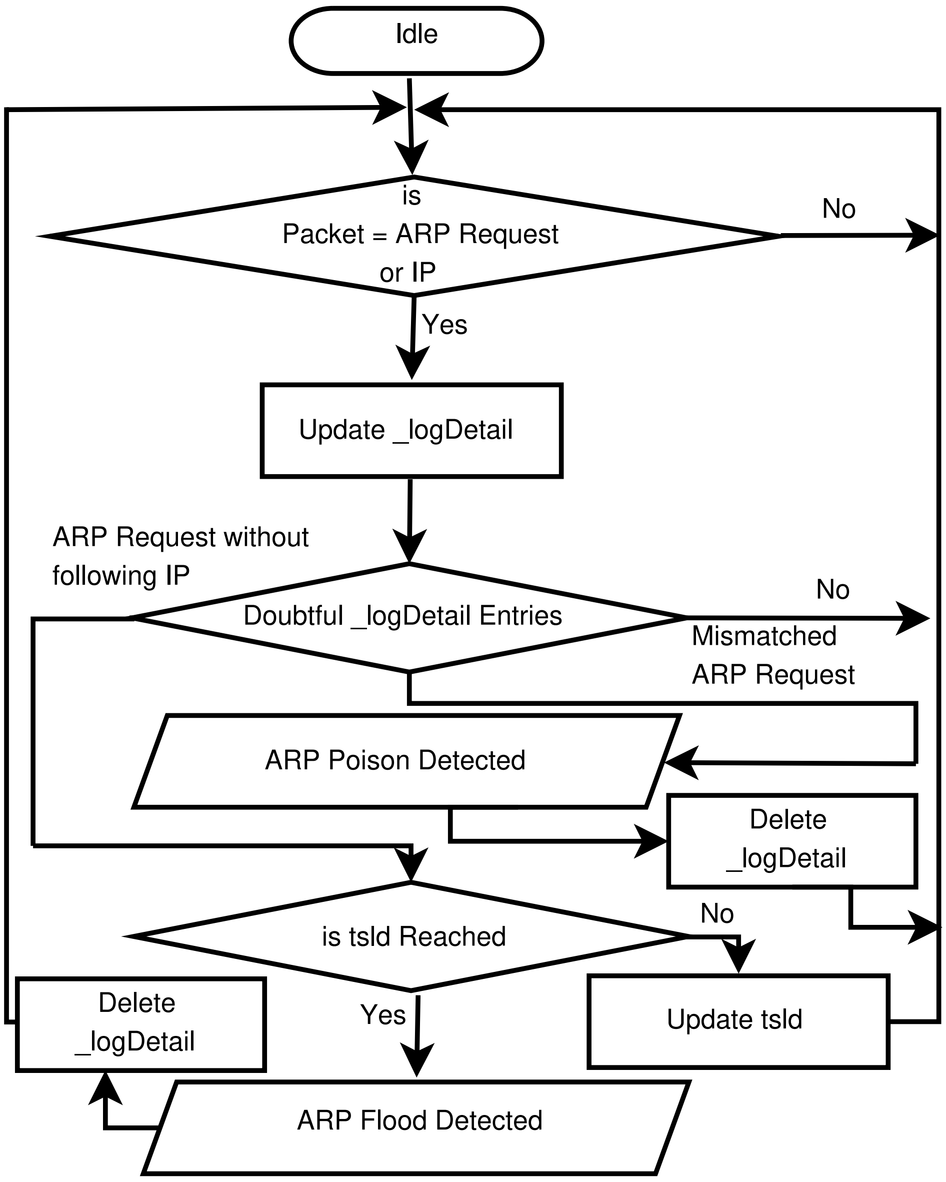


Figure 5.5: Flowchart of attack detection module in FICUR

ARP Flooding attack respectively. Few abstract functions are used in the algorithms. EXTRACT function extracts values that are specified on left side of the statement from the given input. subType (ARP sub type), src_ip (source IP address), src_mac (source MAC address), dst_ip (destination IP address), dst_mac (destination MAC address) are fields of ARP packet header. UPDATE function adds provided objects in _logDetail as one string.

Arguments to Algorithm 12 are Ethernet frame header, forwarding element physical port and log detail file. The log detail file consists of logs of the received packet i.e., IP or ARP. Each log comprises seven entries. In case of IP packet seven entries are string ('IP'), number(0), source IP address, source MAC address, destination IP address, and destination MAC address. For ARP request packet log entries are string ('ARP'), number(1), source protocol address, source hardware address, destination protocol address, and destination hardware address. For ARP log entries, details are extracted from payload and not from Ethernet header. READ function reads the _logDetail file and creates a list of strings. Algorithms 13 and 14 i.e. isARPPoison and isARPFlood uses the list to detect the attacks. Once any ARP attack detected REMOVE function deletes the _logDetail file.

Algorithm 12 Extract information to detect ARP Poison & ARP Flooding

Require: Ethernet frame header(_efh) which comes as PacketIn event parameter, Event port(_port) & a file (_logDetail) to store packet header and event port details.

```
 1: procedure ARPSecurityStructure($\_efh$, $\_port$, $\_logDetail$)
 2:     dst_mac, src_mac, type, payload ← Extract($\_efh$)
 3:     if type='IP' then
 4:         src_ip, dst_ip ← Extract($payload$)
 5:         Update('$IP'$, '$0'$, $\_port$, $src\_ip$, $src\_mac$, $dst\_ip$, $dst\_mac$)
 6:     else if type='ARP' then
 7:         subType, src_pip, src_hmac, dst_pip, dst_hmac= Extract($payload$)
 8:         if subType = '1' then
 9:             Update('$ARP'$, $subType$ , $\_port$, $src\_pip$, $src\_hmac$, $dst\_pip$, $dst\_hmac$)
10:             listd ← Read($\_logDetail$)
11:             if isARPPoison($listd$) = $True$ then
12:                 Generate Alert for ARP Poison
13:                 Remove($\_logDetail$)
14:             end if
15:             if isARPFlood($listd$) = $True$ then
16:                 Generate Alert for ARP Flooding
17:                 Remove($\_logDetail$)
18:             end if
19:         end if
20:     end if
21: end procedure
```

Algorithm 13 is a detailed outline for ARP Poison attack. isARPPoison checks ARP entries for same source IP address and different MAC address. The function i.e. isARPPoison is called with list of strings which is examined for any mismatch i.e., same IP different MAC. For this, only ARP strings from list is selected.

Algorithm 13 Detect ARP Poison attack.

