

VERIFYING NETWORK TOPOLOGY IN SOFTWARE
DEFINED NETWORKS USING STEALTHY
PROBING-BASED VERIFICATION (SPV)

AMIR ALIMOHAMMADIFAR

A THESIS

IN

THE DEPARTMENT

OF

CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

IN INFORMATION SYSTEMS SECURITY AT

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

FEBRUARY 2018

© AMIR ALIMOHAMMADIFAR, 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Amir Alimohammadifar**

Entitled: **Verifying Network Topology in Software Defined Networks Using Stealthy Probing-based Verification (SPV)**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Chun Wang _____ Chair

Dr. Amr Youssef _____ Examiner

Dr. Anjali Agarwal _____ External Examiner

Dr. Lingyu Wang _____ Supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 2018 _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Verifying Network Topology in Software Defined Networks Using Stealthy Probing-based Verification (SPV)

Amir Alimohammadifar

Since a key advantage of Software Defined Networks (SDN) is providing a logically centralized view of the network topology, the correctness of such a view becomes critical for SDN applications to make the right management decisions. However, recently discovered vulnerabilities in OpenFlow Discovery Protocol (OFDP) show that malicious hosts and switches can poison the network view of the SDN controller and consequently lead to more severe security attacks, such as man-in-the-middle or denial of service. Several solutions have been proposed to address such topology poisoning attacks, but their scope is mostly limited to malicious hosts injecting or relaying fake Link Layer Discovery Protocol (LLDP) packets. In this work, we propose Stealthy Probing-based Verification (SPV), a novel stealthy probing-based approach, to significantly extend the scope of existing solutions. Specifically, SPV incrementally verifies legitimate links and detects fake links by sending probing packets. Such packets are sent in a stealthy manner to deceive malicious hosts or switches who may be trying to identify the probing attempts among normal traffic. To illustrate the feasibility of our approach, we implement SPV in an emulated SDN environment using Mininet and OpenDaylight. We further evaluate the applicability and the performance of SPV in a real SDN/cloud topology. We show that SPV can achieve a

very low verification time (i.e., less than 120 milliseconds) in both real and emulated environments which makes SPV a scalable solution for large SDN networks.

Acknowledgments

I would first like to thank my thesis advisor Prof. Lingyu Wang. His continuous availability and guidance helped me the most to finish this thesis work. I am grateful to him for introducing me to the research world and feeding me the basics of this world. He always listens to my problems patiently and tries his best to rescue me with his inspiring speeches. Moreover, he gave his attention to almost all of my requests, which shows his great kindness. I feel very lucky to have him as my supervisor.

I would also like to thank the Audit Cloud Ready (ARC) members specially Suryadipta Majumdar and Taous Madi, who helped me from the beginning of my master's program in all the research projects I did. Without their passionate, participation, and input, my master's thesis could not have been successfully conducted.

I would also like to thank all other faculty members of the CIISE department. I really enjoyed and learned a lot from the courses that I attended during my master's program.

Finally, I must express my very profound gratitude to my parents and to my sister for providing me with unfailing support and continuous encouragement throughout my life and years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Contents

List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Statement	5
1.3 Contributions	5
1.4 Thesis Organization	6
2 Preliminaries and Related Work	7
2.1 Background	7
2.1.1 SDN Overview	7
2.1.2 The OpenFlow Protocol	10
2.1.3 SDN Topology Management Service	13
2.2 Threat Model	14
2.3 Related Work	15
2.3.1 SDN Topology Poisoning Attack Detection Mechanisms	16
2.3.2 OFDP Security Enhancement Mechanisms	19
2.3.3 Active Probing Techniques	21

3	Methodology	22
3.1	Design goals	22
3.2	SPV Design	23
3.2.1	Architecture	23
3.2.2	Link Verification Algorithm	26
3.2.3	Stealthy Packet Handler Algorithm	31
4	Implementation and Experiments	37
4.1	Implementation	37
4.1.1	Environment Setup	37
4.1.2	SPV's Implementation Details	43
4.2	Experiments	47
4.2.1	The Efficiency of SPV	47
4.2.2	Resource Consumption by SPV	49
4.2.3	Evaluating SPV Against Different Attacks	50
4.2.4	Applicability of SPV in a Real SDN/Cloud Topology	53
5	Security Analysis and Discussion	55
5.1	Security Analysis	55
5.1.1	Stealthiness Feature of SPV_PKT packets	55
5.1.2	Relaying or Dropping Packets	56
5.1.3	Replaying SPV_PKT Packets	57
5.1.4	Learning SPV_PKT Packet's Structure	57
5.1.5	Injecting Packets by a Malicious Host	58
5.2	Discussion	59
5.2.1	Exhausting Flow Table Capacity of SDN Switches	59

5.2.2	Effect of Encrypted Communication Between Control and Data Planes	59
5.2.3	Implementing SPV within SDN Controller	60
6	Other Contributions	61
6.1	Runtime Verification of Cloud-Wide VM-Level Network Isolation . . .	61
6.2	Auditing Virtual Networks Isolation Across Cloud Layers	62
6.3	Proactive Security Auditing for Clouds	64
6.4	Preserving Both Privacy and Utility in Prefix Preserving Anonymiza- tion of Network Traces	65
6.5	A Quantitative Approach to Security Compliance Auditing for SDN- Based Cloud	66
7	Conclusion	68
	Bibliography	74

List of Figures

1	Fake link creation in SDN topology	3
2	SDN 3-Layer Architecture	9
3	Bootstrapping Process	10
4	Packet Forwarding Process for a Packet with no Match in Flow Tables	12
5	OFDP protocol and LLDP packet format	14
6	Web Client Harvesting [19]	17
7	SPHINX Flow Graphs Between H1 and H4 [19]	18
8	SDN and stealthy probing-based verification (SPV) architecture . . .	23
9	Verification of a newly added link utilizing SPV	28
10	PKT_DB database table schema	33
11	Line sweep mechanism for selecting stealthy probing packets used in second or further rounds of verification	34
12	SPV_PKT packet generation procedure flow chart	35
13	Installing required features on ODL controller	38
14	A fat-tree topology with four core switches	40
15	An iPerf TCP traffic generation between hosts h1 and h2	41
16	A fat-tree topology with two core switches in the view of the ODL Controller	42
17	Flow rules of switch <code>openflow:2002</code> before adding LLDP relay rules .	42
18	Flow rules of switch <code>openflow:2002</code> after adding LLDP relay rules .	43

19	Afake link creation in the view of the ODL Controller	43
20	An output of opendaylight-topology upon a request to query the ODL's network view	45
21	Details of a stored link and an OpenFlow switch	45
22	Time required by SPV to verify a new link while varying the number of switches up to 40 and number of links up to 96 in both single and multi-thread modes	48
23	Time required by SPV to verify all existing links while varying the number of switches up to 40 and number of links up to 96 in both single and multi-thread modes	49
24	Average CPU usage by SPV to verify all existing links while varying the number of switches up to 40 and the number of links accordingly	50
25	Average memory usage by SPV to verify all existing links while varying the number of switches up to 40 and the number of links accordingly	51
26	Evaluating SPV with the presence of attackers in the network that tend to increase packet loss while varying the links packet loss rate (%)	52
27	Evaluating SPV with the presence of attackers in the network that tend to congest the network by showing percentage of unverified links while varying traffic throughput (Mbps)	53
28	Evaluating SPV with the presence of attackers in the network that tend to relay the traffic by measuring the percentage of relayed SPV_PKT packets while varying the percentage of the time that the attacker may relay all the traffic	54

List of Tables

1	Comparing existing solutions with SPV	4
2	Summary of SDN topology poisoning attack detection and OFDP security enhancement mechanisms	20
3	SPV's performance on a real SDN/cloud topology and Mininet in both single (ST) and multi (MT) thread modes	54

Chapter 1

Introduction

The hope to overcome the current limitations of traditional networks, e.g., complexity and dynamic nature, difficulties to configure and manage the network, fault tolerance issues, etc., lies in the emergence of a new network paradigm, namely, Software Defined Networks (SDN), by separating the network's control and data planes [27, 21, 57]. The empowerment of SDN controllers has become a double-edged sword leading to both convenience in network management and an increase in dependability on these SDN controllers. More specifically, the SDN controller is meant to be the operating system of SDN and provides a logical view of the network to its applications, which run on top of SDN controller by making use of the programmable interface that the SDN controller provides to them [27, 57]. Thus, these applications heavily rely on the SDN controller for the correctness of the logical view of the network to make proper network management decisions, such as routing, load balancing, firewalling, monitoring, etc. [18]. Therefore, the validity of the network view provided by the SDN controller becomes critical for the proper functionalities of its applications and in general, of SDN.

However, the recently discovered vulnerabilities in the OFDP protocol show that malicious hosts or switches can create fake links to poison the network view of the

SDN controller [25]. More specifically, there exist at least three different variations of such attacks [14, 19, 6, 11]: i) injecting fake LLDP packets, ii) relaying LLDP packets using compromised hosts and iii) relaying LLDP packets using compromised switches. We further illustrate the topology poisoning attacks in the following subsection.

1.1 Motivation

Figure 1 illustrates a simple SDN topology with the presence of an application plane, a logically centralized SDN controller, and an underlying data plane which includes OpenFlow switches, and existing hosts. In this example, we assume there are two malicious hosts H1 and H2, and a malicious switch S22. The link discovery is performed via the OFDP protocol along with LLDP packets. These switches and their links (solid lines in the figure) consist of the network view of the data plane from the SDN controller. In the following, we utilize this network view to demonstrate three different poisoning attacks.

- The relaying LLDP packets attack by hosts, involves two malicious hosts, i.e., H1 and H2, to create a fake link between the switches S13 and S14 (the dashed line between switches S13 and S14 in Figure 1). To this end, whenever host H1 receives an LLDP packet from switch S13 (in OFDP protocol, a switch advertises the received LLDP packet in all its outgoing ports that may be connected to hosts or other switches), host H1 may send the received LLDP packet to host H2 using an out-of-band link. In the next step, host H2 may forward the same packet to the switch S14, and according to the OFDP protocol, switch S14 forwards the LLDP packet to the SDN controller. Upon receiving the LLDP packet (that was initially sent to switch S13) from switch S14, the SDN controller assumes a link, which actually does not exist, between switches S13

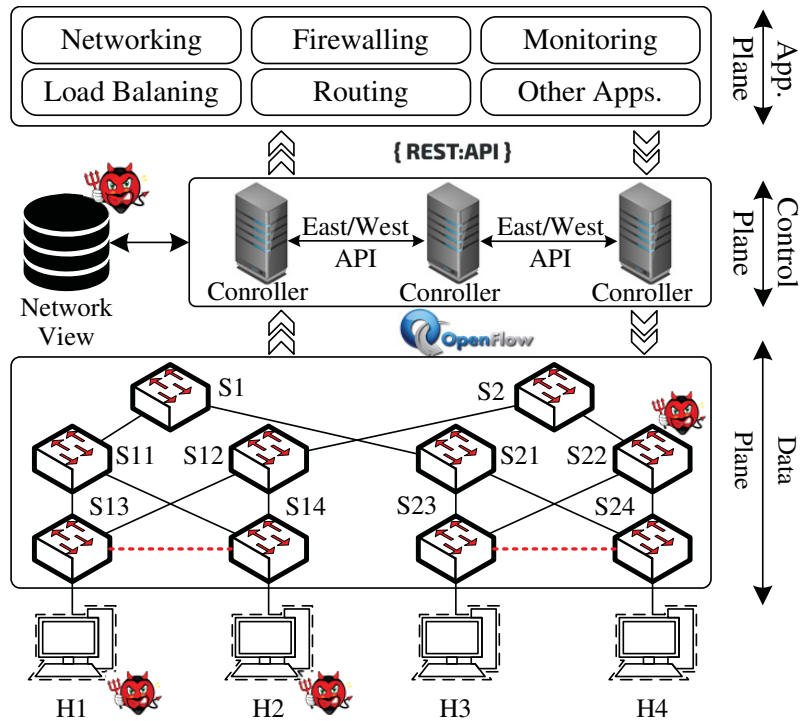


Figure 1: Fake link creation in SDN topology

and S14 (details of SDN link discovery and OFDP protocol are discussed in Section 2.1.3).

- The fake LLDP packet injection attack involves forging the LLDP packet received from switch S13 by the malicious host H1 to create a fake link between switches S13 and S14, shown by the dashed line in Figure 1. To this purpose, host H1 should initially discover some information about specifications, e.g., MAC address, of switch S13 to be able to forge the LLDP packet received from switch S13 and sends it (using an out-of-band link) to switch S14 (or any other switch to create the fake link between switch S13 and the target switch). Utilizing the discovered information, host H1 masquerades itself as switch S13 and sends the forged LLDP packet towards switch S14. Similarly as in the first attack, switch S14 assumes the packet is sent from switch S13 and forwards the

received LLDP packet to the SDN controller which creates a fake link between switches S13 and S14 in the network view of the SDN controller.

- The relaying LLDP packet attack by switches, involves a malicious switch i.e., S22 as shown in Figure 1, to create a fake link by modifying its flow tables to relay LLDP packets received from switch S23 towards switch S24. Being unaware of this attack, switch S24 legitimately sends the relayed LLDP packet received from switch S22 back to the SDN controller; which results in a fake link creation between switches S23 and S24 (shown by dashed line in Figure 1 between switches S23 and S24) in the network view.

Table 1 summarizes the above-mentioned attacks and their existing solutions. To overcome the fake LLDP packet injection attack, several works (e.g., [19, 14, 4, 8]) mitigate this problem to some extent, by extending the functionality of the SDN controllers or modifying the OFDP protocol to harden it. To tackle the relaying LLDP packets using compromised hosts, Dhawan et al. [14] address this issue by assuming the flow of a packet in the network is not manipulated. Finally, detecting poisoning the network view by making use of compromised switches that relay LLDP packets for this purpose, is still an open issue. In summary (as shown in Table 1), no existing work completely overcomes the above-mentioned attacks.

Table 1: Comparing existing solutions with SPV

Detection Approach	Methods of Poisoning Network View		
	Fake LLDP Packet Injection	LLDP Relay by Hosts	LLDP Relay by Switches
TopoGuard [19]	Yes	No	No
SPHINX [14]	Yes	Yes	No
HMAC Authenticated LLDP [4]	Yes	No	No
Detecting a Compromised Switch [12]	No	No	No
OFDPv2 [45]	No	No	No
sOFTDP [8]	Yes	No	No
Stealthy Probing-based Verification (SPV)	Yes	Yes	Yes

Poisoning the network view of the SDN controller can result in more severe attacks such as man-in-the-middle, denial of service, as well as incorrect network management decisions such as wrong routing, wrong load balancing, etc. [19, 18, 25], that might put the network into a critical situation. Therefore, the need for link verification to identify these poisoning attacks is crucial for proper functionalities of an SDN infrastructure.

1.2 Thesis Statement

In this work, we propose a novel approach of actively sending probing packets in a stealthy manner in order to incrementally verify legitimate links and to identify fake links in the SDN network topology view. Specifically, we first design stealthy probing packets in a manner that these packets remain indistinguishable from normal traffic. Second, we provide detailed methodology and algorithms of our approach, namely, SPV. As a proof of concept, we implement and test SPV in an emulated SDN environment using Mininet [38] network emulator and OpenDaylight [1] SDN controller.

1.3 Contributions

Our main contributions in this work are as follows.

- To the best of our knowledge, this is the first solution which can detect all three types of topology poisoning attacks (as demonstrated in Figure 1) based on the innovative idea of employing active probing in a stealthy manner.
- We design the SPV to be separated from the SDN controller implementations such that SPV can be easily adapted to any implementation of SDN controller.

Moreover, the implementation of SPV using OpenFlow-based SDN environment demonstrates the practicality of our approach.

- We evaluate SPV through extensive experiments both by making use of the topology of a real SDN/cloud hosted at one of the largest telecommunication vendors and by using synthetic data. The obtained results show that SPV achieves a constant verification time, i.e., less than 120 milliseconds for link verification, which confirms the scalability of our solution when applied to large size SDN networks.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 considers preliminaries of SDN and its topology discovery mechanism, our threat model, and the related work to this dissertation. Chapter 3 provides the design goals of SPV and discusses our methodology, i.e., SPV. Chapter 4 provides the implementation details and the experiments. Chapter 5 analyzes the security of SPV and discussions SPV's limitations. Contributions to other projects are carried over Section 6 and finally, Section 7 concludes this thesis.

Chapter 2

Preliminaries and Related Work

The first section of this chapter provides preliminaries of our work. Then we define our thread model in the second section of this chapter. The third section, discusses literature related to our work.

2.1 Background

This section provides an extensive overview of OpenFlow-based Software Defined Networks (SDN) and its topology discovery mechanism, i.e., OpenFlow Discovery Protocol (OFDP). We also discuss some security issues related to OFDP protocol which leads to topology poisoning attacks in SDN and its related consequences.

2.1.1 SDN Overview

For several years, variations of devices, such as repeaters, bridges, hubs, routers, and switches, have been developed and deployed to be utilized in networks to keep filtering and forwarding network packets from one to another destination. However, the demand of recent evolving technologies is to have faster and more resilient networks through modern data centers that can carry over millions of Virtual Machines

(VM) and users which can be no longer achieved through traditional networks due to some reasons, such as the complexity, owning, management and maintenance costs, dynamic nature that needs regular and manual configuration of devices, and so on [17, 27, 57].

To overcome the above-mentioned issues and achieve the goals, computer scientists came up with a solution to separate the control and forwarding tasks from networking devices by leaving the forwarding plane inside the networking devices while handing the control plane to another device. This idea was first presented by [15] which defines different elements for forwarding and controlling those forwarding devices through a protocol, namely, ForCES. The idea of separating the control and forwarding planes leads the networking devices to be simply forwarding units which are managed by a logically centralized controller, namely, SDN controller, which works as an operating system of the network. The SDN evolution leads it to a three-layered architecture [17, 27, 57, 21, 51, 2] (as shown in Figure 2): i) the application layer, which consists of different applications that manage the entire data plane, ii) the control plane (i.e., SDN controller), and iii) the data plane, which comprises the forwarding devices and their connecting links.

