Topology Validator - Defense Against Topology Poisoning Attack in SDN

Abhay Kumar(✉) and Sandeep Shukla

Department of CSE, IIT Kanpur, Kanpur 208016, India
{abhkum,sandeeps}@iitk.ac.in

Abstract. SDN controller in the SDN (Software Defined Network) environment needs to know the topology of the whole network under its control to ensure successful delivery and routing of packets to their respective destinations and paths. SDN Controller uses OFDP to learn the topology, for which it uses a variant of LLDP packets used in the legacy network. The current implementations of OFDP in popular SDN controllers suffer mainly two categories of attacks, namely Topology Poisoning by LLDP packet injection and Topology Poisoning by LLDP packet relay. Several solutions have been proposed to deal with these two categories of attacks. Our study found that, while most of these proposed solutions successfully prevented the LLDP packet injection-based attack, none could defend the relay-based attack with promising accuracy. In this paper, we have proposed a solution, namely Topology Validator, along with its implementation as a module of Floodlight SDN controller, which, apart from preventing LLDP injection-based attack, was also able to detect and thwart the LLDP relay-based attack successfully.

Keywords: Software Defined Network · SDN · SDN security · Topology attack

1 Introduction

SDN is a novel networking paradigm that has gained momentum during the last decade. It decouples the data plane and control plane against the standard networking concept, where the control plane and data plane are tightly coupled and present on each device. SDN makes the network programmable allowing it to be more dynamic, cost-effective and controllable. While it has added many features, being an evolving concept, it is also open to several explored and unexplored vulnerabilities. Many of such attacks have been discovered by multiple researchers [11, 14, 15, 17, 18, 21, 25] distributed across control plane, data plane and applications connected to the SDN eco-system via different APIs. Further countermeasures corresponding to each discovered attack has been proposed by researchers [1, 12, 13, 21] with various degree of success.

In this paper, we have explored an attack related to how topology is learned and maintained in the SDN environment. We further have proposed a countermeasure, namely Topology Validator, against the relay-based ghost link creation attack that not just defends against the attack completely but also requires minimal overhead compared to prior

The original version of this chapter was revised: an error in the name of one of the authors has been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-91424-0_21

X. Yuan et al. (Eds.): QShine 2021, LNICST 402, pp. 241–260, 2021.
https://doi.org/10.1007/978-3-030-91424-0_15

solutions for this type of attack. In this work, we have assumed that only hosts can be compromised, switches and controller remain immune.

This paper further is organized as below. Section 2 is about the background of the topic, where we have briefed about SDN (Software Defined Network) and methods used for learning topology in SDN. Section 3 explores topology poisoning attacks and their experimental verifications. Section 4 discusses some of the previously proposed state-of-art works. Section 5 details our proposed solution, experimental observations and related works. Section 6 discusses the result and analyze different parameters that may be affecting our solution adaptation. Section 7 concludes this paper.

2 Background

SDN is a novel networking paradigm that has gained momentum during the last decade. It decouples the data plane and control plane against the standard networking concept, where the control plane and data plane are tightly coupled and present on each device. SDN makes the network programmable allowing it to be more dynamic, cost-effective and controllable. While it has added many features, being an evolving concept, it is also open to several explored and unexplored vulnerabilities. Many of such attacks have been discovered by multiple researchers [11, 14, 15, 17, 18, 21, 25] distributed across control plane, data plane and applications connected to the SDN eco-system via different APIs. Further countermeasures corresponding to each discovered attack has been proposed by researchers [1, 12, 13, 21] with various degree of success.

In this paper, we have explored an attack related to how topology is learned and maintained in the SDN environment. We further have proposed a countermeasure, namely Topology Validator, to the said attack that not just defends against the attack but also requires minimal overhead compared with prior solutions for this category of attacks. In this work, we have assumed that only hosts can be compromised, switches and controller remained immune.

2.1 Software Defined Network

Software Defined Network is a growing architecture that is dynamic, programmable, cost-effective and versatile, making it perfect for the high data transfer capacity networking. The thought behind SDN is to isolate the control plane and the data plane of the networking devices. The primary distinction between SDN and traditional systems is that in the conventional systems, the data plane and the control plane are integrated inside the networking devices, i.e. each networking device has its controlling functions and switching functions. As packets enter the networking device, the networking device itself decides the further activity. However, in the case of SDN, the control plane is logically centralized. The control plane contains controllers that offer guidance to the data plane about how to act whenever a packet arrives. The data plane is included only in forwarding devices where they simply forward the arrived packet as per the flow rules present in the forwarding table.

2.2 Topology Learning in SDN

SDN networks have their intelligence put inside the SDN Controller, which keeps the network functional by commanding switches to route/forward packets to their respective destinations. To function appropriately, SDN controllers need to learn a lot of information from the network. One such important piece of information is the network's topology. Contrary to a legacy network where each switch requires to learn the topology, SDN requires only the controller to know the complete topology of the network. Though the process of learning the topology under both forms of networking is different, SDN OFDP (OpenFlow Discovery Protocol) also uses LLDP (Link Layer Discovery Protocol) packets to learn the topology. The SDN controller for each connected switch generates number of LLDP packets equal to the number of active ports on that switch and sends these to the switches via the controller-switch links at regular intervals. The information about the number of active ports and other details of the switch are learned by the controller at the time the switch registers with the controller and keeps it updated by its host tracking service.

2.2.1 LLDP Packet Format

Figure 1 describes the standard LLDP packet format, while Fig. 2 is an LLDP packet capture from our experimental setup. Apart from common fields such as Preamble, Destination MAC address, Source MAC address, Ethernet type (which describes the link-layer protocol, here 0x88cc) and Frame checksum, it contains four compulsory TLV (Type-Length-Values)s and variable numbers of optional TLVs. DPID and Port ID TLVs are set with Data-Path-ID and port number of the switch respectively, for which the SDN controller has generated that particular LLDP packet.

