Stealthy Probing-based Verification (SPV): An Active Approach to Defending Software Defined Networks against Topology Poisoning Attacks

Amir Alimohammadifar[1], Suryadipta Majumdar[1], Taous Madi[1], Yosr Jarraya[2], Makan Pourzandi[2], Lingyu Wang[1], and Mourad Debbabi[1]

[1] CIISE, Concordia University, Montreal, QC, Canada
{ami_alim,su_majum,wang,debbabi}@encs.concordia.ca
[2] Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
{yosr.jarraya,makan.pourzandi}@ericsson.com

Abstract. 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. Existing solutions mostly rely on passive techniques, which only work for known attacking methods. In this paper, we propose a novel stealthy probing-based verification approach, namely, *SPV*, to detect fake links regardless of the attacking methods used to fabricate them. Specifically, SPV incrementally verifies legitimate links and detects fake links by sending stealthy probing packets designed to be indistinguishable from 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 based on a real SDN/cloud topology. The experimental results show that SPV can respond in near real-time (e.g., less than 120 milliseconds) in both real and emulated environments, which makes SPV a scalable solution for large SDN networks.

Keywords: SDN security, topology poisoning, link verification, active probing

1 Introduction

The Software Defined Networks (SDN) paradigm is gaining momentum as a promising solution with various benefits, such as increasing network resource utilization, simplifying network management, and reducing operating cost [17, 21, 35]. As the central idea of SDN, i.e., separating the network's control and data planes, brings most of those benefits, it also unavoidably empowers the SDN controllers and increases the dependence of SDN applications on those controllers [21, 35]. Specifically, as the operating system of an SDN network, the SDN controller is responsible for maintaining a logically centralized view of the network, which serves as the basis for many SDN applications to make important network management decisions, such as routing, load balancing, firewalling, and monitoring [15]. The validity of such a view is thus critical for the proper functionalities of the SDN applications, and SDN in general.

However, as the SDN controller discovers changes to its network view by sending out special Link Layer Discovery Protocol (LLDP) packets, malicious switches or hosts can easily poison such a view by either manipulating legitimate LLDP packets or injecting fake LLDP packets. As evidenced by recently discovered vulnerabilities in the OpenFlow Discovery Protocol (OFDP) [19], such attacks may take various forms depending on the capabilities of attackers [4, 8, 10, 16, 33]. To make our discussions more concrete, we first illustrate such attacks through an example.

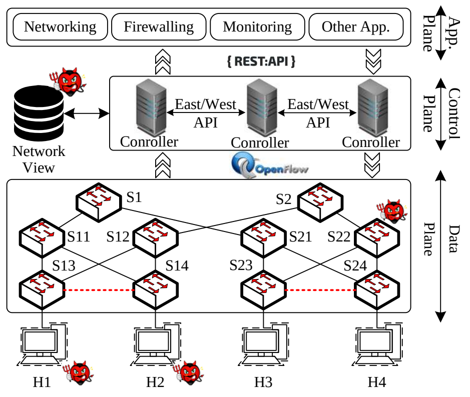


Fig. 1. Topology poisoning attacks in SDN

Motivating Example. Fig. 1 illustrates a simple SDN topology which includes two malicious hosts `H1` and `H2`, and a malicious switch `S22`. We first describe how the SDN controller may discover its network view (i.e., the switches and their links illustrated as solid lines in the figure) and how such a view may be poisoned through attacks.

– **Legitimate topology discovery protocol.** We show how a legitimate run of the topology discovery protocol works to discover a newly added link between switches `S12` and `S13`. First, the controller sends an LLDP packet to all the switches. Each switch is supposed to broadcast this packet to all its ports (except the port connected to the controller). Therefore, switches `S12` and `S13` receive a copy of this packet from each other, upon which they add their identity information to the packets and return the packets to the controller. Based on those packets, the controller concludes that there is a new link between the two switches `S12` and `S13`.

– **Attack 1: Relaying LLDP packets using hosts.** According to the topology discovery protocol, a host is supposed to ignore any LLDP packets broadcasted by the switches. However, upon receiving every such packet from switch `S13`, a malicious host `H1` immediately sends it to an accomplice, host `H2`, through an out-of-band channel. Host `H2` then forwards the packet to switch `S14`, which, unaware of such an ongoing attack, adds its identity information and returns the packet to the SDN controller. Receiving such a packet from `S14` tricks the SDN controller into adding the fake (non-existent) link between `S13` and `S14` into its network view.

– **Attack 2: Injecting fake LLDP packets.** In this attack, upon receiving every LLDP packet from switch `S13`, a malicious host `H1` forges the packet to masquerade itself as switch `S13`, and sends the forged packet (using an out-of-band channel) to switch `S14`. The SDN controller is again tricked into believing there exists a link between `S13` and `S14`.

– **Attack 3: Relaying LLDP packets using switches.** This attack involves a compromised switch [3], i.e., `S22`, to create a fake link between switches `S23` and `S24`. Instead of sending back to the controller, switch `S22` forwards the LLDP packet received from `S23` to `S24`. Switch `S24` returns this packet to the SDN controller, which results in a fake link between `S23` and `S24` to be added.

By poisoning an SDN controller's network view, those attacks may open doors to more severe threats, such as man-in-the-middle and DoS attacks [15, 16, 19]. Most existing works [3, 5, 10, 16] rely on a passive approach that only works for known attacking methods. For example, TopoGuard [16] is a powerful tool that can detect a wide range of attacks on both hosts and links, and it detects the first two attacks above by identifying compromised hosts for unexpectedly sending LLDP packets, and by authenticating the LLDP packets, respectively. However, if the attacker changes his/her methods, e.g., relaying legitimate LLDP packets for the first attack or sending forged LLDP packets for the second from a host, whose connection with a switch is unknown to TopoGuard, then he/she may easily evade the detection. Our key observation is that, such a passive approach can be complemented with an active approach that detects a fake link simply based on its non-existence, regardless of how such a link is fabricated.