Each layer comprises some components. The application layer consists of different applications that manage the entire data-plane network through the control layer. These applications can be monitoring, security, networking or other kinds of business applications. The SDN controller(s) lies in the control layer. Each SDN can have one or multiple SDN controllers, that can be seen as one logically centralized controller. There are different controllers available as of today, POX [47], NOX [40], OpenDaylight [1], Floodlight [48], Beacon [13], Ryu [49], etc. The underlying data plane comprises forwarding devices or switches and the links between them. There are different commercial and open source implementations of SDN devices.

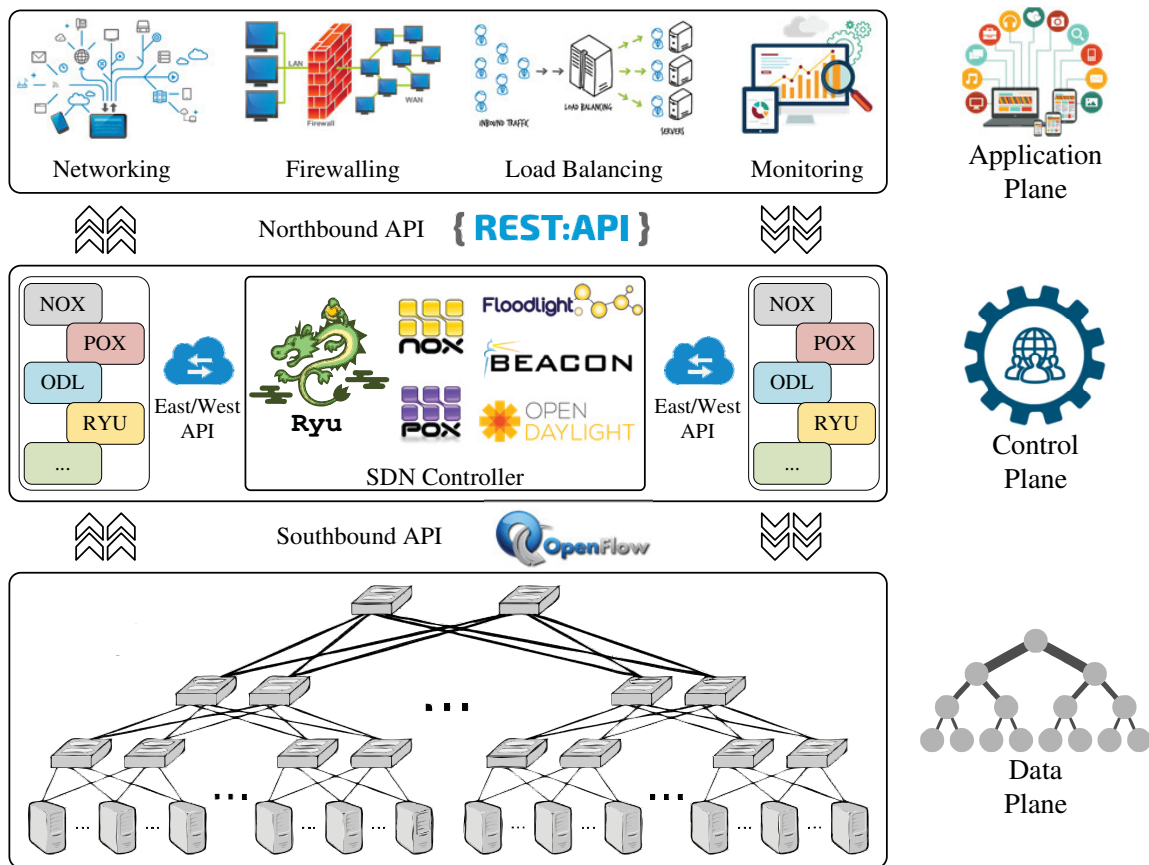


Figure 2: SDN 3-Layer Architecture

A widely used software-based open source SDN device is OVS [42] switches. The mentioned layers communicate through APIs. The application and controller layers communicate through northbound API that has not been standardized yet; However, the most widely used API for the controller and data-plane communications or the southbound API is the OpenFlow protocol [36, 17]. Other available southbound interfaces are OVSDB [46], ForCES [15], POF [53], PCEP [29], NETCONF [16], etc. The OpenFlow specification [41] defines how the controller should communicate with underlying data-plane devices i.e., OpenFlow switches as well as providing the definition and specifications of OpenFlow switches. The overview of the OpenFlow protocol is detailed in the following subsection.

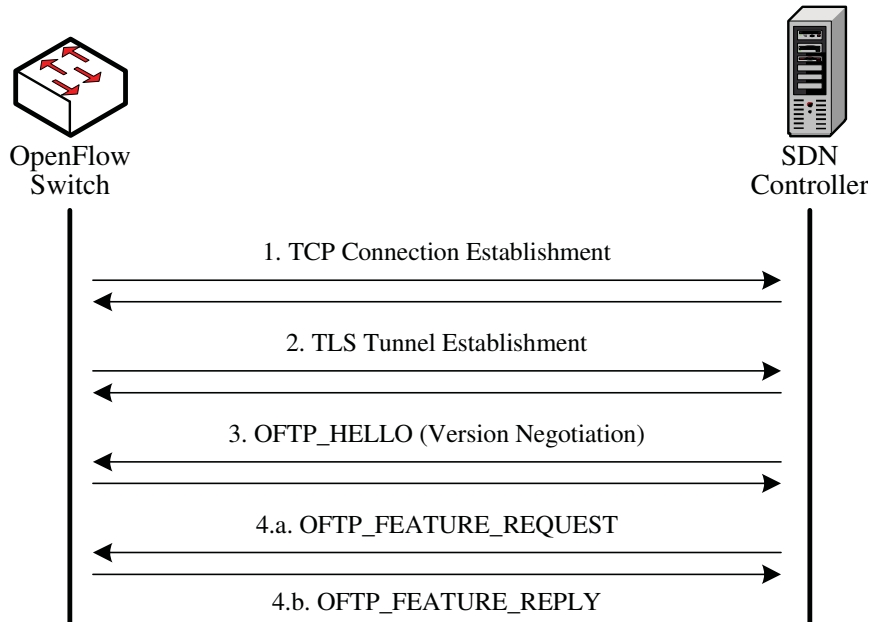


Figure 3: Bootstrapping Process

2.1.2 The OpenFlow Protocol

The general description of OpenFlow protocol [36, 41] is discussed in this section. In general, the OpenFlow protocol defines the communications between data-plane devices and the control plane SDN controllers. It also provides how the data-plane devices should react under certain circumstances such as actions they should do when they receive a packet. Note that the OpenFlow protocol is a small subset of massive SDN technology, therefore, its job is not to define the SDN controller behavior [17, 36]. The description of the important processes defined in OpenFlow protocol, which are utilized in this thesis, is discussed in the following.

2.1.2.1 OpenFlow Switch Bootstrapping Process

The very first step of an OpenFlow switch to get involved in an SDN is to introduce itself to the SDN controller and receives instructions of how to react in the network upon receiving a packet [36, 41, 11]. Therefore, as depicted in Figure 3, whenever

an OpenFlow switch joins an SDN, it first establishes a TCP connection with the SDN controller through a secure channel (i.e., TLS, which is optional) to start bootstrapping process which involves exchanging some control messages. Afterwards, the SDN controller and the newly joined OpenFlow switch exchange `OFTP_HELLO` messages to agree on the OpenFlow version they use for further communications. After agreement on the OpenFlow version, the SDN controller requests for switch's specifications, such as MAC address of its ports, through the `OFTP_FEATURE_REQUEST` message. The switch informs the SDN controller of its specifications using the `PFTP_FEATURE_RESPLY` message. Finally, the SDN controller installs some configurations on the switch by making use of an `OFTP_SET_CONFIG` message. At this point, if there are any proactive flows to be installed on the switch, the SDN controller installs them using an `OFTP_FLOW_MOD` message (note that we do not show the exchange of this message in the above-mentioned figure since it is not part of the bootstrapping process).

2.1.2.2 OpenFlow Switch Packet Forwarding Process

After successful installation of an OpenFlow switch in an SDN, as described in Section 2.1.2.1, the OpenFlow switch starts its functionality by forwarding the received packets based on its flow tables. However, different situations might occur based on the matching results. As a basic functionality of an OpenFlow switch is to receive a packet on one port, modify it if necessary, and forward it on another port. Therefore, each OpenFlow switch has a set of flow tables that comprise flow entries for packet-matching function. In this packet-matching function, whenever an OpenFlow switch receives a packet, it matches the packet with the flows of its tables, which there could be a match or a no-match, to make a decision which might be either of the following [17, 36]: (i) forwarding the packet on an outgoing port, (ii) dropping the packet, or

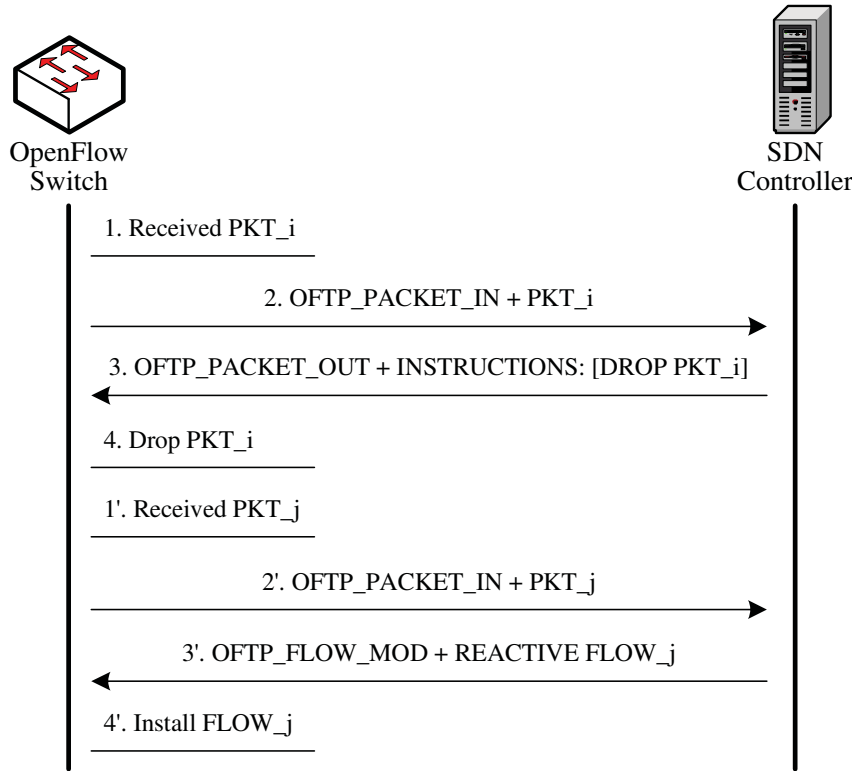


Figure 4: Packet Forwarding Process for a Packet with no Match in Flow Tables

(iii) forwarding the packet to the controller for further instructions.

Note that in an OpenFlow switch, there are different types of flows in its flow tables for packet matching which are proactive, reactive, or hybrid flows. Proactive flows are flow entries that are installed by SDN controller upon joining a switch to the network whereas reactive flow entries are installed with the help of the SDN controller upon receiving a packet that there is no match for it in the existing flow entries. The hybrid flow is a combination of both proactive and reactive flows. To utilize reactive flows, an OpenFlow switch must communicate with the SDN controller to receive proper instructions for forwarding the packets that there is no match for them in its flow tables. To this end, whenever a new packet is received, after performing packet-matching, if there is no match for the packet, the OpenFlow switch encapsulates the received packet in an `OFTP_PACKET_IN` message and forwards it to the SDN

controller and waits for further instructions. After the SDN controller receives the `OFTP_PACKET_IN` message, it makes a decision of what has to be done with the packet and sends the proper instructions towards the mentioned switch. For example, if new flows are to be installed, the SDN controller sends an `OFTP_FLOW_MOD` message along with the new flows to be installed, which are called reactive flows, on the switch so that the switch can utilize those flows to forward the received packet or similar further packets that might be received afterward. As another example, if the packet should be dropped, the SDN controller sends an `OFTP_PACKET_OUT` message along with the instructions to drop the packet, towards the switch [17, 36, 11]. The above-mentioned examples are depicted in Figure 4.

2.1.3 SDN Topology Management Service

One of the main advantages of SDN controller is the centralized network view that it offers, which enables reliable topology management, traffic engineering, resource provisioning, etc. [25]. In OpenFlow-based SDN, after an OpenFlow switch joins to the network, it establishes a TCP connection with the SDN controller. Afterwards, the SDN controller requests the switch for its active ports and their respective MAC addresses using the `OFTP_FEATURE_REQUEST` message. The switch replies with an `OFTP_FEATURE_REPLY` message containing the requested information which is needed for topology discovery.

Although there is no specific standard for discovering the topology of an OpenFlow-based SDN, most SDN controllers' implementations follow the OFDP protocol relying on LLDP packets [19]. Figure 5 shows the steps of how an SDN controller utilizes the OFDP protocol and LLDP packets to discover the network topology:

Step (1): the SDN controller encapsulates an LLDP packet (Figure 5 shows the LLDP packet format) into `OFTP_PACKET_OUT` message and sends it to switch S11

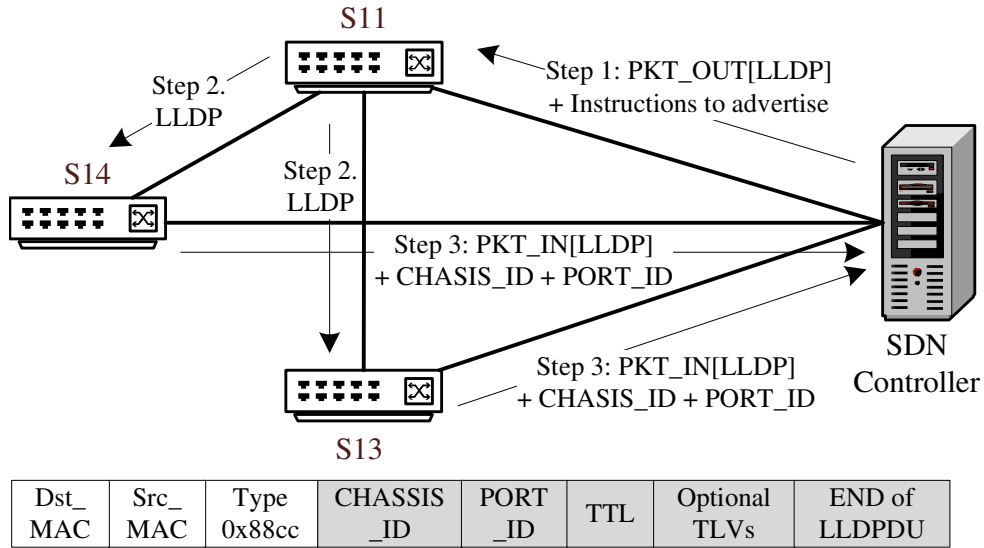


Figure 5: OFDP protocol and LLDP packet format

along with a set of instructions. The fields `CHASSIS_ID` and `PORT_ID` indicate which switch the LLDP packet is designated to. Step (2): the instructions in the received `OFTP_PACKET_OUT` message, instructs switch `S11` to advertise the received LLDP packet in all its ports except the port it receives the packet from. Step (3): switches `S13` and `S14` send the LLDP packet along with their `CHASSIS_ID` and `PORT_ID` of the port they have received the LLDP packet from, back to the SDN controller. based on the received packets, the SDN controller can discover the link between switches `S11` and `S13` as well as the link between switches `S11` and `S14`. SDN controller discovers all the links, in both directions, by performing the similar procedure for all network switches and updates its view of the network periodically.

2.2 Threat Model

Due to the lack of authentication for LLDP packets, malicious hosts or malicious switches can create fake links in the view of the SDN controller by injecting false

LLDP packets or relaying LLDP packets. Those attacks have been already discussed in several works [4, 8, 9, 11, 19, 25]. Similar to those works, we assume that an adversary may compromise one or more host(s) and/or switch(es) in the network. S/he can send information through the compromised hosts using the out-of-band links, and modify the flows of the compromised switches. Furthermore, s/he is able to distinguish between host-generated packets and SDN control packets. S/he can sniff packets, modify them, and inject them into the network. Consequently, the in-scope threats in our work include all the three types of poisoning attacks mentioned in Section 1.

We assume the SDN controller is uncompromisable and the established control channels between the SDN controller and OpenFlow switches are trusted, and thus attackers can only attempt to distinguish probing packets based on their contents. On the other hand, the confidentiality and integrity of public-private key pairs that are installed on switches are preserved, and the implementation or specification of switch software remains unmodified.

2.3 Related Work

There has been a considerable effort tackling traditional networks' topology poisoning issues since the latter might have a significant impact on the network functionality. For instance, poisoning attacks can manipulate routers' network view [39] to cause forwarding black holes, traffic redirection (e.g., to a compromised destination for eavesdropping purposes) [55, 23] or to further mount Spanning Tree Protocol (STP) mangling attacks [44], etc.

Although SDN is a recent networking paradigm, several approaches have already been proposed to address the SDN controller's network view poisoning problem either

as attack detection mechanisms or as security enhancement techniques. In the following, we provide a review of both approaches. We also discuss active probing-based techniques.