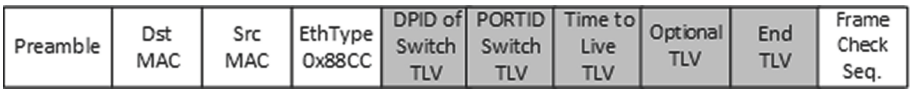

Fig. 1. LLDP packet format

This further generalizes to the fact that all LLDP packets sent to a specific switch will have the same DPID TLV but different PORT ID TLV corresponding to each port. TTL (Time-To-Live) TLV decides the lifespan of the LLDP packet, after which it will be discarded.

In Fig. 2, the captured LLDP packet has four optional TLVs put in for multiple purposes; the last optional TLV, however, will be of our interest which has been discussed in Sect. 4. From the DPID and Port ID TLVs, it can be learned that this LLDP packet was sent to port number 2 of the switch with DPID - 00:00:00:00:00:02.

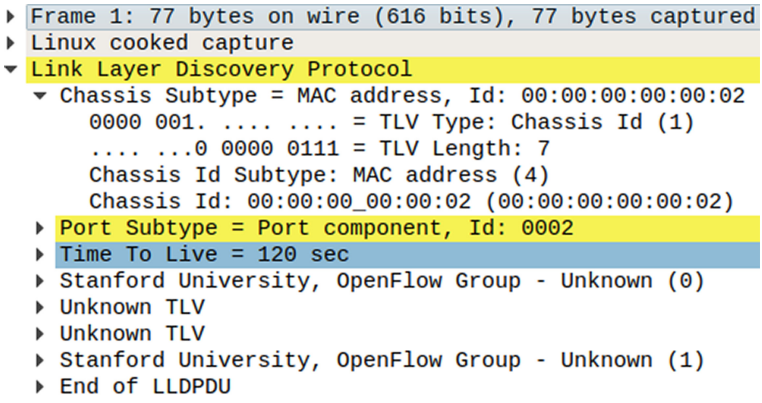

Fig. 2. Wireshark capture of LLDP packet

2.2.2 Link Discovery Service (LDS)

LDS is accomplished by OFDP (OpenFlow Discovery Protocol), which uses LLDP packets to learn the network's topology and keeps it updated as and when new links are added, existing links fail or are removed. LDS in the SDN controller can do so by sending LLDP packets to switches corresponding to each active port on respective switches at regular interval. Intervals after which LLDP packets are sent may be SDN controller dependent. However, in our experimental setup where we had used FloodLight [7, 8] SDN controller, it sends LLDP packets at an interval of 15 s which can be seen in Fig. 3 which is a capture of the FloodLight console output.

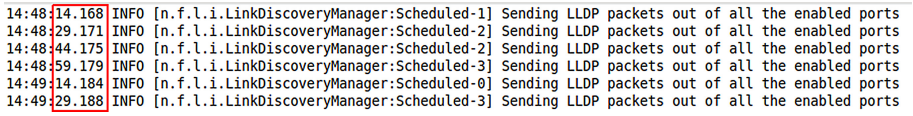

Fig. 3. FloodLight console output LLDP frequency

As illustrated in Fig. 4, let's consider a topology with one controller, two switches and two hosts connected to each switch making the total number of active ports three on each switch. SDN controller will send three (corresponding to each active port on each switch) LLDP packets to each switch embedded inside individual OFPT PACKET OUT messages. Switch extracts out the LLDP packets and forwards them to their respective destination ports. Ports connecting hosts will ignore these LLDP packets. However, ports connecting to other switches will make the LLDP packet marked for that particular port forwarded to other connected switches; in Fig. 4, the LLDP packet sent to switch S1 and port number 3 (S1,3) will reach the switch S2 on port number 1 (S2,1). On arrival at S2, these packets are embedded inside an OFPT PACKET IN message and forwarded to the controller. We found some authors [30] attributed this action of switch S2 to some pre-set rule inside the switch Flow Table or some firmware. However, our observation differed from that.

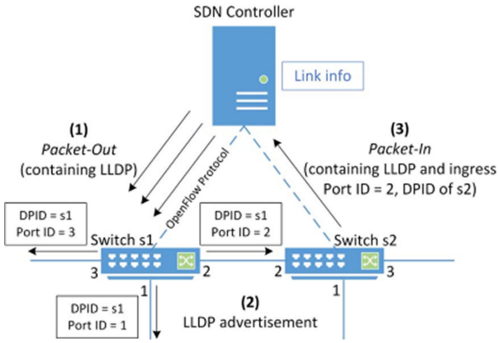

Fig. 4. Working of link discovery service

We could not see any specific flow rules for handling LLDP packet in particular, ref. Fig. 5, instead they got forwarded to the controller following the primary/default rule (which exists in the flow table of all switches all the time, Fig. 5), which says when a switch does not find a match for some arrived packet; it forwards it to the controller after embedding it inside an OFPT PACKET IN message.

```
mininet> dpctl dump-flows
*** s2 ------------------------------------------------
-----------------------
 cookie=0x0, duration=81258.874s, table=0, n_packets=61
47, n_bytes=481648, priority=0 actions=CONTROLLER:65535
*** s1 ------------------------------------------------
-----------------------
 cookie=0x0, duration=81258.889s, table=0, n_packets=61
44, n_bytes=481391, priority=0 actions=CONTROLLER:65535
```

Fig. 5. Flow tables of S1 and S2 with LLDP traffic only

Once this forwarded packet arrives at the controller, the controller can learn that the LLDP packet it has sent to (S1,3) is being received at (S2,1) confirms a unidirectional link between S1 and S2 via port number 3 and 1, respectively. Similarly, the LLDP packet sent by the controller to Switch S2 on port number 1 will be received via (S1,3), confirming another unidirectional link between (S2,1) and (S1,3). As both unidirectional links will be processed in one round, this will result in the discovery of the bi-directional link between S1 and S2. Accordingly, the controller will update the overall topology of the network.

3 Topology Poisoning Attacks in SDN

Poisoning the system's topology isn't altogether new in SDN. A compromised host or a compromised switch can start poisoning the topology. Our paper mainly focuses on the attack started by a compromised host.

