Link Latency Attack in Software-Defined Networks

Sanaz Soltani*, Mohammad Shojafar*, Habib Mostafaei†, Zahra Pooranian* and Rahim Tafazolli*
* 5GIC & 6GIC, University of Surrey, Guildford, UK
{s.soltani, m.shojafar, z.pooranian, r.tafazolli}@surrey.ac.uk
† Technische Universität Berlin, Berlin, Germany
habib@inet.tu-berlin.de

***Abstract*—Software-Defined Networking (SDN) has found applications in different domains, including wired- and wireless networks. The SDN controller has a global view of the network topology, which is vulnerable to topology poisoning attacks, e.g., link fabrication and host-location hijacking. The adversaries can leverage these attacks to monitor the flows or drop them. However, current defence systems such as *TopoGuard* and *TopoGuard+* can detect such attacks. In this paper, we introduce the Link Latency Attack (LLA) that can successfully bypass the systems' defence mechanisms above. In LLA, the adversary can add a fake link into the network and corrupt the controller's view from the network topology. This can be accomplished by compromising the end hosts without the need to attack the SDN-enabled switches. We develop a Machine Learning-based Link Guard (MLLG) system to provide the required defence for LLA. We test the performance of our system using an emulated network on Mininet, and the obtained results show an accuracy of 98.22% in detecting the attack. Interestingly, MLLG improves 16% the accuracy of *TopoGuard+*.**

***Index Terms*—Software-defined Networking (SDN), Topology Poisoning, Link Fabrication Attack, Link Latency, Machine Learning.**

I. Introduction

Software-Defined Networking (SDN) facilitates the management of the network devices, e.g., switches, through an interface and utilises a logically centralised entity to control the entire network. SDN brings flexibility to program the forwarding behaviour of the network devices [1] and reduces the configuration complexities of the network. The OpenFlow [2] is one of the widely employed realisations of SDN in commercial networking devices.

The centralised entity of an SDN-based network is a target for many security attacks, such as topology poisoning attack [3]. This attack targets corrupting the view of the SDN controller on the connected devices, e.g., switches or hosts, to the network. Link Fabrication Attack (LFA) [4] is an example of a topology poisoning attack in which the adversary intends to add a fake link between two switches. The adversary uses the security vulnerabilities of the OpenFlow Discovery Protocol (OFDP) [5] to attack the network since the SDN controller leverages this protocol to obtain the topology information.

In OFDP, the controller sends periodic Link Layer Discovery Protocol (LLDP) [6] messages to the switches. The controller issues the LLDP packets and send them to all switches. By receiving LLDP packets via a `Packet-Out` message, each switch distributes it to all interfaces. When the destination switch receives the LLDP, the switch encapsulates it as a `Packet-In` message and sends it to the controller. Upon receiving LLDP, the controller realises a link between two switches.

The above procedure lacks authenticity. The lack of authentication mechanism in OFDP makes the network vulnerable to the topology poisoning attack since an adversary can insert a fake LLDP message to disturb the global view of the controller [7]. Furthermore, with the adoption of SDN, a large number of enterprises will benefit from its advantages, and according to the report in [8], the SDN market value will grow to 32+ billion USD by 2025. Therefore, the security of the SDN networks becomes crucial for many businesses since the attack can impact many of them. Consequently, the LFA and other types of topology poisoning attacks should be seriously analysed because the cost of an attack can be very high. According to the report in [9], the average cost of a cyber attack is 3.86 million USD per incident.

A. Motivations

Several defences have been proposed in the literature to mitigate LFA risks [4], [10], [11]. For example, *TopoGuard* [4] and *SPHINX* [11] monitor the packets of flows targeted to the controller to detect topology tampering attacks. *TopoGuard+* [10] shows the defence systems of *TopoGuard* and *SPHINX* can be bypassed by introducing *Port Amnesia* and *Port Probing* attack. The adversary uses *Port Amnesia* attack to reset the port type of the device used by *TopoGuard* to detect the link advertisements and relay the LLDP toward the controller. Port Amnesia in *TopoGuard+* can bypass the port-labelling technique of *TopoGuard* by disconnecting and reconnecting the host. Following this way, the label of a port in a switch resets to ANY, i.e., one label type in *TopoGuard* [10]. The defence mechanism of *TopoGuard+* can detect and prevent this attack. Nevertheless, *TopoGuard+* is vulnerable to other topology attacks. Motivated by this, we introduce a Link Latency Attack (LLA) that can bypass the defence system of *TopoGuard+*. The adversary can use this attack to add a fake link into the network and corrupt the controller's view from the network topology. We analysed the passive and active monitoring techniques used in *TopoGuard* and *TopoGuard+* to detect the LFA and report that such systems' defence systems cannot prevent this attack.

B. Contributions

This paper first introduces the LLA. Then, it develops and implements a machine learning-based (ML) system on top of *TopoGuard+* to detect and prevent the LFA. We employ ML-based classifiers to train our system using a dataset to detect the LFA. Our main motive is to take advantage of ML in implementing outlier detection techniques to identify a dynamic threshold for attack classification. We test our system in a network topology built using Mininet [12] and Floodlight [13] controller. The obtained results show that our system can report the attack with an accuracy of 98.22%.