Require: List of IP and ARP request packet header detail(_listd)

```
function isARPPoison($\_listd$)
    for $item$ in $\_listd$ do
        if $item[0]$ = ‘ARP’ and $item[1]$ = ‘1’ then
            for $eachItem$ in $\_listd$ do
                if $eachItem[0]$ = ‘ARP’ and $eachItem[1]$ = ‘1’ and $eachItem[4] = item[4]$ and $eachItem[3]$ != $item[3]$ then
                    return $True$
                end if
            end for
        end if
    end for
end function
```

Algorithm 14 explains detection of ARP Flooding attack. If number of ARP packets without following IP packets exist in log file , is used as Flooding attack detection. In isARPFlood function getObservedThreshold gives average of observed thresholds from history. The isARPFlood function checks if the number of ARP requests for which no IP communication exists is more than threshold i.e. ‘tsld’ then it indicates the possibility of ARP Flooding.

Threshold calculation can be understood with Figure 5.6. A window is to be filled with received flags i.e. the number of IP packets following any ARP packet. Total zero flags are calculated and averaged with the previous result. This threshold is to be provided to the Algorithm 14 for ARP flooding attack detection. In the first iteration, there is no previous result. Therefore, the previous result is considered equal to the present calculated result.

5.2.2 Attack Source Localization

Security measures for any threat can be prevention, detection, and detection with mitigation. The FICUR considers detection with mitigation as a security measure. For mitigation, FICUR locates vulnerable ports and prevent further ARP request packets on same ports. For ARP Poison two ARP requests are generated with the

Algorithm 14 Detect ARP Flooding attack.

Require: List of IP and ARP request packet header detail(_listd)

```
1: function isARPFlood(_listd)
2:     tsld ← getObservedThreshold( )
3:     for item in _listd do
4:         flag ← 0
5:         if item[0] = 'ARP' and item[1] = '1' then
6:             for eachItem in _listd do
7:                 if eachItem[0] = 'IP' and eachItem[2] = item[2] and eachItem[3] = item[3] and eachItem[4] = item[4] and eachItem[5] = item[5] and eachItem[6] = item[6] then
8:                     flag ← flag + 1
9:                     setThreshold(flag)
10:                    if flag $\neq$ 0 then
11:                        tsld ← tsld -1
12:                    end if
13:                    if tsld $\geq$ 0 then
14:                        return True
15:                    end if
16:                end if
17:            end for
18:        end if
19:    end for
20: end function
```

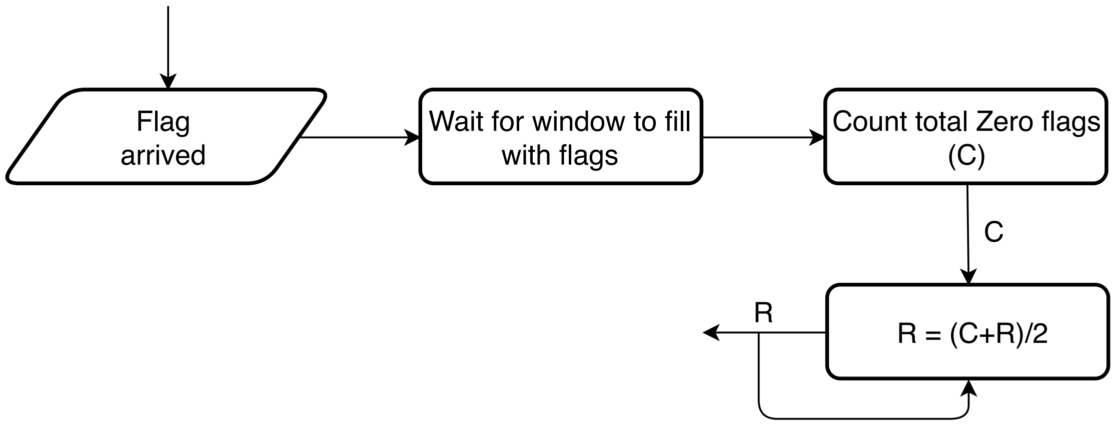

Flag arrived
Wait for window to fill with flags
Count total Zero flags (C)
C
R
R = (C+R)/2

Figure 5.6: Threshold calculation

different IP addresses and common MAC address. For ARP Flooding, a number of ARP request packets without following IP packets crosses the threshold value. Figure 5.7 illustrates source localization in FICUR for both ARP Poison and ARP Flooding. In the case of ARP Poison detection, event port information from both