3.1 Topology Poisoning by Fake LLDP Packet Injection

Assumption 1: To keep it simple, let us assume that all LLDP packets sent by a controller all the time have all Optional TLVs same (This assumption is not very much leveraged, as we have found an old version of FloodLight, the controller under our study doing the same). We further assume that any one host on any of switches, say h1 (connected on port number 1) on S1, is under an attacker's control, using which attacker can inject some packet to the switch.

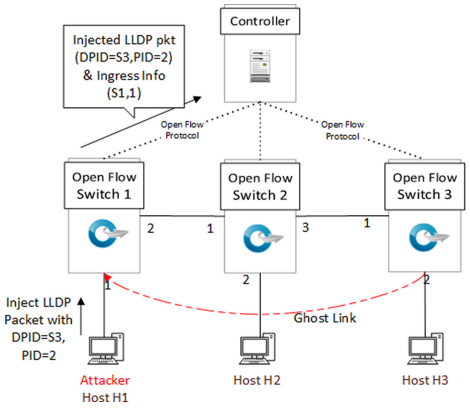

Fig. 6. Ghost link creation by fake LLDP packet

On receiving a genuine LLDP packet, this compromised host injects an LLDP packet with DPID switch S3 and port number 2. When this injected LLDP packet reaches switch S1, unaware of any malicious activity in the network, switch S1 will treat this packet as if it is coming from some switch connected on port 1 of switch S1 and will forward it to the controller after embedding it inside PACKET IN message with source information as (S1,1). This will lead the controller to assume that there exists a link between (S3,2) & (S1,1) and update the overall topology of the network accordingly. Post this topology update, when a host on S3 will try to communicate with a host on S1, the controller will route it through just discovered non-existent link resulting in communication failure.

Assumption 2: Restricting Assumption 1 to cases from all optional TLVs same in all round for all switches, to at least one Optional TLV unique in each round but same for all switches, will not make any difference on the ease of attack and attacker will be able to create a fake link without any new cost.

Assumption 3: On further restricting our Assumption 2 to have at least one unique Optional TLV corresponding to each switch will force the attacker to change its strategy. It will be no longer the same easy task using this approach to create a fake link as it was earlier. For creating a fake link between S3 and S1 under this updated assumption with the compromised host on (S1,1), the attacker on (S1,1) will need to learn the unique Optional TLV sent to S3, and then only it can inject an LLDP packet with learned unique

optional TLV to succeed in creating the fake link. The attacker may learn it by going through the controller's source code if it is open-source or listening to the traffic and making a guess about it (unless it is some random value each time). It is evident that the latest added restriction will rule out the possibility of an attack of nature discussed in this subsection to a larger extent; however, this will not rule out the kind of attack we have discussed in Subsect. 3.2.

We were able to execute the attack under the first assumption by deploying a C executable on host h1. Executable actively listened for the LLDP packet multicast and injects it to S1 after making changes as illustrated in the Fig. 6. With this setup, we were not just able to create the fake link for once, but maintain it in a controlled manner also. The same setup as able to execute the attack under second assumption as well.

3.2 Topology Poisoning by LLDP Packet Relay

The first assumption for the attack under this category remains the same as what we had after adding the second restriction (all rounds of topology discovery will have at least one optional TLV unique to each switch in every round) to our basic assumption (only hosts can be compromised, switches and controllers remain immune). We further assume that we have at least one host compromised on each switch between which attacker wishes to create the fake link. In our example topology illustrated in Fig. 7, we assume hosts on (S1,1) and (S3,2) are under attacker control.

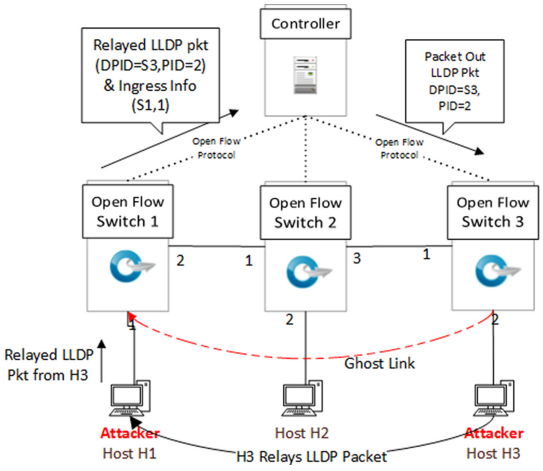

Fig. 7. Ghost link creation by LLDP packet relay

For the execution of the attack, on receiving LLDP packet from the controller, compromised host on (S3,2) uses already existing connectivity between S1 and S3 to relay received LLDP packet to (S1,1). A host connected to (S1,1) on receiving this relayed packet, injects it to the switch S1, which will get forwarded to the controller (as this will not match any of existing flow-table rules), making the controller assume that there

exists a unidirectional link from (S3,1) to (S1,1). This unidirectional link can easily be converted to bidirectional by making the host on (S1,1) to relay the received LLDP packet to host at (S3,2) and host on (S3,2) later injecting it to S3.

We must note here that, restricting our first assumption even to the extent that the LLDP packet corresponding to each port has some unique Optional TLV, this type of attack cannot be prevented.

We were able to execute this attack even with the latest version of FloodLight. SDN controller under our experimental setup. We installed a C program executable on both compromised hosts; each of these C programs had two threads. One thread was used for listening for LLDP packets, while the other one for creating a UDP server. First thread on each compromised host, when receives an LLDP packet relays it to UDP server running on another compromised host. The second thread used for running the UDP server on receipt of relayed LLDP packet was injecting it to the network via the switch they were connected to. In this case, as well we were just not able to create the fake link but were able to maintain it as well.

4 Existing Solutions and Analysis

In this section, we give a concise review of applicable state-of-the-art endeavours that are accomplished for the relief of topology poisoning attacks in SDN. Mostly, these methodologies depend on the consideration of hash or static functions.