The rest of the paper is as follows. In Section II, we present LLA and the weakness of *TopoGuard+* in attack detection. Our proposed MLLG interaction is described in Section III and evaluated in Section IV. A discussion on LLA and MLLG limitations are described in Sections V. We present the related work and conclusion in Sections VI and VII, respectively.

II. LLA: Link Latency Attack

This section introduces a new attack to corrupt the network topology view of the SDN controller. First, we define the threat model used in the paper (see Section II-A). Then, we explain two phases, namely overload phases and relay phase, in sections II-B and II-C, respectively.

A. Threat Model

In this paper, we assume that an adversary can compromise one or more hosts or virtual machines in the SDN through viruses, trojans and malware infections or even in a worse case, the adversary could be an insider. Moreover, the adversary provides an out-of-band communication channel, e.g., cable or wireless connection, between two hosts to relay the LLDP packets. If it is not feasible, a multi-homed single host could be used alternatively [14].

We call this attack Link Latency Attack (LLA). The adversary aims to add a fake link between switches through the out-of-band channel using malicious hosts in the network. To do so, the adversary leverages the end-hosts to inject unwanted traffic, e.g., ARP, to the network to increase the packet processing time of the switches. Consequently, the switches' response time to the controller packets increases since modern proposed defences by the SDN controllers such as *TopoGuard+* rely on probe packets to keep the updated view of the network topology. The adversary uses this long response time to relay LLDP packets to the network to add the corresponding fake link among the switches.

The LLA impacts the performance of the network by misleading the traffic that can have several consequences such as poor Quality of Service (QoS) or Quality of Experience (QoE), to state a few [15], [16].

We now explain how *TopoGuard+* detects LFA. *TopoGuard+* includes a Link Latency Inspector (LLI) module to track latency values of links between switches. The LLI could detect fake links by checking the latency of links imposed by an out-of-band channel when propagating LLDP packets. It periodically issues probe packets to measure the round trip time (RTT) between the controller and switches. Upon receiving the packet, each switch provides a suitable response to that packet. Assume that we have two switches, namely, s_1 and s_2, that are connected to a controller. Also, suppose that T_{p_1} and T_{p_2} are the corresponding link latency of the probe packets sent to s_1 and s_2. The LLI computes the inter-switch link latency T_l using eq. (1).

$$T_l = T_{LLDP} - T_{p_1} - T_{p_2}, \tag{1}$$

where T_{LLDP} indicates the propagation delay of the LLDP packet. To calculate the T_{LLDP}, the controller adds a timestamp to the issued LLDP packet toward the switches and takes the difference when receiving it. Moreover, LLI stores the values of inter-switch latency T_l for previous LLDPs, and measures a latency threshold T_h as shown in eq. (2).

$$T_h = q_3 + 3 * (q_3 - q_1), \tag{2}$$

where q_1 and q_3 indicate the lower and upper quartiles of latencies, respectively. LLI verifies the link's validity by comparing latency T_l and threshold T_h and raises a security alarm in case of suspicious delay, i.e., $T_l > T_h$.

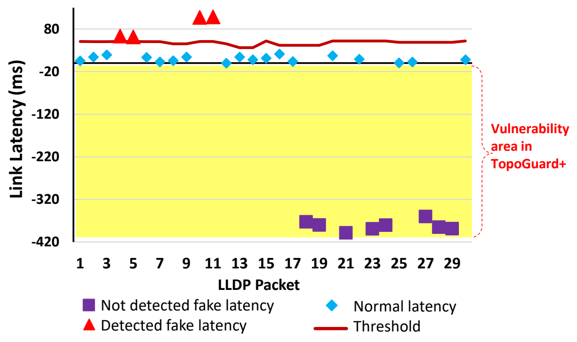


Fig. 1: An example of the weakness of *TopoGuard+* in detecting LLA.

Running example. The adversary can bypass the *TopoGuard+* by launching LLA. Fig. 1 presents an example of the weakness of *TopoGuard+* in detecting LLA. In this figure, the red line indicates the calculated threshold by LLI based on eq. 2. *TopoGuard+* marks a link latency value located above the threshold as a fake link (red triangles in Fig. 1). In addition, a latency value situated between the red and black lines is categorised as a valid link (blue diamonds in Fig. 1). However, *TopoGuard+* fails to detect a negative link latency value located in the yellow area (purple squares in Fig. 1), which is a vulnerability area of the *TopoGuard+*. LLA exploits this vulnerability to impose a fake link into the network.