2.3.1 SDN Topology Poisoning Attack Detection Mechanisms

TopoGuard. In [19], two new network topology poisoning attacks are introduced, namely, *Host Location Hijacking* and *Link Fabrication*, exploiting vulnerabilities in current link discovery and host tracking services in OpenFlow-based SDN. In Host Location Hijacking Attack, an attacker abuse the Host Tracking Service (HTS) of an SDN controller, which is responsible to track the location of hosts in the network, by spoofing a hosts identity. Using HTS, an SDN controller keeps the track of the hosts' locations by processing `OFTP_PACKET_IN` messages so that whenever a host moves to a new location, the SDN controller can update its view over the network. If an adversary manages to craft the identity of a host, she can send spoofed packets to deceive the SDN controller of the target host's location. To this end, the adversary sends the spoofed packets into the network which will trigger `OFTP_PACKET_IN` messages to be sent to the SDN controller. Once the SDN controller receives those messages, it updates the location of the host to the new location which is, in fact, the attacker's location. The result is hijacking the traffic by the attacker which is in fact destined to the original host. An example of the result of this attack is phishing an impersonated website which is depicted in Figure 6 [19]. In the latter case, i.e., the Link Fabrication attack, an adversary utilizes fake LLDP packets or relaying the LLDP packets to maintain fake links in the view of the SDN controller. In fake LLDP packet injection attack, a malicious host stores LLDP packets that are received from an OpenFlow switch, then, the malicious host forges the LLDP packet with the identity of an existing OpenFlow switch in the network and forwards the forged LLDP packets to

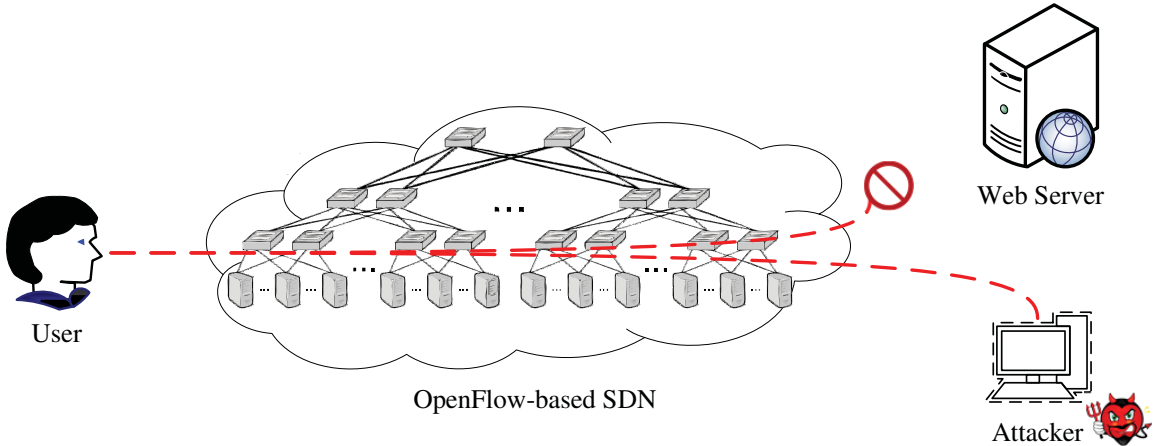


Figure 6: Web Client Harvesting [19]

other switch(es). Or, if the host has access to other switches, it relays the original received LLDP packet to the target switch. Also, a malicious host may forward the received LLDP packets between its ports towards target switch(es) without sending them back to the controller. These mentioned attacks create fake links in the view of the SDN controller (refer to Figure 1 to better understand the Link Fabrication attacks).

Although TopoGuard mainly introduces the above-mentioned attacks, the authors propose an OpenFlow-based SDN controller extension to prevent those vulnerabilities. TopoGuard checks the legitimacy of switch ports and host migration to tackle Host Location Hijacking, as well as verifying LLDP packets' integrity by adding the signature of switch data path ID (DPID) and port in an extra TLV field of LLDP packets using a cryptography hash function. Moreover, to verify host migration correctness, TopoGuard adds authentication to LLDP packets in order to identify those packets generated by hosts aiming at poisoning the network view. However, unlike SPV, TopoGuard cannot detect fake links caused by LLDP relay attacks.

SPHINX. SPHINX [14] allows detecting both known and potentially unknown attacks on network topology by verifying data plane forwarding and network topology

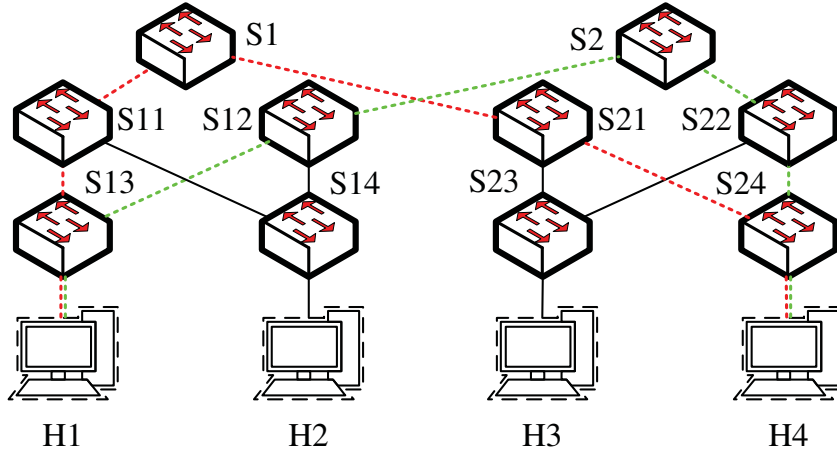


Figure 7: SPHINX Flow Graphs Between H1 and H4 [19]

attacks, and defining administrative policies and verifying them against attacks on SDN. To this end, SPHINX makes use of specific graphs, namely, *flow graphs*, to first model the data plane. A flow graph is a graph representation of data-plane flows with nodes being the endpoints and switches being the nodes. Due to the programmable ability of SDN, different flow graphs might exist between two endpoints. For example, as depicted in Figure 7, between two endpoints H1 and H4, two possible flow graphs are S13-S12-S2-S22-S24 and S13-S11-S1-S21-S24.

To incrementally build the flow graphs, SPHINX processes specific OpenFlow control messages, i.e., `OFTP_FLOW_MOD`, `OFTP_STATUS_REPLY`, `OFTP_PACKET_IN`, and `OFTP_FEATURE_REPLY`, in order to model the data plane. In the last step, with the help of the flow graphs metadata and the mentioned pre-defined administrative policies, SPHINX verifies the discovered flow graphs against the history of the metadata and the pre-defined administrative policies. Note that in this work the communications between the SDN controller and OpenFlow switches are assumed to be trusted. In contrast to SPV, SPHINX cannot detect the creation of fake links in the topology, which would falsify the generated flow graphs based on which the data plane verification is performed. Furthermore, malicious switches that tend to relay LLDP packets

remain undetected.

HMAC Authenticated LLDP. In [4], the authors propose an LLDP packets authentication approach based on adding the HMAC of a switch ID and the corresponding port ID to the LLDP packet into one of the unused TLV fields of an LLDP packet. The packet format of an LLDP packet is already shown in Figure 5. Although this method prevents the forged LLDP packets created by malicious hosts or soft switches from creating fake links in the network view, contrarily to SPV, it cannot handle fake link creation caused by relayed LLDP packets by malicious hosts or switches since it solely depends on HMAC authenticated LLDP packets since relaying an LLDP packet with a valid HMAC value in it, is still verified.

2.3.2 OFDP Security Enhancement Mechanisms

OFDPv2. In [45], an improved version of OFDP, namely, OFDPv2, is proposed. OFDPv2 requires the SDN controller to send only one `OFTP_PAKCKET_OUT` message containing an LLDP packet to a switch and instructs the switch to advertise the LLDP packet on all its ports, instead of sending multiple `OFTP_PAKCKET_OUT` messages to carry an LLDP packet to be forwarded on every port of each switch. Although in OFDPv2 the topology discovery mechanism is changed and improved in comparison to original the OFDP protocol, it lacks the authentication for LLDP packets and hence suffer the same topology poisoning vulnerabilities and unlike SPV, there is no verification mechanism for data-plane links.

sOFTDP. The authors in [8] propose an improvement to the OFDP protocol, namely, sOFTDP. The main idea is to transfer the burden of topology discovery from the SDN controller to the data-plane switches. sOFTDP applies minimal changes to the OpenFlow switch design to allow switches to detect topology changes and notify the controller accordingly. Authenticated LLDP packets are used to avoid packet forgery,

Table 2: Summary of SDN topology poisoning attack detection and OFDP security enhancement mechanisms

Name	Detection Approach	Addressed Poisoning Attacks
TopoGuard [19]	Authenticated LLDP packets	Fake LLDP packet injection
SPHINX [14]	<ul style="list-style-type: none"> Authenticated LLDP packets Flow graphs Administrative-policies 	<ul style="list-style-type: none"> Fake LLDP packet injection LLDP relay by hosts
HMAC Authenticated LLDP [4]	Authenticated LLDP packets	Fake LLDP packet injection
Detecting a Compromised Switch [12]	Artificial packets	None
OFDPv2 [45]	Simplifying OFDP	None
sOFTDP [8]	<ul style="list-style-type: none"> Authenticated LLDP packets BFD sessions 	Fake LLDP packet injection
SPV	<ul style="list-style-type: none"> Authenticated stealthy packets Active probing 	<ul style="list-style-type: none"> Fake LLDP packet injection LLDP relay by switches LLDP relay by hosts

and as port liveliness detection mechanism, whenever a switch joins the network, by making use of some control messages through a 3-way handshake procedure, a Bidirectional Forwarding Detection (BFD) session is established between switches in the network to notify the controller of any port changes. Furthermore, to detect link updates in the network, instead of sending LLDP packets periodically, sOFTDP utilizes BFD session and notify the controller of the link updates. Note that sOFTDP only relies on BFD session for link removals and ports failures through its designed `BFD_STATUS` control message, and for link additions, it uses `OFTP_PORT_STATUS` messages. The reason behind this is to detect port or link failures that are also originating from non-administrative decisions, i.e., failure of physical links or switch failures. Unlike SPV, sOFTDP[8] do not verify fake link creation and furthermore, its vulnerable to relaying BFD packet attack which is similar to relaying LLDP packet attack, so, it still suffers from topology poisoning attacks.

Table 2 summarize and compares the above-mentioned related works.

2.3.3 Active Probing Techniques

Probing techniques are typically used to maliciously infer network specifications (e.g., firewall rules, OpenFlow rules, bandwidth estimation, flow tables usage and capacity, etc.), in contrast to SPV where network probing is used as a defensive mechanism to deceive malicious hosts and SDN switches. For instance, in [52], the authors utilize the delay required for flow installation on SDN switches to detect whether a network is an SDN. INSPIRE [31] relies on some senders located inside the network, a receiver deployed outside the network and a line sweep algorithm to select forged probing packets to be sent to the network in order to infer OpenFlow rules. INSPIRE can infer the flow rules installation mode (i.e., proactive or reactive), by measuring the delay between a packet sending time and its reception, then an apriori algorithm is used to discover the rules. In [10], the authors use active probing techniques based on crafted packets to trigger switch-controller communications, then they use round trip time (RTT) and packet-pair dispersion features to infer information about flow rules. The authors in [30] also use probing and RTT measurement to infer the OpenFlow switches' tables capacity and usage along with the flow rules' hard and idle timeouts. They also trigger controller-switch interactions by sending probing packets to infer the processing time of a specific rule.

Similarly to our work, in [12], network probing is used as a defensive technique. Therein, a periodic sampling-based approach is proposed to detect malicious OpenFlow switches in an SDN. First, a switch is randomly chosen, then a set of flow rules are randomly selected from the switch's flow tables. Afterwards, artificial packets are sent to the target switch to investigate whether the latter behaves properly upon receiving those packets. Our effort can be seen as complementary to this work since the latter checks the legitimacy of switches but cannot verify the links between them.

Chapter 3

Methodology

In this chapter, first we identify the goals to achieve with our proposal. Then we discuss our methodology.

3.1 Design goals

The primary goal of SPV is near real-time verification of data-plane links in an SDN environment in an incremental manner to overcome the topology poisoning attacks that are already mentioned in Section 1. Our target features include:

1. Designing a tool to verify data-plane links in an SDN while keeping the tool independent of any SDN infrastructure specially the SDN controller.
2. Detecting network topology changes in the view of the SDN controller as soon as they occur.
3. Generate stealthy probing packets out of the packets that are already traversing the network to deceive the potential malicious switches which might try to poison the network view of the SDN controller.

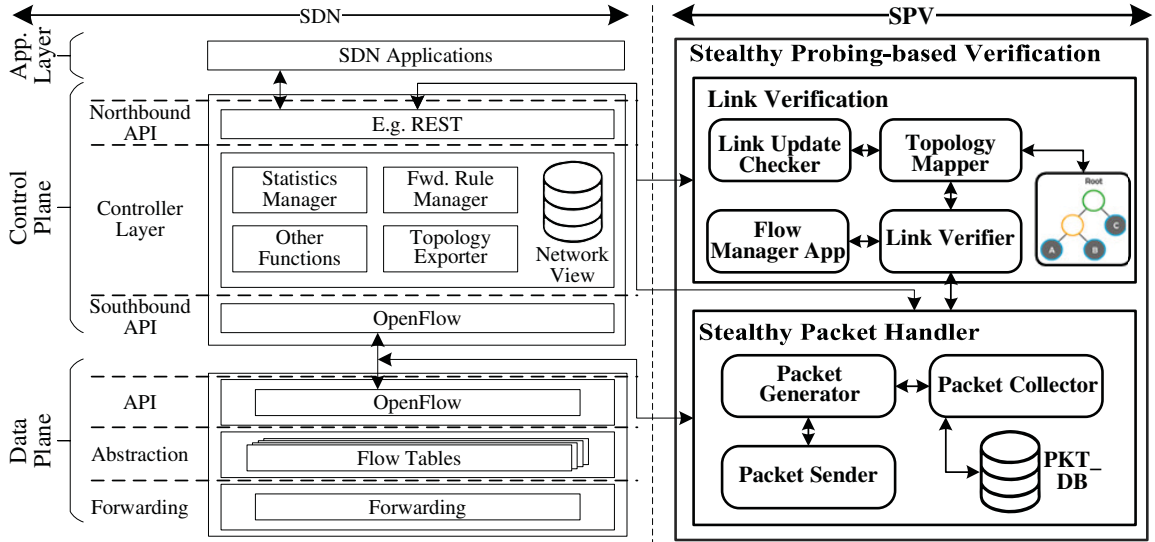


Figure 8: SDN and stealthy probing-based verification (SPV) architecture

4. Designing the stealthy probing packets indistinguishable from other host-generated packets inside the network.
5. Verifying the whole network topology with a very low response time.

3.2 SPV Design

This section elaborates our stealthy active probing-based verification approach, i.e., SPV, to verify OpenFlow-based SDN data-plane links.

3.2.1 Architecture

Figure 8 depicts the architecture of SPV including its interactions with SDN. In order to achieve our goals, i.e., verifying SDN data-plane topology using stealthy probing, we indeed need to pursue certain requirements. The very first requirement is to acquire the knowledge of the network topology, which is data-plane switches, their flows,

and their connecting links. In the next step, we need to design and send stealthy probing packets towards the discovered switches and their connecting links. Finally, the last step is collecting the stealthy packets and verifying the topology. To this end, we design the SPV with two major modules each having corresponding sub-modules to fulfill the above-mentioned requirements. Therefore, SPV has two major modules: Link Verification and Stealthy Packet Handler. The Link Verification module is mainly responsible for identifying and verifying each update in the network topology which comprises Link Update Checker, Topology Mapper, Flow Manager Application and Link Verifier sub-modules. The Stealthy Packet Handler module performs packet collection, packet generation, and packet transmission through the Packet Listener, the Packet Generator, and the Packet Sender sub-modules, respectively. In the following, we describe each module in details.

Link Verification. This module is responsible for tracking and verifying the data plane changes, by making use of stealthy probing packets, which comprises the following modules.

1. **Link Update Checker.** As the very first step of SPV, the Link Update Checker module is responsible for identifying updates in the network through communications with the SDN controller¹ and informing the Link Verification module of any changes made to the network.

This module is installed from the initialization of the network to verify the network topology in an incremental manner.

2. **Topology Mapper.** This module maintains a tree data structure to locally store the up-to-date topology information provided by Link Update Checker. The tree stores the information of data-plane devices, i.e., if it is a host or a switch or a link, along with their specifications, e.g., their status that is

¹The communications is done via northbound API

up/down, and other useful information such as Device IDs, Port IDs and so on. The stored information is later used by the Link Verifier module for the purpose of link verification. Also, the tree is responsible to store the received updates after verification procedure to maintain the validity status of every switch and their connecting links.

3. **Flow Manager.** This module works as an application to the SDN controller to communicate through the northbound API for querying the flows and statistics of a given switch and installing a given flow on a given switch. This module interacts with the Link Verifier module to perform the above-mentioned functionalities and provide the results.
4. **Link Verifier.** This module is the key component in SPV's link verification procedure. The Link Verifier module interacts with Topology Mapper, Flow Manager Application and Stealthy Packet Handler modules. Based on the input data from the Link Update Checker module, the Link Verifier module communicates with the Topology Mapper module to get the link endpoints, and to query the flows and the statistics of them via the Flow Manager Application module. Also, the Link Verifier relies on the Stealthy Packet Handler module, more specifically Packet Collector and Packet Generator modules, which are responsible to generate the stealthy probing packets, and Packet Sender, which is responsible to transmit the packets to be traversed through the links to be verified.

Stealthy Packet Handler. This module is responsible for generating, sending and collecting stealthy probing packets to/from data plane. Details of corresponding modules are discussed below.

1. **Packet Generator.** This module is responsible to generate stealthy probing

packets, namely, `SPV_PKT` packets, with the help of the Packet Collector module upon receiving a request from the Link Verifier module. The generation of stealthy probing packets is performed using two different algorithms (discussed in Section 3.2.3) depending on two possible situations. (i) A link is being verified for the first time, and (ii) a certain link is being verified again, i.e., further rounds of verification for a certain link which is not yet verified, due to the reasons such as loss of `SPV_PKT` packets.