1. Adding Unique TLV (Type Length Value) to each LLDP packet: This solution we found in the SDN controller Floodlight V2.3 implemented. This method can defend the attack cases where an adversary fabricates the LLDP packet to be injected. It fails to counter relay-based attacks.
2. S. Hong, L. Xu, H. Wang, G. Gu [33] proposes a solution to the link fabrication attack, which distinguishes the reason for LLDP packet-based attacks as a failure in identifying the origin and ensuring the integrity of the LLDP packet. It additionally states that no host should inject LLDP packet to the switch, i.e. it stops injection of LLDP packet from the port to which a host is connected. Moreover, to prevent the Link fabrication attack, the authors added one HMAC to the LLDP packet through one extra TLV (Type Length Value).

   Defense proposed by Hong et al. [33] relies heavily on the categorization of the port as switch-port or host-port. However, there is no promising way to find the same has been described in their work. Any assumption like the amount of traffic flowing through a switch-port is higher when compared with host-port, to do the port categorization may not work in situations where intra-switch traffic on a switch is higher than inter-switch traffic and will result in a high number of false-positive and negative cases. Labelling of hosts based on pre-condition and post-condition validation too will not be an effective solution when multiple virtual hosts are connected to the same physical port in bridge mode.
3. Skowyra et al. [12] discovered and demonstrated two topology attacks called port amnesia and port probing in Topoguard proposed by Hong et al. They later proposed and implemented Topoguard+, a modified version of Topoguard after fixing the

cause for attacks they discovered. A module named Link Latency Inspector (LLI) was used to differentiate between genuine and fake switch links. Eduard Marin et al. [1], in their work, discovered two attacks involving TopoGuard and TopoGuard+. In the first, they were able to remove a genuine link, while in the second, a fake link was established, taking advantage of the way LLI works. They proposed that overloading a genuine switch link by overloading two involved switches will end in LLI computing a latency which can cause a genuine link removed. Shrivastava et al. were able to successfully create fake link even in the presence of Topoguard+ by increasing the overall latency of the network such that it becomes comparable to that of the out-of-band link that the attacker is using to relay the packets to create fake links.

4. SPHINX [31] utilizes a form of flow graphs, which is created by PACKET-IN and FEATURES-REPLY messages. Flow graphs are utilized to approve all the updates done in networks and the given requirements. To prevent Link fabrication attack, SPHINX utilizes static switchport binding. Hence it doesn't support the SDN Dynamic evolution.
5. SPV proposed by Alimohammadifar et al. [7] attempted to detect link fabrication attack by sending probing packets indistinguishable from standard packets. While most of the work assumed the switch be trusted, this work claimed to work even with few switches compromised.

   Authors in their work itself have acknowledged that SPV will not work when the compromised relay host is forwarding all packets using out-of-band channels. Their solution worked with an assumption that the attacker is using low bandwidth out-of-band channel for relaying packet, which we consider a very impractical consideration as the attacker may use one of the already existing links to relay the packets.
6. OFDP [8], unlike the previous methods here, the HMAC TLV (Type Length Value), which is being added, provides both authentication and integrity. The HMAC is nothing but a hash function that is appended in every LLDP packets. Here the key which is generated for LLDP packets is dynamic, i.e. a key value is calculated for every LLDP packets. Hence it prevents LLDP Replay attacks.

Although It is better than previous TLV based methods, it utilizes more resources as it calculates HMAC for every LLDP packets; further, it adds no defense against relay-based attacks.

5 Proposed Solution and Implementation

In this section, we provide the details of our proposed solution, which we found not just prevents topology learning based attack more robustly but also with minimum overhead compared to existing solutions.

Before we go into further details, we define here the two terminologies link and switch-port (or port). Link represents the presence of physical connectivity on a particular switch-port to an end-user device/host or some other switch. Port is an administratively controllable entity utilizing the software. It may seem that the status of a link and the status of a port is tightly coupled in both directions, i.e. according to the status of a port

as UP or DOWN, the corresponding link will be UP or DOWN or vice-versa; in real they are not. Table 1 describes the relation between port status and link status. When the link status is down, the port status will be down implicitly. However, if the link is UP, a switch port can be kept administratively UP or DOWN. The administrative controllability of a port plays a vital role in our proposed solution.

Table 1. Port and link status relation

Port status	Link status	Effective port status	Effective link status
UP	UP	UP	UP
UP	DOWN	DOWN	DOWN
DOWN	UP	DOWN	DOWN
DOWN	DOWN	DOWN	DOWN

5.1 Motivation

The real challenge in mitigating LLDP based topology poisoning attacks is identifying if the end device connected to a particular switch-port is an end-user device/host or some other switch. Had there been an accurate way of distinguishing between a switch and end-user device/host, LLDP based topology poisoning attack could have been thwarted easily by simply dropping any LLDP packet coming from any device but switches, as switches are the only device which are supposed to forward the LLDP packets. As discussed in Sect. 4, some efforts were made in this direction using Machine Learning techniques, but they were not accurate enough. Without handling false positive and false negative cases, it may end up collapsing the whole network topology. Other heuristics-based approaches such as traffic on a switch-port to switch-port link should, in general, be higher compared to traffic on switch-port and host link, will not have promising accuracy either as there can be genuine cases where intra-switch communication has more traffic than inter-switch communication. Distinguishing between a switch and a host based on MAC Address will equally be a bad idea, as it is effortless for an attacker to tamper the MAC address of a compromised host to present itself as some switch. Hence our prime focus is to develop a way to decide if the device connected on a switch-port is another switch or some end-user device/host with no false-positive or false-negative cases.

Figure 8(a) describes the cases when all links are good, Fig. 8(b) when a link between one switch and a host goes down, and Fig. 8(c) when a link between two switches goes down. Link *down* activity in Fig. 8(b) will cause the Host 1 and connecting port on switch S1 to go down as well. While there will be no reporting by the Host H1, the involved switch, i.e. S1, will report this new status change to the controller. Similarly, the Link *down*, as shown in Fig. 8(c), will make both the involved switch-ports at respective switches status change to DOWN owing to link failure, and the same will be reported by both the switched to the controller.