The proposed LLA consists of two phases, namely, *overload phase* and *relay phase*. In the former phase, the adversary injects a huge amount of ARP traffic into the network, while in the latter phase, it relays the received LLDP packets from the switches via the out-of-band channel. The adversary leverages at least two compromised hosts for this purpose and frequently switches between two phases based on the LLDP propagation

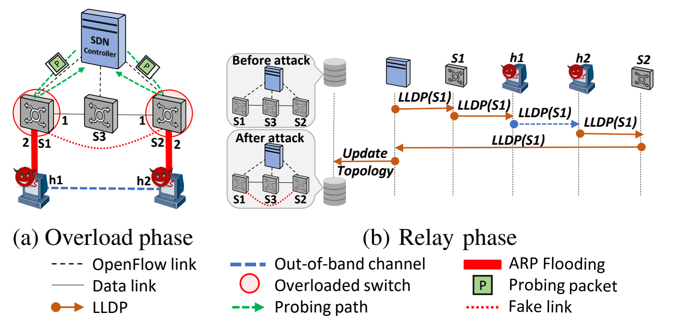


Fig. 2: The considered schematic of LLA. Fig. 2a includes compromised hosts send huge ARP traffic toward s_1 and s_2 to increase the RTT of probing packets. Fig. 2b explains compromised hosts relays the LLDP packet through out-of-band channel.

interval. To measure the LLDP packets interval, the adversary keeps the time difference between two consecutive packets.

The compromised hosts take two different roles during the attack; *flooder* and *listener*. In the *overload phase*, they play the role of *flooder* and send ARP floods to the switches. In the *relay phase*, as a *listener* role, they both listen to LLDP packets and relay them toward each other and the peer switches. Fig. 2 shows how the attack takes place in our scenario.

B. Overload Phase

During this phase, *flooder* hosts send ARP flooding traffic to switches s_1 and s_2 (see Fig. 2a). This traffic significantly increases the number of table-miss entries on the switches and directs a huge number of *Packet_In* messages toward the controller. Handing such an amount of packets results in increasing resource usage of the Open vSwitch (OVS) daemon from the switches. Consequently, the daemon pushes the incoming packets into the queues to be processed later. This results either in growing the probing packet's RTT or even dropping the packets due to the congestion on the ingress port of the switch. The increment in the RTT of the probe packets is enough for the adversary to launch the *relay phase*.

C. Relay Phase

In this phase, both hosts, i.e., h_1 and h_2, stop flooding ARP packets and change their role to *listener* for the incoming LLDP packets. When the controller issues LLDP packets, the adversary changes the attack phase from the *overload phase* to *relay phase*. Fig. 2b shows that upon receiving the LLDP packet in the *relay phase* by host h_1, it forwards this packet to host h_2 through a dedicated link. Host h_2 does the same task and forwards the LLDP packet to switch s_2. At this point, in the view of switch s_2, this is a new LLDP packet from a switch, and it has to forward this packet to the controller.

The controller receives the LLDP response packet, finds a change in the network topology, and updates it. To do so, it performs a check on the threshold and received LLDP packet latency using eq. 1 and eq. 2. Here, the values of T_{p_1} and T_{p_2} are high compared to the normal LLDP packets since they experience high latency in the *overload phase*. However, by applying eq. (1), the latency of the extra link between switches s_1 and s_2, i.e., T_l, stays in the valid range from the controller point of view, i.e, $T_l \leq T_h$. Even in some cases, the calculation shows a negative value for T_l. Hence, the LLI module in *TopoGuard+* fails to detect the LLA, and finally, the controller updates its view of the network topology by adding an extra link between switches s_1 and s_2.

By deeper investigation through the *TopoGuard+* source code, we realised that the implementation strategy used in this framework for measuring the control link latency leads our proposed LLA more cost-efficient for the adversary. By initiating the first *overload phase*, control link latency, i.e., T_{p_1} and T_{p_2}, increases to a high value. However, the abnormal observation is that *TopoGuard+* freezes on this value and never decreases it even after stopping the *overload phase*. It means that the adversary does not need to sustain or repeat the *overload phase* to keep the latency values high. This vulnerability in *TopoGuard+* implementation is because the controller does not initiate a new probing packet toward the switch without receiving the answer for the previous one.

Example Scenario. LLA can be applied in real-world attack scenarios such as SDN-based vehicular network [17]. In such scenarios, OpenFlow switches take the role of roadside units (RSUs). Hosts could connect to the RSUs and can be surveillance computers, roadside control platforms and edge servers. These hosts communicate with each other through wired or wireless channels. LLA creates a fake link that misleads the shortest path decision between two RSUs. It detours the traffic to a different path for the vehicles.

III. Countermeasure

In this section, we explain the detail of our contribution in protecting the network topology from LLA. We first state the architecture of the proposed system. Then, we describe how we collect our dataset and use ML techniques to classify the information of the dataset. Finally, we explain the detail of our implementation.

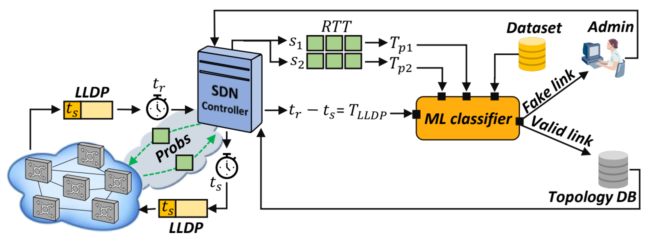


Fig. 3: The architecture of MLLG system to protect LLA.

