Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures

Sungmin Hong*
SUCCESS Lab
Texas A&M University
ghitsh@tamu.edu

Lei Xu*
SUCCESS Lab
Texas A&M University
xray2012@cse.tamu.edu

Haopei Wang
SUCCESS Lab
Texas A&M University
haopei@cse.tamu.edu

Guofei Gu
SUCCESS Lab
Texas A&M University
guofei@cse.tamu.edu

***Abstract*—Software-Defined Networking (SDN) is a new networking paradigm that grants a controller and its applications an omnipotent power to have holistic network visibility and flexible network programmability, thus enabling new innovations in network protocols and applications. One of the core advantages of SDN is its logically centralized control plane to provide the entire network visibility, on which many SDN applications rely. For the first time in the literature, we propose new attack vectors unique to SDN that seriously challenge this foundation. Our new attacks are somewhat similar in spirit to spoofing attacks in legacy networks (e.g., ARP poisoning attack), however with significant differences in exploiting unique vulnerabilities how current SDN operates differently from legacy networks. The successful attacks can effectively poison the network topology information, a fundamental building block for core SDN components and topology-aware SDN applications. With the poisoned network visibility, the upper-layer OpenFlow controller services/apps may be totally misled, leading to serious hijacking, denial of service or man-in-the-middle attacks. According to our study, all current major SDN controllers we find in the market (e.g., Floodlight, OpenDaylight, Beacon, and POX) are affected, i.e., they are subject to the Network Topology Poisoning Attacks. We then investigate the mitigation methods against the Network Topology Poisoning Attacks and present *TopoGuard*, a new security extension to SDN controllers, which provides automatic and real-time detection of Network Topology Poisoning Attacks. Our evaluation on a prototype implementation of *TopoGuard* in the Floodlight controller shows that the defense solution can effectively secure network topology while introducing only a minor impact on normal operations of OpenFlow controllers.**

I. Introduction

Software-Defined Networking (SDN) has emerged as a new network paradigm to innovate the ossified network infrastructure by separating the control plane from the data plane (e.g., switches), as well as providing holistic network visibility and flexible programmability. As the brain of the network, a SDN controller grants users a great tool to design and control the network using their own applications atop the controller's core services. Not only in academic environments, but also in real-world production networks, SDN, particularly its popular realization OpenFlow[1], has been increasingly employed. Many application scenarios have been studied and deployed since then, ranging from campus network innovation to cloud network virtualization and datacenter network optimization.

Since the controller is the core of the SDN architecture, if the OpenFlow controller suffers from any serious vulnerability in its design/implementation, the entire network would be thrown into chaos, or even totally under the control of attackers. To date, several approaches to SDN security have been proposed. FortNOX [27] solves the rule conflicts that violate existing security policies, and provides role-based authorization and security constraint enforcement in the controller kernel. FRESCO [30] provides a composable programming framework to facilitate fast development of SDN security apps. FlowVisor [28] provides isolation of different network slices and VeriFlow [19] verifies network-wide correctness by checking the forwarding graph representation derived from flow modification messages. AvantGuard [31] provides security protection against data-to-control-plane saturation attacks. However, these approaches have primarily concentrated on network/rule consistency/authorization, conflict resolution, app developing, or network resource consumption/scalability, not on the fundamental vulnerabilities inside OpenFlow controllers.

In this paper, we study network topology services/apps of the mainstream OpenFlow controllers and identify several new vulnerabilities that an attacker can exploit to poison the network topology information in OpenFlow networks. The whole network-wide visibility is one of the key innovations provided by SDN compared to legacy networking technologies. As a fundamental building block for network management, the topology information is adopted to most controller core services and upper-layer apps, e.g., those related to packet routing, mobility tracking, and network virtualization and optimization. However, if such fundamental network topology information is poisoned, all the dependent network services will become immediately affected, causing catastrophic problems. For example, the routing services/apps inside the OpenFlow controller

* The first two authors contribute equally to the paper.

[1] In this paper, we may use SDN and OpenFlow interchangeably.

NDSS '15, 8-11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23283

can be manipulated to incur a black hole route or man-in-the-middle attack. In this paper, we uncover new security loopholes existing in current Host Tracking Service and Link Discovery Service in OpenFlow controllers. Furthermore, We introduce two Network Topology Poisoning Attacks, i.e., Host Location Hijacking Attack and Link Fabrication Attack. Upon the exploitation of the Host Tracking Service, an attacker can hijack the location of a network server to phish its service subscribers. By poisoning the Link Discovery Service, an adversary can inject false links to create a black-hole route or launch a man-in-the-middle attack to eavesdrop/manipulate messages in the network. Our new attacks share some similarities in spirit to traditional spoofing attacks in legacy networks (e.g., ARP Poisoning Attack), however with significant differences in exploiting unique SDN vulnerabilities. According to our study, all current major open source SDN controllers in the market (i.e., Floodlight, OpenIRIS, OpenDayLight, Beacon, Maestro, NOX, POX and Ryu) are affected. This raises a serious alarm because these vulnerabilities could significantly impact the deployment of current SDN networks and greatly hurt the future of SDN.

In order to mitigate such attacks, we investigate possible defense strategies. We note that it is difficult to simply use static configuration to solve the problem (similar to using static ARP entry for hosts or the port security feature for switches [2] to solve ARP poisoning attacks), because it requires tedious and error-prone manual effort and is not suitable for handling network dynamics, which is a valuable innovation of SDN. To better balance the security and usability, in this paper, we propose *TopoGuard*, a new security extension to the existing OpenFlow controllers to provide automatic and real-time detection of network topology exploitation. By utilizing SDN-specific features, *TopoGuard* checks precondition and postcondition to verify the legitimacy of host migration and switch port property to prevent the Host Location Hijacking Attack and the Link Fabrication Attack.

In short, our paper makes the following contributions:

- We perform the first security analysis on the SDN/OpenFlow Topology Management Service. In particular, we have discovered new vulnerabilities in the Device Tracking Service and Link Discovery Service in eight current mainstream SDN/OpenFlow controllers.
- We propose Network Topology Poisoning Attacks to exploit the vulnerabilities we have found. We demonstrate the feasibility of those attacks both in the Mininet emulation environment and a hardware SDN testbed.
- We investigate the defense space and propose automatic mitigation approaches against Network Topology Poisoning Attacks, along with a prototype defense system, *TopoGuard*, currently implemented in Floodlight, but could be easily extended to other controllers. Our evaluation shows that *TopoGuard* imposes only a negligible performance overhead.

The rest of the paper is organized as follows: Section II provides background information about the SDN/OpenFlow and its Topology Management Service. Section III describes vulnerabilities in existing SDN Topology Management Service and presents the Network Topology Poisoning Attacks. Section IV investigates the defense strategies against Network Topology Poisoning Attacks. Section V proposes our design and implementation of *TopoGuard* along with its effectiveness and performance evaluation. Section VI discusses possible issues and limitations. Section VII reviews related work about current security research in the SDN/OpenFlow area and similar attacks in the literature. Section VIII concludes this paper.

II. Background

In this section, we provide an introduction to SDN/OpenFlow and its Topology Management Services implemented in the existing OpenFlow controllers.

A. SDN/OpenFlow Background

The Basic Operation of SDN. Software-Defined Networking (SDN) is a new programmable network framework that decouples the control plane from the data plane. An SDN application in the control plane generates complicated network functions such as computing a routing path, monitoring network behavior, and managing network access control. The data plane handles hardware level network packet processing based on high level policies from the control plane. SDN enables users to design and distribute innovative flow handling and network control algorithms conveniently, and add much more intelligence and flexibility to the control plane. We can implement new control functions or protocols just as writing a normal application (analogous to writing an app for smartphone/Android OS). OpenFlow, as a leading reference implementation of SDN, defines the communication protocol between the control plane and the data plane. An OpenFlow switch must establish a TCP connection (with an option of TLS/SSL) to the OpenFlow controller before exchanging symmetric/asynchronous OpenFlow messages. When a new packet comes into an OpenFlow switch, the switch checks if the packet matches any existing flow rules. If so, the switch will process the packet based on the matching rule with the highest priority. Otherwise, the switch sends a *Packet-In* OpenFlow message to the OpenFlow controller to ask for proper actions according to network policies specified in the SDN apps. Once the specific decision is made, the OpenFlow controller either issues a *Packet-Out* message for the one-time packet processing or instructs the OpenFlow switch to install new flow rules by sending a *Flow-Mod* message. In addition, whenever any change on a switch port is detected, a *Port-Status* OpenFlow message must be sent to the controller.