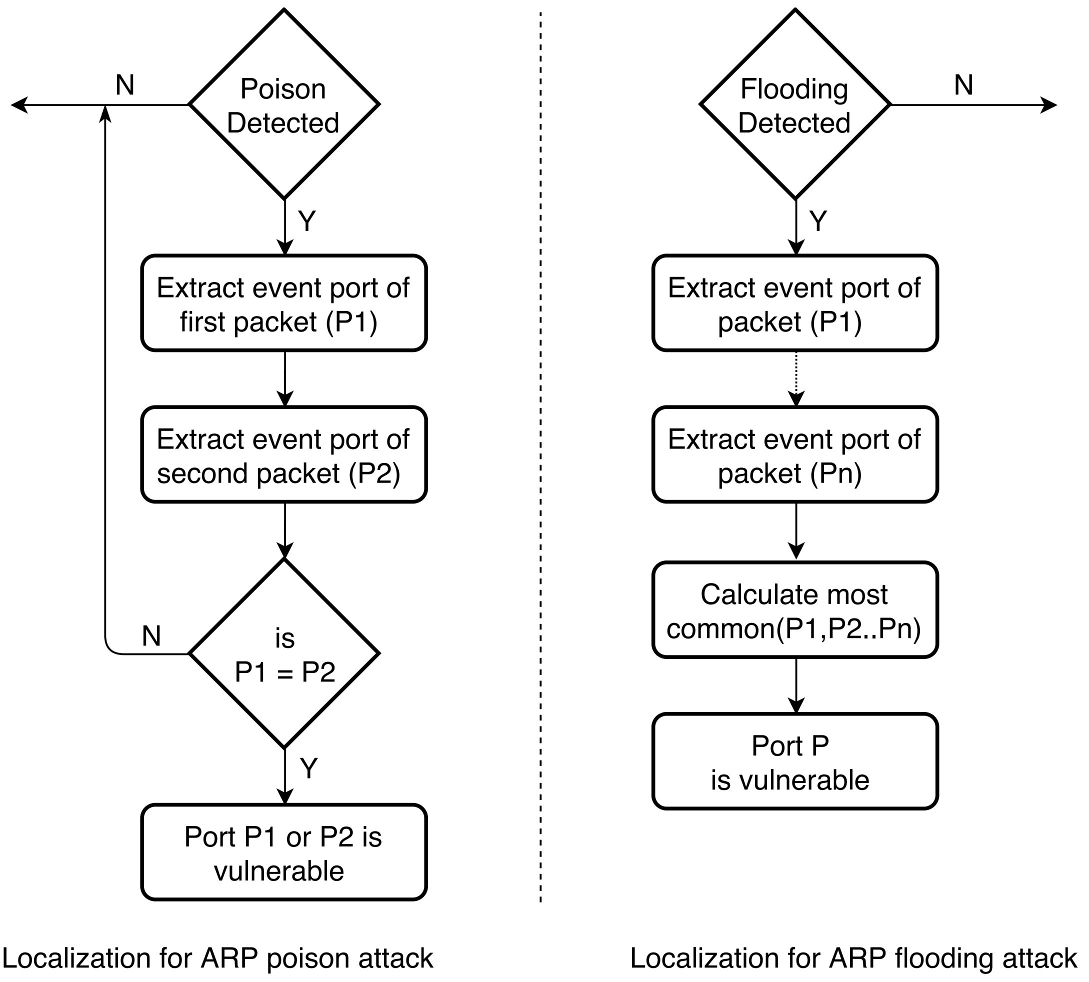

Poison Detected
N
Y
Extract event port of first packet (P1)
Extract event port of second packet (P2)
is P1 = P2
N
Y
Port P1 or P2 is vulnerable
Localization for ARP poison attack
Flooding Detected
N
Y
Extract event port of packet (P1)
Extract event port of packet (Pn)
Calculate most common(P1,P2..Pn)
Port P is vulnerable
Localization for ARP flooding attack

Figure 5.7: Attack source localization

malicious ARP packets is extracted. If both ports are same, it infers that port is vulnerable for ARP communication. In case of ARP Flooding attack, single suspicious ARP request packet is examined for event port. This port declared as vulnerable for ARP Flooding attack.

5.2.3 Attack Mitigation

After successful detection and attack source localization, any strategy to mitigation can be done on the fly because SDN supports dynamic programming. Various strategies can be applied e.g. completely block the port, block the port for outgoing communication, block the port for ARP communication, or block the port for ARP request only. Each has distinguished impact and can support with SDN programmability. In FICUR, ports are blocked for outgoing ARP request packet only. On blocking of outgoing ARP request, any previous communication will not hinder. Any ARP request for a host attached to vulnerable port also serve. Any new ARP request from the vulnerable port is dropped silently. Figure 5.8 illustrates the same.

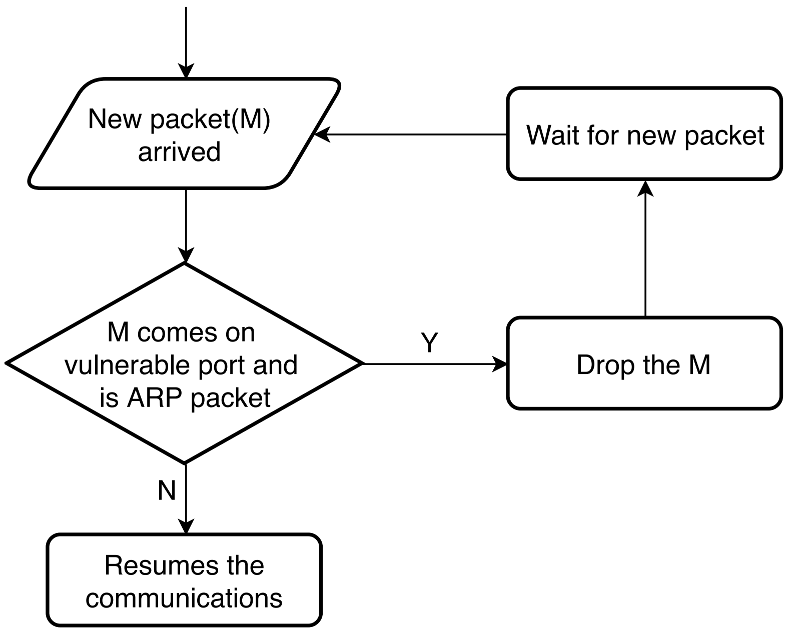

New packet(M) arrived
Wait for new packet
M comes on vulnerable port and is ARP packet
Y
Drop the M
N
Resumes the communications

Figure 5.8: Mitigation flow diagram

5.3 Implementation

Validation of any proposal depends on the working prototype and theoretical proof of correctness. This section discusses the experimental setup, working conditions, used resources, and topology.

5.3.1 Experimental Setup

In FICUR, an application program is written in python which restrain the malicious user to poison the ARP entries for the legitimate hosts lying on the same network segment. An attack module is also written in python, to perform the attacks. These attacks are also validated in the live network.

Various resources used in the experiment is listed in Table 5.1. For the implementation of detection, we choose Mininet[90] environment running on Ubuntu

16.04LTS 64 bit 8GB RAM. OpenFlow enabled HP5406R switch attached with hosts and controller is used to perform attack and detection. Each host has core i5 processor with 8GB RAM and 64-bit Ubuntu 16.04LTS. The POX[8] controller is used to run proposed module.

Resource	Configuration
Test-bed	Mininet(emulation) and HP5406R
Victim/Attacker OS	UBUNTU 16.04$LTS(64bit)^*$
Victim configuration	4CPUs and 8 GB
Attacker configuration	4CPUs and 8 GB
Controller	POX
Attack traffic	150000 Packet/Sec
Software switch	OpenVSwitch(2.5.0)
Network	1 Gbps

Table 5.1: Experimental environment for FICUR

The topology used for experimentation purpose is shown in Figure 5.9. It consists of a few hosts attached to OpenFlow enabled switch. Host M can act as an attacker while host A and B are legitimate machines. To check the proposed solution, we run two cases on the same topology. For the first case, M does not perform any malicious activity. In second case, M was enabled with our attack module to create ARP Flooding and ARP Poison (MitM) attack scenarios. MitM module requires two target machine to perform the attack. This module sends two requests to the specified targets to poison their ARP table. ARP Flooding module needs only one target machine and floods the ARP table of that machine with a large number of ARP request packet.

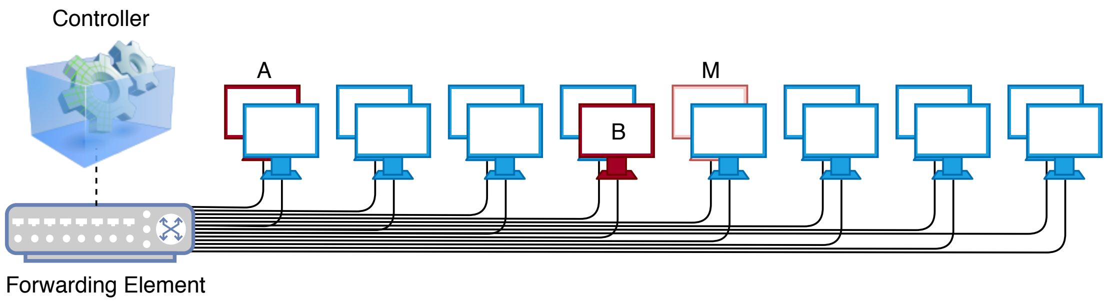


Figure 5.9: Experiment topology

In non-attack scenario, it is observed that normal communication occurs without any disturbance. While in case of attack, target hosts are seen with corrupt ARP table entries. FICUR not only identifies the spoofed ARP messages but also filters these messages at the identified port. After sending a few spoofed ARP packets to the victim, the attacker is not able to send the subsequent false packets. Because all false packets will be denied.