TopoGuard+ uses a time interval threshold to detect the anomalies in the incoming LLDP packets to the controller. The adversary can bypass the threshold by launching LLA. We now describe the architecture of our proposed defence system to avoid bypassing this threshold value.

Architecture. We use ML techniques to detect the LFA and call our system MLLG. Fig. 3 presents an architecture of the MLLG system. In this figure, the MLLG has an offline ML classification model to train the system using a dataset containing various types of LFAs. Upon receiving the LLDP packet, the controller forwards it to the MLLG to verify its validity with the associated link. The controller either drops the LLDP packet or updates the topology database using the outcome of the MLLG.

Dataset. We modify the source code of *TopoGuard+* to extract three following features in nanosecond time scale. First, on receipt of a `Packet-in` message, LLDP propagation time (T_{LLDP}) is captured by taking difference between *sendTime* and *receiveTime* parameters of LLDP packet. Second, we capture the round trip time between the controller and switches to measure the control link latency for ingress (T_{p_1}) and egress (T_{p_2}) switches. We run *TopoGuard+* for nearly three days and collect all mentioned features to prepare our dataset. We develop a script to initiate three types of LFAs, namely, link fabrication attack [4], gradual link fabrication attack [18], and LLA. Our script collects 123,053 LLDP packets including 15,638 (12.8%) for the fake links and 107,415 (87.3%) for the normal inter-switch links.

Packet Classifier. We classify the packets in our dataset using seven ML classification techniques, namely, Logistic Regression (LR), Support Vector Machine (SVM), K-Nearest Neighbors (K-NN), Naive Bayes (NB), Random Forest (RF), and Multi-Layer Perceptron (MLP). The rationale for choosing these classifiers is that they have data-driven algorithms to identify a dynamic threshold in volatile network conditions. Additionally, the ML classifiers use the complex outlier detection algorithms, e.g., the distance-based algorithm in KNN and the Kernel-based approach in SVM, to improve the classification performance [19]. For example, when the data are not linearly separable, SVM applies a kernel function to detect the outliers. We train the classifiers using 80% of packets in the dataset and use the remaining 20% of traffic to test the learned model. Our ML classifiers show the best results for the following parameters; the `max depth` of 4 for RF and the `neighbour number` of 10 for KNN. In addition, we run SVM using linear classification.

Implementation. We now explain the implementation of MLLG. Algorithm 1 presents the pseudo-code of our system that is implemented on top of *TopoGuard+* in the Floodlight controller. To run our detection, the controller needs the LLDP packet data, the egress switch DPID of the switch forwarding the LLDP packet to the controller, and the arrival time of the LLDP packet as the input. Upon receiving the LLDP packets, the controller extracts all required information from the received LLDP packet and calculates the T_{LLDP}. Then, it has to measure the control link latency of the switches connected to the controller. At this point, the controller has all the needed parameters to verify the link's validity using our pre-trained model. It passes the values of the parameters to the `Defence` module and waits for the verification results. If the model detects the fake link, it first drops the LLDP, then informs the network administrator by raising a major security alarm and returns `FakeLink` as the status of the link.

Algorithm 1: ML-based Link Guard (MLLG)

```
Data: Incoming LLDP, LLDP received time ($t_r$), Egress switch
      DPID ($s_E$).
Result: $LinkState$.
1  $t_s \leftarrow$ Extract timestamp of initiated LLDP packet;
2  $s_I \leftarrow$ Extract ingress switch DPID from LLDP packet;
3  $T_{LLDP} \leftarrow t_r - t_s$;
4  $T_{p_1} \leftarrow$ link latency between controller and switch $s_I$;
5  $T_{p_2} \leftarrow$ link latency between controller and switch $s_E$;
6  $Defence \leftarrow$ Load pre-trained ML model;
7  $IsAttack \leftarrow Defence.predict(T_{LLDP}, T_{p_1}, T_{p_2})$;
8  if $IsAttack == 1$ then
9      Drop $LLDP$;
10     Set $LinkState \leftarrow$ "$FakeLink$";
11     Raise "Fake Link" major security alarm;
12 else
13     $LinkState \leftarrow$ "$ValidLink$";
14     Update topology database;
15 end
16 Retrun $LinkState$;
```

Otherwise, it updates the topology database of the network and returns `ValidLink` as the status of the link.

Topology update. The MLLG system can detect the topology updates if the change in the topology, e.g., adding a link with its properties such as delay, is in the same delay range as the current dataset. However, for the changes that impose higher link latency, the update of the dataset is necessary. Specifically, we can categorise the topology updates into *two* sub-categories; a) low-impact changes and b) high-impact changes. In the former case, the MLLG can detect the topology changes without the need to update the dataset since the updated links' delays are within the same range as the current links' delays.In the latter case, the MLLG needs an updated dataset because the updated topology has some links with higher link latencies than the current dataset.