Operational Distinctions Between SDN and Legacy Networks. SDN/OpenFlow introduces many networking innovations, as described in its latest specification [4]. In this paper, we do not emphasize all of those details between the

Distinctions	OpenFlow Networks	Legacy Networks
Source MAC Address	Unchanged when passing the OpenFlow switches	Changed when passing layer 3 devices
Control Message Authentication	Between the OpenFlow switches and controllers	Among layer 3 devices
Spanning Tree Implementation	Centralized calculation based on topology	In a distributed manner

TABLE I: The Distinctions Between Legacy Networks and OpenFlow Networks Highlighted in This Paper

OpenFlow networks and the legacy networks. Instead, we only list some distinctions relevant to the context of this paper, as shown in Table I, which are helpful to explain the idea of this paper. First, the legacy layer 3 devices (e.g., routers) rewrite the layer 2 identity (i.e., source MAC address) of a packet when forwarding it, however the normal behavior of an OpenFlow switch is to keep the layer 2 identity unchanged.[2] For example, the legacy layer 3 devices enforce Proxy ARP [7] (i.e., they reply ARP requests intended for destination hosts) by "falsifying" the layer 2 identity inside ARP replies; the OpenFlow controller, nevertheless, provides ARP proxy without manipulating ARP replies due to its global view of host information. Second, in legacy networks, the Control Message Authentication is among layer 3 devices; however, in the OpenFlow networks, the Control Message Authentication is enforced between switches and controllers because of the separation of the control plane and the data plane. Third, unlike legacy networks that utilize BPDUs (Bridge Protocol Data Units) to compute Spanning Tree Protocol (STP) in a distributed manner, the OpenFlow controller is capable of centralized calculation of Spanning Tree based on the topology view.

B. Unique Topology Management in OpenFlow Controllers

Different from legacy networks, topology management is unique in SDN networks due to the newly added, logically centralized network controller. In order to facilitate network management and programmability, the OpenFlow controller maintains topology information and provides such visibility to upper services/apps, as shown in Figure 1. More importantly, not only all controllers use the same topology discovery mechanism, but also both core controller components and SDN applications are tightly coupled with the topology information. The more OpenFlow applications are developed, the more critical dependencies would affect the whole components in the controller.

Generally, in an SDN/OpenFlow network, topology management includes three parts: (1) switch discovery, (2) host discovery, (3) internal link (i.e., switch-to-switch link) discovery. The switch discovery does not require any additional protocol since when an OpenFlow switch establishes a connection to the OpenFlow controller, the switch information should be stored in the OpenFlow controller for future management. When a switch receives any packet from a host and it does not match any flow entry in the flow table, a *Packet-In* message encapsulating the packet is sent to the OpenFlow controller. The OpenFlow controller then learns the information about the host and its location (i.e., the corresponding attached switch port) from the message. For internal link discovery, the OpenFlow controller herein utilizes OpenFlow Discovery Protocol (OFDP). In this paper, we mainly focus on the Host Tracking Service and Link Discovery Service inside the OpenFlow controller.

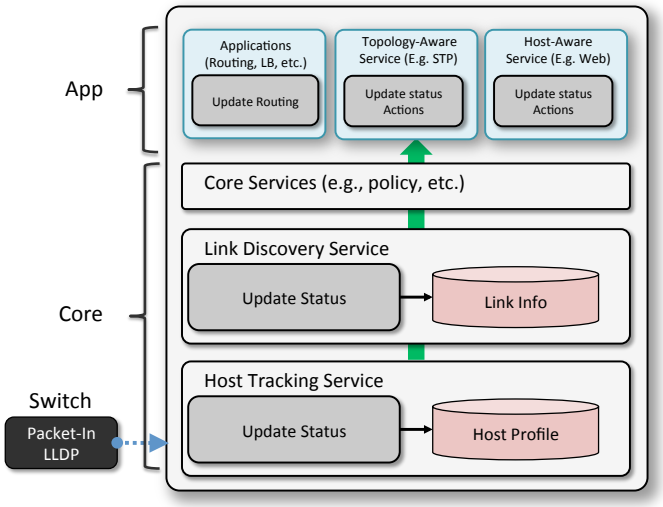


Fig. 1: Service Dependencies among Layered Controller Components

Host Tracking Service. Inside an OpenFlow controller, Host Profile is maintained to track the location of a host. There are significant advantages for Host Tracking. For example, in a data center, it is tedious and error-prone to manually maintain the locations of virtual machines due to their frequent migration. Also, as demonstrated in [3], the OpenFlow controller with host location tracking can provide seamless handoff among different access points. In this regard, the Host Tracking Service (HTS) in the controller is to provide an easy way to guarantee flexible network dynamics by dynamically probing *Packet-In* messages and updating Host Profiles.

Let us take a close look at how an OpenFlow controller tracks dynamics of host devices. In an OpenFlow network, the OpenFlow controller reactively listens to *Packet-In* messages to maintain Host Profile. During this procedure, the OpenFlow controller mainly handles two relevant host events (i.e., JOIN and MOVE). The scenario for the first event is that, when the OpenFlow controller fails to locate an existing Host Profile

[2]The OpenFlow switches can manipulate packet headers by using the *Set-Field* action if the application needs to do so. However, *Set-Field* is typically optional for the OpenFlow switch implementations to increase their functionalities.

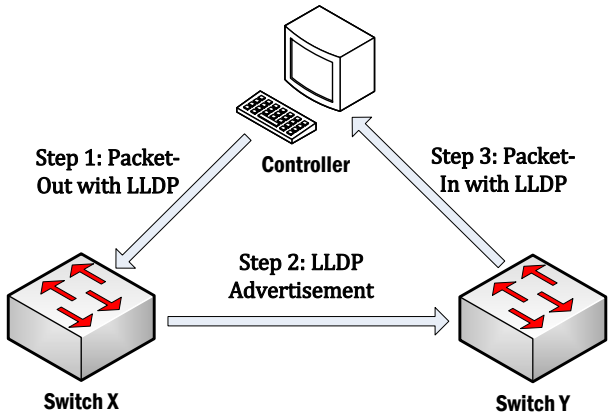


Fig. 2: Link Discovery Procedure in OpenFlow Networks

according to the information from incoming *Packet-In* messages, it creates a new Host Profile. In such case, the controller assumes a new host joins the network. The second scenario occurs when the OpenFlow controller successfully locates a Host Profile but finds mismatched location information between the Host Profile and *Packet-In* messages. In the case, it assumes the host has moved to a new location and then updates the location information inside the corresponding Host Profile.

Table III shows the Host Tracking Services in current OpenFlow controller platforms. In order to handle host mobility, the existing OpenFLow controllers maintain a profile for each host. In detail, the Host Profile includes: (1) MAC address, (2) IP address, and (3) Location information (i.e., the DPID and the port number of the attached switch as well as the last seen timestamp). Normally, a Host Profile is indexed by the MAC address. Floodlight, for example, indexes the Host Profile with MAC address and VLAN ID. Beacon and the old version of Host Tracking Service in OpenDayLight support both MAC address and IP address as the index.

Link Discovery Service. To dynamically discover topology, the Link Discovery Service (LDS) inside OpenFlow controllers uses Open Flow Discovery Protocol (OFDP), which refers to LLDP (Link Layer Discovery Protocol) packets with format shown in Table II, to detect internal links between switches.

Figure 2 depicts the link discovery procedure in an OpenFlow network. Note that here we illustrate only a unidirectional link discovery for simplicity (the discovery of opposite link is performed in a similar fashion). Initially, the OpenFlow controller sends out *Packet-Out* messages to Switch X with the payload of a controller-specific LLDP packet. The payload of the LLDP packet contains DPID and the output port of Switch X. Upon receiving the LLDP packet, Switch X advertises it to all other ports in a broadcast manner. Typically, in an OpenFlow network, this kind of broadcast is achieved by iterative transmissions of one LLDP packet to each broadcast-enabled port of a switch. Then, the next-hop Switch Y, driven by its firmware or under explicit instructions of the attached OpenFlow controller, reports the incoming LLDP packet to the controller with the ingress Port ID and DPID of Switch Y via a *Packet-In* message. When receiving the *Packet-In* message from Switch Y, the OpenFlow controller can detect a unidirectional link from Switch X to Switch Y. Table III shows link discovery components in existing OpenFlow controller platforms. We find that all of these controllers embrace the internal link discovery procedure as we describe above.

In addition to the internal link discovery, some OpenFlow controller implementations, e.g., Floodlight and OpenIRIS, also propose a scheme to detect *multi-hop links*, which refers to links traverse across a Non-OpenFlow island. In order to detect such links, Floodlight leverages BDDP packets(i.e., a broadcast version of LLDP packets with a broadcast destination MAC address) to overcome unpredictable forwarding behaviors of Non-OpenFlow switches.

Finally, we note that these topology management services are critical building blocks to provide important information to other topology-dependent services (e.g., shortest-path routing and spanning tree) and apps (e.g., network routing management/optimization). For interested readers, we provide more details of two representative topology-dependent services (among many others), i.e. shortest-path routing and spanning tree in SDN, in the Appendex A.