5.3.2 Performance Metrics

FICUR is detection and mitigation based solution for ARP based security threats. Before understanding results, a brief on performance metrics is presented.

Attack Detection Time: It is a time from the first suspicious packet arrived to a conclusion i.e., attack detected. ARP Poison and ARP Flooding attack detect using different approaches, hence varies in detection time. A good algorithm always takes minimum time for detection. Early detection leads to effective mitigation.

Computational Overhead: Every security extension creates an overhead in terms of CPU cycles. ARP Poison and ARP Flooding detection and mitigation time are computed from various observations. These observations are averaged to get a final overhead.

5.3.3 Results and Discussion

Few results increase the confidence over functional correctness of attack scenarios and proposed solutions. Mostly, experiments are done on both Mininet and live test bed, i.e., using switch HP5406R. Figure 5.10 illustrates a conventional scenario and ARP Flooding attack scenario. Here, a number of ARP packets received at the controller is plotted against time. Initially, a regular ARP communication is shown. But after some time attack script started to show the difference in the pattern. In Figure 5.10 a sharp change in slope in both curves shows the starting of ARP Flooding attack. Standard communication rate at which ARP request packets are reaching to the controller is less than after the attack. Due to scale variation, zoomed plots are also shown for in-depth observations.

Figure 5.11 also illustrates the manifestation of the ARP Flooding attack. It

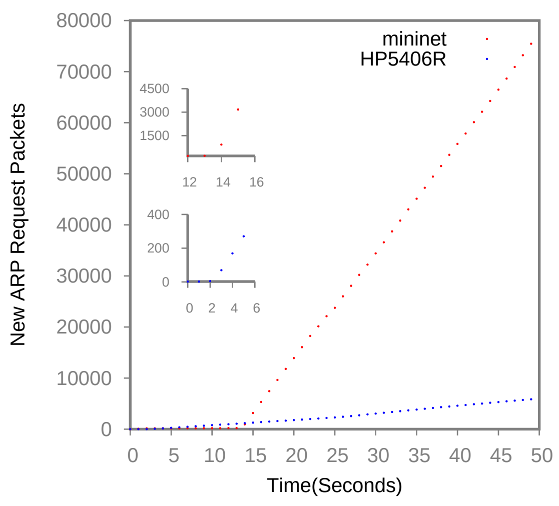


Figure 5.10: Arrival of every ARP request at controller

exhibits the number of MAC entries in infected hosts over time. Initially, the rate at which MAC entries are installed in the infected host is steady. ARP Flooding can easily be identified by the sudden rise in the rate at which MAC entries are stored in ARP cache.

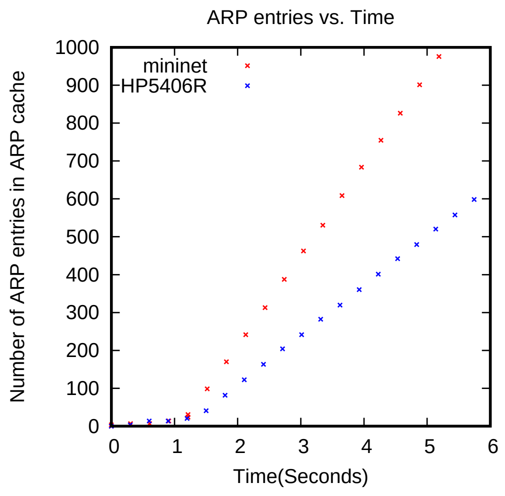


Figure 5.11: Number of ARP entries on affected machine

In Figures 5.10 and 5.11, actual hardware performs slower than Mininet. In Mininet, packets are traveled in an emulated environment, but in the case of actual hardware, the packet must pass through network interfaces which introduce transmission and propagation delays.

FICUR successfully detect both ARP Poison and ARP Flooding. Figure 5.12

demonstrates detection time of these attack in microseconds. ARP Poison and ARP Flooding attack are detected in 16056, 42840 and 422517, 1028160 microseconds respectively. All these statistics are taken with both Mininet, and HP5406R OpenFlow enabled switch. These numbers are averaged from different observations.

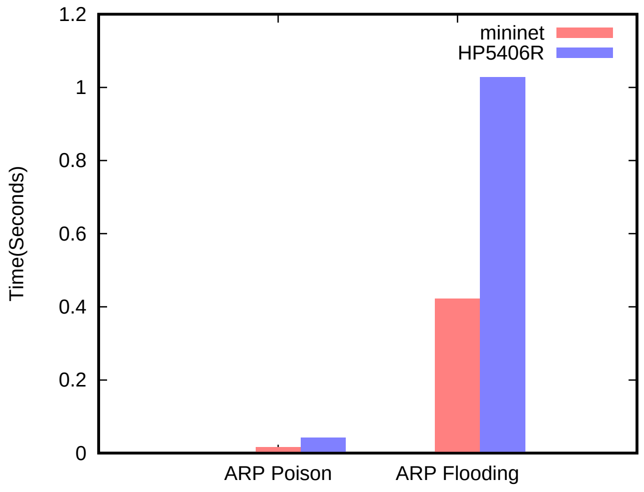


Figure 5.12: Attack detection time

Any security extension comes with the cost. In the default configuration, the controller has to install flow entries for ARP communication. In FICUR extension, extra processing is performed at the controller. Figure 5.13 depict the total overhead due to FICUR algorithms.

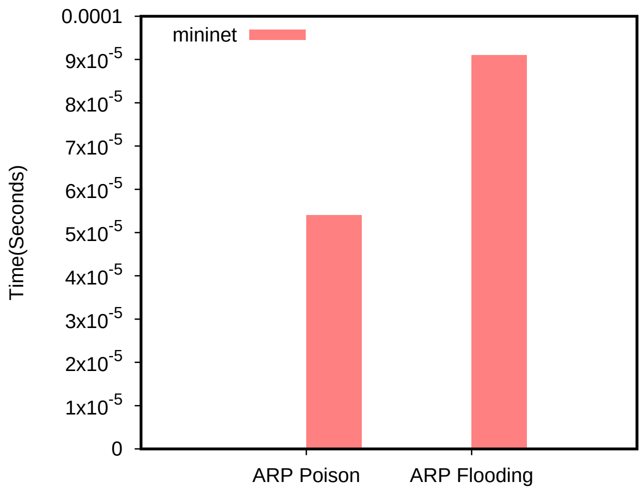


Figure 5.13: FICUR computation overhead

Each IP and ARP request packet reaches to the controller for the attacks detection. In the switched environment where the number of hosts is limited in a network segment to 48. The overall performance of the controller will be sustainable. However, for a large network with multiple segments, the controller performance may have serious challenges. One more aspect for many packets reaches to the controller is prone to Denial of Service (DoS). However, this solution may be employed along with DoS prevention, detection, and mitigation system. This collaboration will help to rule out DoS threat.