In this paper, we propose the first active approach of sending probing packets in a stealthy manner to incrementally verify legitimate links and identify fake links independent of how a fake link is fabricated. Specifically, the probing packets are designed to be stealthy in the sense that they are indistinguishable from normal traffic. We provide detailed methodology and algorithms for both link verification and packet generation. As a proof of concept, we implement and test our approach based on both emulated (using Mininet [27] network emulator) and real SDN environments using the OpenDaylight [1] SDN controller. Finally, we evaluate both the efficiency and effectiveness of our solution against reactive adversaries using both synthetic and real data.

Our main contributions are as follows.

– To the best of our knowledge, SPV is the first active probing-based solution to SDN topology poisoning attacks, and the first comprehensive solution that works regardless of the attacking methods.
– Our implementation using OpenFlow-based SDN environment demonstrates the practicality of our approach. Moreover, we have designed SPV to be separated from the SDN controller implementation, such that it can be easily adapted to other controller implementations.
– As demonstrated through experiments using both synthetic data and the topology of a real SDN/cloud hosted at one of the largest telecommunication vendors, SPV responds in near real-time (e.g., less than 120 milliseconds) for each new link, which confirms its scalability for large SDN networks.

The rest of the paper is organized as follows. Section 2 introduces SDN and our threat model. Section 3 details our methodology. Section 4 discusses security analysis of SPV. Sections 5 and 6 describe the implementation details and experimental results, respectively. Section 7 gives more discussions. Section 8 discusses the related work, and Section 9 concludes the paper.

[3] Note those are software-based switches in SDN/cloud environments which can easily be compromised by malware infections or remote attacks.

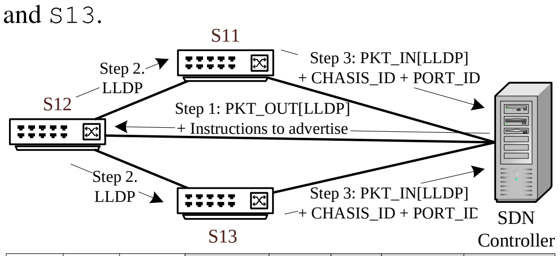
2 Preliminaries

This section provides a brief overview of SDN, and discusses our threat model.

2.1 Background

SDN Overview. As demonstrated in Fig. 1, SDN adopts a three-layered architecture [14, 17]: i) the application layer (the upper layer in the figure), which consists of different applications that manage the data plane, ii) the control plane (i.e., SDN controller), and iii) the data plane, which consists of the forwarding devices and their connecting links. The SDN controller communicates with the application layer and the data plane layer through the northbound and the southbound APIs, respectively. The OpenFlow protocol [13, 25] is the most widely used southbound API for SDN [14, 17].

SDN Topology Management. Most SDN controller implementations follow the OFDP protocol to send LLDP packets for topology management [16]. To facilitate further discussions, the upper figure in Fig. 2 demonstrates the three basic steps of the OFDP protocol as follows. Step (1): the SDN controller encapsulates an LLDP packet (the format of an LLDP packet is shown in the lower figure in Fig. 2) into a message, and sends it to switch `S12`. Step (2): the received message instructs switch `S12` to advertise the enclosed LLDP packet in all its ports except the port it receives the packet from. Step (3): switches `S11` and `S13` send back the LLDP packet in an encapsulated message along with their specifications to the SDN controller. Based on the received packets, the SDN controller discovers the link between switches `S11` and `S12` and the link between switches `S12` and `S13`.



Dst_ MAC	Src_ MAC	Type 0x88cc	CHASSI S_ID	PORT _ID	TTL	Optional TLVs	END of LLDPDU

LLDP Packet Format

Fig. 2. The OFDP protocol and LLDP packet format

2.2 Threat Model

Similar to existing works [3, 5, 6, 8, 16, 19], 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.

On the other hand, we assume the SDN controller is not compromised and the established control channels between the SDN controller and OpenFlow switches are trusted.

Consequently, the attackers can only attempt to distinguish probing packets based on their contents. we also assume 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.

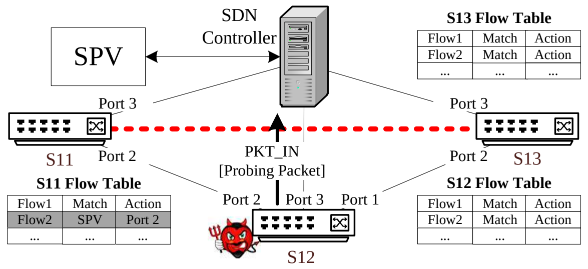
3 Methodology

This section details our methodology including how to verify a newly added link using probing packets and how to generate the probing packets in a stealthy manner.

3.1 Link Verification

Before SPV can verify a newly added link, it must keep track of updates made to the network view. For this purpose, SPV communicates with the SDN controller via the northbound API in order to be notified about any updates to the network view including link creation (a detailed discussion of the SPV architecture and how it interacts with SDN can be found in the Appendix due to page limitations). Once a new link is created, SPV collects relevant information for generating the probing packet, and prepares necessary flows for sending the packet. Finally, SPV sends the probing packet and, depending on the outcome, it marks the link as either legitimate or fake. In the following, we first build intuitions through an example, and then describe the details of our link verification algorithm.

Example 1. Fig. 3 shows a simplified SDN topology with three switches and one SDN controller. We assume switch `S12` is malicious and it has created a fake link between switches `S11` and `S13` (shown by dashed line in Fig. 3) through the aforementioned attack. The following explains how SPV can detect this fake link.



S13 Flow Table

Flow1	Match	Action
Flow2	Match	Action
...	...	...