III. Topology Poisoning Attack

In this section, we describe the vulnerabilities in Topology Management Services of existing OpenFlow controllers that we have found. Based on such vulnerabilities, we propose and measure two OpenFlow Network Topology Poisoning Attacks, i.e., Host Location Hijacking Attack and Link Fabrication Attack.

A. Threat Model and Experimental Environment

We assume an adversary possesses one or more compromised hosts or virtual machines (e.g., through malware infection) in the SDN/OpenFlow network and has the read and write privilege on packets in the operating system.[3] Note that, in this paper, we assume the transmission of OpenFlow messages via the control channel can be properly protected by SSL/TLS.

Furthermore, we demonstrate the SDN-specific Network Topology Poisoning Attacks both in Mininet 2.0 [24] and a physical environment (with hardware OpenFlow switches). Mininet 2.0 is widely used for emulating an OpenFlow network environment. Our hardware testbed includes several OpenFlow-enabled hardware: TP-LINK(TL-WR1043ND) and LINKSYS(WRT54G) which run OpenWRT firmware with an OpenFlow extension and PCs with Intel Core2 Quad processor and 2GB memory.

[3]In the extreme case, the adversary can be an insider.

Dl_dst	Dl_src	Eth_type	Chassis ID TLV	Port ID TLV	TTL TLV	Optional TLVs	End TLV
01:80:C2:00:00:0E	Outgoing Port MAC	0X88CC	DPID of Switch	Port Number of Switch	Time to Live	E.g., System Description	End Signal of LLDP

TABLE II: The Format of LLDP Packets

Controller Platform	Link Discovery Service	TLVs	Host Tracking Service	Host Profile
Ryu	switches.py	DPID, Port ID, TTL	host_tracker.py	MAC, ***IP***, Location
Maestro	DiscoveryApp.java	DPID, Port ID, TTL	LocationManagementApp.java	***MAC***, Location
NOX	discovery.py	DPID, Port ID, TTL	hosttracker.cc	***MAC***, Location
POX	discovery.py	DPID, Port ID, TTL, System Description	host_tracker.py	***MAC***, IP, Location
Floodlight	LinkDiscoveryManager.java	DPID, Port ID, TTL, System Description	DeviceManagerImpl.java	***MAC***, ***VLAN ID***, IP, Location
OpenDayLight	DiscoveryService.java	DPID, Port ID, TTL, System Description	DeviceManagerImpl.java	***MAC***, ***VLAN ID***, IP, Location
OpenIRIS	OFMLinkDiscovery.java	DPID, Port ID, TTL, System Description	OFMDeviceManager.java	***MAC***, ***VLAN ID***, IP, Location
Beacon	TopologyImpl.java	DPID, Port ID, TTL, Full Version of DPID	DeviceManagerImpl.java	***MAC***, VLAN ID, ***IP***, Location

TABLE III: Topology Management Services (the bold attributes in Host Profile column denotes the identifier of a host)

B. Host Location Hijacking Attack

In this part, we detail the Host Location Hijacking Attack which is a kind of spoofing attack by exploiting the Host Tracking Service in the OpenFlow network.

Exploitation in Host Tracking Service. As described in Section II, HTS in the OpenFlow controllers maintains Host Profile for each end host to track network mobility. As long as hosts (or virtual machines) migrate, HTS can quickly react to such event. In particular, HTS recognizes the motion of hosts by monitoring *Packet-In* messages. Once being aware that a particular host migrates to a new location, i.e., DPID or ingress Port ID is different from the corresponding entry of the Host Profile, HTS updates Host Profile and optionally raises a HOST_MOVE event to its subscriber services. However, such update mechanism is vulnerable due to the ignorance of authentication.

In order to investigate security issues when HTS updates Host Profile, we manually analyze the source code of HTS in current mainstream OpenFlow controllers. Our study shows that existing OpenFlow controllers have few security restrictions on host location update. For instance, Floodlight and the old version of OpenDayLight controller provides an empty-shell API, called *isEntityAllowed*, which accepts every host location update rather than blocking possible spoofing attacks. The POX controller throws a warning if the observed time for device migration is less than a minimum expected time (60 seconds by default). However, we assume that such simple verification is easy to bypass if the adversary recognizes this feature in advance. The lack of consideration on security provides an opportunity for an adversary to tamper host location information by simply impersonating the target host. What is worse, all OpenFlow controllers have a routing module that utilizes the host location information to make the packet forwarding decision. That is, if an adversary can tamper the location information, he/she has a potential to hijack the traffic towards the host.

Here, we propose an attacking strategy where the adversary crafts packets with the same identifier of the target host. Once receiving the spoofed packet, the OpenFlow controller will be tricked to believe that the target host moves to a new location, which actually is the attacker's location. As a result, future traffic to the target is hijacked by the adversary. Next, we introduce a practical example of harvesting web clients by exploiting the vulnerability in HTS, as shown in Figure 3.

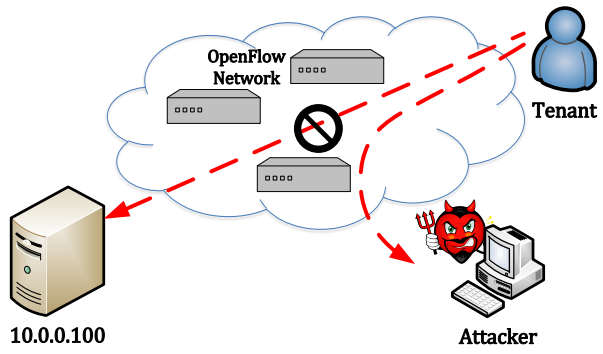


Fig. 3: Attacker impersonates a specific web server to phish users

Web Clients Harvesting. In order to conduct a Web Clients Harvesting Attack, we firstly need to retrieve the identifier of the target. From Table III, we find that the host identifier varies among MAC address, VLAN ID, and IP address depending on the platform and version of OpenFlow controllers. It is trivial to know the IP address if we have already chosen an attacking target. Besides, the VLAN ID is normally unused during the update procedure of Host Profile. On the other hand, as MAC address is regarded as the (or part of) identifier for hosts in most OpenFlow controllers (except for Ryu), we can use ARP request packets to probe the MAC address of our target. Note that such simple probe method is feasible because the OpenFlow network does not change the source MAC address during packet transmission.

In addition, one difficulty to successfully exploit HTS

(a) Connected to Genuine Server (b) Hijacked by Malicious Server

Fig. 4: Web Impersonation Attack

lies in that the adversary needs to race with the target host, because any traffic initiated from the target host can correct host location information in the controller. To overcome the non-determinism of such situation, we could set our target as a server. This is because a server normally runs in a passive mode, i.e., it opens specific port(s) and waits for remote connections from clients.

In this paper, we launched a Host Location Hijacking Attack in our experimental environment. We chose Floodlight (master) as the OpenFlow controller, atop which the Host Tracking Service and Shortest Path Routing Service are enabled by default. We deployed an Apache2 [1] web server with IP address "10.0.0.100" and several hosts in our customized OpenFlow topology. The reachability test is shown in Figure 4(a), that is, before we launch the Host Location Hijacking Attack, clients can visit the genuine web server with our assigned IP address. Upon a compromised host, we also run a Web service and send ARP request to probe the corresponding MAC address of "10.0.0.100". We then use Scapy [8] to periodically inject fake packets in the name of our target (the genuine web server "10.0.0.100"). After that, we find the new coming client attempting to visit the web server "10.0.0.100" is directed to the malicious server, as shown in Figure 4(b).

C. Link Fabrication Attack

In this part, we show how an adversary can fabricate a link into the network topology to threaten normal network activities.

Exploitation in Link Discovery Service. To build the entire network topology and handle dynamics of a network, OpenFlow adopts OFDP for topology management. Typically, OpenFlow controllers utilize LLDP packets to discover links among OpenFlow switches. However, there exist security flaws during the link discovery procedure.

As implicated in Section II, the LDS in OpenFlow controllers is subject to two invariants: 1) The integrity/origin of LLDP packets must be ensured during the Link Discovery procedure; 2) The propagation path of LLDP packets can only contain OpenFlow-enabled switches. Unfortunately, those two security invariants are poorly enforced in current instantiations of OpenFlow controllers. In our study, we find that the syntax of LLDP packets varies among different OpenFlow controller platforms. For example, POX and Floodlight use an integer variable to represent the port number of a remote switch whereas the form of the port number in OpenDayLight is the ASCII value of characters. In addition, some OpenFlow controllers add extra TLVs (Type-Length-Values), e.g., system description, to enrich the semantics of LLDP packets. The controller-uniqueness of LLDP packets to some extent protects the LLDP "origin invariant." However, we argue that it is not enough when taking into account the open source nature of OpenFlow controllers and simple semantics of LLDP. Also, the Floodlight controller adds an origin authenticator as an extra TLV of LLDP packets to verify the origin of LLDP packets. However, the authenticator keeps unchanged after the setup of Floodlight controllers, which allows an adversary to violate the origin property. More seriously, we find that there is no mechanisms in current OpenFlow controllers to ensure the integrity of LLDP packets.