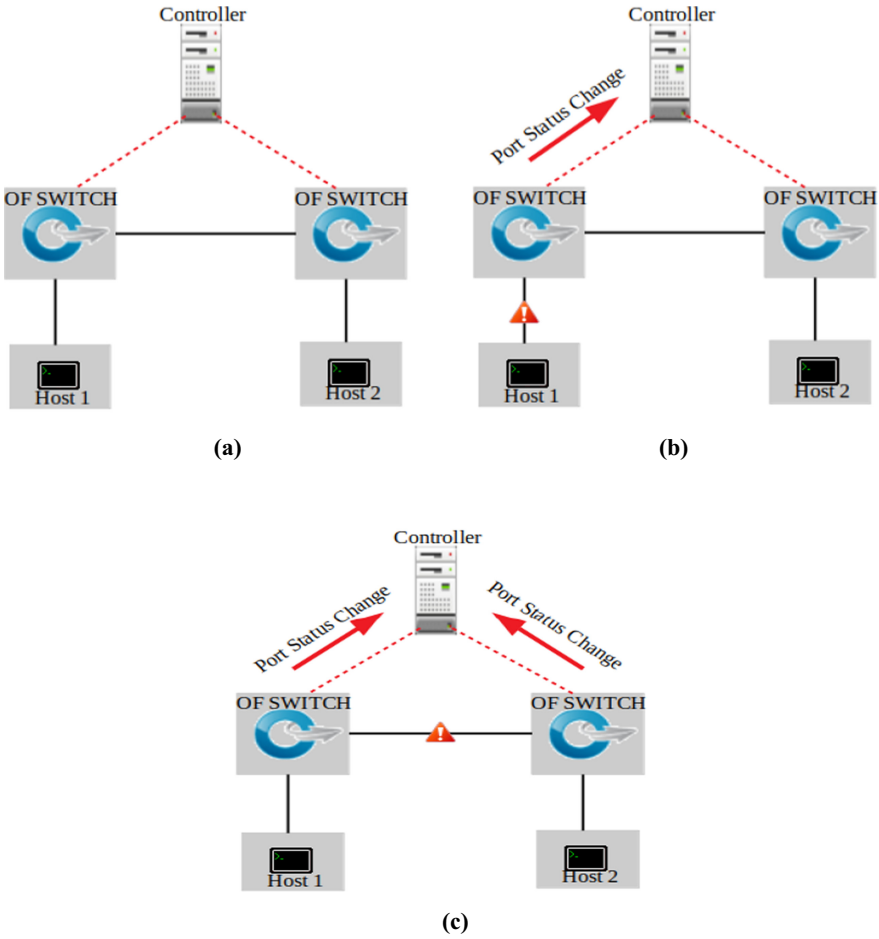

Fig. 8. **(a)** Topology under normal scenario. **(b)** Switch to host link failure. **(c)** Switch to switch link failure

We can summarize the above observation as Switch Port status change are reported by switches and recorded by the controller, while port status change at hosts are not reported. This observation can further be used to identify if the other device connected on a switch port is a host or some other switch by the controller.

5.2 Experimental Observations

Experimental topology used for recording the observation was as illustrated in Fig. 10.

1. When a link between an OF-Switch and End-Host is brought DOWN: As part of this experiment, we brought down the link between OF-Switch S1 and Host h1 in our experimental topology. This link down action triggered the switch to send two OFPT PORT STATUS messages corresponding to the switch-port on which host h1

was connected, the first one with OFPC PORT DOWN and OFPPS LINK DOWN of config and state descriptors respectively set to 0, while second OFPT PORT STATUS message has both flags set to 1. The first OFPT PORT STATUS message represents the state of the switch-port before the link down event, while the second represents the current state of the switch-port, i.e. after the link down event was recorded.
2. When a link between an OF-Switch and End-Host is brought UP: As part of this experiment, we restored the link we had brought down in step 1 above. This resulted in the switch sending another OFPT PORT STATUS message for the involved switch-port where it has config descriptor in which other element values including OFPC PORT DOWN reset to 0, while state descriptor has all its elements value reset to 0 except OFPPS LIVE set to 1.
3. When a link between two OF-Switch is brought DOWN: As part of this experiment, we brought down the link between OF-Switch-1 and OF-Switch-2 established using ports (S1,3) and (S2,1), respectively. This action triggered both the involved switches to send OFPT PORT STATUS message corresponding to each involved port on respective switches with values for flags OFPC PORT DOWN and OFPPS LINK DOWN of config and state descriptors respectively set to 1. Before sending the updated values post-event, each switch first sends OFPT PORT STATUS messages with flags of OFPC PORT DOWN and OFPPS LINK DOWN of config and state descriptors respectively set to 0.
4. When a link between two OF-Switch is brought UP: As part of this experiment, we restored the link that we brought down in step 3 above. This triggered both the involved switches to send OFPT PORT STATUS messages to the controller for their corresponding ports with flags OFPC PORT DOWN and OFPPS LINK DOWN values set to 0 and OFPPS LIVE set to 1.

With experimental verification, we can conclude that if a link that connects a switch-port to a host goes down, the controller receives one OFPT PORT STATUS message with flags OFPC PORT DOWN and OFPPS LINK DOWN set to 1. While a link that connects one switch-port to another switch-port goes down, the controller receives two OFPT PORT STATUS message with flags OFPC PORT DOWN and OFPPS LINK DOWN set to 1, one OFPT PORT STATUS message corresponding to each involved switch. Similarly, when a link is restored for each involved switch-port, the controller receives an OFPT PORT STATUS message with OFPPS LIVE set to 1.

5.3 Algorithm

Algorithm 1 describes the steps involved in validating the topology.

Lemma 1. Algorithm 1, if it detects a new link, always ensure that it is not a ghost link.

Proof: Suppose the algorithm considers a ghost link as the genuine one. To validate its consideration, it will make any of the involved ports on the link connecting switches administratively down. As per the algorithm and assumption, it should result in only one PORT DOWN message. However, in actual it will result in two PORT DOWN messages, making our initial consideration incorrect.