2. **Packet Collector.** This module collects two types of packets. Firstly, it collects and stores `OFTP_PACKET_IN` messages, that contain host generated packets in their content, in its local database, namely, `PKT_DB`, which are later used by Packet Generator in the stealthy packet generation algorithms. Secondly, it collects and stores `OFTP_PACKET_IN` messages that have an `SPV_PKT` in their content and report them to the Link Verifier as soon as it collects them.
3. **Packet Sender.** This module manages to send stealthy probing packets towards the source endpoint switches of the links to be verified. More specifically, the first responsibility of this module is to receive a stealthy probing packet and information of the link to be verified from the Link Verifier module. The second responsibility is to send the received packet to be traversed through the link to be verified.

3.2.2 Link Verification Algorithm

This section describes the steps of Link Verification mechanism.

Step 1: Tracking Network View Updates. The first step of SPV is to capture every change in the network topology and to maintain a local view of the network state. To this end, the Link Update Checker continuously monitors the network view

of the SDN controller to detect updates in the network topology starting from the initialization of the network. As soon as an update is detected, the Topology Mapper is informed to keep track of them by storing them locally. Then, the Topology Mapper notifies the Link Verifier of the emergence of any new link added to the network view.

Example 1. Suppose in Figure 9 switch S14 is a malicious switch and tends to poison the network view by creating a fake link between switches S11 and S12 (shown by dashed lines in Figure 9) by relaying LLDP packets between switches S11 and S12. In SPV’s design, the very first step for link verification is detecting updates in the network topology view. So, Link Update Checker communicates with SDN controller to get the new link information i.e., link S11:Port2-S12:Port2 (which is a fake link and need to be validated). After detecting link S11:Port2-S12:Port2 via the Link Update Checker module, the Topology Mapper is informed of the existence of this new link and the link information is stored locally with the status of the link set to “Not-Validated”. Then the Link Verifier is informed of the new link that must be validated.

Step 2: Collecting Data for Packet Generation. In this step, the Link Verifier obtains information about the newly added link along with the information of its endpoint switches, their flows, and statistics through the Flow Manager Application module. Upon a request from the Link Verifier, Stealthy Packet Handler generates an SPV_PKT packet and sends it back (the details of generating the packet is discussed in Section 3.2.3). This packet is a stealthy probing packet designed for the verification of a specific data-plane link.

Example 2. In this step (as shown in Figure 9), the Link Verifier queries the flows and statistics of switches S11 and S12 via the Flow Manager Application module and sends the queried information to the Packet Generator module. Upon receiving a request from the Link Verifier module, Packet Generator generates a stealthy probing

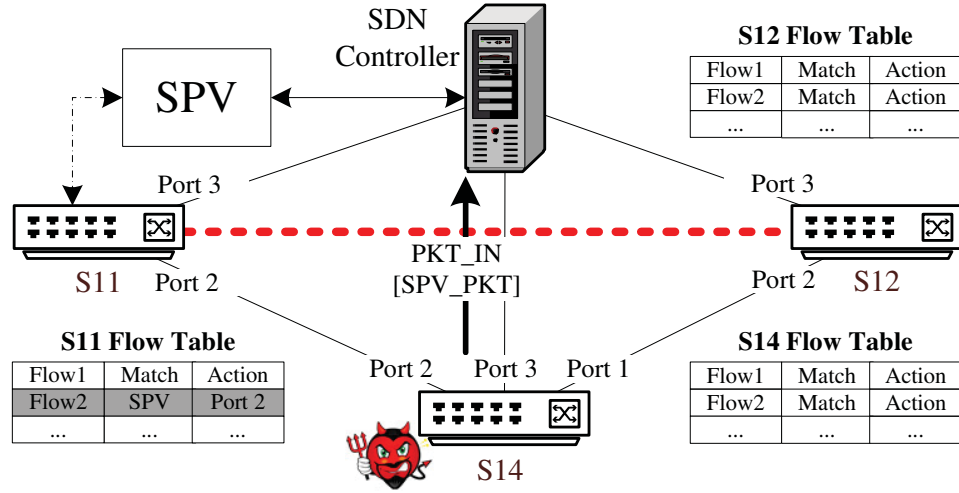


Figure 9: Verification of a newly added link utilizing SPV

packet, i.e., SPV_PKT packet, specific for verifying link S11:Port2-S12:Port2 (details of generating SPV_PKT packet is discussed in Section 3.2.3) and send it back to the Link Verifier.

Step 3: Installing Flow to Forward SPV_PKT. The third step is mainly to install a flow to source endpoint switch of the link to be verified. Utilizing the link information, i.e., link endpoints and their connecting ports, and the returned SPV_PKT packet from the Packet Generator module, the Link Verifier creates a flow. Then, with the help of the Flow Manager Application module, the created flow is installed on the source endpoint of the link to be verified. This flow helps the source endpoint switch, of the link to be verified, to forward the received SPV_PKT packet on its outgoing port towards the destination endpoint switch of the link. Note that the flow is generated to forward a specific SPV_PKT packet towards another switch and is deleted from the switch after a successful round of verification of the given link. For the bi-directional links, SPV follows the similar approach, but in both directions.

Example 3. Based on the received SPV_PKT packet and the information of link S11:Port2-S14:Port2, Link Verifier module generates and installs a flow (shown

by a shaded row in the flow table, **S11 Flow Table** in Figure 9) on switch **S11**, i.e., the flow to match the mentioned specific **SPV_PKT** to forward it on port **S11:Port2**, with the help of the Flow Manager Application module.

Step 4: Collecting SPV_PKT Packets. Based on Step 3, this step mainly waits until it receives the response from Stealthy Packet Handler module within a given time threshold, and obtains the packet information of a received **SPV_PKT** packet. Within the mentioned time threshold, if the Stealthy Packet Handler receives an **OFTP_PACKET_IN** message with a content of the **SPV_PKT** packet, it informs the Link Verifier by sending the packet to it. Alternatively, if the packet is dropped for any reason, after a certain timeout, the Link Verifier is informed with timeout message for the given **SPV_PKT** packet that had been sent for verifying a specific link.

Example 4. After successful installation of the flow on switch **S11**, Link Verifier requests the Packet Sender to send the **SPV_PKT** packet to switch **S11**, as depicted in Figure 9. The switch **S11** receives the **SPV_PKT**, finds a match in its flow table for it, and forwards the packet on its port **S11:Port2**. After the packet is received by switch **S14**, since the port **S11:Port2** is connected to **S14:Port2** and not **S12:Port2**, switch **S14** performs a lookup in its flow tables to find a match for it. However, switch **S14** does not find any match for the received stealthy **SPV_PKT** packet, therefore, it encapsulates the **SPV_PKT** in an **OFTP_PACKET_IN** message and forwards it to the SDN controller. At this point, the Packet Collector module is continuously listening to SDN controller and data plane communications and upon receiving an **OFTP_PACKET_IN** with an **SPV_PKT** in its contents, it adds the **OFTP_PACKET_IN** message to the **PKT_DB** database (the details of **PKT_DB** database is discussed in Section 3.2.3) and send a copy of it to the Link Verifier module.

Step 5: Validating a Specific Link. The final step of the link verification procedure is to decide on the validity of a link. To this end, there are two possible scenarios.

In the first scenario, the Link Verifier receives an `OFTP_PACKET_IN` message with a content of the `SPV_PKT` packet and examines the `SPV_PKT` packet to match it with the link that the packet is originally designed for, more specifically, the matching is performed by utilizing a hash value that has been inserted into the payload of `SPV_PKT` packets. At this point, if the `SPV_PKT` packet is sent back from the destination endpoint switch of a given link, the link is validated and marked as “Legitimate”. Otherwise, the `SPV_PKT` packet is sent back from a switch other than the destination endpoint of a given link. In this case, the link and its sender switch are marked as “Malicious”.

More intuitively, since SPV does not rely on LLDP packets, and because `SPV_PKT` packets are indistinguishable from other packets that are already traversing the network, the malicious switches are deceived and caught. In the second scenario, the `SPV_PKT` packet is not received by the Packet Collector for any reason, e.g., it is dropped due to network congestion or packet loss possibilities, and the link stays in “Not-Validated” state until the next verification round.

Example 5. After the Link Verifier module receives information from Packet Collector module, it queries for the `SPV_PKT` packets with the `Link_ID` of the `S11:Port2-S12:Port2` from the `PKT_DB` database (depicted in Figure 10). Then it confirms the identity of the `SPV_PKT` packet using the hash value provided in its payload, i.e., the hash of the ID of the `SPV_PKT` packet and the timestamp that it has been generated, by comparing the mentioned hash value with the calculated hash value earlier in the process of `SPV_PKT` packet generation. If both values match, the link `S11:Port2-S12:Port2` is validated and marked as “Validated” in the tree data structure with the help of the Topology Mapper module. However, since in this example the received `SPV_PKT` packet is sent from switch `S14`, the `DPID` of the `OFTP_PACKET_IN` message sender and `Switch_ID` value in the `PKT_DB` database for the received `SPV_PKT`

Algorithm 1 Link Verification Algorithm

```
1: Input: links: List of links to be verified
2: procedure VERIFICATIONPREPARATION(links)
3:   for each l ∈ links do
4:     srcSwStat = flowMgr.getStat(l.srcSwID)
5:     dstSwStat = flowMgr.getStat(l.dstSwID)
6:     SPV_PKT = decPktHandler.pktGen(srcSwStat, dstSwStat, l)
7:     SPV_Flow = genFlow(SPV_PKT.header, l.srcSwID, l.srcSwPort)
8:     flowMgr.installFlow(SPV_Flow, l.srcSwID)
9:     while true do
10:      if decPktHandler.pktSender(SPV_PKT, l.srcSwID, l.srcSwPort)
    == "Successful" then
11:        l.SPVPKT = SPV_PKT    ▷ storing the latest SPV_PKT for
each link
12:        break


---


13: Input: PKTIN_SPV: collected OFTP_PACKET_IN sent from Packet Collec-
tor that contains SPV_PKT
14: Input: SwID: the switch ID of the OFTP_PACKET_IN message sender
15: procedure LINKVALIDATION(PKTIN_SPV, SwID)
16:   for each l ∈ links do
17:     if l.SPVPKT == PKTIN_SPV.payload then
18:       if SwID == l.dstSwID then
19:         l.Status = "Legitimate"
20:       else if SwID ≠ l.dstSwID then
21:         l.Status = "Malicious"
22:         maliciousSwList.add(SwID)
```

packet does not match, and both the link S11:Port2-S12:Port2 and the switch S14 are marked as "Malicious".

Algorithm 1 shows Steps 1-4 through the *VerificationPreparation* procedure and Step 5 through the *LinkValidation* procedure.

3.2.3 Stealthy Packet Handler Algorithm

This section discusses the Stealthy Packet Handler algorithm.

Step 1: Collecting OFTP_PACKET_IN Messages. The first step of the Stealthy Packet handler is to collect OFTP_PACKET_IN messages that have a host-generated packet in their content, via the Packet Collector module, and build a pool of packets stored in a locally maintained database, i.e., PKT_DB. These packets are

utilized in designing the SPV_PKT packets in a way that a malicious host or switch is not able to distinguish those packets from a regular host-generated packet that has already traversed the network. Hence, upon receiving an OFTP_PACKET_IN message that contains a host-generated packet in its content, the Packet Collector extracts the header information of the OFTP_PACKET_IN message and the host-generated packet inside the message along with its size and other useful information, and stores them in its local database. The PKT_DB database contains a schema as depicted in Figure 10.

In Figure 10, for each entry in PKT_DB, the PKT_ID specifies a unique identifier for each packet that is stored in the database. The PKT_ID is never exposed to the outside of SPV since it is a secret information which is used for authentication purposes (details are discussed later in this section). PKT_type can be either "Host Packet" or "SPV_PKT", since we collect OFTP_PACKET_IN messages with contents of either host-generated packets or SPV_PKT packets. The Switch_DPID value has the DPID of the switch that has sent the OFTP_PACKET_IN message, this information can be achieved by making use of the MAC address of the OFTP_PACKET_IN message sender switch. PKT_IN_hdr and Data_hdr contain the header of the OFTP_PACKET_IN message and the corresponding packet in its content, respectively. Link_ID contains the ID of the link that the SPV_PKT has been sent to verify its validity. Finally, the timestamp shows the timestamp that an SPV_PKT is generated. Some of the above-mentioned attributes might not contain any value depending on the packet type, i.e., either HostPacket or SPV_PKT.

Step 2: Generating SPV_PKT Packets. This step is to generate stealthy probing SPV_PKT packets for data-plane link verification purpose. Depending on two possible scenarios, i.e., (i) first round of verification of a link and (ii) further rounds of verification of a link, either **Step 2a** or **Step 2b** is performed respectively.

PKT_ ID	PKT_ type	Switch_ DPID	PKT_IN _hdr	Data_ hdr	Data_ size	Link_ ID	Times tamp
------------	--------------	-----------------	----------------	--------------	---------------	-------------	---------------

Figure 10: PKT_DB database table schema

Step 2a. The Packet Collector module searches in its database for the packets that do not have the `Switch_DPID` of the link endpoint switches, i.e., the endpoint switches of a link to be verified must not appear in the `Switch_DPID` field of the packet that is selected from the PKT_DB database. The list of the selected packets, satisfying the mentioned condition, is sent to the Packet Generator module. Then the Packet Generator randomly chooses a packet from the list, forges the source and destination IP and/or MAC addresses (depending on an IP or Ethernet packet), and ports, and sends it to the Link Verifier module. At this point, a unique ID i.e., `PKT_ID`, is assigned to the packet to be used in packet matching. Since OpenFlow switches do not process packet payloads, the Packet Generator calculates the hash value of concatenation of `PKT_ID` and the timestamp of the moment that the packet is generated, i.e., $\text{Hash}(\text{PKT_ID}||\text{timestamp})$, and stores it as the `SPV_PKT` packet payload. If the payload size of the original packet is larger than the size of the hash value, we append dummy data to the `SPV_PKT` packet payload. A copy of the sent `SPV_PKT` packet is stored in the PKT_DB database along with the link information, i.e., `Link_ID`, which is used in further rounds of link verification (Step 2b).

Step 2b. If there is a match for a given link's ID inside the PKT_DB database, the Packet Generator fetches the related `SPV_PKT` packet and sends the packet to a Line Sweep algorithm which produces the next probing packet for verifying this link. Upon receiving the returned packet, the payload is added similarly as mentioned in Step 2a. Then, the newly generated `SPV_PKT` packet is sent to the Link Verifier and replaced with the previous `SPV_PKT` packet inside the PKT_DB database for a given `Link_ID`. We further discuss the details of the line sweep algorithm in the following.

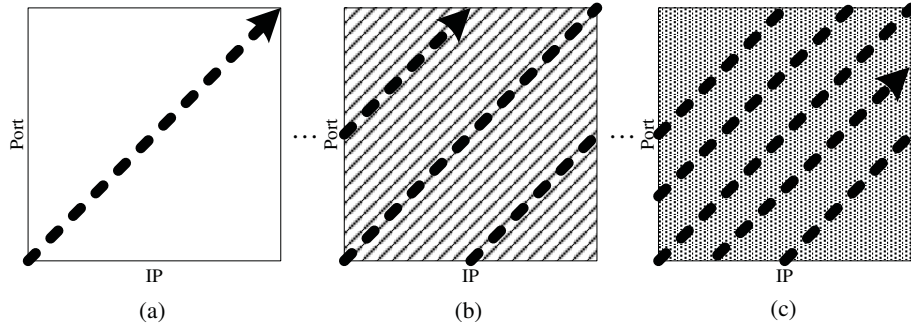


Figure 11: Line sweep mechanism for selecting stealthy probing packets used in second or further rounds of verification

As mentioned earlier, the Packet Generator records the latest SPV_PKT packet for each link to be used to generate the next probing packets. After generating an SPV_PKT packet for the first round of verification for a certain link, the generated packet is sent to a sweep line algorithm proposed in [26] to generate further probing packets. Usually, a sweep line algorithm is used to solve a problem by sweeping an imaginary line over a dimension stopping at some points. In the context of our work, the problem is to choose the next probing packets in a way that we keep the stealthiness of the chosen SPV_PKT packets. In [26], the line sweep algorithm is used to choose probing packets to infer firewall rules, and the method searches for packet headers in $(n-1)$ shapes in an n -dimensional space, by searching for a straight line that is not diagonal but having the characteristics of vertical or horizontal. The method generates packets that cover the sweeping line with a 45-degree angle between starting and ending points. After the algorithm searched for all packets (Figure 11(a)), it chooses another line and repeats the procedure till the whole space is covered (Figures 11(b) and 11(c)).

We leverage the line sweep algorithm in [26] with some variations similar to [31] for selecting SPV_PKT packets' header information for second or further rounds of verification procedure of links that are not verified in the first round of verification

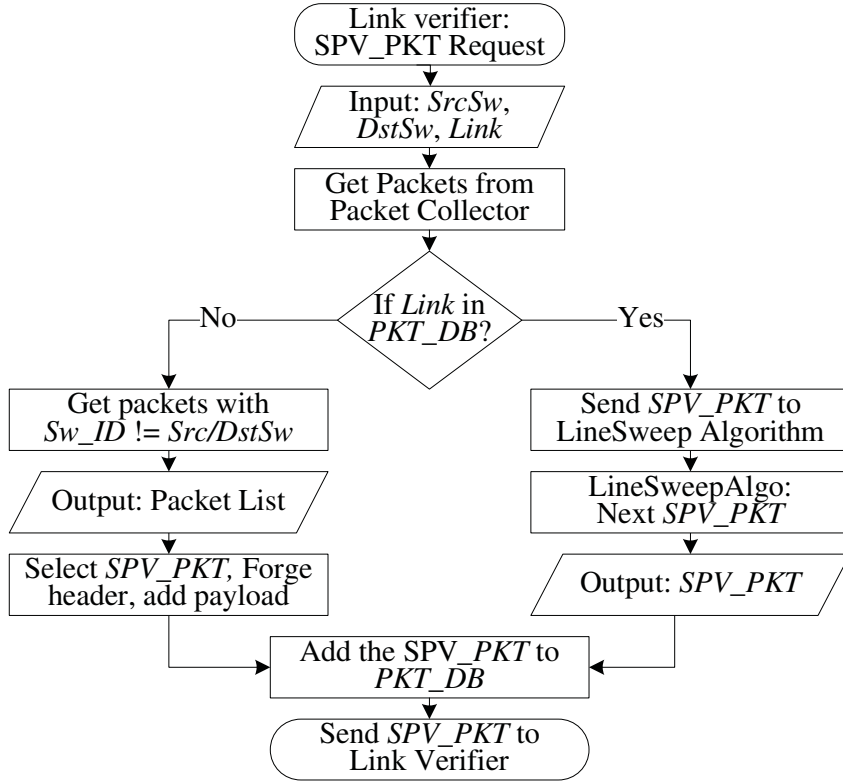


Figure 12: SPV_PKT packet generation procedure flow chart

procedure. The algorithm is used to select the IP addresses, and port numbers of the SPV_PKT packet header. So, based on each SPV_PKT packet of a given link, to select the next probing packet, the IP addresses and port numbers are incremented sequentially and simultaneously.