S11 Flow Table

Flow1	Match	Action
Flow2	SPV	Port 2
...	...	...

S12 Flow Table

Flow1	Match	Action
Flow2	Match	Action
...	...	...

Fig. 3. An example of how SPV verifies a link

- First, since SPV continuously monitors updates to the network topology view, it obtains information about the new link between switches `S11` and `S13` (noted as `S11:Port2-S13:Port2`) as soon as it is created.
- Second, SPV generates a probing packet based on the flow tables of switches `S11` and `S13` (the packet generation will be further elaborated in the next section).
- Third, SPV installs a new flow in the flow table of `S11` (shown as a shaded row in the table) stating that the packets matching this flow must be forwarded to `S11:Port2`, as the link to be verified is `S11:Port2-S13:Port2`.

- Fourth, SPV sends the probing packet to switch `S11`, which forwards the packet on its port `S11:Port2` due to the matching flow. At this point, if there were indeed a legitimate link, the probing packet would be forwarded to switch `S13`. However, as the link is fake and `S11:Port2` is in fact connected to `S12:Port2`, switch `S12` receives the probing packet instead and it returns the packet to the controller. Clearly, no matter how the fake link is fabricated, SPV can always detect it, as long it is non-existent in the sense that not all packets will pass through it (we will discuss the effect of dropping packets in Section 4) and attackers cannot identify our probing packets (we will explain why `S12` cannot identify this probing packet in the next section).
- Finally, once the probing packet is received, SPV discovers that it is sent to the controller from switch `S12` instead of switch `S13`. Therefore, SPV reports link `S11:Port2`-`S13:Port2` to be fake and switch `S12` to be compromised.

The ideas illustrated in Example 1 are more formally described in Algorithm 1 and explained below.

Algorithm 1 Link Verification Algorithm

```
1:  Input: links: Network view of the SDN controller
2:  procedure UPDATECHECK(network view)
3:      while true do
4:          newLink=checkUpdtae(network view)
5:          if newLink == "NOT NULL" then
6:              links = links+newLink
7:              VerificationPreparation(links)
8:  Input: links: List of links to be verified
9:  procedure VERIFICATIONPREPARATION(links)
10:     for each $l \in links$ do
11:         $srcSwStat$ = getStat($l.srcSwID$)          ▷ collecting flows for the source switch
12:         $dstSwStat$ = getStat($l.dstSwID$)          ▷ collecting flows for the destination switch
13:         $ProbingPacket$ =generatePacket($srcSwStat$, $dstSwStat$, $l$)
14:         $SPV\_Flow$ = generateFlow($ProbingPacket.header$,$l.srcSwID$, $l.srcSwPort$)
15:         installFlow($SPV\_Flow$, $l.srcSwID$)       ▷ installing a flow on the source switch
16:         while true do
17:             if sendPacket($ProbingPacket$, $l.srcSwID$, $l.srcSwPort$) == "Successful" then
18:                 $l.ProbingPacket = ProbingPacket$   ▷ storing the latest ProbingPacket
19:                 break
20:         while true do
21:             $receivedPacket = receivePacket()$
22:             if $time \leq threshold$ then
23:                 LinkValidation(receivedPacket, receivedPacket.SwID)
24:                 break
25:             else if $time > threshold$ then
26:                 sendPacket($ProbingPacket$, $l.srcSwID$, $l.srcSwPort$)
27: Input: RET_PKT: the returned probing packet to SPV via controller
28: Input: SwID: the switch ID of the probing packet sender to the controller
29: procedure LINKVALIDATION(RET_PKT, SwID)
30:     for each $l \in links$ do
31:         if $SwID == l.dstSwID$ then
32:             $l.Status$ = "Legitimate"
33:         else if $SwID \neq l.dstSwID$ then
34:             $l.Status$ = "Malicious"
35:             $maliciousSwList.add(SwID)$
```

Lines 2-6: 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 facilitate efficient processing. As soon as an update is reported, SPV stores the link information and initiates the verification preparation.

Lines 11-13: Generating a Probing Packet. Based on information about the newly added link obtained from the previous step, SPV identifies the source and destination switches of the new link. Then, SPV collects flows and other relevant statistics of these two switches (lines 11-12). Afterwards, it generates a probing packet based on these flows and statistics (the packet generation step is further detailed in Section 3.2).

Lines 14-15: Installing Flow to Forward Probing Packet. Utilizing the link information, i.e., link switches and their connecting ports, and the generated probing packet, SPV creates a flow and installs it on the source switch so that this switch can forward the received probing packet on its outgoing port towards the destination switch. Note that the flow is deleted from the switch after a successful round of verification of the given link (omitted from the algorithm).

Lines 16-19: Sending Probing Packets. Once the flow is successfully installed, SPV sends the probing packet to the source switch. SPV keeps sending the packet till the transmission is successful, and then stores the latest probing packet for verification and future packet generation.

Lines 20-26: Receiving Probing Packets. After transmission, SPV waits until it receives the probing packet within a given time threshold. Alternatively, if the packet is dropped for any reason (e.g., an active adversary dropping probing packets purposely), after a certain timeout, SPV re-sends the probing packet to the same switch with randomly chosen time intervals so that an active adversary cannot later predict the probing packets (this defense mechanism of SPV is further discussed in Section 4). Upon a successful return of a probing packet, SPV triggers the link validation procedure to verify the returned packet.

Lines 30-35: Validating a Specific Link. Finally, SPV verifies the validity of a link. To this end, SPV first checks the switch ID of the sender switch of the returned packet. If the ID is the same as that of the destination switch of the link to be verified, then the link is a legitimate link. Otherwise, the link is considered fake and the sender switch of the returned packet is considered compromised. There is another possibility (omitted from the algorithm), where the probing packet is not returned to SPV for certain reason, e.g., dropped due to network congestion or an active adversary. In such a case, SPV marks the link as "Not-Validated" and conducts further rounds of verification with randomly chosen time intervals to prevent an active adversary (discussed more in Section 4).