Each incoming packet is logged in a log file. The log file is the data structure that is continuously monitored, used and updated by the controller. Whenever an attack is detected, the data structure is deleted, and a new copy is made available for further incoming packets. Even if the attacks has not happened, this data structure will reduce its size by removing old entries.

FICUR detects ARP Poison for Man in the Middle attack. If an attacker only poisons a single host, the current version of FICUR will not detect it. Now consider the case that an attacker infects a single host which implies that the attack affects nothing. And as we all know that “There are no free lunches”, which means to perform an attack, there should be a specific benefit.

FICUR detect ARP Flooding attack by monitoring ARP traffic followed by IP traffic. If any port producing ARP traffic without following IP traffic leads to detection followed by mitigation. Consider the case, that the attacker also produces IP traffic along with ARP traffic to fool the solution. To generate IP traffic the attacker has to invest resources. But this will lead to degradation of the attack magnitude.

5.4 Summary

Nowadays SDN is considered a unique networking paradigm. Existing communication protocols are same, but devices can use them in a novel way to solve problems. In this chapter ARP based threats are discussed. ARP has no built-in mechanism to ensure authentication and integrity, hence ARP-based attacks are serious attacks even for SDN. Although several different mechanisms are present in literature to detect such kind of attack but most of them are resource intensive. The proposed method utilizes SDN programmability with present ARP format. In the nutshell, the chapter can summarized as follows.

- This chapter discusses a novel mechanism to detect ARP based attacks effectively with restricted overhead in space and time. The proposed method utilizes the match and filter ability of SDN to detect and mitigate the ARP-based attacks.

- This chapter also discusses experimental evidence. The Mininet emulator with POX controller is used to validate proposal. Hardware test bed which consist of HP5604R OpenFlow enabled switch also used to perform the experiments. Obtained detection time and overall computation overhead assure to use in a small size network.

Conclusions are limited to known facts. Previous chapters capture some facts about topology discovery. In the next chapter few conclusions are made on these facts. Future work for the reader is also suggested in the next chapter.

Chapter 6

Conclusions and Future Scope

Software Defined Networking (SDN) gives unique opportunity to handle communication needs and provides effective, scalable and reliable solutions. To achieve this more effectively a secure, efficient and lightweight topology discovery is important. This chapter ends the thesis with conclusions. However, conclusions are always restricted to known facts. Future directions are also illustrated for the reader of the thesis.

6.1 Conclusions

In the thesis, four main contributions are made namely an empirical state of the art, preventive solution, i.e. TILAK, A new protocol, i.e. SDN Link Discovery Protocol (SLDP) and detection and mitigation for Address Resolution Protocol (ARP) based threat i.e. FICUR. From each such contribution, few conclusions are drawn as follows.

- Several attacks on link discovery such as Link Layer Discovery Protocol (LLDP) Poison, LLDP Flooding, and LLDP Replay attacks are possible on current deployments. In particular, the link discovery process is examined on various SDN controllers including POX, Ryu, OpenDayLight, Floodlight, ONOS, Beacon, and HPE-VAN. Our work confirms that each aforementioned controller is vulnerable to one or more types of LLDP attacks.

- Available literature for secure link discovery is inadequate. From the literature, various gaps are identified, i.e. insecure, if secure then costly, or maybe necessitating to change existing OpenFlow specification.

- A novel solution called TILAK can prevent LLDP based security threats. It is the only preventive solution among the array of state of the art solutions. TILAK uses Media Access Control (MAC) address to binding randomness to assure prevention for all the discussed attacks. Comparing with the existing solutions, TILAK produced resource penalty in negative. The theoretical analysis is also helping to prove the correctness of TILAK.

- The current version of link discovery is using borrowed LLDP frame and generated LLDP packets are more than required. Few fields in LLDP packet are of no use in link discovery. It also creates a vulnerable environment to get exploited. Possible causes for these vulnerabilities are the use of static LLDP frame and lack of authentication, integrity.

- SDLP is a secure, efficient and lightweight link discovery protocol for SDN. SLDP is probably secure link discovery process against Poison, Replay, and Flooding attacks. SLDP uses token-based prevention with lower number of packets to provide link discovery. SLDP provides a design of discovery packets which are lightweight. Resource penalty for overall computation is far less than the original implementation.

- Without securing data link layer host discovery, secure topology discovery is hard to imagine. At the data link layer, ARP based host discovery is performed. ARP is vulnerable to ARP Poison and Flooding attacks. ARP has no built-in mechanism to ensure authentication and integrity hence ARP-based attacks are serious attacks even for SDN.

- FICUR utilizes the match and filter-ability of SDN to detect and mitigate the ARP-based attacks. This solution neither does put any extra overhead on the network nor it requires any changes in OpenFlow specifications.

6.2 Future Directions

Nothing is perfect. Even after investing a long time to find facts and make conclusions in any research, there are possible improvements. The thesis also suggests few future directions to make topology discovery more secure, efficient and robust.

- The controller assumes all received information is genuine. In link discovery, if the controller or switch is malicious, it is more challenging to prevent, detect or mitigate Poison, Flooding and Replay attacks. Consider the case when a switch is sending false information or the controller is creating false information database. Then, detection of any trust breach is interesting.

- The controller assumes that connected switch follows OpenFlow specifications. But if not, then what? How the controller discovers the topology information even after this uncertainty.

- If two hosts attached to different switches started before switch awakes, LLDP based threat could happen in link discovery. Although this is a low probability event but can this resolved? If it is, SLDP can be more robust.

- In SLDP, packets generation, flow entry installation etc. are periodic tasks. Optimal time for SLDP period is yet to prove by experiments. Optimal time will reduce the overhead and make it less vulnerable to attacks.

- Distributed link discovery is challenging, and if it will be possible, it will reduce the overhead of a single controller.

- Wired network is mostly stable, the need of periodic packets seems avoidable. Lesser the packets, make link discovery more efficient. We can also explore the possibility of packet less link discovery process.

- FICUR is a detection and mitigation system. In the attack scenario, a machine is busy with false packets and in the same time detection system is also running. Hence, a preventive solution for different test cases is needed.

- Any ARP based attack prevention, detection, mitigation system is challenging if the controller or a switch is malicious or is performing arbitrary.
- FICUR can be improved if research will continue to focus on the optimization of threshold as well as by incorporating some extensive diagnosis capabilities to the present scheme.
- FICUR is not scalable for the enterprise network. So we can also look forward to make it scalable, with a lesser number of ARP, IP packets to the controller.
- FICUR is ARP based attack detection and mitigation system. However, a combination with DoS detection system can be investigated.

Appendix A

Details of Examined Controllers

In this appendix, details for the various examined controller are listed. All inspected files are at the leaf of each directory tree.