Figure 12 provides the flow chart for SPV_PKT packet generation procedure.

Step 3: Sending SPV_PKT Packets. This step is to transmit SPV_PKT packets towards a given switch for the purpose of data-plane link verification. Upon receiving a request from the Link Verifier module, a generated SPV_PKT packet is sent towards the source endpoint of the link to be verified, via the Packet Sender module.

Algorithm 2 details the above-mentioned steps.

Algorithm 2 Stealthy Packet Handler Algorithm

```
1: Input: PKT_IN: the exchanged OFTP_PACKET_IN messages between Open-
   Flow switches and the SDN controller
2: Input: SwID: the switch ID of the OFTP_PACKET_IN message sender
3: procedure PKTCOLLECTOR(PKT_IN, SwID)
4:   if PKT_IN.payload  $\neq$  controlMsg then
5:     pktDataBase.add(PKT_IN, SwID)
6:     if PKT_IN.payload  $==$  SPV_PKT then
7:       LinkVerification.LinkValidation(PKTIN_SPV, SwID)


---


8: Input: srcSwStat: flows and status of source endpoint of the link
9: Input: dstSwStat: flows and status of destination endpoint of the link
10: Input: link: the link information
11: procedure PKTGEN(srcSwStat, dstSwStat, link)
12:   pkts = getPktsFromDB(srcSwID, dstSwID)
13:   if link.SPVPKT  $\neq$  null then
14:     pkt = lineSweepPktGen(l.SPVPKT)
15:   else
16:     pkt = pktSelector(pkts)  $\triangleright$  select and forge the header of a random packet
17:   return pkt


---


18: Input: PKT: a packet to be sent
19: Input: SwID: the switch ID to send the packet to
20: Input: SwPort: the port number of the switch to send the packet on
21: procedure PACKETSENDER(PKT, SwID, SwPort)
22:   if NorthBoundAPI.SendPacket(PKT, SwID, SwPort)  $==$  "Successful"
   then
23:     return "Successful"
24:   else
25:     return "Unsuccessful"
```

Chapter 4

Implementation and Experiments

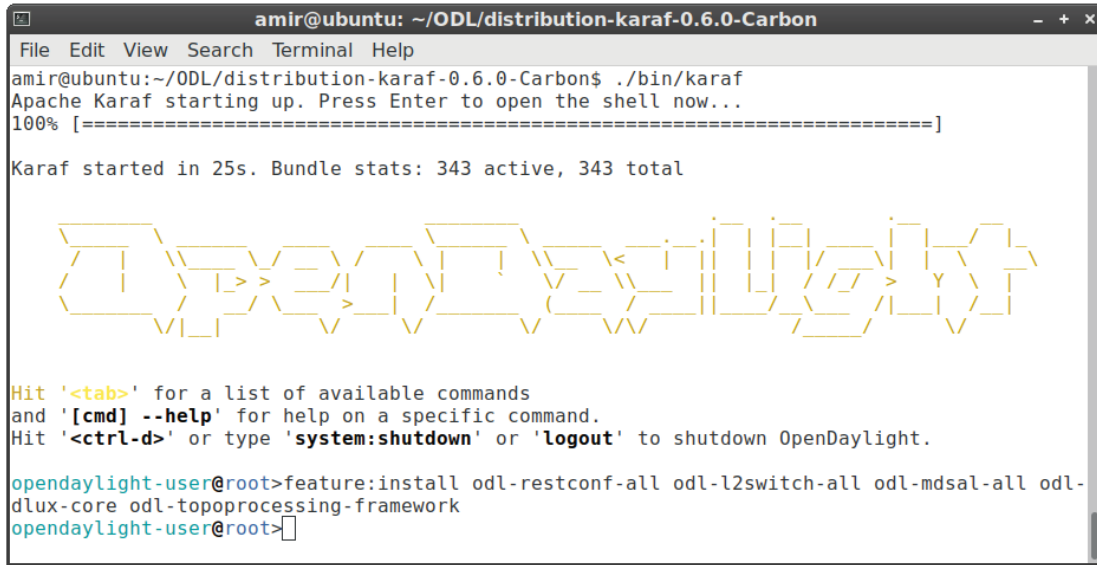
This chapter describes the implementation details of SPV and provides the experiments conducted to evaluate the SPV.

4.1 Implementation

Before we delve into the implementation details of SPV, we provide an overview of the environment and the tools we use to implement SPV, as well as an LLDP relay attack implementation.

4.1.1 Environment Setup

We use OpenDaylight (ODL) [1] as the SDN controller and implement the data plane using Mininet network emulator [28, 38]. ODL is an open platform for automating large scale networks. ODL has a model-driven service adaptation layer (MD-SAL) architecture, in which network applications and network devices are considered as objects and communicate with each other by using the northbound and southbound API plugins' functions, that are provided by SAL [37, 1]. Mininet [24, 38] is a network emulator that can be used to deploy OpenFlow switches and virtual hosts while

A terminal window titled "amir@ubuntu: ~/ODL/distribution-karaf-0.6.0-Carbon" showing the startup of Apache Karaf. The terminal output includes: "Apache Karaf starting up. Press Enter to open the shell now...", "100% [=====]", "Karaf started in 25s. Bundle stats: 343 active, 343 total", a yellow ASCII art logo for OpenDaylight, and instructions: "Hit '<tab>' for a list of available commands and '[cmd] --help' for help on a specific command. Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight." The user then enters the command "feature:install odl-restconf-all odl-l2switch-all odl-mdsal-all odl-dlux-core odl-topoprocessing-framework" and the prompt returns to "opendaylight-user@root>".

```
amir@ubuntu: ~/ODL/distribution-karaf-0.6.0-Carbon
File Edit View Search Terminal Help
amir@ubuntu:~/ODL/distribution-karaf-0.6.0-Carbon$ ./bin/karaf
Apache Karaf starting up. Press Enter to open the shell now...
100% [=====]

Karaf started in 25s. Bundle stats: 343 active, 343 total

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>feature:install odl-restconf-all odl-l2switch-all odl-mdsal-all odl-
dlux-core odl-topoprocessing-framework
opendaylight-user@root>
```

Figure 13: Installing required features on ODL controller

connecting them using virtual links in a single machine.

4.1.1.1 Control Plane Setup

In our setup, we install ODL Carbon release on a virtual machine running a Linux Ubuntu server 16.04 with two Intel(R) Xeon(R) E3-1271 v3 CPUs and 6GB of RAM. We mainly focus on the SDN controller functionality of ODL by making use of the REST northbound API and OpenFlow southbound API. Furthermore, in our work, we utilize some of ODL’s features such as a) *odl-restconf*, b) *odl-l2switch*, c) *odl-mdsal*, d) *odl-dlux*, e) *network-topology* and f) *packet-processing*. Figure 13 illustrates the installation of the above-mentioned features on ODL controller.

The mentioned features are installed on ODL to mainly provide access to rest API, optimization of packet forwarding for learning switches, generic support for applications, graphical user interface, network topology and processing and forwarding packets along with instructions, respectively. We configure ODL to instruct data-plane switches that join the network to install reactive flows instead of proactive flooding

flows and to forward the newly received packets to the ODL for further instructions through the following configuration file. Moreover, proactive flows for LLDP packets (Ether type 0x88cc) is installed upon joining a switch to the network for discovering the network topology. The above-mentioned configurations are depicted in Listings 5.1, 5.2 and 5.3, respectively.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <arp-handler-config xmlns="urn:opendaylight:packet:arp-handler-config">
3   <is-proactive-flood-mode>false</is-proactive-flood-mode>
4 </arp-handler-config>
```

Listing 4.1: Disabling proactive flooding flow installation

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <l2switch-config xmlns="urn:opendaylight:packet:l2switch-config">
3   <is-install-dropall-flow>false</is-install-dropall-flow>
4 </l2switch-config>
```

Listing 4.2: Sending new packets to controller for further instructions

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <loop-remover-config xmlns="urn:opendaylight:packet:loop-remover-config">
3   <is-install-lldp-flow>true</is-install-lldp-flow>
4   <lldp-flow-table-id>0</lldp-flow-table-id>
5   <lldp-flow-priority>100</lldp-flow-priority>
6   <lldp-flow-idle-timeout>0</lldp-flow-idle-timeout>
7   <lldp-flow-hard-timeout>0</lldp-flow-hard-timeout>
8 </loop-remover-config>
```

Listing 4.3: Send LLDP packets to controller

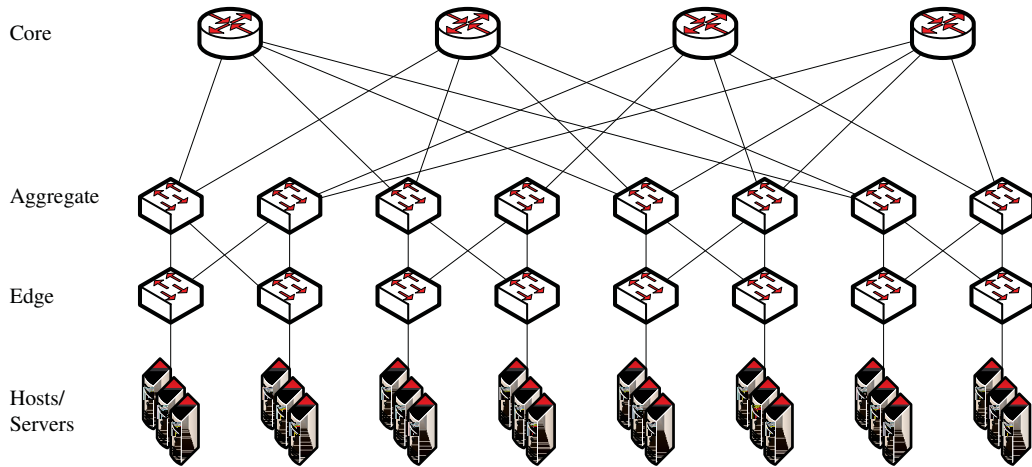


Figure 14: A fat-tree topology with four core switches

4.1.1.2 Data Plane Setup

To set up data-plane devices, we utilize Mininet 2.2.1 on a separate Linux virtual machine running Linux Ubuntu server 16.04 with two Intel(R) Xeon(R) E3-1271 v3 CPUs and 4GB of RAM. We utilize OpenFlow version 1.3, as it is the latest supported version by Mininet 2.2.1. The OpenFlow switches in the data plane are chosen to be software based Open vSwitch [42] switches. We consider a fat-tree topology [3], which is one of the most used network topologies in nowadays large data centers [58], for our data plane. A fat-tree topology comprises four layers; core switches on top, aggregate switches at the second layer, edge switches at the third layer and hosts or servers are at the lowest layer. Figure 14 illustrates a fat-tree topology with four core switches.

4.1.1.3 Traffic Generator

To better simulate an SDN environment, it is required to generate and exchange traffic between hosts, passing through OpenFlow switches in our emulated environment. To this purpose, we utilize iPerf [20], which is a powerful tool to generate both UDP and

```
"Node: h1" (on mininet-vm)
root@mininet-vm:~/mininet_topos# iperf -s -t -w 20m
iperf: ignoring extra argument -- 20m
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 20] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 44080
[ ID] Interval      Transfer    Bandwidth
[ 20] 0.0-10.0 sec  58.7 GBytes 50.4 Gbits/sec
█

"Node: h2" (on mininet-vm)
root@mininet-vm:~/mininet_topos# iperf -c 10.0.0.1 -t -b 500m -t 10
iperf: ignoring extra argument -- 500m
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 19] local 10.0.0.2 port 44080 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 19] 0.0-10.0 sec  58.7 GBytes 50.4 Gbits/sec
root@mininet-vm:~/mininet_topos# █
```

Figure 15: An iPerf TCP traffic generation between hosts h1 and h2

TCP traffic between two endpoints. This tool is later used in Section 4.2 to generate traffic between different pairs of hosts. Figure 15 shows the details of generating a TCP traffic for the duration of 10 seconds between two given hosts, h1 as the TCP server and h2 as the TCP client, and the report after the termination of the session.

4.1.1.4 Attack Implementation

In this section, we implement an LLDP relay packet attack using a malicious switch to show the feasibility of the attack that can occur in an SDN environment. In this implementation, we create a fat-tree topology with two core switches by making use of a Python script which leverages Mininet's python libraries for this purpose. We choose the ODL to be the SDN controller and OpenvSwitch as the underlying data-plane switches. Figure 16 illustrates the network view of the ODL controller on the above-mentioned fat-tree topology.

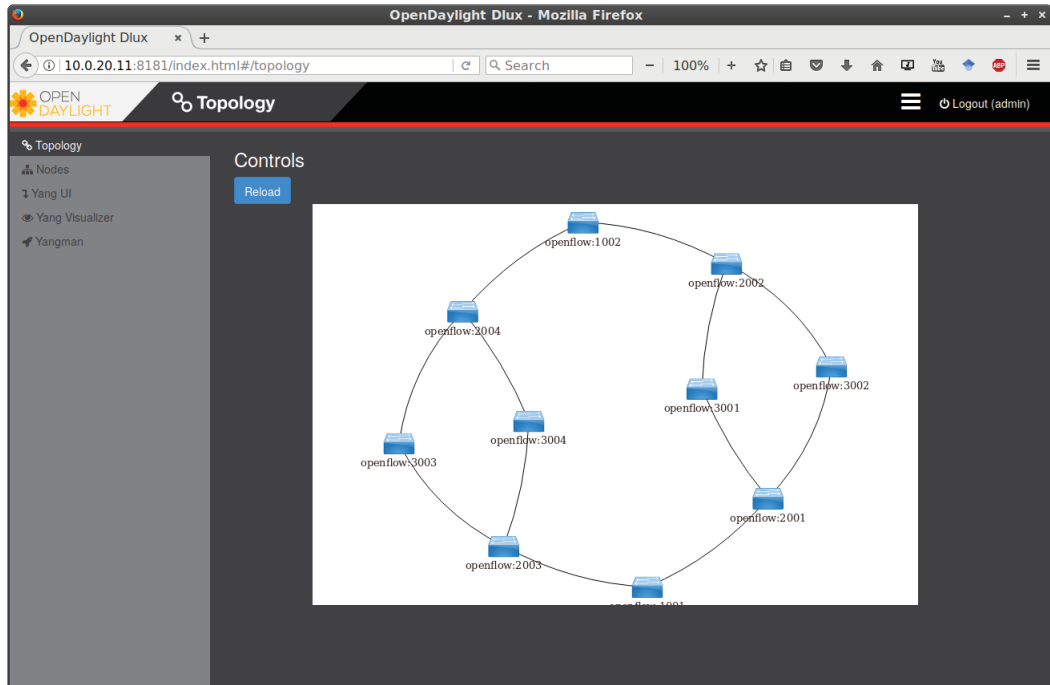


Figure 16: A fat-tree topology with two core switches in the view of the ODL Controller

```

"Node: 2002" (root) (on mininet-vm)
root@mininet-vm:~/mininet_topos# sudo ovs-ofctl dump-flows 2002 -0 openflow13
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0:2b0000000000000a, duration=895.533s, table=0, n_packets=720, n_bytes=65520, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0:2b0000000000000a, duration=895.557s, table=0, n_packets=0, n_bytes=0, priority=1,arp actions=CONTROLLER:65535
root@mininet-vm:~/mininet_topos#

```

Figure 17: Flow rules of switch `openflow:2002` before adding LLDP relay rules

In the shown topology in Figure 16, we try to create a fake link between switches `openflow:3002` and `openflow:1002` in the view of the SDN controller by relaying the LLDP packets between them by modifying the flow rules of the switch `openflow:2002`. Unmodified flow rules of the switch `openflow:2002` are shown in Figure 17.

As it is illustrated in Figure 18, the two flow rules with higher priority are added to relay LLDP packets between ports `port 1` and `port 4` in switch `openflow:2002`.