3.2 Probing Packet Generation

The link verification method introduced in the previous section critically depends on the fact that attackers cannot effectively distinguish the probing packets from normal traffic. This section details the generation of such stealthy probing packets. Again, we first build intuitions through an example and then present the formal packet generation algorithm with the detailed description.

Example 2. Following Example 1, a stealthy probing packet is generated to validate the link between switches `S11` and `S13`. SPV generates this probing packet as follows.

– First, SPV builds a packet pool by collecting the `OFTP_PACKET_IN` messages sent from switches towards the SDN controller, which contain a host generated packet inside them (as shown in Table 1 with PKT_TYPE equal to `HOST`). Each entry of this pool contains different attributes to store useful information about the collected packets (details of such attributes are discussed later in this section).

– Second, upon detecting a new link between switches `S11` and `S13`, SPV prepares a list of candidate packets from the pool in a manner such that none of the packets in the list have traversed through `S11` and `S13`.
– Third, SPV randomly chooses a packet from the list of candidates, forges its header information, and assigns a unique ID to the chosen packet.
– Fourth, SPV calculates a hash value over the unique ID and the timestamp at the moment of packet generation, i.e., `Hash`(`PKT_ID`||`Timestamp`) and uses it as the packet's data field. This generated probing packet is then sent to verify the link between `S11` and `S13` as explained in Example 1. The packet is also added to the pool, shown as the shaded row in Table 1.

#	PKT_ID	PKT_TYPE	Switch_DPID	PKT_IN_hdr	Data_hdr	Data_size	Link_ID	Timestamp
1	-	HOST	S_12	ip:mac	ip:mac	1024	-	-
2	-	HOST	S_13	ip:mac	ip:mac	74	-	-
4	adr341...w34	SPV	-	-	ip:mac	256	Link_1	1522643791
5	-	HOST	S_11	ip:mac	ip:mac	74	-	-
6	dgw213...a78	SPV	-	-	ip:mac	256	Link_3	1522644976

Table 1. An example packet pool

The ideas illustrated in Example 2 are more formally described in Algorithm 2 and explained below.

Lines 4-7: Building the Packet Pool. To build the packet pool, SPV first identifies the attributes shown in Table 1, which are needed for probing packet generation. Second, SPV collects all incoming packets sent to the controller. Third, SPV extracts the header information of each packet and other metadata (e.g., size). Finally, SPV stores the extracted attributes for each packet into the packet pool. Specifically, for each entry in the packet pool, the `PKT_ID` attribute specifies a unique identifier, which is randomly generated for each probing packet in the pool. The `PKT_ID` attribute serves as the secret information used for authentication purposes in SPV. The `PKT_TYPE` attribute can be either "Host" or "SPV", since we collect only these two types of packets. SPV obtains the `Switch_DPID` attribute of the switch that has sent the packet from the MAC address of the packet sender's switch. The `PKT_IN_hdr` and `Data_hdr` attributes contain the headers of the `OFTP_PACKET_IN` and the host generated packets, respectively. The `Link_ID` attribute contains the ID of the link that the probing packet has been sent to verify its validity. Finally, the `Timestamp` attribute shows the timestamp of the probing packet generation.

Lines 12-20: Generating Probing Packets. From the packet pool, SPV first chooses a list of packets which have not traversed through the two switches between which the link to be verified situates to avoid a possible mapping between the probing packet and these reference packets (discussed more in Section 4). If there does not exist any previously stored probing packet for this link (meaning that this is the first round of verification for this link), then SPV randomly chooses a packet from the above-mentioned list, and forges the source and destination IP and/or MAC addresses and ports of the packet header. Thus, SPV prevents a compromised switch from identifying probing packets (discussed further in Section 4). At this point, a unique ID, i.e., `PKT_ID`, is assigned to the packet. SPV also calculates the hash value `Hash`(`PKT_ID`||`Timestamp`) and stores it as part of the probing packet's payload to prevent adversaries from forging or replaying probing packets as will be discussed in Section 4. This concludes the packet generation. However, if this is not the first round of verification for this link, then SPV fetches the related probing packet from the pool. Afterwards, SPV utilizes the line sweep algorithm [20] (in a similar way as in [24]) to produce the next probing packet for verifying this link so that the header information remains less different (e.g., same subnet) than the previous one(s).

Lines 25-28: Sending Probing Packets. This step is to transmit probing packets towards a given switch at the specific port for the purpose of link verification.

Algorithm 2 Probing Packet Generation Algorithm

```
 1: Input: PKT_IN: the exchanged packets between switches and the SDN controller
 2: Input: SwID: the switch ID of the packet sender
 3: procedure PKTCOLLECTOR(PKT_IN, SwID)
 4:     if $PKT\_IN.payload \neq controlMsg$ then
 5:         pktPool.add($PKT\_IN, SwID$)
 6:         if $PKT\_IN.payload == ProbingPacket$ then
 7:             $LinkVerfication.LinkValidation(PKTIN\_SPV, SwID)$
 8: Input: srcSwStat: flows and status of source switch of the link
 9: Input: dstSwStat: flows and status of destination switch of the link
10: Input: link: the link information
11: procedure PKTGEN(srcSwStat, dstSwStat, link)
12:     $pkts = getPktsFromPool(srcSwID, dstSwID)$
13:     if $link.ProbingPacket == null$ then              ▷ first round of verification
14:         $pkt = selectPkt(pkts)$          ▷ select and forge the header of a random packet
15:         $pkt\_ID = generateID(pkt)$
16:         $pkt.payload = hash(pkt\_ID||Timestamp)$
17:     else                                             ▷ further rounds of verification
18:         $l.ProbingPacket = fetchPacket(pool, link)$
19:         $pkt = lineSweepPktGen(l.ProbingPacket)$
20:     return $pkt$
21: Input: PKT: a packet to be sent
22: Input: SwID: the switch ID to send the packet to
23: Input: SwPort: the port number of the switch to send the packet on
24: procedure PACKETSENDER(PKT, SwID, SwPort)
25:     if NorthBoundAPI.SendPacket($PKT, SwID, SwPort$) == "Successful" then
26:         return "Successful"
27:     else
28:         return "Unsuccessful"
```

4 Security Analysis

This section discusses how attackers may attempt to evade SPV and how such attempts are addressed in SPV.