Algorithm 1: Topology Validator Algorithm

```
while a new link discovered do
        Get the details of switches and ports involved;
        Controller sends ofp port config to any of participating switch with OFPPC
        PORT DOWN set to 1;
        if ofp port state with OFPP PORT DOWN set to 1 is generated by both
        involved ports then
          Discovered Link is a genuine one;
        Else
          Discovered Link is an attack attempt;
        end
end
```

Lemma 2. Algorithm 1 ensures a genuine link is never declared a ghost link.

Proof: We assume that the algorithm considers a newly discovered genuine link a ghost link. For validation as per the stated algorithm, any of the involved ports on link connecting switches will be made down administratively. As this has been considered a ghost link, it should result in only one PORT DOWN message. However, it will result in two PORT DOWN message making our assumption incorrect.

5.4 Implementation

Figure 9 describes the port presentation of ports in the SDN environment as per the latest implemented specification of OpenFlow protocol V1.3.0. From the list of descriptors in Fig. 9, config and state are the ones we are interested in.

The config descriptor as defined above records the administrative setting of the port. OFPPC PORT DOWN field tells about the administrative status of the port, which takes a binary value, where 0 represents the port being administratively DOWN while 1 represents the port as administratively UP. Further specification demands this field to be set/reset by the controller, and the switch should not change it. However, if some external interface or activity causes a change of this field value, the switch must convey this change to the controller using the OFPT PORT STATUS message.

The state descriptor as defined above records the physical link state of the port. OFPP LINK DOWN value is set to 1 when a physical link is present and reset to 0 when there is no physical link present. This descriptor's value cannot be changed by the controller. The switch sends an OFPT PORT STATUS message to update the controller if there is any change in the physical state of the port. Our proposed solution exploits the above two discussed descriptors for thwarting topology poisoning attack, described in Sect. 1 (Table 2).

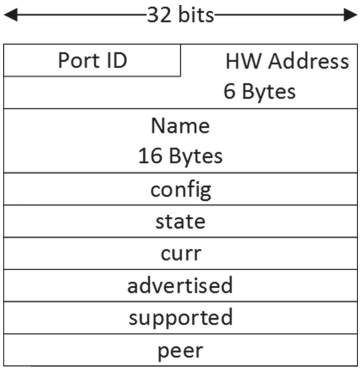

Fig. 9. Switch port representation in SDN

Table 2. Config and state descriptor

Config Descriptor	State Descriptor
enum ofp_port_config { OFPPC_PORT_DOWN = 1 << 0 , OFPPC_NO_RECV = 1 << 2 , OFPPC_NO_FWD = 1 << 5 , OFPPC_NO_PACKET_IN = 1 << 6 } ;	enum ofp_port_state { OFPPS_LINK_DOWN = 1 << 0 , OFPPS_BLOCKED = 1 << 1 , OFPPS_LIVE = 1 << 2 , } ;

Case 1: The Normal Scenario: We assume that the initial topology is depicted in Fig. 10, and a new link between (S1,3) and (S2,1) is being made live by connecting a physical wire. This activity will trigger two OFPT PORT STATUS messages from each of switches s1 and s2 for the ports (S1,3) and (S2,1). Following algorithm 1, a controller will make any of the randomly chosen switch port, say (S2,1) administratively DOWN by sending a port down message. As per Table 1, this will cause the link between (S1,3) and (s2,1) to down, and it should follow events as in experiment no 4, and the controller will receive one port status change message for each involved port on both switches. This will confirm to the controller that the newly discovered link is a genuine one and will make the port (S2,1) from administratively DOWN to UP which will re-enable the link between (S1,3) and (S2,1). As the newly discovered link has been verified between two switches, it can now be incorporated in the topology, and topology routing may be updated accordingly.

We achieved this scenario under our experimental setup by first disabling the link between switches S1 & S2 and enabling them again. Enabling action of the link between two switches can be considered equivalent to connecting the two switched with a physical wire.

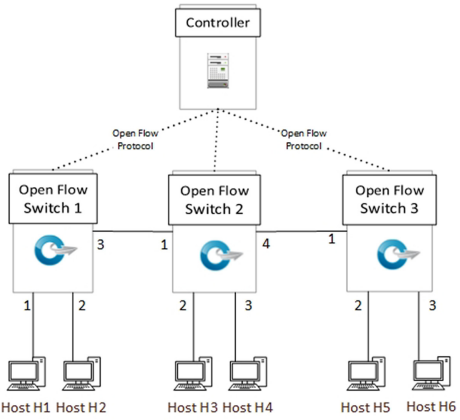

Fig. 10. Experimental topology

Case 2: The Attack scenario: We assume that in one round of LLDP discovery message sent and received. On analysis, it discovered a link between (S1,2) and (S3,2), and this link is an attack attempt by compromised hosts at (S1,2) and (S3,2) where host at (S1,2) relaying the copy of LLDP packet received by it to host (S3,2) by using the existing link (out-of-band) that connects switches S1 and S3 as depicted in Fig. 10. On discovery of the new link, going by the algorithm, the controller will turn any of the ports (S1,2) and (S3,2) chosen at random, say (S3,2) administratively DOWN by sending port down message to the switch S3. On receipt of this message going by the Table 1 link between host h3 and (S3,2) will go down too, which will make switch S3 send one status change message to the controller corresponding to (S3,2). However, turning DOWN will have no effect on the link between (S1,2) and host h2; hence no activity will be recorded by the controller against switch S1. Had this newly discovered link been a genuine one, failure of the link would have triggered status messages from switch S1. As the controller did not receive two status messages from both allegedly involved switches, it will ignore the newly discovered link, and no update will be made to the existing topology of the network. Moreover, the port connecting to the link may administratively be turned down depending upon the network's policy to avoid further poisoning attack by these machines.

This case under our experimental setup was achieved in the same fashion it was used for verifying relay-based topology poisoning attack as mentioned in Sect. 3.2. We implemented our proposed solution as a separate module of floodlight. We did not change the normal functioning of the topology manager; instead, we confirmed if we can detect the attack attempt. Table 3 captures floodlight console output, where activities corresponding to our implementation can be seen.