POX(0.2.0)

```
/pox/pox/openflow
 └── discovery.py
      ├── class Discovery
      ├── class LLDPSender
      ├── class LinkEvent
      └── class Link
```

Ryu(4.12)

```
/ryu/ryu/topology
 └── switches.py
      ├── class LLDPPacket
      └── class LinkState
```

Floodlight(1.2)

```
/floodlight
 └── floodlight.jar
      └── class LinkDiscoveryManager
```

ONOS(1.9.0)

```
/onos-1.9.0
 └── onos-providers-lldp-1.9.0.jar
      └── class LinkDiscovery
```

OpenDayLight(3.0.7)

```
/distribution-karaf-0.5.2-Boron-SR2
 ├── lldp-speaker-0.3.2-Boron-SR2.jar
 │    ├── class LLDPSpeaker
 │    └── class LLDPUtil
 ├── topology-lldp-d..ry-0.3.2-Boron-SR2.jar
 │    ├── class LLDPDiscoveryUtils
 │    ├── class LLDPActivator
 │    └── topology-lldp-discovery.xml
 ├── sal-0.7.0.jar
 │    ├── class LLDP
 │    └── class LLDPTLV
 └── liblldp-0.11.2-Boron-SR2.jar
      └── class CustomTLVKey
```

Beacon(1.0.4)

```
/beacon-1.0.4
 └── net.beaconcontroller.topology_1.0.4.jar
      └── class TopologyImpl
```

HPE-VAN(2.7.18)

```
/hpe-van-sdn-ctlr
 └── sdn-ctl-oflinkdisco-2.7.18.jar
      ├── class BddpPacketBuilder
      ├── class DiscoveryPacketListener
      └── class BddpPacketInfo
```

Appendix B

List of Publications

Journal

J1 Ajay Nehra, Meenakshi Tripathi, M.S.Gaur, Ramesh Babu Battula and Chhagan Lal. "SLDP: A Secure and Lightweight Link Discovery Protocol for Software Defined Networking". In: *Computer Networks.* Elsevier, 2018.

J2 Ajay Nehra, Meenakshi Tripathi, M.S.Gaur, Ramesh Babu Battula and Chhagan Lal. "TILAK: A Token based Prevention Approach for Topology Discovery Threats in SDN". In: *International Journal of Communication Systems.* Wiley, 2018.

J3 Prashant Kumar, Meenakshi Tripathi, Ajay Nehra, Mauro Conti and Chhagan Lal. "SAFETY: Early Detection and Mitigation of TCP SYN Flood Utilizing Entropy in SDN". In: *Transactions on Network and Service Management.* IEEE, 2018.

Conference

C1 Ajay Nehra, Meenakshi Tripathi and M.S.Gaur. "'Global View' in SDN: Existing Implementation, Vulnerabilities & Threats". In: *Proceedings of the* 10th *International Conference On Security Of Information And Networks.* SIN'17. Jaipur, India: ACM, 2017.

C2 Ajay Nehra, Meenakshi Tripathi and M.S.Gaur. "FICUR: Employing SDN Programmability to Secure ARP". In: *Proceedings of the* 7th *Annual Comput-*

ing and Communication Workshop and Conference. CCWC17. Las Vegas, USA: IEEE, 2017.

C3 Ajay Nehra, Meenakshi Tripathi and M.S.Gaur. “Requirement Analysis for Abstracting Security in Software Defined Network”. In: *Proceedings of the* 8th *International Conference on Computing Communication and Networking Technologies.* ICCCNT17. IIT Delhi, India: IEEE, 2017.

Bibliography

[1] Ann Bosche et al. *Unlocking Opportunities in the Internet of Things*. Aug. 7, 2018. URL: `https://www.bain.com/insights/unlocking-opportunities-in-the-internet-of-things/` (visited on 01/01/2019).

[2] *Software-defined networking (SDN) market revenue worldwide from 2016 to 2022 (in billion U.S. dollars)*. URL: `https://www.statista.com/statistics/668394/worldwide-software-defined-networking-market-revenue/` (visited on 01/01/2019).

[3] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76.

[4] B. A. A. Nunes et al. "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks". In: *IEEE Communications Surveys Tutorials* 16.3 (2014), pp. 1617–1634.

[5] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. "Sdn Security: A Survey". In: *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. Nov. 2013, pp. 1–7.

[6] Ajay Nehra, Meenakshi Tripathi, and Manoj Singh Gaur. "Requirement analysis for abstracting security in software defined network". In: *2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. July 2017, pp. 1–8.

[7] P. Kumar et al. "SAFETY: Early Detection and Mitigation of TCP SYN Flood Utilizing Entropy in SDN". In: *IEEE Transactions on Network and Service Management* 15.4 (Dec. 2018), pp. 1545–1559.

[8] *POX*. URL: https://github.com/noxrepo/pox (visited on 01/01/2019).

[9] *RYU*. URL: https://osrg.github.io/ryu/ (visited on 01/01/2019).

[10] *OpenDaylight*. URL: https://www.opendaylight.org/ (visited on 01/01/2019).

[11] *Floodlight*. URL: http://www.projectfloodlight.org (visited on 01/01/2019).

[12] *Beacon*. 2011. URL: https://github.com/bigswitch/BeaconMirror (visited on 01/01/2019).

[13] *ONOS*. URL: http://onosproject.org/ (visited on 01/01/2019).

[14] *HPE-VAN*. URL: https://community.arubanetworks.com/t5/HPE-VAN-SDN-Controller-OVA-Free/ct-p/HPEVANSDNControllerOVAFreeTrial (visited on 01/01/2019).

[15] Ajay Nehra et al. "TILAK: A token-based prevention approach for topology discovery threats in SDN". In: *International Journal of Communication Systems* (), e3781.

[16] Ajay Nehra et al. "SLDP: A secure and lightweight link discovery protocol for software defined networking". In: *Computer Networks* 150 (2019), pp. 102–116.

[17] Ajay Nehra, Meenakshi Tripathi, and Manoj Singh Gaur. "FICUR: Employing SDN programmability to secure ARP". In: *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*. Jan. 2017, pp. 1–8.

[18] "Thomas Nadeau and Ken Gray". *SDN: Software Defined Networks. An Authoritative Review of Network Programmability Technologies*. O'Reilly, 2013.

[19] "Paul Goransson and Chuck Black". *Software Defined Networks. A Comprehensive Approach*. Morgan Kaufmann, 2014.

[20] Dell White Paper. *Open Networking: Dell's Point of View on SDN*. 2015. URL: https://i.dell.com/sites/csdocuments/Business_solutions_whitepapers_Documents/en/us/dell-networking-sdn-pov.pdf (visited on 01/01/2019).

[21] AMD White Paper. *Enabling Smart Software Defined Networks.* 2015. URL: https://www.amd.com/Documents/SDN-Whitepaper.pdf (visited on 01/01/2019).

[22] Siamak Azodolmolky. *Software Defined Networking with OpenFlow.* Packt Publishing, 2013.