Stealthiness Feature of SPV. The security of SPV relies on the so-called stealthiness feature of a probing packet, meaning the packet cannot be distinguished from other normal host-generated packets traveling through the network. To achieve this, SPV forges its probing packets based on normal host-generated packets (discussed in Section 3.2) in a special way. Specifically, a probing packet is designed to have some fields matching those of normal host-generated packets, whereas other fields must not match. For example, when a probing packet is generated based on a reference packet chosen from the candidate pool (detailed in Section 3.2), the `Eth_Type` field and the packet's payload size would both match those of the reference packet, whereas the `Src_IP`, `Dst_IP`, `Src_MAC`, `Dst_MAC`, `Src_Port` and `Dst_Port` fields must not match those of the reference packet. The former ensures that the probing packet would look exactly like a legitimate packet, and the latter prevents a compromised switch from correlating the probing and reference packets based on common source and destinations.

Relaying or Dropping Packets. A malicious switch may be able to identify probing packets, if SPV always sends them immediately after a new link creation. To address this concern, SPV keeps sending probing packets to the same switch with randomly chosen time intervals so that an active adversary cannot predict the probing packets during its further rounds. Alternatively, a malicious switch may evolve by forwarding traffic including SPV packets passing through it hoping that the fake link remains undetected. However, since the probing packets are indistinguishable, the malicious switch may need to forward every single packet traversing the fake link in order to completely evade detection. However, by forwarding all the traffic, the fake link essentially serves the purpose of a true link, which is against the original objective of the attacker. Hence, a more practical adversary might partially forward the traffic within certain time intervals after the link creation, hoping that the probing packet is among those forwarded. We measure the effect of such attacks on SPV in Section 6. Moreover, to detect such attacks, one potential solution is to employ the timing channel (i.e., comparing time differences between packets traversing on different links), which is considered as a future work. Finally, a malicious switch may choose to only relay the LLDP packets and drop all other packets including SPV packets. However, such DoS attacks can be more easily detected and such detection is beyond the scope of this work.

Compromising Multiple Adjacent Switches. An attacker may compromise multiple switches adjacent to each other, which, however, may not provide any advantage due to the following. i) If the compromised switches relay packets independently, then the chance they all happen to relay a probing packet becomes even lower. ii) Alternatively, if the compromised switches act in coordination, then they essentially become one switch so the chance of relaying a probing packet is the same as with one compromised switch.

Forging or Replaying Probing Packets. A strong adversary may employ malicious switches to mimic the link verification mechanism of SPV by either forging or replaying probing packets, and having them sent back to the SDN controller as if those packets are returned from the other end of the fake link. However, as discussed in Section 3.2, SPV addresses this issue by authenticating each received probing packet using a unique hash value over the secret PKT_ID and timestamp, i.e., $\mathtt{Hash(PKT\_ID||timestamp)}$. Since SPV keeps track of the secret PKT_ID values of all its probing packets, it will not accept any probing packets forged by the adversary. Moreover, the freshness proof (i.e., timestamp) also prevents the adversary from replaying previous probing packets.

Learning Probing Packets. 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 patterns of probing packets by processing the network traffic. For example, if we were to use only the aforementioned hash value as the payload of probing packets, then a compromised switch may process the host-generated packets to measure their payload size in order to distinguish the probing packets. To address this issue, we ensure the payload size of each probing packet is the same as its reference packet by padding dummy data to the aforementioned hash value. Moreover, a strong adversary might keep a history of packets, and leverage learning methods to classify probing packets based on their header information (e.g., source/destination addresses). Therefore, through the line sweeping algorithm, SPV ensures that the header information of probing packets for the same switch remain very close (e.g., in the same subnet).

Injecting Packets by a Malicious Host. A malicious host may inject packets into the network hoping to influence the probing packet generation. However, this attack only works when the following conditions are satisfied: i) the injected packet is used as the reference packet to generate a probing packet. Since SPV chooses reference packets randomly, this can only be achieved with a considerable amount of injected packets, which may be detected by other 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 (since probing packets are never chosen to verify the same switch from where the original packet has been collected), and iii) the probing packet is sent to verify the connecting links of the same compromised switch, which involves a considerable amount of uncertainty.

5 Implementation

This section describes the implementation details of SPV.

Background. We use OpenDaylight (ODL) [1], which is an open platform for automating large-scale networks[26], as the SDN controller, and implement the data plane using Mininet network emulator [22, 27], which is a network emulator that can be used to deploy OpenFlow switches and virtual hosts.

SDN Setup. In this paper, we mainly focus on the SDN controller functionality of ODL by making use of the REST northbound API and OpenFlow southbound API. 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*. 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 configure ODL to instruct data-plane switches that join the network to install reactive flows and to forward the newly received packets to the ODL for further instructions. Moreover, proactive flows for LLDP packets (Ether type 0x88cc) is installed upon joining a switch to the network for discovering the network topology. To set up data plane devices, we utilize Mininet 2.2.1 on a separate Linux virtual machine running Ubuntu server 16.04 with two Intel(R) Xeon(R) E3-1271 v3 CPUs and 4GB of RAM. We utilize OpenFlow version 1.3, since 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 [34] switches.