```

"Node: 2002" (root) (on mininet-vm)
root@mininet-vm:~/mininet_topos# sudo ovs-ofctl dump-flows 2002 -O openflow13
OFFST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b00000000000000a, duration=8.64s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=4,dl_type=0x88cc actions=output:1
 cookie=0x2b00000000000000a, duration=1.888s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=1,dl_type=0x88cc actions=output:4
 cookie=0x2b00000000000000a, duration=994.893s, table=0, n_packets=800, n_bytes=72800, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000000a, duration=994.917s, table=0, n_packets=0, n_bytes=0, priority=1,arp actions=CONTROLLER:65535
root@mininet-vm:~/mininet_topos#

```

Figure 18: Flow rules of switch openflow:2002 after adding LLDP relay rules

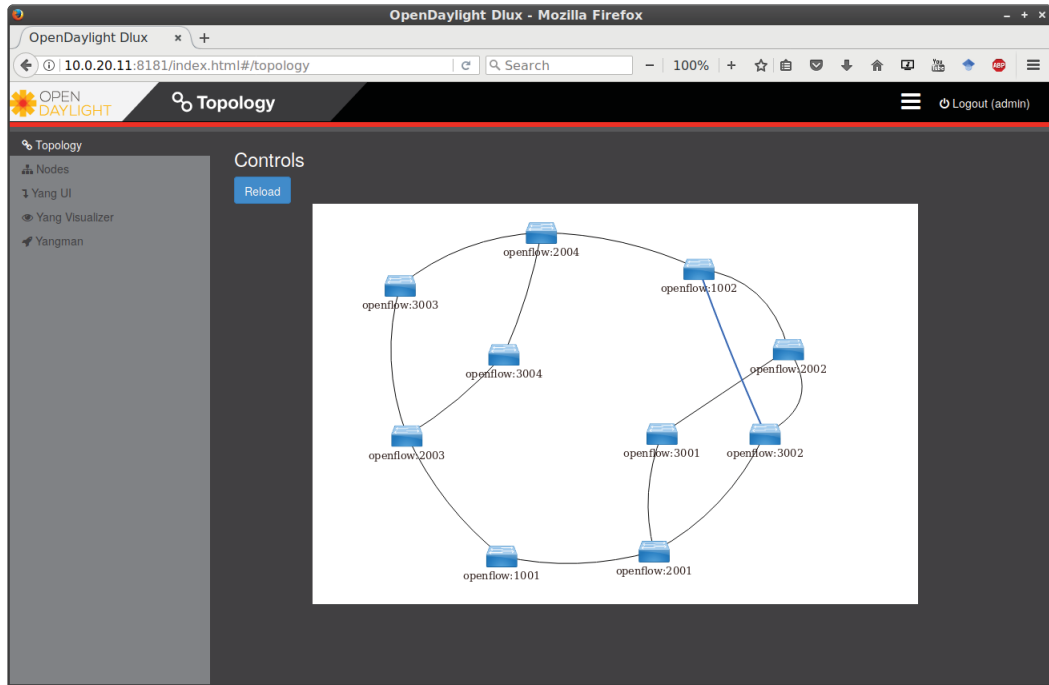


Figure 19: Afake link creation in the view of the ODL Controller

This action creates a fake link in the view of the controller between switches openflow:3002 and opneflow:1002 as depicted with a blue line in Figure 19.

4.1.2 SPV's Implementation Details

We install SPV on the same machine that runs the ODL (i.e., a virtual machine running a Linux Ubuntu server 16.04 with two Intel(R) Xeon(R) E3-1271 v3 CPUs and 6GB of RAM). Note that although SPV and ODL are installed on the same machine, they work independent of each other, meaning that they can be installed on

two separate machines as well. The main programming language used to implement SPV is Java [22]. We utilize Scapy’s [50] Python libraries for the purposes of SPV_PKT packet generation and encoding. We implement the SPV in both single thread and multi-thread modes, by utilizing Java’s multi-threading capabilities in order to enhance the response time of link verification when there are more than one links to be verified simultaneously. The following discusses implementation details of main SPV’s components.

4.1.2.1 Link Verification

The Link Verification module (see Figure 8 for more details) is implemented to communicate with ODL’s northbound API for querying the changes in network topology, storing the topology locally, installing flows on data-plane switches and performing link verification (an example of ODL’s rest API output for a linear network topology with three switches is depicted in Figure 20). More specifically, the Link Update Checker interacts with ODL’s *opendaylight-topology* module to keep track of changes in the data-plane network. Then, SPV’s Topology Mapper module keeps the latest topology locally by utilizing regular expression techniques to parse the output of the *opendaylight-topology*. The data structure to keep the topology is a map of the network topology which consists of links and nodes and their connections along with other useful information, e.g., switch statistics or links’ status. Note that the validity information of a link is also stored within the tree data structure. A part of the output of a linear network topology with three switches after parsing and storing the data locally in SPV is shown in Figure 21 which is an illustration of collected information over a link and an OpenFlow switch.

The next step in Link Verification procedure is to go through the unverified links, send a request to Stealthy Packet Handler module for a generation of an SPV_PKT

```

amir@ubuntu:~$ curl -H Accept:application/json -u admin:admin http://localhost:8181/restconf/operational/network-topology:network-topology/
{"network-topology":{"topology":[{"topology-id":"flow:1","node":[{"node-id":"openflow:3","opendaylight-topology-inventor
y:inventory-node-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendaylight-inventory:id='openflow:3']
","termination-point":[{"tp-id":"openflow:3:2","opendaylight-topology-inventory:inventory-node-connector-ref":"/opendayl
ight-inventory:nodes/opendaylight-inventory:node[opendaylight-inventory:id='openflow:3']/opendaylight-inventory:node-con
nector[opendaylight-inventory:id='openflow:3:2']"}, {"tp-id":"openflow:3:1","opendaylight-topology-inventory:inventory-no
de-connector-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendaylight-inventory:id='openflow:3']/ope
ndaylight-inventory:node-connector[opendaylight-inventory:id='openflow:3:1']"}, {"tp-id":"openflow:3:LOCAL","opendaylight
-topology-inventory:inventory-node-connector-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendayligh
t-inventory:id='openflow:3']/opendaylight-inventory:node-connector[opendaylight-inventory:id='openflow:3:LOCAL']"}]}, {"n
ode-id":"openflow:2","opendaylight-topology-inventory:inventory-node-ref":"/opendaylight-inventory:nodes/opendaylight-in
ventory:node[opendaylight-inventory:id='openflow:2']","termination-point":[{"tp-id":"openflow:2:3","opendaylight-topolog
y:inventory:inventory-node-connector-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendaylight-invent
ory:id='openflow:2']/opendaylight-inventory:node-connector[opendaylight-inventory:id='openflow:2:3']"}, {"tp-id":"openflo
w:2:2","opendaylight-topology-inventory:inventory-node-connector-ref":"/opendaylight-inventory:nodes/opendaylight-invent
ory:node[opendaylight-inventory:id='openflow:2']/opendaylight-inventory:node-connector[opendaylight-inventory:id='openfl
ow:2:2']"}, {"tp-id":"openflow:2:1","opendaylight-topology-inventory:inventory-node-connector-ref":"/opendaylight-invento
ry:nodes/opendaylight-inventory:node[opendaylight-inventory:id='openflow:2']/opendaylight-inventory:node-connector[opend
aylight-inventory:id='openflow:2:1']"}, {"tp-id":"openflow:2:LOCAL","opendaylight-topology-inventory:inventory-node-conne
ctor-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendaylight-inventory:id='openflow:2']/opendayligh
t-inventory:node-connector[opendaylight-inventory:id='openflow:2:LOCAL']"}]}, {"node-id":"openflow:1","opendaylight-topol
ogy:inventory:inventory-node-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendaylight-inventory:id='
openflow:1']","termination-point":[{"tp-id":"openflow:1:LOCAL","opendaylight-topology-inventory:inventory-node-connector
-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendaylight-inventory:id='openflow:1']/opendaylight-in
ventory:node-connector[opendaylight-inventory:id='openflow:1:LOCAL']"}, {"tp-id":"openflow:1:2","opendaylight-topology-in
ventory:inventory-node-connector-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:node[opendaylight-inventory:
id='openflow:1']/opendaylight-inventory:node-connector[opendaylight-inventory:id='openflow:1:2']"}, {"tp-id":"openflow:1:
1","opendaylight-topology-inventory:inventory-node-connector-ref":"/opendaylight-inventory:nodes/opendaylight-inventory:
node[opendaylight-inventory:id='openflow:1']/opendaylight-inventory:node-connector[opendaylight-inventory:id='openflow:1
:1']"}]}, {"link":[{"link-id":"openflow:3:2","source":{"source-node":"openflow:3","source-tp":"openflow:3:2"},"destinatio
n":{"dest-node":"openflow:2","dest-tp":"openflow:2:3"}}, {"link-id":"openflow:2:3","source":{"source-node":"openflow:2","
source-tp":"openflow:2:3"},"destination":{"dest-node":"openflow:3","dest-tp":"openflow:3:2"}}, {"link-id":"openflow:1:2",
"source":{"source-node":"openflow:1","source-tp":"openflow:1:2"},"destination":{"dest-node":"openflow:2","dest-tp":"open
flow:2:2"}}, {"link-id":"openflow:2:2","source":{"source-node":"openflow:2","source-tp":"openflow:2:2"},"destination":{"d
est-node":"openflow:1","dest-tp":"openflow:1:2"}}]}]}amir@ubuntu:~$ ]

```

Figure 20: An output of opendaylight-topology upon a request to query the ODL's network view

```

<terminated> OSDP [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Dec 31, 201
Printing Link
*****
link-id = openflow:1:2
link-srcNode = openflow:1
link-srcPort = openflow:1:2
link-dstNode = openflow:2
link-dstPort = openflow:2:2
link-status = 1
link-type = switchLink
*****
Printing OpenFlow Switch
*****
node-id = openflow:2
node-ports = [openflow:2:3, openflow:2:2, openflow:2:1, openflow:2:LOCAL]
port-status = 0
node-ip = -
node-mac = -

```

Figure 21: Details of a stored link and an OpenFlow switch

packet, and forward it towards the link that is to be verified. The procedure of packet generation and forwarding it towards the link to be verified is discussed in the next subsection.

4.1.2.2 Stealthy Packet Handler

The Stealthy Packet Handler module consists of multiple Java classes along with Python scripts which utilize Scapy [50], i.e., a packet manipulation tool aimed to send, sniff, dissect and forge network packets. The interaction with the Link Verification module is done via a Java class. After a request for packet generation is sent from Link Verifier, a Python script utilizes Scapy's [50] libraries to generate and encode SPV_PKT packets in the Base64 format and send them towards specific links using ODL's northbound API. In Listing 5.4 we illustrate the content of an API request to send a Base64 encoded SPV_PKT packet, using the northbound API of the ODL.