In our study, we also find some OpenFlow controllers, e.g., Floodlight and OpenDaylight, provide an API *suppressLinkDiscovery* to block LLDP propagation to particular ports connected to hosts. This kind of method is similar to the BPDU Guard security feature in legacy Ethernet switches, which prevents BPDU frames from sending to those ports enabled with the PortFast feature (i.e., manual configuration of switch ports connected to hosts). However, depending only on static port settings is not enough for diverse OpenFlow network environments, varying from a home network to an enterprise or cloud/data-center network and from stationary networks to mobile networks.

In order to deceive the LDS, an adversary can violate the "integrity/origin invariant" and "path invariant" of LLDP packets. In particular, the adversary originates falsified LLDP packets or simply relays LLDP packets between two switches to fabricate a non-existing internal link. At a first glance, it does not seem practical to inject arbitrary packets into the network from hosts or virtual machines because they are normally isolated by specific mechanisms, e.g., VLAN and Firewall. However, it appears feasible for hosts and virtual machines to inject or relay LLDP packets in OpenFlow networks. The OpenFlow networks allow LLDP packets to be sent outside all switch ports to dynamically track internal links between switches. Thus, the current design of OpenFlow controllers accepts LLDP packets from each switch port, even though it is connected to a host, which leaves a room for an adversary to inject fake internal links on compromised hosts or virtual machines. Next, we describe two methods an adversary can utilize to inject fake links into network topology.

Fake LLDP Injection. In this case, an adversary generates fake LLDP packets into an OpenFlow network to announce bogus internal links between two switches. By monitoring the traffic from OpenFlow switches, the adversary can obtain the genuine LLDP packet. Afterwards, he/she can violate the origin invariant of an LLDP packet, while OpenFlow controllers leverage specific syntax and extra TLVs for verification. Due to the open source nature of most OpenFlow controllers, the adversary can find out the reference LLDP syntax. Although the source code of OpenFlow controllers could be veiled and

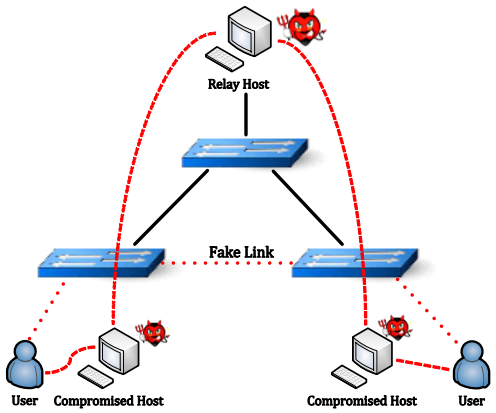


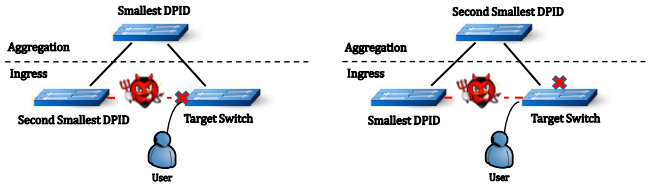
Fig. 5: Link Fabrication in an LLDP relay manner

a network administrator could customize the source code, it is also available to decode the LLDP packets by using differential tools. Moreover, as described above, the weak authenticator of LLDP packets imposed by several OpenFlow controllers can be bypassed. As long as the adversary acquires the genuine LLDP packet along with its syntax, he/she can modify the specific contents of the LLDP packet, e.g., the DPID field or the port number field, and launch the Link Fabrication Attack. In order to circumvent the possible anomalous traffic detection, the adversary could tune the LLDP injecting rate to the LLDP sending rate monitored from the OpenFlow controller.

LLDP Relay. Instead of injecting forged LLDP packets, a stronger adversary can also fabricate internal links in a relay fashion (without packet modification). That is, when receiving an LLDP packet from one target switch, the adversary repeats it to another target switch without any modification. In the case, the adversary constructs a fake topology view to the OpenFlow controller as if there is an internal link between those two target switches. This kind of fake link injection incurs future attack possibilities which we will describe more in detail as follows.

Here, we discuss two ways to build a communication channel to relay LLDP packets, i.e., by physical links and by a tunnel. An intuitive relay method is that an adversary sets up physical links (e.g., cable or wireless) between two switches. If this is not feasible, the adversary can use another more feasible approach, which relays LLDP packets by reusing the existing OpenFlow network infrastructure as illustrated in Figure 5. Particularly, the dotted line is the communication channel between two users in the view of an OpenFlow controller, whereas the dashed line is the actual traffic route. To successfully launch an LLDP relay attack, the adversary first needs to find suitable relay host(s), which can be achieved by a connection test. Another thing we need to note is that, some OpenFlow controllers, e.g., Floodlight and POX, disable the Host Tracking Service on internal link switch ports, which hinders the deployment of the LLDP relay channel. However, we cannot ignore the tunnel-based LLDP relay attack on those controllers in a hybrid network scenario (i.e., a network contains both OpenFlow islands and Non-OpenFlow islands), as the OpenFlow controller can hardly stop Host Tracking Service on the Multi-hop link ports (i.e., switch port outgoing to another Non-OpenFlow switch).

Next, we illustrate two attack possibilities on the top of Link Fabrication Attack, i.e., Denial of Service attack and man-in-the-middle attack.



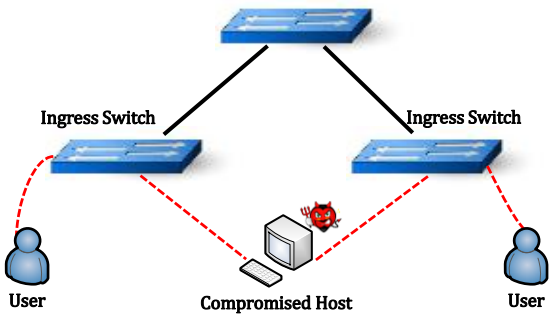
(a) The chosen switch has the second smallest DPID (b) The chosen switch has the smallest DPID

Fig. 6: Denial of Service Attack

Denial of Service Attack. To prevent a broadcast storm and save energy, OpenFlow controllers provide Spanning Tree Service (as detailed in Appendix A). When any topology update occurs, Spanning Tree Service is triggered to block those redundant ports. However, this capability can be leveraged by an adversary to launch a Denial of Service attack. In particular, by injecting a fake link into existing topology, the adversary can borrow the knife of Spanning Tree Service to "kill" normal switch ports.

One challenge to launch this type of attack is the non-deterministic characteristics of Spanning Tree Calculation after fake link injection. We note that the Spanning Tree Algorithm always excludes the link that connects largest DPID switches. Hence, we introduce an attack strategy tailored to a practical scenario, where an adversary possesses several compromised hosts connected to ingress switches. By listening to LLDP packets, the adversary can acquire the DPIDs of two ingress switches. Then, the adversary controls the compromised host which connects to the ingress switch with a lower DPID and injects a fake LLDP to announce a link with the target switch. As a result, there may be two consequences: if the DPID of the aggregation switch is smaller than that of our chosen switch, the adversary could shut down an arbitrary port of the target switch, as shown in Figure 6(a); otherwise, if the chosen switch has the smallest DPID, the link between the target switch and the aggregation switch is excluded from the spanning tree and also the corresponding ports are blocked, as shown in Figure 6(b).

We demonstrated a Denial of Service attack in our experimental environment. We chose POX as the OpenFlow controller, enabling routing module (l2_learning.py), link discovery module (discovery.py) and spanning tree module (spanning_tree.py). Note that the action for non-spanning-tree ports was configured as Port_Down. Then, we deployed a FatTree-like topology, where we controlled two hosts connecting to two



(a) Attack Topology

```
root@mininet-vm:~# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
08:58:40.733076 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
08:58:40.891255 ARP, Reply 10.0.0.3 is-at b2:af:fb:e9:a0:69 (oui Unknown), lengt
h 28
08:58:40.905558 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2757, seq 1, lengt
h 64
08:58:40.911964 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2757, seq 1, length
64
08:58:41.731296 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2757, seq 2, lengt
h 64
08:58:41.731520 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2757, seq 2, length
64
08:58:42.730103 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2757, seq 3, lengt
h 64
08:58:42.730156 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2757, seq 3, length
64
08:58:42.830172
08:58:43.731298 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2757, seq 4, lengt
h 64
08:58:43.731347 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2757, seq 4, length
64
```

(b) Attack Result

Fig. 7: Man-In-The-Middle Attack

sibling ingress switches. We ran Wireshark with the OpenFlow Dissector extension [5], which helps to parse OpenFlow messages, and dumped the *Packet-Out* messages with the payload of LLDP packets. We also ran an attacking script to craft fake LLDP packets based on the dumped genuine LLDP packets and injected them to the switch with the smaller DPID. As a result, we noticed that the users who are connected to our target switch port could not access the network resource any more.