SPV's Implementation Details. SPV is mostly implemented in Java. We leverage Scapy [31], a packet manipulation tool, for the purposes of probing packet generation and encoding. We implement the SPV in both single-threading and multi-threading modes to enhance the response time of link verification while verifying multiple links simultaneously. We provide further implementation details of SPV in the following.

SPV communicates 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. More specifically, SPV interacts with ODL's *opendaylight-topology* module to keep track of changes in the data plane network. SPV stores the topology in a tree data structure which consists of links, nodes (e.g., switches and hosts) and their connections along with other useful information, e.g., switch statistics or links' status. SPV verification results are also stored in this tree. To send, sniff, dissect and forge network packets, SPV utilizes Scapy. Also, Scapy is used to generate probing packets, encode them in the Base64 format and send them towards specific links using ODL's northbound API.

6 Experiments

This section first discusses the network topology used in our experiments, and then presents different experimental results.

Network Topology. We consider a fat-tree topology [2], which is one of the mostly used network topologies in nowadays large data centers [36], 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 [2, 36]. To further stress the SPV and evaluate its accuracy, we conduct further set of experiments to measure SPV's performance up to for 5,000 link verifications in the network.

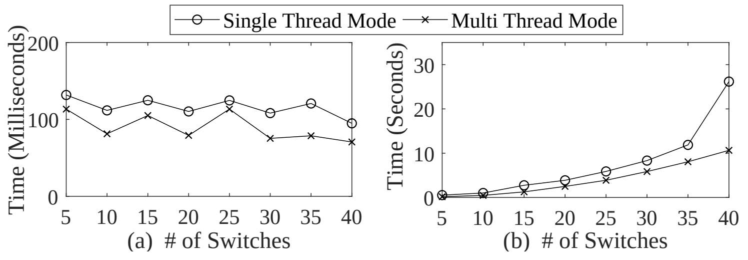


Fig. 4. Time required by SPV to verify (a) a new link and (b) all existing links while varying both the number of switches and number of links in both single and multi-threading modes

The Efficiency of SPV. In the first set of our experiments, we measure the time requirement of SPV. The reported verification time includes the time for performing all steps mentioned in Section 3.1. Fig. 4(a) depicts the time in milliseconds to incrementally verify a newly added link in both single threading and multi-threading 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). Fig. 4(b) shows the time required by SPV to verify a group of links in both single threading and multi-threading 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 threading mode is 26.1 seconds. We further improve the verification time by leveraging the multi-threading mode, which reduces the verification time to 10.6 seconds. Even though the increase in the number of switches in the network results in the increase of verification time for both modes, the increase of the verification time remains almost linear in the multi-threading mode. These results show the practicality of SPV in medium-sized data centers to verify their topology.

Resource Consumption by SPV. The second set of the experiments is to measure the resource consumption (i.e., CPU and memory usage) by SPV. Fig. 5 depicts the average CPU and memory usage to verify all existing links in the network by varying the number of switches for single threading mode. More specifically, Fig. 5(a) shows that SPV on average requires about 20% of the CPU, which is a reasonable amount. Fig. 5(b) depicts the memory consumption by SPV to verify all existing links 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 paper.

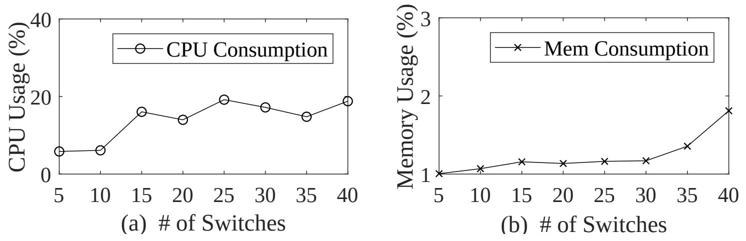


Fig. 5. Average (a) CPU usage and (b) memory usage by SPV to verify all existing links while varying the number of switches up to 40 and the number of links accordingly

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. Fig. 6(a) and 6(b) show the percentage of unverified links in the first round of SPV for 5,000 links in the network while varying the packet loss rate and increasing traffic throughput, respectively. More specifically, Fig. 6(a) depicts that the increase of the packet loss rate, the percentage of the unverified links linearly increases due to the fact that the probing 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 are 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 Fig. 6(b) by increasing the traffic throughput to the maximum allowed bandwidth while keeping the loss rate to 5%, the percentage of unverified links stays almost constant. The reason is that the flow installed by SPV to forward the probing 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. To further alleviate this effect, SPV periodically sends probing packets until a link is verified and hence, all of the network links are eventually verified.

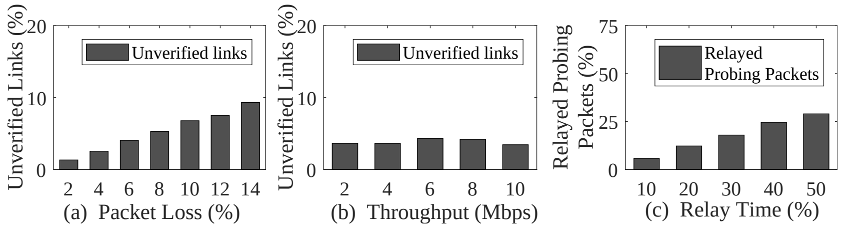


Fig. 6. Evaluating SPV with the presence of attackers in the network that tend to increase packet loss or congest the network by showing percentage of unverified links while varying the (a) packet loss rate (%) and (b) traffic throughput (Mbps), and (c) measuring the percentage of relayed probing packets while varying the percentage of the time that the attacker may relay all the traffic

In this part of the experiments, we measure the percentage of probing packets being affected in the presence of one or multiple malicious switch(es) that forward(s) 10% to 50% of the traffic at different time intervals. The effect of tampering probing packets by an attacker will depict quite similar results, as an attacker cannot distinguish a probing packet from host-generated packets. Fig. 6(c) shows that the amount of forwarded probing packets remains less than 30% for the case a malicious switch forwards all the packets in 50% of the time, however, in this experiments we consider the worst case where the malicious switch is not detected until the last round of verification. The resilience of SPV to this attack can be further improved by using different probabilistic functions with different distributions in choosing the time intervals for sending the probing packets.