[23] HP White Paper. *HP SDN hybrid network architecture.* 2015. URL: https://community.arubanetworks.com/aruba/attachments/aruba/SDN/43/1/4AA5-6738ENW.PDF (visited on 01/01/2019).

[24] William Stallings. *Foundations of Modern Networking. SDN, NFV, QoE, IoT, and Cloud.* Pearson Education, 2016.

[25] Microsoft White Paper. *Software-Defined Networking.* 2015. URL: http://download.microsoft.com/download/4/a/0/4a01102f-c83a-4cf6-824b-c7e21d6a0160/software_defined_networking_white_paper.pdf (visited on 01/01/2019).

[26] Juniper White Paper. *Network Transformation with NFV and SDN.* 2017. URL: https://www.juniper.net/assets/fr/fr/local/pdf/whitepapers/2000628-en.pdf (visited on 01/01/2019).

[27] *OpenFlow Switch Specification, Version 1.5.0.* Dec. 19, 2014. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf (visited on 01/01/2019).

[28] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.0.0.* Dec. 31, 2009. URL: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf (visited on 01/01/2019).

[29] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.1.0.* Feb. 28, 2011. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf (visited on 01/01/2019).

[30] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.2.0*. Dec. 5, 2011. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf (visited on 01/01/2019).

[31] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.0*. June 25, 2012. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf (visited on 01/01/2019).

[32] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.1*. Sept. 6, 2012. URL: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf (visited on 01/01/2019).

[33] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.2*. Apr. 25, 2013. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.2.pdf (visited on 01/01/2019).

[34] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.3*. Sept. 27, 2013. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.3.pdf (visited on 01/01/2019).

[35] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.4*. Mar. 27, 2014. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.4.pdf (visited on 01/01/2019).

[36] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.5*. Mar. 26, 2015. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf (visited on 01/01/2019).

[37] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.4.0*. Oct. 14, 2013. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf (visited on 01/01/2019).

[38] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.4.1*. Mar. 26, 2015. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.4.1.pdf (visited on 01/01/2019).

[39] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.5.1.* Mar. 26, 2015. URL: `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf` (visited on 01/01/2019).

[40] Cisco White Paper. *Software-Defined Networking: Why We Like It and How We Are Building On It.* 2013. URL: `https://www.cisco.com/c/dam/en_us/solutions/industries/docs/gov/cis13090_sdn_sled_white_paper.pdf` (visited on 01/01/2019).

[41] Intel White Paper. *Adopting Software-Defined Networking in the Enterprise.* 2014. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/best-practices/adopting-software-defined-networking-in-the-enterprise-paper.pdf` (visited on 01/01/2019).

[42] ONF White Paper. *Software-Defined Networking: The New Norm for Networks.* 2012. URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf` (visited on 01/01/2019).

[43] Linux Foundation White Paper. *Harmonizing Open Source and Standards in the Telecom World.* 2017. URL: `https://go.pardot.com/l/6342/2017-04-27/3tjbm4/6342/173369/LF_StandardsOpenSource_Whitepaper.pdf` (visited on 01/01/2019).

[44] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. "On scalability of software-defined networking". In: *IEEE Communications Magazine* 51.2 (Feb. 2013), pp. 136–141.

[45] Brandon Heller, Rob Sherwood, and Nick McKeown. "The Controller Placement Problem". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks.* HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 7–12. ISBN: 978-1-4503-1477-0. DOI: `10.1145/2342441.2342444`. URL: `http://doi.acm.org/10.1145/2342441.2342444`.

[46] Diego Kreutz, Fernando M.V. Ramos, and Paulo Verissimo. "Towards Secure and Dependable Software-defined Networks". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking.*

HotSDN ’13. Hong Kong, China: ACM, 2013, pp. 55–60. ISBN: 978-1-4503-2178-5. DOI: 10.1145/2491185.2491199. URL: http://doi.acm.org/10.1145/2491185.2491199.

[47] Y. Hu et al. “Reliability-aware controller placement for Software-Defined Networks”. In: *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on.* May 2013, pp. 672–675.

[48] Rob Sherwood et al. *FlowVisor: A Network Virtualization Layer.* 2009. URL: https://www.gta.ufrj.br/ensino/cpe717-2011/openflow-tr-2009-1-flowvisor.pdf (visited on 01/01/2019).

[49] I. Ahmad et al. “Security in Software Defined Networks: A Survey”. In: *IEEE Communications Surveys Tutorials* 17.4 (Oct. 2015), pp. 2317–2346.

[50] S. Khan et al. “Topology Discovery in Software Defined Networks: Threats, Taxonomy, and State-of-the-Art”. In: *IEEE Communications Surveys Tutorials* 19.1 (2017), pp. 303–324.

[51] George Tarnaras, Evangelos Haleplidis, and Spyros Denazis. “SDN and ForCES based optimal network topology discovery”. In: *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)* (2015), pp. 1–6.

[52] Leonardo Ochoa Aday, Cristina Cervelló Pastor, and Adriana Fernández Fernández. “Current Trends of Topology Discovery in OpenFlow-based Software Defined Networks”. In: *International Journal of Distributed Sensor Networks* 5.2 (2015), pp. 1–6.

[53] Sungmin Hong et al. “Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures”. In: *Proceedings 2015 Network and Distributed System Security Symposium* February (2015), pp. 8–11.

[54] *Link Layer Discovery Protocol and MIB.* URL: http://www.ieee802.org/1/files/public/docs2002/lldp-protocol-00.pdf (visited on 01/01/2019).

[55] Tri Hai Nguyen and Myungsik Yoo. “Analysis of link discovery service attacks in SDN controller”. In: *International Conference on Information Networking* (2017), pp. 259–261.

[56] Ajay Nehra, Meenakshi Tripathi, and M. S. Gaur. "'Global View' in SDN: Existing Implementation, Vulnerabilities & Threats". In: *Proceedings of the 10th International Conference on Security of Information and Networks*. SIN '17. Jaipur, India: ACM, 2017, pp. 303–306.

[57] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. New York, NY, USA: ACM, 2010, 19:1–19:6.

[58] J. Lin et al. "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications". In: *IEEE Internet of Things Journal* 4.5 (Oct. 2017), pp. 1125–1142. ISSN: 2327-4662. DOI: `10.1109/JIOT.2017.2683200`.

[59] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A Survey". In: *Comput. Netw.* 54.15 (Oct. 2010), pp. 2787–2805. ISSN: 1389-1286. DOI: `10.1016/j.comnet.2010.05.010`. URL: `http://dx.doi.org/10.1016/j.comnet.2010.05.010`.

[60] A. Gupta and R. K. Jha. "A Survey of 5G Network: Architecture and Emerging Technologies". In: *IEEE Access* 3 (2015), pp. 1206–1232. ISSN: 2169-3536.

[61] A. Gohil, H. Modi, and S. K. Patel. "5G technology of mobile communication: A survey". In: *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*. Mar. 2013, pp. 288–292.