We implement the ML classification part of the *MLLG* in Python using the Scikit-learn library [20].

IV. Performance Evaluation

In this section, we present the network setup to run MLLG and report the performance of our system.

A. System Setup

We do our experiments on a virtual machine equipped with an Intel Core i5-6500 3.2GHz with 2 CPU cores and 8GB RAM running Ubuntu 14.04 LTS-64bit. We use Mininet for the simulation and build the network topology of Fig. 4. All the links connecting the hosts to the switches have 5 milliseconds (ms) of delay, including the inter-switch links. There are two compromised hosts connected to switches s_1 and s_3. They play the role of the *flooder* and *listener* in the *overload* and *relay* phases of the LLA. The two compromised hosts communicate via a dedicated out-of-band channel with 10ms of link delay. We run the Floodlight controller which includes *TopoGurad+* defence on the same VM to protect the network.

B. Running the Tests

To launch the *overload phase* of the LLA, we use *arping* to send a considerable number of ARP requests toward the

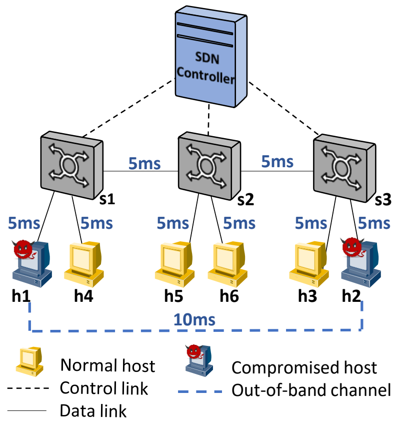
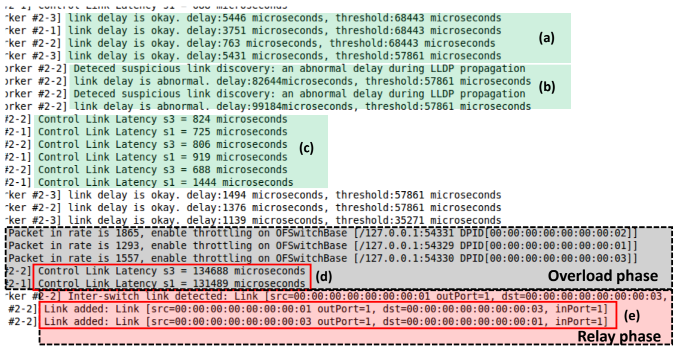


Fig. 4: Test environment topology and configuration.

switches (1,000,000 ARP request messages with 1-us interval). For *relay phase*, we use *scapy* [21] library in Python to relay the received LLDP packets via the out-of-band channel toward s_1 and s_3.

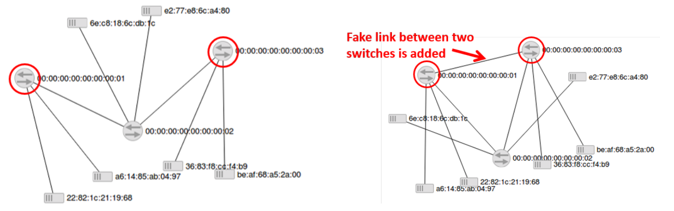


```
-2-1] Control Link Latency s1 = 688 microseconds
-ker #2-3] link delay is okay. delay:5446 microseconds, threshold:68443 microseconds
-ker #2-1] link delay is okay. delay:3751 microseconds, threshold:68443 microseconds
-ker #2-2] link delay is okay. delay:763 microseconds, threshold:68443 microseconds
-ker #2-3] link delay is okay. delay:5431 microseconds, threshold:57861 microseconds
orker #2-2] Deteced suspicious link discovery: an abnormal delay during LLDP propagation
orker #2-2] link delay is abnormal. delay:82644microseconds, threshold:57861 microseconds
orker #2-2] Deteced suspicious link discovery: an abnormal delay during LLDP propagation
orker #2-2] link delay is abnormal. delay:99184microseconds, threshold:57861 microseconds
#2-2] Control Link Latency s3 = 824 microseconds
#2-1] Control Link Latency s1 = 725 microseconds
#2-2] Control Link Latency s3 = 806 microseconds
#2-1] Control Link Latency s1 = 919 microseconds
#2-2] Control Link Latency s3 = 688 microseconds
#2-1] Control Link Latency s1 = 1444 microseconds
-ker #2-3] link delay is okay. delay:1494 microseconds, threshold:57861 microseconds
-ker #2-2] link delay is okay. delay:1376 microseconds, threshold:57861 microseconds
-ker #2-3] link delay is okay. delay:1139 microseconds, threshold:35271 microseconds
Packet in rate is 1865, enable throttling on OFSwitchBase [/127.0.0.1:54331 DPID[00:00:00:00:00:00:00:02]]
Packet in rate is 1293, enable throttling on OFSwitchBase [/127.0.0.1:54329 DPID[00:00:00:00:00:00:00:01]]
Packet in rate is 1557, enable throttling on OFSwitchBase [/127.0.0.1:54330 DPID[00:00:00:00:00:00:00:03]]
2-2] Control Link Latency s3 = 134688 microseconds
2-1] Control Link Latency s1 = 131489 microseconds
-ker #2-2] Inter-switch link detected: Link [src=00:00:00:00:00:00:00:01 outPort=1, dst=00:00:00:00:00:00:00:03,
 #2-2] Link added: Link [src=00:00:00:00:00:00:00:01 outPort=1, dst=00:00:00:00:00:00:00:03, inPort=1]
 #2-2] Link added: Link [src=00:00:00:00:00:00:00:03 outPort=1, dst=00:00:00:00:00:00:00:01, inPort=1]
```