```

1 { "input": {
2     "connection-cookie": 123456,
3     "egress": "/opendaylight-inventory:nodes/opendaylight-inventory:node[
         opendaylight-inventory:id='openflow:1']/opendaylight-inventory:
         node-connector[opendaylight-inventory:id='openflow:1:2']",
4     "node": "/opendaylight-inventory:nodes/opendaylight-inventory:node[
         opendaylight-inventory:id='openflow:1']",
5     "payload": "
         AAAAAACAAAAAABCABFAABQAAEAAEAGZqUKAAAABCgAAAp2cQqMAAAAAAAAFACI
6     AAIuAAANTM3NzdCQkJEQzY3RDQ5RDc2QzNERjJENUQ3OTQzN0EzQTBFNzk3OQ=="
7     }}

```

Listing 4.4: Content of an API request to send an SPV_PKT packet using ODL's northbound API

4.2 Experiments

This section presents our experiments for evaluating the performance of SPV. We conduct four groups of experiments to evaluate the performance and accuracy of SPV. We first measure the efficiency of SPV by varying the complexity of the network. Then, we evaluate the resource consumption, i.e., CPU and memory usage, of SPV. We also measure the effect of different kinds of attackers that tend to increase packet loss, congest the network and relay the traffic, on SPV. We further implement SPV in a real SDN/cloud topology to verify its applicability and evaluate its performance. To this end, first we discuss the network topology used in our experiments, and then each group of experiments is detailed.

Network Topology. As mention in Section 4.1.1.2, we consider a fat-tree topology [3] for our data plane. We vary the switches from five to 40 where the largest topology has eight core switches, 16 aggregate switches, and 16 edge switches, which comply with the size of a medium-sized data center to accommodate tens of thousands of servers [3, 58]. Even though the reported results in the next section show that we could expect SPV to be expandable to large data centers, we only vary the number of switches up to 40 due to the resource limitations of our environment. However, to stress the SPV and evaluate its accuracy, we conduct further set of experiments to measure SPV’s performance for up to 5,000 link verifications in the network.

4.2.1 The Efficiency of SPV

In the first group of our experiments, we measure the time requirement of SPV in different situations. The reported verification time includes the time for selecting the link, generating SPV_PKT packet, installing flow rules, sending and receiving the SPV_PKT packet, and finally verifying both the SPV_PKT packet and the link. Figure 22 shows the time required for SPV to verify one link and Figure 23 shows the

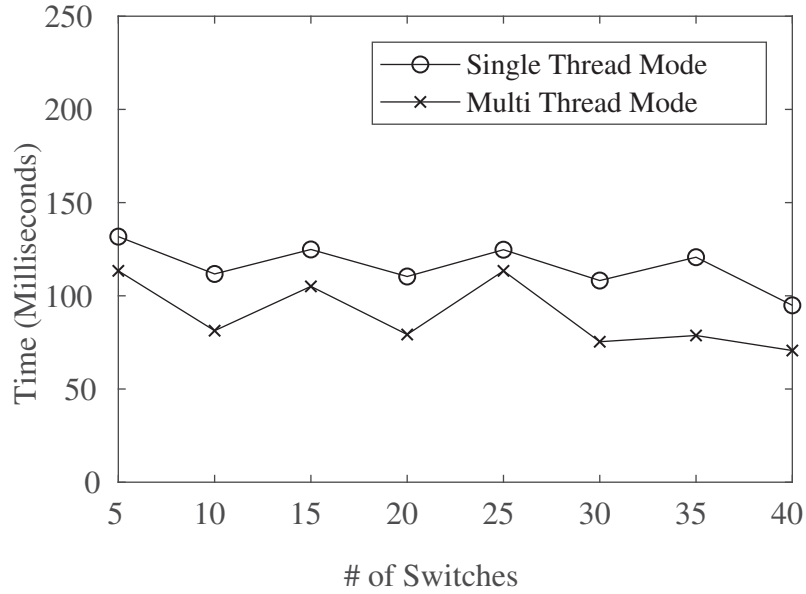


Figure 22: Time required by SPV to verify a new link while varying the number of switches up to 40 and number of links up to 96 in both single and multi-thread modes

time required for SPV to verify all existing links in the network. More specifically, Figure 22 depicts the time in milliseconds to incrementally verify a newly added link in both single thread and multi-thread modes while varying the size of the network by increasing the number of switches from five to 40 with a maximum of 96 data-plane links. In this case, the average verification time is 102 milliseconds, which shows the near-real-time nature of SPV to verify a newly added link. Also, the verification time is independent of the size of the network (e.g., number of switches and their connected links). Whereas Figure 23 shows the time required by SPV to verify a group of links in both single thread and multi-thread modes while varying the size of the network by increasing the number switches from five to 40 with a maximum 96 data-plane links. The verification time for the largest dataset in the single thread mode is 26.1 seconds. We further improve the verification time by conducting them through the multi-thread mode, which reduces the verification time to 10.6 seconds.

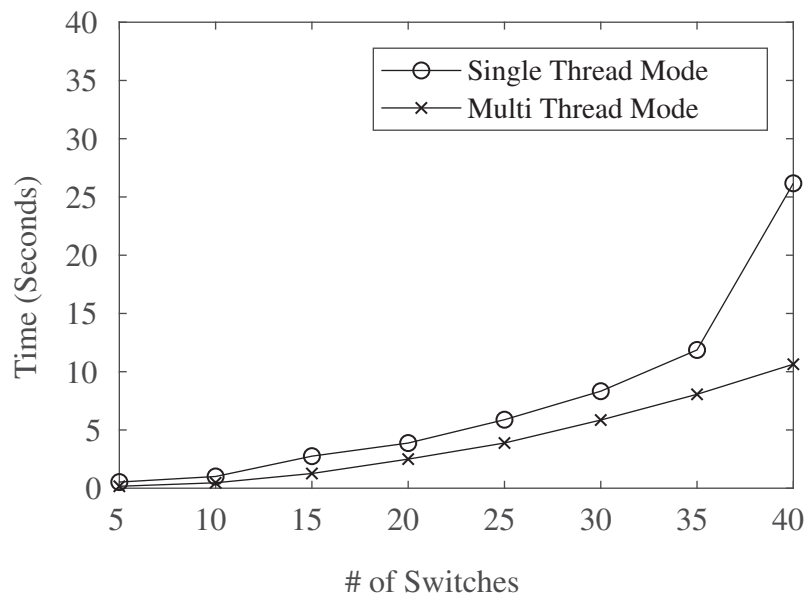


Figure 23: Time required by SPV to verify all existing links while varying the number of switches up to 40 and number of links up to 96 in both single and multi-thread modes

Even though the increase of the number of switches in the network results in the increase of verification time for both single and multi-thread modes, the increase of the verification time remains almost linear in the multi-thread mode. These results show the practicality of SPV in medium-sized data centers to verify their topology.

4.2.2 Resource Consumption by SPV

The second group of the experiments is to measure the resource consumption (i.e., CPU and memory usage) by SPV. Figure 24 depicts the average CPU and Figure 25 depicts the average memory usage to verify all existing links in the network by varying the number of switches for single thread mode. More specifically, Figure 24 shows that SPV on average requires about 20% of the CPU, which is a reasonable amount. Figure 25 depicts the memory consumption by SPV while verifying all existing links

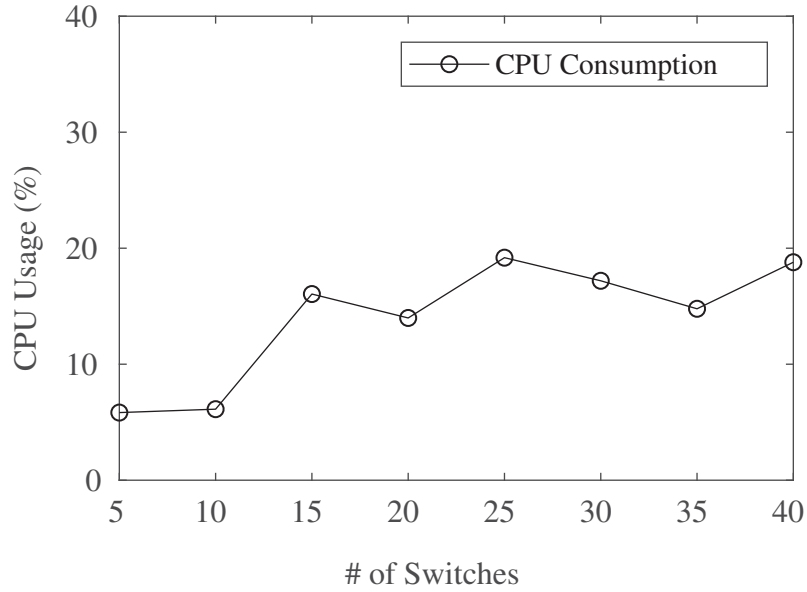


Figure 24: Average CPU usage by SPV to verify all existing links while varying the number of switches up to 40 and the number of links accordingly

for different network sizes. Even though we observe an increase in the memory consumption for larger datasets, it still remains below 2%. Note that, the CPU and memory consumption for verifying a single link is negligible and hence not reported in this thesis.

4.2.3 Evaluating SPV Against Different Attacks

The objective of the third set of experiments is to investigate the effect of packet loss, network traffic and the packet relay attack on SPV. To this end Figures 26 show the percentage of unverified links in the first round of SPV for 5,000 link verifications in the network while varying the packet loss rate and Figure 27 shows the percentage of unverified links in the first round of SPV for 5,000 link verifications in the network while increasing traffic throughput. To reduce this effect, SPV periodically sends SPV_PKT packets until a link is verified and hence, all of the network links are eventually verified. In this set of experiments, we utilize a fat-tree topology with two core

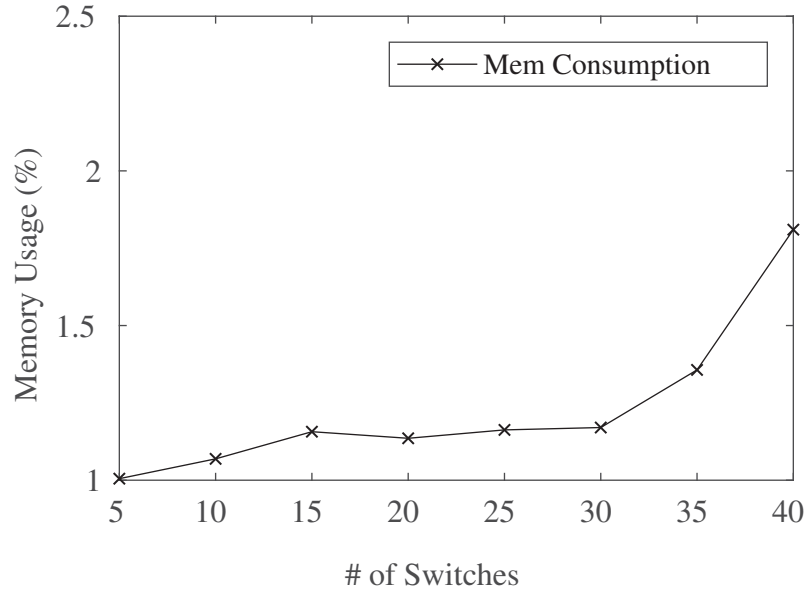


Figure 25: Average memory usage by SPV to verify all existing links while varying the number of switches up to 40 and the number of links accordingly

switches, four aggregate switches and four edge switches where the bandwidth of core switches to aggregate switches are set to 100Mbps and the bandwidth for aggregate switches to edge switches is set to 10Mbps. The spanning tree protocol is used to eliminate loops in the network routing. To generate traffic in this set of experiments, we set up two pairs of iPerf [20] clients and servers in the network and exchange both UDP and TCP traffic between them while changing the iPerf client/server pairs between existing hosts in the network.

More specifically, as it is illustrated in Figure 26 with the increase of the packet loss rate, the percentage of the unverified links linearly increases due to the fact that the SPV_PKT packets may be also lost; However, since the distribution of packet loss rate is among all links in the network, and SPV only deals with one link at a time, the percentage of unverified links is always less than packet loss rate in the network which concludes that our approach is resilient to high packet loss rates in the network. In Figure 27 by increasing the traffic throughput to the maximum allowed

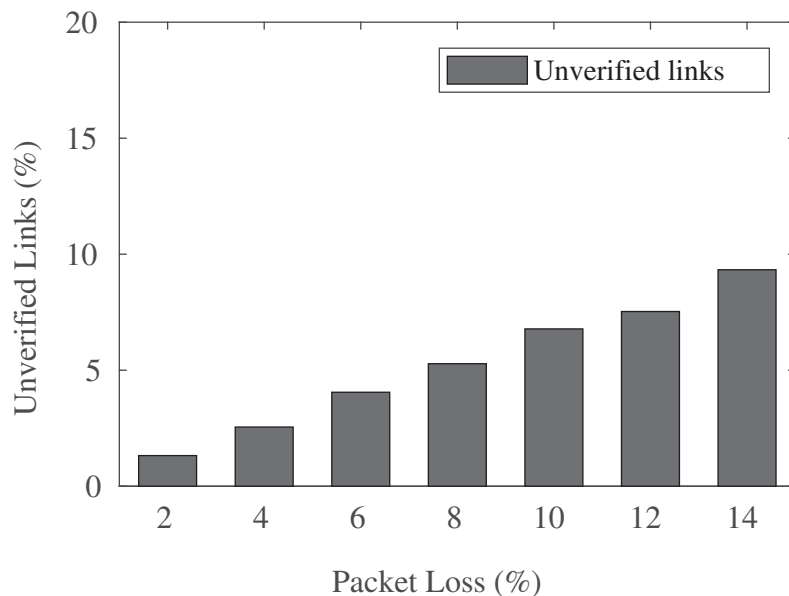


Figure 26: Evaluating SPV with the presence of attackers in the network that tend to increase packet loss while varying the links packet loss rate (%)

bandwidth while keeping the loss rate to 5%, the percentage of unverified links stays almost constant and with a very small amount of change. The reason is that the flow installed by SPV to forward the SPV_PKT packet has a higher priority than other flows on a given switch and this fact makes SPV be independent of network congestion and only be dependent on network packet loss rate in this set of experiment.

As discussed later in Section 5, a malicious switch may forward selected traffic that may or may not include SPV_PKT packets. Hence, we measure the percentage of SPV_PKT packets being affected in the presence of a malicious switch that forwards 10% to 50% of the traffic at different time intervals.

Figure 28 shows that the amount of forwarded SPV_PKT packets remains less than 30% in the case a malicious switch forwards all the packets in 50% of the time, however, in this experiments we considered an extreme case by assuming the malicious switch is not detected until last round of verification which not always true and SPV might detect it much earlier. The resilience of SPV to this attack can be

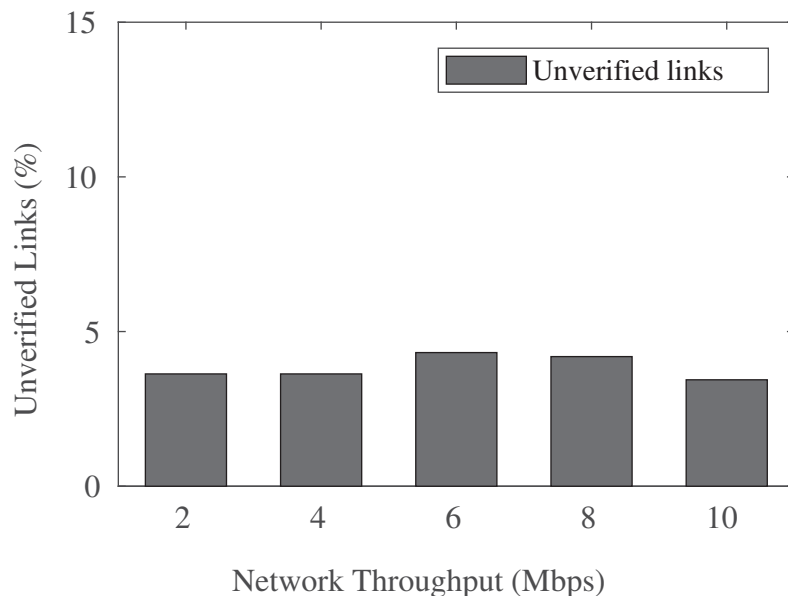


Figure 27: Evaluating SPV with the presence of attackers in the network that tend to congest the network by showing percentage of unverified links while varying traffic throughput (Mbps)

further improved by using different probabilistic functions with different distributions in choosing the time intervals for sending the SPV_PKT packets.

4.2.4 Applicability of SPV in a Real SDN/Cloud Topology

The objective of our last set of experiments is to validate the applicability of SPV in a real SDN/cloud topology. To this end, we utilize an accessible part of the topology of a real SDN/cloud hosted at one of the largest telecommunication vendors that comprises of OpenStack [43] cloud with 22 compute nodes, each having a software-based OpenFlow switch, to reside thousands of VMs. All 22 OVS switches are connected to each other in a mesh architecture having 231 bi-directional links. We further run the mentioned topology with ODL controller to be able to run SPV. Table 3 reports the results obtained based on this real topology and compares them with the results for our largest fat-tree topology having 40 switches and 96 links. The results illustrate

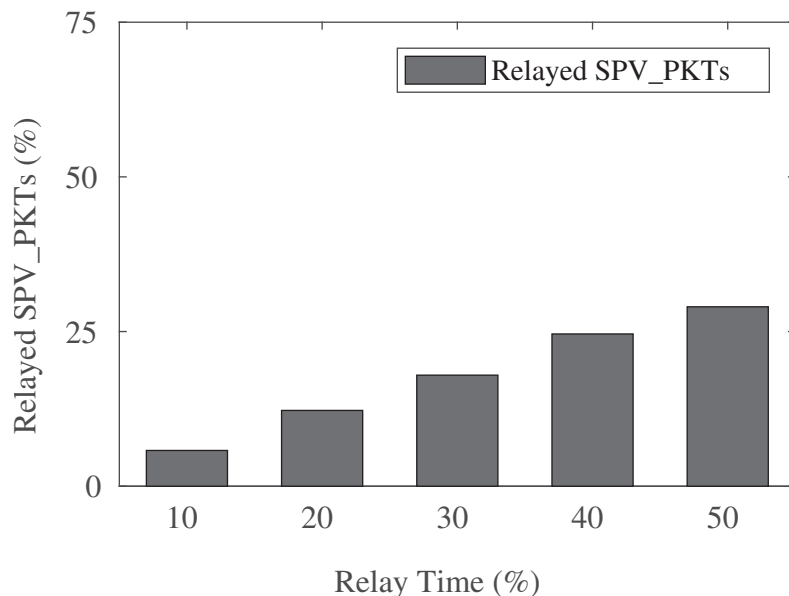


Figure 28: Evaluating SPV with the presence of attackers in the network that tend to relay the traffic by measuring the percentage of relayed SPV_PKT packets while varying the percentage of the time that the attacker may relay all the traffic

Table 3: SPV’s performance on a real SDN/cloud topology and Mininet in both single (ST) and multi (MT) thread modes

Results	SDN/Cloud Mesh	Mininet Fat-Tree
All link Verification Time (MT)	13.2052 s	10.6406 s
All link Verification Time (ST)	134.147 s	26.1725 s
Single Link Verification Time (ST)	100.306 ms	94.8919 ms
CPU Consumption (ST)	8.50294 %	18.79099 %
Memory Consumption (ST)	1.81817 %	1.81003 %

that the verification time for an incremental link verification in the network remains almost constant (i.e., in average around 100ms) independent of network topology, and the verification time for all links in the multi-thread mode can rise up to 13 seconds. Table 3 also shows other measurements such as CPU and memory consumption of SPV in both topologies. Those results indicate a reasonable performance and demonstrate the practicality of our approach in a real-world SDN/cloud.

Chapter 5

Security Analysis and Discussion

This chapter discusses the security analysis of SPV as well as further discussions over SPV.

5.1 Security Analysis

This section discusses how SPV may overcome various attacks.

5.1.1 Stealthiness Feature of SPV_PKT packets

Security of SPV relies on the so-called stealthiness property of a probing packet, meaning the packet cannot be distinguished from other host-generated packets traveling through the network. To this end, SPV builds a pool of packets, by collecting the traversed packets between the hosts. Afterwards, SPV selects packets from the pool, modifies their header and forwards them as SPV_PKT packets to selected switches to verify the links between them (Section 3.2.3 discusses the SPV_PKT generation procedure). The following discusses how different fields of SPV_PKT packet may affect the stealthiness property.

In an SPV_PKT packet, some fields must not match with those of any host-generated

packets, whereas the rest of the fields must match, in order to ensure the stealthiness property. Suppose P is a pool of host-generated packets, that have been traversed through the network, decapsulated from `OFTP_PACKET_IN` messages sent to the SDN controller. An `SPV_PKT` packet, `SPV_PKT_i` is generated based on a reference packet from the pool P . In `SPV_PKT_i`, `Eth_Type` field and packet's payload size match with those of the reference packet. Otherwise, there is a high chance that a malicious switch might identify the `SPV_PKT_i` based on the unusual fields of the packet. On the other hand, `Src_IP`, `Dst_IP`, `Src_MAC`, `Dst_MAC`, `Src_Port` and `Dst_Port` fields in `SPV_PKT_i` must not match with the same fields of the reference packet. Otherwise, a malicious switch might be suspicious when it observes similar packets with the same source and destination addresses multiple times in the network.

In the following, we consider different threats to the stealthiness property and discuss how SPV can address them.

5.1.2 Relaying or Dropping Packets

A malicious switch may evolve to react to SPV by forwarding traffic including SPV packets passing through it so that the fake link remains undetected.

However, to be undetected by SPV, the malicious switch must forward every single packet traversing the fake link. Therefore, this reaction is against the original objective of the attacker, because by forwarding all the traffic (i.e., creating a fake link to attack availability), the malicious switch is actually working as a link. Hence, a more practical adversary might partially forward the traffic within certain time intervals (since SPV randomly chooses time intervals to send `SPV_PKT` packets) hoping that it might also forward the `SPV_PKT` packets among the forwarded traffic to keep the fake link undetected. We further measured the effect of such attack on SPV in Section 4.2. Moreover, to detect such attacks, one potential solution is to compare

time differences between packets traversing on different links; which is considered as a potential future work.

Alternatively, a malicious switch may choose to only relay the LLDP packets and drop every other packet including SPV packets in an attempt to confuse SPV. However, this kind of attack is beyond the scope of this work, since SPV focuses on the integrity of SDN controller’s view over the network topology and this attack targets the availability of the network functionality.

5.1.3 Replaying SPV_PKT Packets

With the presence of an intelligent malicious switch (i.e., a soft switch), a forged SPV_PKT packet might be generated and used to degrade the accuracy of SPV. A malicious attacker may mimic the packet generation mechanism of SPV and utilize the history of packets to generate forged SPV_PKT packets and send them back to the SDN controller, pretending the SPV_PKT packet is returning from the other end of the fake link, to deceive SPV. However, as discussed earlier in Section 3.2.3, SPV addresses this issue by authenticating each received SPV_PKT packet. More specifically, to enable authentication of SPV_PKT packets, SPV adds a unique hash value, i.e., $\text{Hash}(\text{PKT_ID}||\text{timestamp})$, for each SPV_PKT packet in their payload to prevent the replay attack on SPV_PKT packets. Moreover, through the aforementioned unique hash value, which includes a freshness proof (i.e., timestamp), SPV prevents an attacker from replaying already used and valid SPV_PKT packets to hide another fake link.

5.1.4 Learning SPV_PKT Packet’s Structure

Even though OpenFlow switches by design are not meant to process packet payloads, the presence of soft switches in the network allows an attacker to learn the

nature of SPV_PKT packets by processing the network traffic. For example, an intelligent malicious switch might process the OFTP_PACKET_IN messages that contain host-generated packets to measure their payload size to distinguish SPV_PKT packets and stop forwarding them. This might happen since the hash value that is inserted into the payload of an SPV_PKT packet has a constant size. To address this issue, we keep the payload size of the SPV_PKT packet the same as the payload size of the reference packet from which the SPV_PKT packet is generated, by padding dummy data to SPV_PKT packet's payload. Thus, SPV overcomes malicious switches that might abuse the packet payload size to breach the stealthiness property of SPV_PKT packets.

Moreover, selecting packets from the packet pool and forging their header fields randomly might not help SPV to hold the stealthiness property. Since an intelligent malicious switch might keep a history of packets and leverage learning methods to keep track of packet types and their header to correctly identify SPV_PKT packets. To overcome this problem, we send the chosen SPV_PKT packets to a line sweep algorithm (as discussed in Section 3.2.3) to choose the header information of next probing packets for further verification rounds. This helps the SPV to hold the stealthiness property of SPV_PKT packets by keeping the source and destination addresses of each SPV_PKT packet very close to a previously sent SPV_PKT packet for a specific link for a certain period of time.

5.1.5 Injecting Packets by a Malicious Host

A malicious host may inject unique packets into the network to influence the SPV_PKT packet generation in order to breach the stealthiness feature of SPV_PKT packets. For this purpose, a malicious host may inject packets with some specific fingerprints such as unique payload size or type. However, this attack only works when the following conditions are satisfied: i) the SPV_PKT packet is generated out of the injected packets

by the malicious host, which can be achieved by a considerable amount of injected packets into the network without being detected by other network security solutions such as IDS, ii) the owner of the malicious host also has a compromised switch, which is not connected to the same host (as SPV_PKT packets are never chosen to verify the same switch from where the original packet has been collected), and iii) the SPV_PKT packet is sent to verify the connecting links of the same compromised switch which involves a considerable amount of uncertainty added by SPV.

5.2 Discussion

This section discusses different concerns on SPV.

5.2.1 Exhausting Flow Table Capacity of SDN Switches

There exist several attacks (as described in [30, 52]) to exhaust the flow table capacity of SDN switches which may affect the availability by adding new flow rules on those switches. Consequently, these attacks may affect the verification process of SPV by preventing the installation of flow rules related to forwarding the SPV_PKT packets. Even though such flow table overflow attacks are beyond the scope of this work, existing solutions (e.g., [59]) to address them could be leveraged to avoid any effect on SPV.

5.2.2 Effect of Encrypted Communication Between Control and Data Planes

In case TLS is enabled for secure communications, administrators require to share en/decryption keys of control messages with SPV; which might be a practical assumption for an administrator intending to protect his/her network topology from

poisoning attacks. However, if SPV would be implemented within the SDN controller, this explicit sharing of keys is unnecessary.

5.2.3 Implementing SPV within SDN Controller

The rationale behind placing SPV outside of SDN controller is to make it applicable to different implementations (e.g., [48, 13, 47]) of SDN controller with minimal effort. However, placing SPV within the SDN controller may help to further improve the performance of SPV. For example, retrieving control messages or network topology would be much faster in the latter case. The downside of such design is the decrease in the security of SPV, since the independence of SPV makes it even more secure as SPV can be a hardened box/software which is easier to secure than SDN controller with a wide range of functionalities, as well as the redundant efforts required to integrate it to different implementations of SDN controller.