Table 3. Floodlight console output

01.240 INFO [n.f.l.i.LinkDiscoveryManager] Sending LLDP packets out of all the enabled ports
13.605 INFO [n.f.l.i.LinkDiscoveryManager] Interswitch link removed: Link [src=00:00:00:00:00:00:00:02 outPort=1, dst=00:00:00:00:00:00:00:01, inPort=1, latency=2]
13.605 INFO [n.f.l.i.LinkDiscoveryManager] Interswitch link removed: Link [src=00:00:00:00:00:00:00:01 outPort=1, dst=00:00:00:00:00:00:00:02, inPort=1, latency=2]
13.925 INFO [n.f.t.TopologyManager] Recomputing topology due to: link-discovery-updates
16.243 INFO [n.f.l.i.LinkDiscoveryManager] Sending LLDP packets out of all the enabled ports

31.434 INFO [n.f.t.TopologyManager] Recomputing topology due to: link-discovery-updates
31.503 INFO [TopologyValidator] Discovered a new link [src=00:00:00:00:00:00:00:01 outPort=1, dst=00:00:00:00:00:00:00:02, inPort=1, latency=2]
31.620 INFO [TopologyValidator] About to issue Administrative Down to [src=00:00:00:00:00:00:00:01 outPort=1]
31.987 INFO [TopologyValidator] Port down recorded from [src=00:00:00:00:00:00:00:01 outPort=1]
32.105 INFO [TopologyValidator] Waiting for port down from [src=00:00:00:00:00:00:00:02 outPort=1]
32.995 INFO [TopologyValidator] Port Down recorded from [src=00:00:00:00:00:00:00:02 outPort=1]
33.204 INFO [TopologyValidator] Validation Success
33.379 INFO [TopologyValidator] About to issue Administrative UP to [src=00:00:00:00:00:00:00:01 outPort=1]
33.834 INFO [TopologyValidator] Port UP event recorded for [src=00:00:00:00:00:00:00:01 outPort=1]
34.367 INFO [TopologyValidator] Port UP event recorded for [src=00:00:00:00:00:00:00:02 outPort=2]
01.254 INFO [n.f.l.i.LinkDiscoveryManager] Sending LLDP packets out of all the enabled ports
01.447 INFO [n.f.t.TopologyManager] Recomputing topology due to: link-discovery-updates
16.257 INFO [n.f.l.i.LinkDiscoveryManager] Sending LLDP packets out of all the enabled ports

6 Result and Analysis

Our solution was able to mitigate the LLDP relay-based topology poisoning attack with minimal overhead. However, we evaluated it further on the parameters of time taken to update the topology and its effect on the availability of the network. As the update is not being made to the topology immediately after discovering an LLDP packet sent/receive round, in the non-attack scenario, it will take more time than the typical scenario.

Effect on Availability: As the link discovered will be new, it will in no way affect the availability of existing topology, as all the administrative port DOWN/UP activity will be restricted to the port involved in forming a new link. Hence our solution will not have any impact on the existing topology without incorporating a new link.

Delay in the Incorporation of New Link: New link validity check will consume some time. We measured the time lag between the intercept of the new link and final validation by the java (programming language used in Floodlight) logging feature. Figure 11 depicts the time consumed over ten experimental attempts. Mean time delay was found to be 3301.9 ms. The average time elapsed can be considered acceptable, considering the fact that the LLDP round runs every 15 s.

Comparison with Existing Solutions: A parametric comparison is not possible with existing works, as different authors have proposed solutions using different approaches. It is the end result, i.e. if the proposed solution could prevent topology poisoning attack with no false positive and false negative cases, is the only measuring criteria. We have discussed it in Sect. 4 to conclude that all of the already proposed state-of-art works fail to prevent *topology poisoning by relay*-based attacks on some of all cases. However, we have found our solution preventing topology learning based poisoning attacks with no false-positive or false-negative cases.

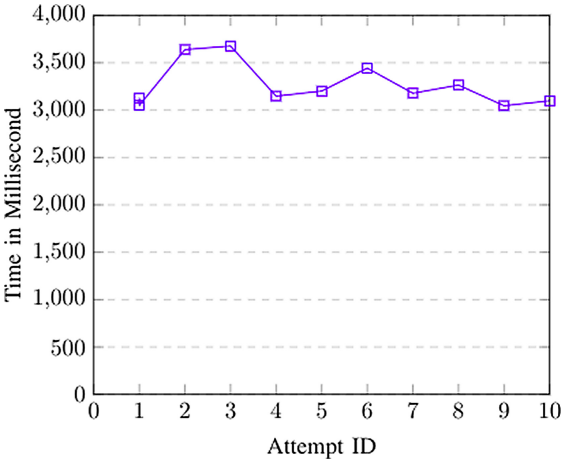

Fig. 11. Validation time consumption plot

7 Conclusion

This paper presented a novel defense method against relay-based poisoning attack in the the SDN environment. The described solution was able to defend the relay-based topology poisoning attack with no false positive or false negative cases. We also were able to keep the overhead minimal as it does not require any additional headers added to the standard LLDP packets, all that it takes is an average of 3301 ms delay in incorporating new topology changes.

Acknowledgements. This research was partially funded by the c3i center (Interdisciplinary Center for Cyber Security and Cyber Defense of Critical Infrastructures, IIT Kanpur) funding SERB, Government of India.