Fig. 5: Floodlight system logs before and after launching LLA. **(a)** Normal inter-switch link latency and threshold. **(b)** *TopoGuard+* detects LFA **(c)** Normal control link latency value. **(d)** The adversary initiates *overload phase* resulted a sharp growth in control link latency. **(e)** *TopoGuard+* fails to detect LLA and adds the fake link to the network topology.

1) LLA Security Performance: We activate the system log of the Floodlight controller to check the behaviour of *TopoGuard+* against LLA. Fig. 5 shows a snapshot of the system logs before and after launching LLA. We separate Fig. 5 with the different coloured boxes to report the impact of the attack on various parts of the log. During normal execution of the system, we observe that all the inter-switch link latency values are less than the predefined threshold value, i.e., $\approx$ 57 ms (see Fig. 5 section (a)). Then, we conduct the LFA and observe that *TopoGuard+* detects the attack (see Fig. 5 section (b)) since the latency of received LLDP, e.g., 82,644us, is greater than the threshold, i.e., 57,861us. At this point, the control link latency value is $\approx$ 1ms in a normal situation (see Fig. 5 section (c)).

We now launch the LLA by applying the *overload phase*. By checking the log of the system, we observe a sharp growth in control link latency value to $\approx$ 130ms (see Fig. 5 section (d)).



(a) Before Link Latency Attack (b) After Link Latency Attack

Fig. 6: An example of LLA using a simple topology on the floodlight controller. 6a before and 6b after launching LLA.

Note that the Floodlight controller applies a simple throttling strategy to prevent itself from being overloaded. The controller cannot detect the floods of ARP packets issued during the *overload phase* since we issue them for a short period. At this point, we apply the *relay phase* (see Fig. 5 section (e)) and observe that *TopoGuard+* fails in detecting the attack. Consequently, it adds a bidirectional fake link between s_1 and s_3 as a valid link to the network topology.

Running example. We now show a running example of LLA using a simple topology on the Floodlight controller. We take a snapshot from our network topology from the WebUI of the Floodlight controller in Fig. 6 before and after launching the LLA. Fig. 6a presents the controller view of the current links in which there is no link between switch s_1 with DPID [00:00:00:00:00:00:00:01] and switch s_2 with DPID [00:00:00:00:00:00:00:03]. However, after launching the attack, the controller misleads into believing a direct link between these two switches (see Fig. 6b).

2) MLLG Defence Results: We evaluate the performance of our proposed MLLG defence by measuring *five* widely used metrics, namely, Accuracy (A), False Alarm (FA) or False Positive Rate (FPR), Recall (R) or True Positive Rate (TPR), Precision (P), and F1-score (FS). Then, we compare the obtained results with those of running *TopoGuard+* to show the effectiveness of MLLG system.

TABLE I: The detection performance comparison (%). A:= Accuracy; P:= Precision; FS:= F1-Score.

Defence	Classifier	A	FPR	TPR	P	FS
MLLG	KNN	**98.22**	0.97	**92.74**	**93.24**	**92.99**
	MLP	96.80	0.98	81.58	92.36	86.64
	RF	96.02	0.92	75.06	92.22	82.76
	LR	88.05	2.03	19.98	58.79	29.82
	SVM	87.42	**0.40**	3.77	57.84	7.08
	NB	87.28	2.65	18.22	49.95	26.70
TopoGuard+	LLI	84.28	5.48	14.03	27.14	18.50

Table I shows the performance of the different classifiers of our systems against *TopoGuard+*. We find that the accuracy of the KNN, MLP, and RF algorithms is at least 96%, and the KNN algorithm has the highest accuracy. The reason is that the feature labels are known, and their values are mostly near each other. Moreover, LLDP traffic has no noise data and has been prepossessed. While the other classifiers, including the one of *TopoGuard+*, have the accuracy of less than 89%. Specifically, the accuracy of *TopoGuard+* is 84.28%. The

KNN algorithm has the highest values for TPR, P, and FS with 92.74%, 93.24%, and 92.99%, while the corresponding parameter values for those of *TopoGuard+* are 14.03%, 27.14%, and 18.50%, respectively. Finally, the SVM classifier reports 0.40% of the packets as FPR, while *TopoGuard+* makes a false alarm for 5.48% of the packets in our dataset.