Man-In-The-Middle Attack. Similarly, The fake link injection can also disturb the operation of Shortest Path Routing Service. An adversary can build an LLDP relay channel to deceive an OpenFlow controller with the illusion of an (actually non-existing) internal link between target switches. Once the OpenFlow controller notices a link up, it re-computes the Shortest Route based on contaminated topology information. As a result, all the traffic traversing through the fake route will fall into the trap of the adversary. Note that unlike legacy Ethernet switches, the OpenFlow switches do not change the source MAC address of packets. As such, in order to circumvent possible anomaly detection, we must keep the source MAC address when relaying both LLDP and normal packets.

Here, we launched a man-in-the-middle attack in our experimental environment. The OpenFlow controller we chose was Floodlight (master) with default settings. We deployed an linear network topology, as shown in Figure 7(a), where we had a compromised host connecting two ingress switches in the network. After the deployment, we ran an attacking script to relay traffic between two target switches (The dashed line in Figure 7(a) is the actual communication traffic route). As shown in Figure 7(b), we successfully wiretapped the traffic of clients connected to the target switches.

IV. Countermeasures

A. Static Defense Strategies

To defeat the proposed Network Topology Poisoning Attacks in SDN networks, we can have two major types of defense strategies: static or dynamic. The static solution is to manually configure/manage the host location and link information beforehand (e.g., assign a host identifier such as a MAC address to a specific switch port), and then manually verify and modify whenever there are changes (new addition or removal). However, this defense is obviously not attractive as the manual management is tedious, error-prone and not scalable in practice. In particular, it is not suitable for SDN networks, in which dynamics could be common and the scalability is important. Thus, in the following sections, we mainly focus on discussing dynamic defense strategies, as briefly summarized in Table IV. We will further introduce our proposed new defense system, *TopoGuard*, and evaluate its effectiveness and performance in the later section.

B. Dynamic Defense Strategies against Host Location Hijack

The problem of Host Location Hijacking lies in that OpenFlow controllers fail to verify the host identifier when the location of a host is updated. In order to tackle this issue, we discuss two possible mitigation methods which can secure HTS in OpenFlow controllers as well as dynamically track network mobility.

Authenticate Host Entity. A cryptographic solution to this problem is to authenticate a host by adding additional public-key infrastructure. In particular, when a host needs to change its location, it encodes the new location information into an unused field of packet (e.g., VLAN ID or ToS) with the encryption using its private key. This solution seems decent to prevent malicious host profile falsification, because it is not practical for an adversary to acquire the private key of the target host. However, there are several restrictions that make it hard to be feasible in practice. First, it begets additional storage overhead for keeping public keys in the OpenFlow controller side as well as computation overhead for handling each *Packet-In* message. The management of all keys of hosts and the dynamic addition/removal also bring a lot of overhead and cost. Moreover, this method requires to modify the implementation on every host, which is tedious and difficult in practical deployment.

Verify the Legitimacy of Host Migration. Another lightweight solution we propose is to verify conditions of a host migration. The idea is based on our two observations. First, the precondition of a host migration is that the OpenFlow

	Host Migration	Comparative Feasibility	Integirty/Origin Invariant of LLDP	Path Invariant of LLDP
Authentication	Yes	Low	Yes	No
Verification	Yes	High	No	Yes

TABLE IV: Defense Capabilities

controller must receive a Port_Down signal before the host migration finishes. Second, the postcondition of a host migration is that the host entity is unreachable in the previous location after the completion of the host migration. Consequently, based on this cause-and-effect relation, we can verify the legitimacy of the host migration by checking the precondition and postcondition. This method also adds performance overhead for *Packet-In* message processing, but compared to Host Entity Authentication, it is lighter and more feasible. In this paper, we adopt this verification approach to secure the host migration.

C. Dynamic Defense Strategies against Link Fabrication

As mentioned earlier, the root causes of the Link Fabrication attack can be summarized as: 1) The integrity/origin of an LLDP packet can be violated during the link discovery procedure in OpenFlow networks; 2) The compromised hosts can involve in the LLDP propagation path. To fix those security omissions, we propose two approaches that can secure the Link Discovery procedure without the burden of manual effort.

Authentication for LLDP packets. The first security omission exploited by an adversary is that the OpenFlow controller fails to verify the integrity of LLDP packets. Also, the adversary can defeat the verification of the origin in current OpenFlow controllers as long as he/she is able to receive LLDP packets from the connected switch. One solution to this problem is to add extra authenticator TLVs in the LLDP packet. Especially, we can add a controller-signed TLV into the LLDP packet and check the signature when receiving the LLDP packets. The signature TLV is calculated over the semantics of the the LLDP packet (i.e., DPID and Port number). In this case, the adversary can hardly manipulate the LLDP packets. However, this approach suffers from the fact that it fails to defend against the Link Fabrication attack in an LLDP relay/tunneling manner.

Verification for Switch Port Property. Another security invariant of the OpenFlow link discovery procedure is that no hosts can participate in the LLDP propagation. An idea to mitigate the relay-based Link Fabrication is to check if any host resides inside the LLDP propagation, e.g., we can add some extra logic to track the traffic coming from each switch port to decide which device is connected to the port. If OpenFlow controllers detect host-generated traffic (e.g., DNS) from a specific switch port, we set the *Device Type* of that port as HOST (details in Section V). Otherwise, we assign those switch ports as SWITCH when LLDP packets are received from those ports. In OpenFlow networks, those two categories are mutually exclusive because LLDP can only transmit on switch internal link ports and ports connected to the OpenFlow controller [4]. One assumption of this method lies in that the compromised machine is not an actual switch thus will generate regular host-generated traffic (e.g., ARP, DNS). This assumption is reasonable and it holds in most cases in practice. While a powerful adversary could theoretically disable all host-generated traffic in compromised hosts or virtual machines, it could also make the machine somewhat non-functional, at least for some normal networking activities/operations, and such non-functional anomaly could be easily noticed by the normal machine user, thus expose the existence of the adversary.

Finally, we note that in the case the adversary mutes all host-generated traffic, our aforementioned switch port property verification may not work. From the controller perspective, the attacking host can act just as a part of a cable, which is very difficult to discover by layer 2 or layer 3 security mechanisms. We could resort to verify some physical layer characteristics (e.g., [20]) to differentiate whether the attached device hardware is a switch or a machine, which is out of the scope of this paper.

V. *TopoGuard* PROTOTYPE SYSTEM

In this section, we detail the design and implementation of a new security extension for the OpenFlow controller, called *TopoGuard*, to protect the SDN networks from Network Topology Poisoning Attacks. *TopoGuard* is certainly not perfect. Our goal is to provide an automatic tool that (i) has a good balance between usability and security, and (ii) can be easily integrated into current mainstream OpenFlow controllers for immediate protection.

A. Overview

The basic idea of *TopoGuard* is to secure OpenFlow controllers by fixing security omissions as described in the previous section. In *TopoGuard*, we design Topology Update Checker to automatically validate the update of network topology, which is dependent on the information provided by Port Manager and Host Tracker.

Architecture. Figure 8 illustrates the architecture of our defense system. The Topology Update Checker verifies the legitimacy of a host migration, the integrity/origin of an LLDP packet and switch port property once detecting a topology update. Specifically, the Port Manager surveils OpenFlow messages to track dynamics of switch ports, which are stored in the Port Property. Afterwards, the Port Property is used to reason about the trustworthiness of a topology update. The Host Prober module is to test the liveness of the host in

[4] In this paper, we consider the control channel of OpenFlow networks could be properly under protection of SSL/TLS.

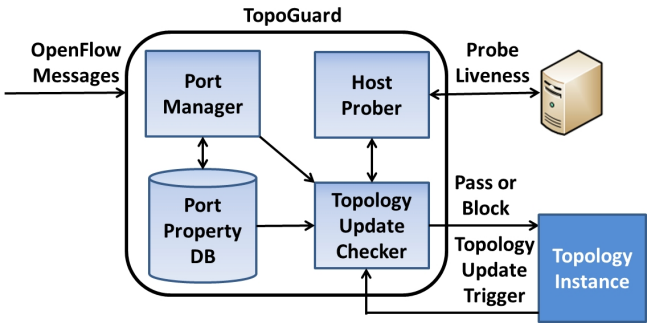


Fig. 8: The Architecture of *TopoGuard*

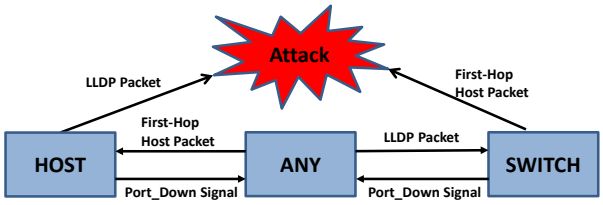


Fig. 9: The Transition Graph of *Device Type*

the specific location of the OpenFlow network, which also provides forensics to judge the host migration.