References

1. Popic, S., Vuleta, M., Cvjetkovic, P., Todorović, B.M.: Secure topology detection in software-defined networking with network configuration protocol and link layer discovery protocol. In: 2020 International Symposium on Industrial Electronics and Applications (INDEL), pp. 1–5 (2020). https://doi.org/10.1109/INDEL50386.2020.9266137
2. Chou, L.-D., et al.: Behavior anomaly detection in SDN control plane: a case study of topology discovery attacks. Wirel. Commun. Mobile Comput. (2020). http://dxp.doi.org/10.1155/2020/8898949
3. Huang, X., Shi, P., Liu, Y., Xu, F.: Towards trusted and efficient SDN topology discovery: a lightweight topology verification scheme. Comput. Netw. **170**, 107119 (2020). ISSN 1389-1286
4. Marin, E., Conti, M.: An in-depth look into SDN topology discovery mechanisms: novel attacks and practical countermeasures. In: CCS 2019, London, United Kingdom (2019)
5. Cao, J., et al.: The crosspath attack: disrupting the SDN control channel via shared links. In: USENIX Security Symposium, pp. 19–36 (2019)
6. Abdou, A., van Oorschot, P.C., Wan, T.: Comparative analysis of control plane security of SDN and conventional networks. IEEE Commun. Surv. Tutor. **20**, 3542–3559 (2018)
7. Alimohammadifar, A., et al.: Stealthy probing-based verification (SPV): an active approach to defending software defined networks against topology poisoning attacks. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11099, pp. 463–484. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98989-1_23
8. Nehra, A., Tripathi, M., Singh Gaur, M., Babu Battula, R., Lal, C.: TILAK: a token based prevention approach for topology discovery threats in SDN. Int. J. Commun. Syst. **32**, e3781 (2018)
9. Big Switch Networks. Floodlight (2020). http://www.projectfloodlight.org/floodlight/
10. Big Switch Networks. Floodlight Git repository (2020). https://github.com/floodlight/floodlight
11. Shrivastava, P., Agarwal, A., Kataoka, K.: Detection of topology poisoning by silent relay attacker in SDN. In: Annual International Conference on Mobile Computing and Networking (MobiCom), pp. 792–794 (2018)
12. Skowyra, R., et al.: Effective topology tampering attacks and defenses in software-defined networks. In: International Conference on Dependable Systems and Networks (DSN), pp. 374–385 (2018)

13. Mininet Team. Mininet (2020). http://mininet.org
14. Ujcich, B.E., et al.: Cross-app poisoning in software-defined networking. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 648–663 (2018)
15. Wang, H., Yang, G., Chinprutthiwong, P., Xu, L., Zhang, Y., Gu, G.: Towards fine-grained network security forensics and diagnosis in the SDN era. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 3–16 (2018)
16. Xue, L., Ma, X., Luo, X., Chan, E.W.W., Miu, T.T.N., Gu, G.: LinkScope: towards detecting target link flooding attacks. IEEE Trans. Inf. Forensics Secur. (TIFS) **13**, 2423–2438 (2018)
17. Zhang, M., Li, G., Xu, L., Bi, J., Gu, G., Bai., J.: Control plane reflection attacks in SDNs: new attacks and countermeasures. In: Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2018)
18. Lee, S., Yoon, C., Lee, C., Shin, S., Yegneswaran, V., Porras, P.A.: DELTA: a security assessment framework for software-defined networks. In: Network and Distributed System Security Symposium (NDSS) (2017)
19. Lin, P.P., Li, P., Nguyen, V.L.: Inferring OpenFlow rules by active probing in software-defined networks. In: International Conference on Advanced Communication Technology (ICACT), pp. 415–420 (2017)
20. Thimmaraju, K., Schiff, L., Schmid, S.: Outsmarting network security with SDN teleportation. In: IEEE European Symposium on Security and Privacy (EuroS\&P), pp. 563–578 (2017)
21. Xu, L., Huang, J., Hong, S., Zhang, J., Gu, G.: Attacking the brain: races in the SDN control plane. In: USENIX Security Symposium, pp. 451–468 (2017)
22. Zhang, P.: Towards rule enforcement verification for software defined networks. In: IEEE Conference on Computer Communications (INFOCOM), pp. 1–9 (2017)
23. Jero, S., Koch, W., Skowyra, R., Okhravi, H., Nita-Rotaru, C., Bigelow, D.: Identifier binding attacks and defenses in software-defined networks. In: USENIX Security Symposium, pp. 415–432 (2017)
24. Jero, S., Bu, X., NitaRotaru, C., Okhravi, H., Skowyra, R., Fahmy, S.: BEADS: automated attack discovery in OpenFlow-based SDN systems. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) RAID 2017. LNCS, vol. 10453, pp. 311–333. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66332-6_14
25. Chen, H., Benson, T.: The case for making tight control plane latency guarantees in SDN switches. In: Symposium on SDN Research (SOSR), pp. 150–156 (2017)
26. Ambrosin, M., Conti, M., De Gaspari, F., Poovendran, R.: Lineswitch: tackling control plane saturation attacks in software-defined networking. IEEE/ACM Trans. Netw. (TON) **25**(2), 1206–1219 (2017)
27. Katta, N., Alipourfard, O., Rexford, J., Walker, D.: CacheFlow: dependency aware rule-caching for software-defined networks. In: Symposium on SDN Research (SOSR), pp. 6:1–6:12 (2016)
28. Sonchack, J., Dubey, A., Aviv, A.J., Smith, J.M., Keller, E.: Timing-based reconnaissance and defense in software-defined networks. In: Annual Conference on Computer Security Applications (ACSAC), pp. 89–100 (2016)
29. Xu, H., Yu, Z., Yang Li, X., Qian, C., Huang, L., Jung, T.: Real-time update with joint optimization of route selection and update scheduling for SDNs. In: International Conference on Network Protocols (ICNP), pp. 1–10 (2016)
30. Alharbi, T., Portmann, M., Pakzad, F.: The (in)security of topology discovery in software defined networks. In: Local Computer Networks (LCN), pp. 502–505 (2015)
31. Dhawan, M., Poddar, R., Mahajan, K., Mann, V.: SPHINX: detecting security attacks in software-defined networks. In: Network and Distributed System Security Symposium (NDSS), pp. 8–11 (2015)
32. He, K., et al.: Measuring control plane latency in SDN-enabled switches. In: Symposium on SDN Research (SOSR), pp. 25:1–25:6 (2015)

33. Hong, S., Xu, L., Wang, H., Gu, G.: Poisoning network visibility in software defined networks: new attacks and countermeasures. In: Network and Distributed System Security Symposium (NDSS), pp. 8–11 (2015)
34. Kreutz, D., Ramos, F., Verissimo, P., Esteve Rothenberg, C., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. ArXive-prints (2014). https://doi.org/10.1109/JPROC.2014.2371999