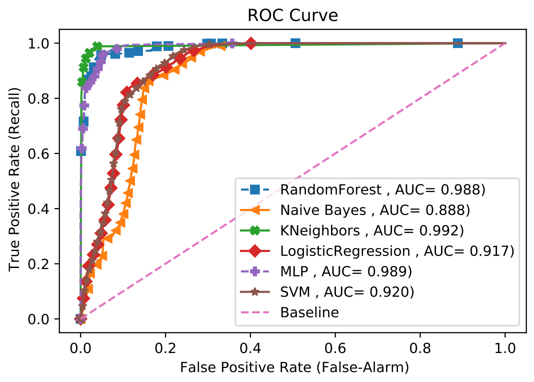


Fig. 7: ROC curves and AUC comparison for different classifiers.

We now report the ability of the classifiers in detecting the LLA using the Receiver Operating Curve (ROC). The ROC illustrates TPR versus FPR in which the *Area Under the Curve (AUC)* is calculated to determine which classifier best predicts the fake or valid links. Fig. 7 indicates that the MLP classifier achieves the highest AUC of 0.994 among all other classifiers.

The rationale to leverage the ML classification technique in link verification is that it makes MLLG different from previous solutions such as *TopoGuard+*. We summarise the differences as follows. First, MLLG improves the detection and accuracy rate compared to the current best solution *TopoGuard+*. Second, MLLG empowers the controller to verify the links based on a massive LLDP traffic and collected latencies dataset over the desired period.

V. Discussion

In this section, we discuss the limitations of our attack scenario in more detail. Our proposed LLA attack and MLLG defence are currently implemented in Floodlight with *TopoGuard+* defence which could be extended to other controllers [4]. We need at least two compromised hosts with an out-of-band channel or one dual-homed host to launch an LLA. In case of having several adversaries in a large network, MLLG still could detect the attack. To do this, we require to store all probe packets and LLDP traffic of all switches in the dataset.

We suppose that the adversary initially conducted some experiments with various duration ranges for the *overload phase* to estimate the frequency of probe packets. Using the default configuration of *TopoGuard+*, the controller sends a probe packet every 5 seconds.

Detecting ARP floods with a low false-positive rate and for a short period is a challenge for most network Intrusion Detection Systems (IDSs), such as Snort and Bro, due to a considerable number of ARP requests in large-scale networks [4]. This weakness helps the adversary mitigate the risk of detecting the ARP flood by a defence system in *overload phase* of LLA.

VI. Related Work

In this section, we briefly describe the state-of-the-art on topology poisoning attack in SDN networks. This attack was first introduced in [4], and it is a type of protocol-based attack in which the adversary does not need to have the control plane access or know the vulnerability of the controller. The two security threats in this attack are link fabrication attack (LFA) and host location hijacking attack (HLHA). The LFA aims to add a virtual fake link among the switches in the network. While HLHA aims to tamper the location of a host in the network to mislead the traffic flows toward the adversary. Therefore, LLA is categorised in LFA group.

TopoGuard protects the network from LFA by a port labelling strategy. The controller uses different labels such as HOST, SWITCH, and ANY to classify the devices based on the received traffic. The main drawback of *TopoGuard* is that the adversary can compromise a host and pretend its label as a switch to relay the LLDP packets. SPHINX [11] is a framework that compares two flow graphs of the network traffic and finds potential anomalies. Nevertheless, the defence mechanism of SPHINX could not detect all types of topology poisoning attacks. In contrast, MLLG detects link latency anomalies by analysing LLDP traffic along with packet latency values.

In [10], a new type of LFA, namely, port amnesia attack, has been introduced. The authors showed that the port-labelling technique of *TopoGuard* could be bypassed if the adversary switches the port status of the compromised host from down to up during the LLDP propagation. Then, they developed *TopoGuard+* framework, which contains a Link Latency Inspector (LLI) module to detect the fake link. The LLI calculates the latency of an inter-switch link and compares it with a latency threshold. However, this threshold could not be updated based on network traffic patterns, resulting in valid link removal.

The work in [18] designs an LFA against *TopoGuard+* which could gradually increase the latency threshold until it becomes greater than the latency of the out-of-band-channel. The adversary in LLA aims to increase the value of control link latency (and not a threshold value) which takes only a few seconds to achieve. However, launching the attack in [18] needs hours of preparation. Moreover, they do not present a specific defence against the attack. The work in [22] also proposes a threshold-based defence by collecting the samples from the latency LLDP packets and compare them with the threshold. Nevertheless, similar to [18], the approach could cause significant false-positive predictions. However, MLLG achieves less than 1% false-positive prediction rate. In [23], a worm-hole attack is proposed to relay the packet over the fake link without using any out-of-band channel.

Alimohammadifar et al. [24] proposed Stealthy Probing-Based Verification (SPV) defence that sends probing packets toward the switches to find the potential fake links. However, the integration of SPV with the current controller decreases its security features. However, other recent tools such as the one in [25] can be used to get more insights on the root of the attacks in the SDN-based networks. A security architecture is developed in [26] to mitigate the risk of attacks caused by malicious end hosts in SDN, such as a set of topology poisoning attacks. The solution works based on the enforcement of security policies in the data and control plane. However, similar to [11], it fails to detect all type of LFAs.