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 [28] 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. Table 2 reports the results obtained based on this real topology and compares them with the results for our largest fat-tree topology. The results illustrate that the verification time for an incremental link verification in the network remains almost constant (i.e., around 100ms) independent of network topology and its size, and the verification time for all links in the multi-threading mode can rise up to 13 seconds. Other results presented in Table 2 also indicate a reasonable performance and demonstrate the practicality of our approach in a real-world SDN/cloud.

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 %

Table 2. Summary of the experimental results with real SDN/cloud topology

7 Discussion

This section discusses different concerns on SPV.

Exhausting Flow Table Capacity of SDN Switches. There exist several attacks (as described in [23, 32]) 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 paper, existing solutions (e.g., [37]) to address them could be leveraged to avoid any effect on SPV.

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.

Implementing SPV within SDN Controller. The rationale behind placing SPV outside of SDN controller is to make it applicable to different implementations (e.g., [12, 11, 30]) of SDN controller with minimal effort. However, placing SPV within the SDN controller may help to further improve SPV's performance. For example, retrieving control messages or network topology would be much faster in the latter case. However, such design may decrease the overall security of SPV, since with the current design, SPV can be a hardened box/software which is easier to secure.

8 Related Work

This section discusses different categories of related works.

SDN Topology Poisoning Attack Detection Mechanisms. There exist several works (e.g., [16, 33, 10, 3]) targeting SDN topology poisoning attacks. TopoGuard [16, 33] proposes an OpenFlow-based SDN controller extension, which checks the legitimacy of switch ports and host migration to prevent host-based topology poisoning attacks. In contrast to SPV, TopoGuard can detect a wider range of attacks on both hosts and links. However, it still adopts a passive approach which means the detection relies on knowledge about the attacking methods used to fabricate the fake links. We consider SPV and TopoGuard complementary solutions due to their different advantages. SPHINX [10] proposes a more generic solution on detecting both known and potentially unknown attacks on network topology. In contrast to our work, 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. In [3], 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. Unlike 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.

OFDP Security Enhancement Mechanisms. Several works propose variations of the OFDP protocol for SDN security enhancement. Pakzad et al. [29] propose a modieifed version of OFDP, namely, OFDPv2, which 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 in all its ports, instead of sending an `OFTP_PAKCKET_OUT` message for every port of each switch. sOFTDP [5] also proposes a variation of the OFDP protocol. The main idea is to transfer the burden of topology discovery from the SDN controller to the data-plane switches. Unlike SPV, both OFDPv2[29] and sOFTDP[5] do not verify fake link creation.

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.). For instance, in [32], the authors utilize the delay required for flow installation on SDN switches to detect whether a network is an SDN. INSPIRE [24] 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 [7], 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 [23] 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. Unlike these works, SPV utilizes network probing as a defensive mechanism to deceive malicious hosts and SDN switches. Similarly to our work, in [9], network probing is used as a defensive technique. Therein, a periodic sampling-based approach is proposed to detect malicious OpenFlow switches in an SDN. 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.

9 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 paper, 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 can respond in near real-time (e.g., 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 [23] could be beneficial. Moreover, the robustness of our packet generation mechanism can be further improved by leveraging machine learning techniques in analyzing network traffic.

Acknowledgements. The authors thank the anonymous reviewers for their valuable comments. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under CRD Grant N01823.

References

1. OpenDaylight (2017), `https://www.opendaylight.org/`
2. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: ACM SIGCOMM Computer Communication Review. vol. 38, pp. 63–74 (2008)
3. Alharbi, T., Portmann, M., Pakzad, F.: The (in)security of topology discovery in software defined networks. In: 40th Conference on Local Computer Networks (LCN). pp. 502–505. IEEE (2015)
4. Antikainen, M., Aura, T., Särelä, M.: Spook in your network: Attacking an sdn with a compromised openflow switch. In: Nordic Conference on Secure IT Systems. pp. 229–244. Springer (2014)
5. Azzouni, A., Boutaba, R., Trang, N.T.M., Pujolle, G.: sOFTDP: Secure and efficient topology discovery protocol for SDN. arXiv preprint arXiv:1705.04527 (2017)
6. Azzouni, A., Trang, N.T.M., Boutaba, R., Pujolle, G.: Limitations of openflow topology discovery protocol. arXiv preprint arXiv:1705.00706 (2017)
7. Bifulco, R., Cui, H., Karame, G.O., Klaedtke, F.: Fingerprinting software-defined networks. In: 23rd International Conference on Network Protocols (ICNP) (2015)
8. Bui, T., et al.: Analysis of topology poisoning attacks in software-defined networking (2015)
9. Chi, P.W., Kuo, C.T., Guo, J.W., Lei, C.L.: How to detect a compromised sdn switch. In: 1st IEEE Conference on Network Softwarization (NetSoft) (2015)
10. Dhawan, M., Poddar, R., Mahajan, K., Mann, V.: Sphinx: Detecting security attacks in software-defined networks. In: NDSS (2015)
11. Erickson, D.: Beacon (2013), `https://openflow.stanford.edu/display/Beacon/Home`
12. Floodlight, P.: Open source software for building software-defined networks (2017), `http://www.projectfloodlight.org/floodlight/`
13. Foundation, O.N.: Openflow switch specification version 1.3.5 (2014), `https://www.opennetworking.org/software-defined-standards/specifications/`
14. Goransson, P., Black, C., Culver, T.: Software defined networks: a comprehensive approach. Morgan Kaufmann (2016)
15. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S.: Nox: towards an operating system for networks. ACM SIGCOMM Computer Communication Review **38**(3), 105–110 (2008)
16. Hong, S., Xu, L., Wang, H., Gu, G.: Poisoning network visibility in software-defined networks: New attacks and countermeasures. In: NDSS (2015)
17. Jarraya, Y., Madi, T., Debbabi, M.: A survey and a layered taxonomy of software-defined networking. IEEE communications surveys & tutorials **16**(4), 1955–1980 (2014)
18. Kaur, K., Singh, J., Ghumman, N.S.: Mininet as software defined networking testing platform. In: International Conference on Communication, Computing & Systems (ICCCS). pp. 139–42 (2014)
19. Khan, S., Gani, A., Wahab, A.W.A., Guizani, M., Khan, M.K.: Topology discovery in software defined networks: Threats, taxonomy, and state-of-the-art. IEEE Communications Surveys & Tutorials **19**(1), 303–324 (2017)
20. Kim, H., Ju, H.: Efficient method for inferring a firewall policy. In: 13th Asia-Pacific Network Operations and Management Symposium (APNOMS) (2011)

21. Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. Proceedings of the IEEE **103**(1), 14–76 (2015)
22. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. p. 19. ACM (2010)
23. Leng, J., Zhou, Y., Zhang, J., Hu, C.: 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)
24. Lin, P.C., Li, P.C., Nguyen, V.L.: Inferring openflow rules by active probing in software-defined networks. In: 19th International Conference on Advanced Communication Technology (ICACT) (2017)
25. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review **38**(2), 69–74 (2008)
26. Medved, J., Varga, R., Tkacik, A., Gray, K.: Opendaylight: Towards a Model-Driven SDN Controller Architecture. In: 15th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM) (2014)
27. Mininet: An instant virtual network on your laptop (or other pc) (2017), `http://mininet.org/`
28. OpenStack: Open source software for creating private and public clouds (2017), `https://www.openstack.org/`
29. Pakzad, F., Portmann, M., Tan, W.L., Indulska, J.: Efficient topology discovery in software defined networks. In: Signal Processing and Communication Systems (ICSPCS), 2014 8th International Conference on. pp. 1–8. IEEE (2014)
30. POX: The pox controller (2013), `https://github.com/noxrepo/pox`
31. Scapy: Packet manipulation program. (2017), `http://www.secdev.org/projects/scapy/`
32. Shin, S., Gu, G.: Attacking software-defined networks: A first feasibility study. In: Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. pp. 165–166. ACM (2013)
33. Skowyra, R., Xu, L., Gu, G., Hobson, T., Dedhia, V., Landry, J., Okhravi, H.: Effective topology tampering attacks and defenses in software-defined networks. In: Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18) (June 2018)
34. vSwitch, O.: Production quality, multilayer open virtual switch (2016), `http://openvswitch.org/`
35. Xia, W., Wen, Y., Foh, C.H., Niyato, D., Xie, H.: A survey on software-defined networking. IEEE Communications Surveys & Tutorials **17**(1), 27–51 (2015)
36. Xia, W., Zhao, P., Wen, Y., Xie, H.: A survey on data center networking (dcn): infrastructure and operations. IEEE Communications Surveys & Tutorials **19**(1), 640–656 (2017)
37. Zhang, M., Bi, J., Bai, J., Dong, Z., Li, Y., Li, Z.: Ftguard: A priority-aware strategy against the flow table overflow attack in sdn. In: Proceedings of the SIGCOMM Posters and Demos. pp. 141–143. ACM (2017)

Appendix

The SPV Architecture. Fig. 7 depicts the architecture of SPV including its interactions with SDN. SPV has two major modules: Link Verification and Stealthy Packet Handler. In the following, we describe each module in details.

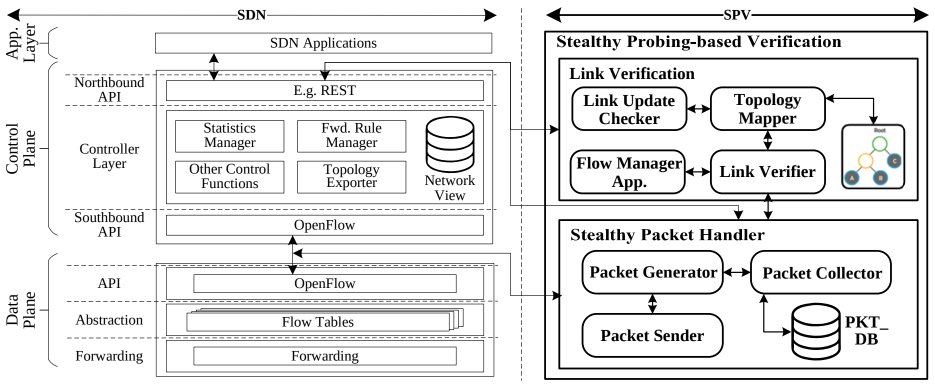


Fig. 7. SDN and stealthy probing-based verification (SPV) architecture

The Link Verification 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 to identify updates in the network through communications with the SDN controller[4] and inform 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 incrementally.
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 up/down, and other useful information such as Device IDs, Port IDs and so on. Also, the tree stores 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.** 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, the Link Verifier communicates with the Topology Mapper to get the link endpoints, and to query the flows and the statistics of them via the Flow Manager Application module. Also, this module relies on the Stealthy Packet Handler module, more specifically Packet Collector and Packet Generator modules, which are 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.

[4] The communication is conducted via northbound API

The Stealthy Packet Handler 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) 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. First, 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. Second, it collects and stores `OFTP_PACKET_IN` messages that contain `SPV_PKT`, and reports them to the Link Verifier.
3. **Packet Sender.** This module is 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 by utilizing the SDN controller's northbound API.