B. Design

Port Property Management. In order to reason about the validness of a topology update, we profile extra properties for each switch port in an OpenFlow controller. These properties include: *Device Type*, *Host List* and *SHUT_DOWN_FLAG*. The *Device Type* depicts which type of device a particular switch port connects to. The value could be ANY, SWITCH and HOST. As illustrated in Figure 9, The initial value of *Device Type* is ANY, which will turn to SWITCH or HOST based on following traffic. When Port Manager receives LLDP packets from a switch port with *Device Type* of ANY, it changes its type to SWITCH. Similarly, the *Device Type* of the switch port is set to HOST when Port Manager receives any first-hop host traffic, i.e., the host identity of the traffic is detected by the OpenFlow controller for the first time. In contrast, the HOST and SWITCH port are set back to ANY when receiving a Port_Down signal indicated in *Port-Status* messages. If Port Manager detects an LLDP packet from a HOST port or a first-hop host packet from a SWITCH port, it raises an attack alert and notifies Topology Update Checker to prevent the relevant topology update. The intuition behind this defense approach is that an LLDP packet is only designed to traverse through switch internal link ports in the data plane.

One challenge in port property management is how to decide the *Device Type* of a port as HOST. An intuitive solution in the OpenFlow networks is to monitor *Packet-In* messages from the switch port to detect host-generated traffic (e.g., ARP and DNS). After detecting host-generated traffic, we consider the port is connected to a host and change its *Device Type* as HOST. However, in our study, we find that different OpenFlow switches may issue multiple replicas of *Packet-In* messages for a specific host flow, i.e., the OpenFlow controller would receive host traffic from switch internal link ports. It could be due to the race condition scenario or specialized packet processing logic of OpenFlow controller applications. To solve this problem, we keep tracking the first-hop host traffic. Especially, we maintain *Host List* in the Port Property for each switch port, which contains host entities (in the form of a MAC address). When receiving *Packet-In* messages, the Port Manager locates the host entity in the existing *Host List* of Port Property. if the host entity is not found, the traffic is regarded as first-hop traffic and the source MAC address is recorded in the *Host List* of Port Property of the ingress switch port. Also, we need to handle network dynamics such as the *Set-Field* action in the OpenFlow flow rule, because any modification of the source MAC address during packet transmission can cause misclassification of first-hop traffic. For this, we also maintain a host-MAC alias map when we observe some flow rule modifying the source MAC address.

Another purpose of keeping *Host List* is to verify the trustworthiness of a host migration. As we mentioned in Section IV, the precondition for a host migration is that the OpenFlow controller receives a Port_Down signal before the host migration finishes. At this point, we set *SHUT_DOWN_FLAG* for hosts in the *Host List* of a switch port once detecting the port is down. The *SHUT_DOWN_FLAG* can be disabled when Port Manager receives correlated host traffic from the port. Furthermore, we can validate the *SHUT_DOWN_FLAG* inside *Host List* for the verification of the host migration.

Host Prober. As the counterpart to checking the precondition of a host migration, we can also leverage Host Prober to verify the postcondition, i.e., the host is unreachable in the previous location after the host migration completes. To achieve this, the Host Prober issues a host probing packet, e.g., ICMP Echo Request, to the former location of the host and waits for a response within a reasonable timeout. The Host Prober sends out a *Packet-Out* message with the payload of a crafted ICMP packet and outputs it to a specific switch port. In order to ensure the successful delivery of the response, the Host Prober also installs a flow rule to direct the ICMP response back to the OpenFlow controller. Also, to lower the overhead, in the current implementation, we set the response timeout as 1 second.

Topology Update Verification. The Topology Update Checker verifies the correctness of a topology update including a host migration and a new link discovery. When a host migration is detected, the checker references Port Property to check the precondition and instructs Host Prober to validate the postcondition. We note that the time overhead of checking the postcondition would be much higher than that of checking the precondition. In order to reduce the overall overhead, we can adopt a roll-back technique in Host Migration verification. That is, the Topology Update Checker updates a host location if the precondition is passed (*SHUT_DOWN_FLAG* for that

host is enabled in Port Property of the former location) without waiting for the result of Host Prober. However, if the response of Host Prober indicates a malicious host migration, the Topology Update Checker withdraws the previous update and raises an attack alert. In this case, the time overhead for verifying the host migration only counts on validating the precondition.

The Topology Update Checker also verifies the link discovery. The first task is to ensure the LLDP integrity/origin. For this sake, we place a signature TLV into an LLDP packet, which is a cryptographic hash value of a DPID and Port number. As soon as a new link is discovered, the Topology Update Checker conducts extra verification logic for the signed hash TLV. Then, the Topology Update Checker detects if the host lies on the path of the LLDP propagation. This task is achieved by checking the *Device Type* of switch ports of the new link. As a result, any internal link update involved in the HOST port will be denied and trigger an attack alert.

C. Implementation

We have developed a prototype implementation of *TopoGuard* on the master version of Floodlight. The Topology Update Checker, including Port Manager and Host Prober, works as a Floodlight service and is approximately 700 lines of Java code. The Topology Update Checker implements *IDeviceListener* and *ILinkDiscoveryListener* to monitor an update event of the topology instance inside Floodlight controller, while the Port Manager implements *IOFSwitchListener* and *IOFMessageListener* to initiate and maintain Port Property for each switch port.

To ensure the origin and integrity of an LLDP packet, we also use a keyed-hash message authentication code (HMAC) as an optional TLV for LLDP packets. In particular, we utilize javax.crypto package and select SHA-256 as the hash function along with controller's secret key. This adds about 130 lines of code in Java.

D. Evaluation

We evaluated a prototype implementation of *TopoGuard* to examine its effectiveness and performance.

1) Effectiveness: We first measured the effectiveness of our implementation against the Network Topology Poisoning Attacks discussed in Section III. Our experiment is conducted in the OpenFlow network environment including the Floodlight controller with *TopoGuard*. We launched aforementioned Network Topology Poisoning attacks in the environment and testified the reactions of the fortified Floodlight controller by observing the console output.

Detecting Host Location Hijacking. An adversary can spoof the identity of a target host to hijack its location information inside OpenFlow controllers. Note that we assume the target host is not compromised by the adversary. With *TopoGuard*, the falsified host migration can be detected due to dissatisfaction of the precondition and the postcondition.

Fig. 10: The Detection of Host Location Hijacking Attack

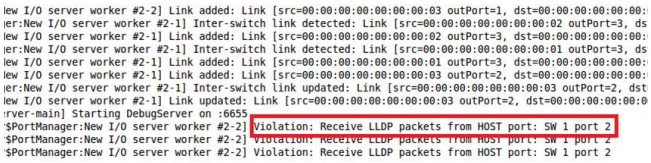


Fig. 11: The Detection of Link Fabrication Attack

That is, the Floodlight controller fails to receive a Port_Down message before receiving a host move event as shown in the first line in the red pane of Figure 10, and it succeeds in probing the target host in the previous location after receiving the host move event, as shown in the second line in the red pane of Figure 10.

Preventing Link Fabrication. An adversary can also falsify LLDP packets to fabricate non-existing links between switches. Under the radar of *TopoGuard*, the attempts to exploit the poor origin check and the omitted integrity assurance of the LLDP packets can be efficiently prevented. To ensure network dynamics, we do not manually block the LLDP packets on switch ports, i.e., the adversary is allowed to receive LLDP packets. However, once either a DPID or Port ID is manipulated by the adversary, the fortified LLDP handler can detect it and fire an alert. Note that we disable the port property verification while checking the integrity of LLDP packets because this LLDP falsification is also launched inside the data plane.

For another way of link fabrication, the adversary utilizes compromised hosts to relay LLDP packets between two target switches. When the compromised hosts start relaying LLDP packets, *TopoGuard* detects the violation of *Device Type* of particular ports, as shown in the red pane of Figure 11.

2) Performance: We further evaluated the performance of *TopoGuard* on Floodlight about the overhead over normal packet processing. In this experiment, we leverage Java System.nanoTime API to measure the running time of program snippets, which provides a precision of 1 nanosecond. Note that the measurement is conducted after all modules of the Floodlight controller are completely booted.