VII. Conclusion

In this paper, we introduced the link latency attack in SDN networks and examined it on the recent version of *TopoGuard*, which is *TopoGuard+*. Experiments show that *TopoGuard+* cannot detect such an attack in the simulated network topology. We designed and developed a machine learning-based system equipped with several packet classifiers to defend against such an attack. Our system uses a dataset to train the ML-based classifier and leverages this information by the controller to provide the countermeasure for the attack. We plan to gradually update the system classification model of our proposed defence algorithm using real-time ML techniques. Using such techniques will allow us to continuously update our dataset and training model. Additionally, we are going to extend our solution with a programmable switching substrate and include other attack classes on SDN that can benefit from our ML approach.

Acknowledgment

This work was partially funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

References

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

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

[3] S. Khan, A. Gani, A. W. A. Wahab, M. Guizani, and M. K. Khan, "Topology discovery in software defined networks: Threats, taxonomy, and state-of-the-art," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 303–324, 2016.

[4] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures." in *NDSS*, vol. 15, 2015, pp. 8–11.

[5] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, "Efficient topology discovery in software defined networks," in *2014 8th International Conference on Signal Processing and Communication Systems (ICSPCS)*. IEEE, 2014, pp. 1–8.

[6] T. Alharbi, M. Portmann, and F. Pakzad, "The (in) security of topology discovery in software defined networks," in *2015 IEEE 40th Conference on Local Computer Networks (LCN)*. IEEE, 2015, pp. 502–505.

[7] N. Kaur, A. K. Singh, N. Kumar, and S. Srivastava, "Performance impact of topology poisoning attack in sdn and its countermeasure," in *Proceedings of the 10th International Conference on Security of Information and Networks*, ser. SIN '17, 2017, p. 179–184.

[8] Research and Market, "Software-defined networking market by component (sdn infrastructure, software, and services), sdn type (open sdn, sdn via overlay, and sdn via api), end user, organization size, enterprise vertical, and region - global forecast to 2025," 2021. [Online]. Available: https://bit.ly/3pDWlJK

[9] Ponemon Institute, "Cost of a data breach report 2020," 2021. [Online]. Available: https://bit.ly/3l0AjR4

[10] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, "Effective topology tampering attacks and defenses in software-defined networks," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 374–385.

[11] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: detecting security attacks in software-defined networks." in *Ndss*, vol. 15, 2015, pp. 8–11.

[12] MININET. An instant virtual network on your laptop (orother pc. [Online]. Available: http://mininet.org/

[13] Floodlight. Open sdn controller. [Online]. Available: http://floodlight.openflowhub.org/

[14] J. Abley, K. Lindqvist, E. Davies, B. Black, and V. Gill, "Ipv4 multihoming practices and limitations," *RFC4116, IETF, July*, 2005.

[15] T. Arnold, M. Calder, I. Cunha, A. Gupta, H. V. Madhyastha, M. Schapira, and E. Katz-Bassett, "Beating bgp is harder than we thought," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19, 2019, p. 9–16.

[16] Z. Akhtar, Y. S. Nam, R. Govindan, S. Rao, J. Chen, E. Katz-Bassett, B. Ribeiro, J. Zhan, and H. Zhang, "Oboe: Auto-tuning video abr algorithms to network conditions," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, 2018, p. 44–58.

[17] J. Wang, Y. Tan, J. Liu, and Y. Zhang, "Topology poisoning attack in sdn-enabled vehicular edge network," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9563–9574, 2020.

[18] E. Marin, N. Bucciol, and M. Conti, "An in-depth look into sdn topology discovery mechanisms: Novel attacks and practical countermeasures," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1101–1114.

[19] H. J. Escalante, "A comparison of outlier detection algorithms for machine learning," in *Proceedings of the International Conference on Communications in Computing*, 2005, pp. 228–237.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[21] "Scapy: Packet crafting for python2 and python3." [Online]. Available: https://scapy.net/

[22] D. Smyth, S. McSweeney, D. O'Shea, and V. Cionca, "Detecting link fabrication attacks in software-defined networks," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–8.

[23] J. Hua, Z. Zhou, and S. Zhong, "Flow misleading: Worm-hole attack in software-defined networking via building in-band covert channel," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1029–1043, 2020.

[24] A. Alimohammadifar, S. Majumdar, T. Madi, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "Stealthy probing-based verification (spv): an active approach to defending software defined networks against topology poisoning attacks," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 463–484.

[25] B. E. Ujcich, S. Jero, R. Skowyra, A. Bates, W. H. Sanders, and H. Okhravi, "Causal analysis for software-defined networking attacks," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[26] V. Varadharajan and U. Tupakula, "Counteracting attacks from malicious end hosts in software defined networks," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 160–174, 2019.