Chapter 6

Other Contributions

During this master thesis work, other than SPV, which is the main contribution of this master thesis project, we also contributed to other projects that are described in the following sections.

6.1 Runtime Verification of Cloud-Wide VM-Level Network Isolation

Multi-tenancy is one of the unique provided features of today's cloud systems which may lead to some security concerns such as network isolation of each cloud tenant's virtual resources. Furthermore, due to some built-in challenges in the cloud environment, such as the sheer size of virtual networks and VM-level and distributed network access control, verifying the network isolation becomes a very difficult task specially when it comes to pair-wise VM-reachability verification. However, TenantGuard [56] propose a scalable run-time solution for cloud-wide, VM-to-VM isolation verification by leveraging the hierarchy of virtual networks, efficient data structures, incremental design and parallel computing for better performance. TenantGuard is implemented

and tested on OpenStack and its performance both in-house and on Amazon EC2 [5], confirms its scalability and efficiency (13 seconds for verifying 168 millions of VM pairs).

Our contribution to this work is to set up Amazon EC2 nodes for parallel computations by leveraging the Apache Ignite [7] version 1.4.0, which is a memory-centric multi-model distributed database, caching, and processing platform. We set up two to 16 Amazon EC2 virtual machines and configured Apache Ignite on each machine to run the TenantGuard auditing tasks in parallel, which takes up to 13 seconds for verifying the reachability of 168 millions of VM pairs. Alongside the setup for parallel processing units, we designed and implemented simulation scripts for conducting different experiments, such as computing the performance of TenantGuard by varying the number of VMs/Subnets, Rules/Routers, and Hops by making use of Linux Bash. Moreover, through experiments, we run and compared the performance of TenanatGuard with NOD [32] network verification’s implementation.

6.2 Auditing Virtual Networks Isolation Across Cloud Layers

Some of the consequence of multiplexing virtual resources in a multi-tenant cloud system is cross-tenants data leakage and denial of service issues. Moreover, due to other factors such as the gap between the existing security properties inside the cloud and the high-level description of defined standards, the multi-layer design of the cloud stack, and the dependencies between those layers makes auditing virtual network isolation a very challenging task. To overcome the above-mentioned issues, ISOTOP is designed to cover cross-layer consistent isolation verification in a multi-tenant cloud

environment by mainly focusing on verification of multiple security properties at virtual network layer 2 and overlay networks, namely topology isolation. The mentioned security properties are developed based on the literature and common knowledge on layer 2 virtual networks isolation in order to relate them to defined security standards and filling the gap between the actual existing properties in a cloud system and those standards. Furthermore, a cross-layer network model with their inter-dependencies and isolation mechanisms are devised to study semantics among those layers as well as identifying and auditing the network isolation related data. More specifically, ISOTOP's main job is to evaluate the consistency of layer 2 virtual network isolation by leveraging the gathered cross-layer information between management and infrastructure layers.

Our contributions to this work are listed in the following.

1. Implementing cloud environment by utilizing OpenStack [43] Mitaka version. In this environment, a controller and a network node with different compute nodes are implemented to run the ISOTOPS experiments. The connecting technology between different nodes is switched from LinuxBridge to OpenvSwitch to leverage the OpenFlow protocol.
2. Implementing automated attack scenarios for Intra-compute and Inter-compute nodes which abuse the VLAN and VXLAN connections within and among compute nodes to maliciously connect VMs of different tenants in virtual network layer-2, which is against the network isolation security standards. These attacks show the need to verify the compliance of network isolation properties in different layers of a cloud system.
3. Implementing automated data collection from different layers of the implemented cloud system for auditing.

4. Implementing 11 properties for isolation properties at the infrastructure management level, isolation properties at the implementation level, and topology consistency properties, using CPS solver, Sugar [54], and different sat-solvers, such as Minisat, Riss, Pingeling, Lingeling, and Treengling.
5. Implementing automated verification of the above-mentioned properties along with conducting extensive experiments by varying different factors such as VMs to measure the performance of our automated tool by making use of Python and Linux Bash.
6. Finally, we conducted further experiments by leveraging Amazon EC2 machines to run ISOTOP in a parallel mode for one of the properties.

Note that ISOTOP is extension of [33] and is currently under revision for ACM Transactions on Privacy and Security (TOPS) journal.

6.3 Proactive Security Auditing for Clouds

The scalability and particularly the on-demand requirements from users and cloud service providers had made the security compliance verification a challenging task due to the sheer size and dynamic nature of cloud systems. The verification of security properties with respect to security standards can take up to minutes or sometimes hours. However, to offer proactive security auditing, PVSC [34] and its successor LeaPS [35], which the latter one utilizes learning-based approaches in comparison to the PVSC, have overcome this issue by studying the dependencies between cloud-wide events and starting the verification process before any violation occurs. More specifically, LeaPS achieves a practical response time of 6ms to audit a cloud of 100,000 VMs which is a 50% improvement over PVSC in which the response time is also in milliseconds.

Our contributions to this work are listed in the following.

1. Implementing cloud environment by utilizing OpenStack [43] Mitaka version. In this environment, a controller and a network node with different compute nodes are implemented to collect input for PVSC and LeaPS. The connecting technology between different nodes is switched from LinuxBridge to OpenvSwitch to leverage the OpenFlow protocol.
2. Configuring and installing the Ceilometer service for the cloud environment to capture the management events.
3. Implementing automated scripts to simulate the occurrences of different cloud-wide events as the input for the verification engine.
4. Identifying the relationships and the dependencies between different cloud-wide events.

6.4 Preserving Both Privacy and Utility in Prefix Preserving Anonymization of Network Traces

Auditing humongous amount of data in recent large-scale data-centers is very costly which leads the cloud service providers to outsource their data to a third-party analyst. However, there are security concerns in outsourcing the data in plain-text. This causes the data-center and cloud service providers to anonymize the data before outsourcing them which leads them to question about the utility and privacy of those anonymization techniques. To preserve both privacy and utility, K-Pseudo Anonymity and N-View Defence propose a technique through allowing the third-party analyst to generate different views of the anonymized original data, which is mainly network traces, that are designed to be indistinguishable from the original data even

though there might be adversaries with a pre-knowledge of the original data. So, the trade-off between privacy and utility is shifted to the trade-off between privacy and computational cost. The result of the analysis is one view, which preserves the utility, among those generated N-views, which preserves the privacy.

Our contribution to this work is to write simulation scripts to evaluate the performance and the privacy of the K-Pseudo Anonymity and N-View Defence by varying different factors such as the number of partitions, views, IP addresses in input data sets and etc. Note that this work is under revision for the Privacy Enhancing Technologies Symposium (PETS) conference.

6.5 A Quantitative Approach to Security Compliance Auditing for SDN-Based Cloud

One of the security concerns in today's cloud systems is the co-residency of different tenants' virtual infrastructures, including VMs and virtual networks. However, none of the existing solutions handle this issue by quantitatively measuring the relative distance of cloud resources. More specifically, Quantic proposes a novel approach to measure the physical and virtual distances between cloud resources in order to quantify the level of tenants' virtual infrastructure co-residency. Moreover, instead of providing a binary result for auditing security compliance, a degree of breaches are provided based on the measured distances.

Note that the Quantic is under revision for IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2018.

Our contributions to this work are listed in the following.

1. Simulating the three-layer fat-tree network topology.
2. Implementing cloud environment by utilizing OpenStack [43] Mitaka version

and OpenvSwitch for leveraging the OpenFlow protocol, and generating the synthetic data based on the implemented OpenStack cloud to provide large-scale input data for evaluating the Quantic.

3. Implementing policy definitions and verification of Quantic.
4. Designing and implementing different sets of simulation scripts for extensive evaluation of Quantic.

Chapter 7

Conclusion

The correctness of SDN controller view on network topology is known to be critical for making the right management decisions. However, recently discovered vulnerabilities in OFDP protocol show that poisoning network view of the SDN controller may lead to severe security attacks, such as man-in-the-middle or denial of service. In this thesis, we proposed SPV, a novel stealthy probing-based approach, to significantly extend the scope of existing solutions, by generating and sending stealthy packets to incrementally verify legitimate links and detect fake links as well as the responsible malicious switches. As a proof of concept of our approach, we implemented SPV in an emulated SDN environment using Mininet and OpenDaylight. Through extensive experiments, we showed that SPV achieved a constant time to verify each incremented link to the network, i.e., less than 120 milliseconds, which makes SPV a scalable solution for large SDN networks. We also measured the performance of SPV in a real SDN/cloud hosted at one of the largest telecommunication vendors to validate the applicability of SPV in a real environment. To further improve the accuracy and performance of SPV, considering traversal time of stealthy packets in the link verification procedure and integrating SPV within the SDN controller (for faster processing of control messages or being independent of public/private key sharing) can

be considered as potential future work. Also, to enhance the security of SPV against certain attacks such as flow table exhaustion, adapting methods such as [30] could be beneficial. Moreover, the stealthiness property of stealthy packets can be further improved by leveraging machine learning techniques in analyzing network traffic to enhance the packet generation mechanism of SPV.

Bibliography

- [1] OpenDaylight. Retrieved December 11, 2017 from <https://www.opendaylight.org/>.
- [2] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov. Security in software defined networks: A survey. *IEEE Communications Surveys & Tutorials*, 17(4):2317–2346, 2015.
- [3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.
- [4] T. Alharbi, M. Portmann, and F. Pakzad. The (in)security of topology discovery in software defined networks. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 502–505. IEEE, 2015.
- [5] Amazon EC2. Secure and resizable compute capacity in the cloud. Retrieved December 11, 2017 from <https://aws.amazon.com/ec2/>.
- [6] M. Antikainen, T. Aura, and M. Särelä. Spook in your network: Attacking an sdn with a compromised openflow switch. In *Nordic Conference on Secure IT Systems*, pages 229–244. Springer, 2014.
- [7] Apache Ignite. Database and caching platform. Retrieved December 11, 2017 from <https://ignite.apache.org/>.
- [8] A. Azzouni, R. Boutaba, N. T. M. Trang, and G. Pujolle. sOFTDP: Secure and efficient topology discovery protocol for SDN. *arXiv preprint arXiv:1705.04527*, 2017.
- [9] A. Azzouni, N. T. M. Trang, R. Boutaba, and G. Pujolle. Limitations of openflow topology discovery protocol. *arXiv preprint arXiv:1705.00706*, 2017.
- [10] R. Bifulco, H. Cui, G. O. Karame, and F. Klaedtke. Fingerprinting software-defined networks. In *Network Protocols (ICNP), 2015 IEEE 23rd International Conference on*, pages 453–459. IEEE, 2015.

- [11] T. Bui. Analysis of topology poisoning attacks in software-defined networking. Master's thesis, Aalto University, School of Science, University in Espoo, Finland, 2015.
- [12] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei. How to detect a compromised sdn switch. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–6. IEEE, 2015.
- [13] David Erickson. Beacon. Retrieved December 11, 2017 from <https://openflow.stanford.edu/display/Beacon/Home>.
- [14] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS*, 2015.
- [15] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. Forwarding and control element separation (forces) protocol specification. Technical report, 2010.
- [16] R. Enns, M. Bjorklund, and J. Schoenwaelder. Network configuration protocol (netconf). *Network*, 2011.
- [17] P. Goransson, C. Black, and T. Culver. *Software defined networks: a comprehensive approach*. Morgan Kaufmann, 2016.
- [18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.
- [19] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS*, 2015.
- [20] iPerf. The ultimate speed test tool for tcp, udp and sctp. Retrieved December 11, 2017 <https://iperf.fr/>.
- [21] Y. Jarraya, T. Madi, and M. Debbabi. A survey and a layered taxonomy of software-defined networking. *IEEE communications surveys & tutorials*, 16(4):1955–1980, 2014.
- [22] Java. Retrieved December 11, 2017 from <https://java.com>.
- [23] Y. F. Jou, F. Gong, C. Sargor, X. Wu, S. F. Wu, H.-C. Chang, and F. Wang. Design and implementation of a scalable intrusion detection system for the protection of network infrastructure. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 69–83. IEEE, 2000.

- [24] K. Kaur, J. Singh, and N. S. Ghumman. Mininet as software defined networking testing platform. In *International Conference on Communication, Computing & Systems (ICCCS)*, pages 139–42, 2014.
- [25] S. Khan, A. Gani, A. W. A. Wahab, M. Guizani, and M. K. Khan. Topology discovery in software defined networks: Threats, taxonomy, and state-of-the-art. *IEEE Communications Surveys & Tutorials*, 19(1):303–324, 2017.
- [26] H. Kim and H. Ju. Efficient method for inferring a firewall policy. In *Network Operations and Management Symposium (APNOMS), 2011 13th Asia-Pacific*, pages 1–8. IEEE, 2011.
- [27] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.
- [28] B. Lantz, B. Heller, and N. 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*, page 19. ACM, 2010.
- [29] J. Le Roux. Path computation element (pce) communication protocol (pcep). 2009.
- [30] J. Leng, Y. Zhou, J. Zhang, and C. Hu. An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network. *arXiv preprint arXiv:1504.03095*, 2015.
- [31] P.-C. Lin, P.-C. Li, and V. L. Nguyen. Inferring openflow rules by active probing in software-defined networks. In *Advanced Communication Technology (ICACT), 2017 19th International Conference on*, pages 415–420. IEEE, 2017.
- [32] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, pages 499–512, 2015.
- [33] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 195–206. ACM, 2016.
- [34] S. Majumdar, Y. Jarraya, T. Madi, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Proactive verification of security compliance for clouds through pre-computation: Application to openstack. In *European Symposium on Research in Computer Security*, pages 47–66. Springer, 2016.
- [35] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Leaps: Learning-based proactive security auditing for clouds. In *European Symposium on Research in Computer Security*, pages 265–285. Springer, 2017.

- [36] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
- [37] J. Medved, R. Varga, A. Tkacik, and K. Gray. OpenDaylight: Towards a Model-Driven SDN Controller Architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*, pages 1–6. IEEE, 2014.
- [38] Mininet. An instant virtual network on your laptop (or other pc). Retrieved December 11, 2017 from <http://mininet.org/>.
- [39] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Detecting and isolating malicious routers. *IEEE Transactions on Dependable and Secure Computing*, 3(3):230–244, 2006.
- [40] NOX. The nox controller. Retrieved December 11, 2017 from <https://github.com/noxrepo/nox>.
- [41] Open Networking Foundation. Openflow switch specification version 1.3.5. Retrieved December 11, 2017 from <https://www.opennetworking.org/software-defined-standards/specifications/>.
- [42] Open vSwitch. Production quality, multilayer open virtual switch. Retrieved December 11, 2017 from <http://openvswitch.org/>.
- [43] OpenStack. Open source software for creating private and public clouds. Retrieved December 11, 2017 from <https://www.openstack.org/>.
- [44] A. Ornaghi and M. Valleri. Man in the middle attacks. In *Blackhat Conference Europe*, 2003.
- [45] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska. Efficient topology discovery in software defined networks. In *Signal Processing and Communication Systems (ICSPCS), 2014 8th International Conference on*, pages 1–8. IEEE, 2014.
- [46] B. Pfaff and B. Davie. The open vswitch database management protocol. 2013.
- [47] POX. The pox controller. Retrieved December 11, 2017 from <https://github.com/noxrepo/pox>.
- [48] Project Floodlight. Open source software for building software-defined networks. Retrieved December 11, 2017 from <http://www.projectfloodlight.org/floodlight/>.
- [49] Ryu. Component-based software defined networking framework. Retrieved December 11, 2017 from <https://osrg.github.io/ryu/>.

- [50] Scapy. Packet manipulation program. Retrieved December 11, 2017 from <http://www.secdev.org/projects/scapy/>.
- [51] S. Scott-Hayward, S. Natarajan, and S. Sezer. A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, 18(1):623–654, 2016.
- [52] S. Shin and G. Gu. Attacking software-defined networks: A first feasibility study. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 165–166. ACM, 2013.
- [53] H. Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013.
- [54] N. Tamura and M. Banbara. Sugar: A csp to sat translator based on order encoding. *Proceedings of the Second International CSP Solver Competition*, pages 65–69, 2008.
- [55] F. Wang, B. Vetter, and S. F. Wu. Secure routing protocols: Theory and practice. Technical report, Technical report, North Carolina State University, 1997.
- [56] Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation. In *NDSS*, 2017.
- [57] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, 17(1):27–51, 2015.
- [58] W. Xia, P. Zhao, Y. Wen, and H. Xie. A survey on data center networking (dcn): infrastructure and operations. *IEEE Communications Surveys & Tutorials*, 19(1):640–656, 2017.
- [59] M. Zhang, J. Bi, J. Bai, Z. Dong, Y. Li, and Z. Li. Ftguard: A priority-aware strategy against the flow table overflow attack in sdn. In *Proceedings of the SIGCOMM Posters and Demos*, pages 141–143. ACM, 2017.