Impact on Performance The performance penalty imposed by *TopoGuard* mainly comes from the Link Discovery Module and the *Packet-In* message processing. Table V shows the average delay for *TopoGuard* added to the Floodlight controller on link discovery snippets, i.e., different functional blocks of Link Discovery Module. For the first round of

Link Discovery Snippet	Impact of *TopoGuard* (Percentage)	Controller Overall Cost
LLDP Construction(First time with computing HMAC)	0.431ms(80.4%)	0.536ms
LLDP Construction	0.005ms(2.92%)	0.171ms
LLDP Verification	0.005ms(1.64%)	0.304ms

TABLE V: HMAC Overhead on the Floodlight controller

the LLDP packets construction, the average overhead of *TopoGuard* is 0.431 ms, which accounts for 80.4% of overall LLDP construction time. However, we note that the following cost of *TopoGuard* imposed on the LLDP construction is much lower, which is about 0.005 ms and only accounts for 2.92% of overall LLDP construction time. The significant discrepancies stem from our implementation strategy because *TopoGuard* computes the HMAC value once and cache the computation result for the future construction and verification of LLDP packets. The strategy also lowers the impact of *TopoGuard* on the verification phase of LLDP packets, which is only about 0.005 ms. On the other hand, the Port Manager incurs a delay over the normal packets processing because it sits in the earlier stage in the OpenFlow message processing pipeline of the Floodlight controller than the Shortest Path Routing Service and the Link Discovery Service. Accordingly, we also measure the time that the Port Manager spends on handling LLDP packets and host-generated packets. The result shows the average delay is 0.02 ms for the LLDP packets processing in the Link Discovery Service and 0.032 ms for the normal packets processing in the Shortest Path Routing Service. From the above result, we conclude that the impact of *TopoGuard* is negligible on the normal operation of Link Discovery Service of the Floodlight controller.

VI. Discussion

Topology management has never been included in the OpenFlow Specification [4]. In other words, it is vendor-implementation-dependent although there is already some discussion about this issue in the community. However, most of existing OpenFlow controllers and switch vendors follow a certain convention for handling the topology management. Since the first reference OpenFlow controller implementation (i.e., NOX [13]), almost all the implementations implicitly follow its design, including OpenFlow Discovery Protocol (OFDP) and Host Tracking Service. This may be a root cause for explaining that all the controllers expose the similar vulnerabilities. We hope that our work can draw attention from the SDN community and more security considerations will be put into the further specification.

Our attacks mainly focus on the data plane communication channel, i.e., an adversary can launch Link Fabrication Attack or Host Location Hijacking Attack on the top of compromised hosts. In fact, the security of OpenFlow control plane is also a security concern. As discussed in [9], most OpenFlow controllers and switch vendors lack full implementation of SSL/TLS. Seizing this security deficiency, an adversary can also launch man-in-the-middle attacks to manipulate control traffic between the controller and switches. We think that the message authentication can be extended to all OpenFlow messages to mitigate potential message falsification.

Finally, the fact that the OpenFlow controller handles all the layer 2 protocols on behalf of switches in OpenFlow networks leaves a room for other potential vulnerabilities from which the traditional network switches do not suffer. For instance, a new kind of Denial of Service attack [31], targeting at the *Packet-In* message handler, may saturate the control channel of OpenFlow as well as overload OpenFlow controllers. In order to systematically investigate the potential security issues, designing a new security fuzzer for SDN (controllers) may help us find more vulnerabilities, which is our future work.

VII. Related Work

In this section, we investigate security research in the SDN domain and related poisoning attacks in legacy networks.

Security Research for SDN networks. Several verification approaches are often used to debug and check network invariants. VeriFlow [19] presents a layer between the control plane and the data plane that monitors network state updates and verifies the violations of invariants dynamically at real time. NetPlumber [18] introduces a realtime network-wide policy checking tool using Header Space Analysis (HSA). NICE [22] uses model checking and symbolic execution to find network software bugs in OpenFlow applications. SOFT [21] introduces an approach for testing the interoperability of OpenFlow switches with reference implementations. [14] designs and presents the first machine-verified SDN controller based on NetCore [25]. [11] introduces a verification tool that takes the software program of a data plane as input and check target properties. These verification solutions only verify the logic correctness of the control plane and data plane, however fail to locate the network topology exploitations discussed in this paper. One insight behind Network Topology Poisoning Attacks stems from the centralized network visibility that OpenFlow Controller offers to lessen onerous network management tasks. Unfortunately, our study in this paper shows that this function could be exploited if not carefully designed, thereby incurring serious security threats.

To date, the security issues of SDN have been being widely discussed in both academia and industry. FortNox [27] introduces a SDN tunneling attack and presents two mechanisms, role-based authorization and security constraint enforcement, to solve corresponding security challenges. OF-RHM [16]

proposes OpenFlow Random Host Mutation to dynamically mutates IP addresses of hosts inside an LAN network. FRESCO [30] introduces an OpenFlow security application development framework which provides modular composable security services for application developers. SDN Scanner [29] and Avant-Guard [31] show a new attack (which is called data-to-control plane saturation attack) against SDN networks and provide solutions to prevent such attacks. Different from the previous work, this paper is the first one to study the network topology visibility exploitation in the SDN network. Concurrently, SPHINX [23] proposes a unified approach to use network flow graphs to detect attacks that violate those learned flow graphs/modules. Different from their work, this paper deeply investigates the vulnerabilities causing Network Topology Poisoning Attacks, as well as a low-overhead real-time defensive solution.

Related Poisoning Attacks in Legacy Networks. One notorious counterpart to the Host Location Hijacking Attack is the ARP Cache Poisoning attack in Ethernet networks. That is, an adversary sends forged ARP messages to associate the IP address of the target host with the MAC address of a malicious host. By doing so, the adversary can hijack the entity of the target host, which is normally a gateway. However, the ARP Cache Poisoning attack has several differences from the Host Location Hijacking Attack as shown in Table VI. First, the attacking scope of the ARP Cache Poisoning is limited to a broadcast domain, i.e., the adversary must stay within the same broadcast domain with its target. By contrast, the adversary can launch the Host Location Hijacking Attack at any location of an OpenFlow network. Second, in addition to ARP reply packets, the Host Location Hijacking Attack can leverage almost all kinds of packets, e.g., ICMP echo, UDP and TCP, to usurp the location of the target host. In this point, the Host Location Hijacking Attack can be concealed in normal traffic to sidestep NIDS (Network Intrusion Detection Systems). Also from the defense perspective, the traditional mitigation strategies for ARP Cache Poisoning, such as the static ARP entry, may not be appropriate to apply directly to the SDN network since its static configuration undermines the dynamics handling capability of the OpenFlow network, e.g., tracking host migration between various OpenFlow access points [3]. To defend against the Host Location Hijacking Attack along with tracking network dynamics, in this paper, we leverage OpenFlow specific capabilities to dynamically verify the host migration.

As illustrated in Table VII, an attack in legacy networks similar to the spirit of the Link Fabrication Attack is the STP Mangling (a.k.a, BPDU Falsification) attack [26], i.e., an adversary falsifies BPDUs with the smallest bridge ID to preempt the root of Spanning Tree. After faking the root, the adversary has potential to elaborate a Denial of Service or man-in-the-middle attack. However, the STP Mangling attack only disrupts the running of STP rather than injecting a fake link into network topology to poison the entire network operation. Also, some prior work about the exploitation of the view of network topology focus on only link-state routing protocols. Jones et al. [17] outline vulnerabilities of the design of OSPF and discuss the possible exploitations. Nakibly et al. [12] introduce two attacks to persistently falsify the topology of an OSPF network, which also incurs denial of service, eavesdropping and man-in-the-middle attacks. Such attacks are launched by compromising the router entity or obtaining the pre-shared keys for the authentication of router. However, the Link Fabrication Attack in this paper can be launched from the hosts residing in the data plane. Apart from a wired network, The link-state routing protocols in Mobile Ad Hoc networks, e.g., Optimized Link State Routing Protocol (OLSR), also incur similar security challenges. As mentioned in [6], an adversary can falsify links into OLSR topology by generating TC (or HNA) messages. Similar to OSPF Link Fabrication, OLSR Link Fabrication requires compromised routing entities, which is not required in our attacks. Another attack avenue in OLSR is the Wormhole attack [10], [32], which artificially creates wormholes in OLSR networks by recording traffic in one location and replaying it in another location. The OLSR Wormhole attack is only launched in a relay/replay manner. In contrast, our Link Injection attack can also be launched by falsifying the LLDP packets. In all, Table VII summarizes the differences of those attack from the Link Fabrication Attack proposed in this paper.