[62] S. An, B. Lee, and D. Shin. "A Survey of Intelligent Transportation Systems". In: *2011 Third International Conference on Computational Intelligence, Communication Systems and Networks*. July 2011, pp. 332–337.

[63] T. Alharbi, M. Portmann, and F. Pakzad. "The (in)security of Topology Discovery in Software Defined Networks". In: *2015 IEEE 40th Conference on Local Computer Networks (LCN)*. 2015, pp. 502–505.

[64] F. Pakzad et al. “Efficient topology discovery in software defined networks”. In: *2014 8th International Conference on Signal Processing and Communication Systems (ICSPCS)*. Dec. 2014, pp. 1–8.

[65] Mohan Dhawan et al. “SPHINX: Detecting Security Attacks in Software-Defined Networks.” In: *NDSS*. The Internet Society, 2015.

[66] T. Alharbi, M. Portmann, and F. Pakzad. “The (in)security of Topology Discovery in Software Defined Networks”. In: *2015 IEEE 40th Conference on Local Computer Networks (LCN)*. Oct. 2015, pp. 502–505.

[67] Zhao Xin, Yao Lin, and Wu Guowei. “ESLD: An efficient and secure link discovery scheme for software defined networking”. In: *International Journal of Communication Systems* 31.10 (), e3552.

[68] A. Azzouni et al. “sOFTDP: Secure and efficient OpenFlow topology discovery protocol”. In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2018, pp. 1–7. DOI: `10.1109/NOMS.2018.8406229`.

[69] L. Ochoa-Aday, C. Cervelló-Pastor, and A. Fernández-Fernández. “Self-Healing Topology Discovery Protocol for Software-Defined Networks”. In: *IEEE Communications Letters* 22.5 (2018), pp. 1070–1073.

[70] P. Thorat et al. “Rapid recovery from link failures in software-defined networks”. In: *Journal of Communications and Networks* 19.6 (2017), pp. 648–665.

[71] E. Rojas et al. “TEDP: An Enhanced Topology Discovery Service for Software-Defined Networking”. In: *IEEE Communications Letters* 22.8 (Aug. 2018), pp. 1540–1543. ISSN: 1089-7798.

[72] L. Ochoa-Aday, C. Cervelló-Pastor, and A. Fernández-Fernández. “Discovering the Network Topology: An Efficient Approach for SDN”. In: *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal* 5.2 (2016).

[73] Y. Jimenez, C. Cervello-Pastor, and A. Garcia. “Dynamic Resource Discovery Protocol for Software Defined Networks”. In: *IEEE Communications Letters* 19.5 (May 2015), pp. 743–746.

[74] L. Ochoa-Aday, C. Cervello-Pastor, and A. Fernandez-Fernandez. “Self-Healing Topology Discovery Protocol for Software-Defined Networks”. In: *IEEE Communications Letters* 22.5 (May 2018), pp. 1070–1073.

[75] David C. Plummer. *Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware*. RFC 826. RFC Editor, Nov. 1982.

[76] Cisco. *Address Resolution Protocol*. 2013. URL: `https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipaddr_arp/configuration/xe-3se/3850/arp-xe-3se-3850-book/arp-config-arp.pdf` (visited on 01/01/2019).

[77] SANS. *Address Resolution Protocol Spoofing and Man-in-the-Middle Attacks*. 2006. URL: `https://www.sans.org/reading-room/whitepapers/threats/address-resolution-protocol-spoofing-man-in-the-middle-attacks-474` (visited on 01/01/2019).

[78] Christian Benvenuti. *Understanding Linux Network Internals*. O’Reilly Media, 2005. Chap. 29, pp. 699–748.

[79] Lawrence Berkeley. *arpwatch(8) - Linux man page*. URL: `https://linux.die.net/man/8/arpwatch` (visited on 01/01/2019).

[80] M. Carnut and J. Gondim. “switched Ethernet networks: A feasibility study”. In: *Proceedings of the 5th Simposio Seguranca em Informatica*. 2003.

[81] Han-Wei Hsiao, Cathy S. Lin, and Ssu-Yang Chang. “Constructing an ARP Attack Detection System with SNMP Traffic Data Mining”. In: *Proceedings of the 11th International Conference on Electronic Commerce*. ICEC ’09. Taipei, Taiwan: ACM, 2009, pp. 341–345.

[82] Vipul Goyal and Rohit Tripathy. “An Efficient Solution to the ARP Cache Poisoning Problem”. In: *Proceedings of the 10th Australasian Conference on Information Security and Privacy.* ACISP’05. Brisbane, Australia: Springer-Verlag, 2005, pp. 40–51.

[83] D. Bruschi E. Rosti A. Ornaghi. *S-ARP: a Secure Address Resolution Protocol.* 2003. URL: `https://www.acsac.org/2003/papers/111.pdf` (visited on 01/01/2019).

[84] Wesam Lootah, William Enck, and Patrick McDaniel. “TARP: Ticket-based address resolution protocol”. In: *Computer Networks* 51.15 (2007), pp. 4322–4337.

[85] Cisco. *Dynamic ARP Inspection.* 2013. URL: `http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/dynarp.html` (visited on 01/01/2019).

[86] S. Y. Nam, D. Kim, and J. Kim. “Enhanced ARP: preventing ARP poisoning-based man-in-the-middle attacks”. In: *IEEE Communications Letters* 14.2 (Feb. 2010), pp. 187–189.

[87] H. Ma et al. “Bayes-based ARP attack detection algorithm for cloud centers”. In: *Tsinghua Science and Technology* 21.1 (Feb. 2016), pp. 17–28.

[88] M. Z. Masoud, Y. Jaradat, and I. Jannoud. “On preventing ARP poisoning attack utilizing Software Defined Network (SDN) paradigm”. In: *Applied Electrical Engineering and Computing Technologies (AEECT), 2015 IEEE Jordan Conference on.* Nov. 2015, pp. 1–5.

[89] T. Alharbi et al. “Securing ARP in Software Defined Networks”. In: *2016 IEEE 41st Conference on Local Computer Networks (LCN).* Nov. 2016, pp. 523–526.

[90] Bob Lantz, Brandon Heller, and Nick McKeown. “A Network in a Laptop: Rapid Prototyping for Software-defined Networks”. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks.* Hotnets-IX. Monterey, California: ACM, 2010, 19:1–19:6.

Brief bio-data

Ajay Nehra completed a Doctoral degree in July 2019 with the Department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur, India. He awarded the M.Tech. Degree in Computer Science and Engineering from the Central University of Rajasthan, India, in June 2012 and received the Bachelor of Engineering degree in Computer Engineering from the University of Rajasthan in June 2008. He awarded scholarships for both the Master's and Doctoral degrees from the Ministry of Human Resource Development, Government of India. His current research area includes Software-Defined Networking, Information Security, and Network Security.