Note that aforementioned work motivates our study, but the problem domain in this paper is totally different. That is, we focus on the SDN-specific security issues which stem from the different operations of SDN networks and legacy networks, as well as from the security omissions in the design and implementation of current OpenFlow Controllers.

VIII. Conclusion

In this paper, we propose new SDN-specific attack vectors, Host Location Hijacking Attack and Link Fabrication Attack, which seriously challenge the core advantage of SDN, i.e., network-wide visibility. We demonstrate that the attacks can effectively poison the network topology information, thereby misleading the controller's core services and applications. We also systematically investigate the solution space and then present *TopoGuard*, a new security extension to the OpenFlow controllers, which provides automatic and real-time detection of Network Topology Poisoning Attacks. Finally, our prototype implementation shows a simple yet effective and efficient defense against the Network Topology Poisoning Attacks. With the publication of this paper, we also plan to release our prototype tool to help fix these vulnerabilities in widely used OpenFlow controllers. We hope that this work will attract more attention to SDN security research and contribute to the standardization of the SDN specification with security considerations.

Acknowledgements

This material is based upon work supported in part by the Air Force Office of Scientific Research under FA-9550-13-1-0077 and a Google Faculty Research award. Any opinions,

Attack Requirement	OpenFlow Host Location Hijacking	ARP Cache Poisoning
Attacker Location Restriction	Anywhere within the OpenFlow domain with the target	Stay within the same broadcast domain with the target
Target Visibility	MAC Address and IP Address	Only IP Address
Attack Avenue	**OpenFlow Host Location Hijacking**	**ARP Cache Poisoning**
Falsified Packet Type	Almost every kind of packets	Only ARP packet
Attack Result	**OpenFlow Host Location Hijacking**	**ARP Cache Poisoning**
Hijack the Target Location	Yes	Yes

TABLE VI: Comparison between Host Location Hijacking and ARP Cache Poisoning

Attack Requirement	OFDP Link Fabrication	STP Mangling	OSPF Link Fabrication	OLSR Wormhole
Attack relies on Compromised Routers/Switches	No	No	Yes	No
Need to defeat Neighbor Discovery/Authentication	No	No	Yes	No
Attack Avenue	**OFDP Link Fabrication**	**STP Mangling**	**OSPF Link Fabrication**	**OLSR Wormhole**
Falsify Control Message	Yes	Yes	Yes	No
Relay Control Message	Yes	No	Yes	Yes
Attack Result	**OFDP Link Fabrication**	**STP Mangling**	**OSPF Link Fabrication**	**OLSR Wormhole**
Injecting False Link into Network Topology	Yes	No	Yes	Yes
Affected Service	All Topology-Based Services	STP	Routing	Routing

TABLE VII: Comparison between Link Fabrication Attack and Previous Counterparts

findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFOSR and Google.

REFERENCES

[1] Apache http server project. https://httpd.apache.org/.

[2] Configuring port security. http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst4500/12-2/25ew/configuration/guide/conf/port_sec.html.

[3] n-casting using openflow. http://archive.openflow.org/wp/n-casting-mobility-using-openflow/.

[4] OpenFlow Specification v1.4.0. http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[5] Openflow wireshark dissector. http://archive.openflow.org/wk/index.php/OpenFlow_Wireshark_Dissector.

[6] Optimized Link State Routing Protocol (OLSR). http://www.ietf.org/rfc/rfc3626.txt.

[7] Proxy ARP. http://www.cisco.com/c/en/us/support/docs/ip/dynamic-address-allocation-resolution/13718-5.html#howdoesproxyarpwork.

[8] Scapy: Packet manipulation program. http://www.secdev.org/projects/scapy/.

[9] K. Benton, L. J. Camp, and C. Small. Openflow vulnerability assessment. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013.

[10] D. Raffo C. Adjih and P. Mhlethaler. Attacks against olsr: Distributed key management for security. In *2005 OLSR Interop and Workshop*, July 2005.

[11] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[12] D. Gonikman G. Nakibly, A. Kirshon and D. Boneh. Persistent ospf attacks. In *In proceedings of the 19th Annual Network & Distributed System Security Conference (NDSS'12)*, 2012.

[13] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. July 2008.

[14] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controller. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[15] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[16] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: Transparent moving target defense using software defined networking. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*.

[17] E. Jones and O. L. Moigne. Ospf security vulnerabilities analysis. In *Internet-Draft draft-ietf-rpsec-ospf-vuln-02, IETF, June 2006*.

[18] P. Kazemian, M. Chang, H. Zeng, S. Whyte, G. Varghese, and N. McKeown. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[19] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[20] L.B. Kish. Protection against the man-in-the-middle-attack for the kirchhoff-loop-johnson(-like)-noise cipher and expansion by voltage-based security. In *Fluctuation and Noise Letters 6 (2006) L57L63*.

[21] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A soft way for openflow switch interoperability testing. In *Proceedings of ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[22] P. Peresini D. Kostic M. Canini, D. Venzano and Jennifer Rexford. A nice way to test openflow applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[23] K. Mahajan M. Dhawan, R. Poddar and V. Mann. Cloudnaas: a cloud networking platform for enterprise applications. In *In proceedings of the 22th Annual Network & Distributed System Security Conference (NDSS'15)*, 2015.

[24] Mininet. Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/.

[25] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proceedings*

of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2012.

[26] A. Ornaghi and M. Valleri. Man In The Middle Attacks. http://www.blackhat.com/presentations/bh-europe-03/bh-europe-03-valleri.pdf.

[27] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, August 2012.

[28] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[29] S. Shin and G. Gu. Attacking software-defined networks: A first feasibility study (short paper). In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013.

[30] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[31] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[32] A. Perrig Y. Hu and D. B. Johnson. Wormhole attacks in wireless networks. In *2005 OLSR Interop and Workshop*, July 2005.

APPENDIX

A. Example Topology-dependent Services in SDN

In this section, we describe two representative topology-dependent services (among many others), i.e. shortest-path routing and spanning tree, to illustrate how topology management services contribute to SDN network operations.

Shortest-Path Routing Service. Packet routing is the cornerstone for network operation. The topology service is crucial for packet forwarding in all OpenFlow instantiations. To explain the packet processing procedure in OpenFlow controllers, we take the example of the routing service in Floodlight. We assume Alice wants to visit a web site "ABC.com". After a successful DNS resolution, Alice receives the corresponding IP address and then sends an HTTP request to "ABC.com". Since the nearest switch is unable to find a flow rule to forward this packet, the OpenFlow switch reports it to the OpenFlow controller as a *Packet-In* message. In order to process this communication, the OpenFlow controller resolves the destination location by referring to the Host Profile stemmed from Host Tracking Service. After resolving location information of the destination, the OpenFlow controller runs a shortest path algorithm (e.g., Dijkstra's algorithm) on the topology information derived from the Link Discovery Service to compute a route (in its networks) from the source to the destination. If the OpenFlow controller successfully derives a route, it pushes a route update to involved OpenFlow Switch(es) for future communication between Alice and "ABC.com."

Spanning Tree Protocol Service Apart from shortest path routing, loop-free is another important concern for network management. For this purpose, Spanning Tree Protocol (STP for short) is used to disable redundant ports and links as a layer 2 extension. Also, it provides an avenue to save energy in data center network [15]. Instead of distributed STP computation (by BPDUs) in traditional networks, OpenFlow controller has the capability to solve the Spanning Tree in a fast-convergence manner. As described in Algorithm 1, the spanning tree calculation in OpenFlow controllers is built upon the topology information.

After calculating spanning tree, OpenFlow controllers leverage *Port-Mod* messages to manage the switch port status. In particular, if particular OpenFlow switch ports stay outside of Spanning Tree, the controller can send out *Port-Mod* message with the ofp_port_config of OFPPC_PORT_DOWN to turn off those ports.

Algorithm 1 Topology-based STP Calculation in SDN

Input: $TOPO$:The topology of current network
Output: ST:The spanning tree of TOPO

1: $ST.switches \leftarrow \emptyset$, $ST.links \leftarrow \emptyset$, $candidateSwitches \leftarrow \emptyset$, $solvedSwitches \leftarrow \emptyset$
2: $root$ = POP(sorted($TOPO.switches$)
3: **while** TRUE **do**
4: **for** each switch $S \in TOPO.switches$ **do**
5: **if** $S \notin solvedSwithes$ **and** $(root, S) \in TOPO.links$ **then**
6: $candidateSwitches = candidateSwitches \bigcup S$
7: $ST.switches = ST.switches \bigcup S$
8: $ST.links = ST.links \bigcup (root, S)$
9: **end if**
10: **if** $candidateSwitches == \emptyset$ **then**
11: break
12: **end if**
13: **end for**
14: $solvedSwitches = solvedSwitches \bigcup root$
15: $root$ = POP(sorted($candidateSwitches$))
16: **end while**
17: **return** ST