Manipulating OpenFlow Link Discovery Packet Forwarding for Topology Poisoning

Mingming Chen
mzc796@psu.edu
The Pennsylvania State University
University Park, PA, USA

Thomas La Porta
tfl12@psu.edu
The Pennsylvania State University
University Park, PA, USA

Teryl Taylor
terylt@ibm.com
IBM Research
Yorktown Heights, NY, USA

Frederico Araujo
frederico.araujo@ibm.com
IBM Research
Yorktown Heights, NY, USA

Trent Jaeger
trentj@ucr.edu
University of California, Riverside
Riverside, CA, USA

Abstract

Software-defined networking (SDN) is a centralized, dynamic, and programmable network management technology that enables flexible traffic control and scalability. SDN facilitates network administration through a centralized view of the underlying physical topology; tampering with this topology view can result in catastrophic damage to network management and security. To underscore this issue, we introduce Marionette, a new topology poisoning technique that manipulates OpenFlow link discovery packet forwarding to alter topology information. Our approach exposes an overlooked yet widespread attack vector, distinguishing itself from traditional link fabrication attacks that tamper, spoof, or relay discovery packets at the data plane. Unlike localized attacks observed in existing methods, our technique introduces a globalized topology poisoning attack that leverages control privileges. Marionette implements a reinforcement learning algorithm to compute a poisoned topology target, and injects flow entries to achieve a long-lived stealthy attack. Our evaluation shows that Marionette successfully attacks five open-source controllers and nine OpenFlow-based discovery protocols. Marionette overcomes the state-of-the-art topology poisoning defenses, showcasing a new class of topology poisoning that initiates on the control plane. This security vulnerability was ethically disclosed to OpenDaylight, and CVE-2024-37018 has been assigned.

CCS Concepts

• **Security and privacy** → Distributed systems security; **Security protocols**; • **Networks** → **Link-layer protocols**.

Keywords

SDN; LLDP; Precise Link Manipulation; Reinforcement Learning

1 Introduction

Software-defined networking (SDN) is a flexible network architecture that offers centralized control and management of a network, in contrast to traditional data networks. SDN enables a controller to manage the network centrally through one of several protocols (e.g., OpenFlow [63]) and decouples control from the switches. A major advantage of SDN is that it provides a real-time view of the entire network, which allows for flexible traffic management. This serves as the foundation for many services, including cloud computing, traffic engineering, and network monitoring.

Despite its many advantages, SDN's centralized architecture suffers from inherent limitations related to scalability and fault-tolerance [4, 73]. To solve these issues, multi-controller architectures were introduced [30, 73] to reduce communication latency between controllers and switches and balance the load across multiple controllers. Furthermore, fault-tolerant controller clustering enables rapid recovery of the control plane to mitigate the impact of a single point of failure or attacks on a single controller. However, multi-controller architectures also introduce attack vectors [2, 38, 43, 58] spanning the *east-westbound* interface (between controllers) and *north-southbound* interface (between controllers and applications/switches).

Specifically, while topology poisoning attacks originating from the data plane have been well-studied [8, 29, 49], the control plane poses challenges for creating sophisticated, long-lasting topology poisoning attacks. For example, a naïve approach, in which a malicious controller shares incorrect information with peer controllers to poison the topology, is not effective [35] because leader controllers periodically (e.g., every 100 milliseconds [68]) re-discover the topology directly from the network, independently [11, 23, 68]. Moreover, the assignment of the leader role is dynamic [30, 62, 73], rendering attacks that rely on a static leader ineffective. Similarly, attacks that *directly* manipulate traffic forwarding through the injection of malicious flow entries are easily detectable [39, 44, 56, 59].

Mingming Chen, Thomas La Porta, Teryl Taylor, Frederico Araujo, and Trent Jaeger

A critical and previously overlooked vulnerability in modern SDN link discovery occurs when the flow entries designed for traffic routing are used to impact link discovery. The OpenFlow Discovery Protocol (OFDP), the de-facto SDN discovery protocol, enables controllers to perform link discovery between any pair of neighboring switches; however, maliciously constructed (i.e., poisonous[1]) flow entries that manipulate the Link Layer Discovery Protocol (LLDP)[2] packet forwarding may cause controllers to discover incorrect links and miss real ones. Further, we find that both the OpenDaylight and ONOS controller clusters[3] allow controllers in any role in a multi-controller architecture to introduce these poisonous flow entries due to an insecure multi-controller implementation. Malicious applications on an SDN controller can also exploit this vulnerability.

To underscore the severity of this security vulnerability, we introduce MARIONETTE, a *persistent*, *stealthy*, *precise*, and *globalized* topology poisoning attack that uses poisonous OpenFlow [63] flow entries to induce controllers to independently discover a poisoned topology using standard discovery protocols. In contrast to techniques that ephemerally mislead controllers with false information, our approach steers benign controllers to independently discover a false, poisoned topology and accept it indefinitely, without flow interruption. Modern topology discovery protocols infer links between switches based on the start and end points of the path traversed by an LLDP packet. MARIONETTE manipulates the traversal's endpoint to fabricate links, circumventing current state-of-the-art detection mechanisms. This enables MARIONETTE to manipulate any discoverable links in the network irrespective of their location. As a result, it can initiate a global attack by computing a target poisoned topology based on the real topology.

MARIONETTE works with standard OpenFlow protocols, attacking OpenFlow-based SDN topology discovery protocols that flood or broadcast discovery packets to discover links (§7.1). The attack first learns the current network topology and forwarding policy. It then uses reinforcement learning (RL) [57] to learn a deceptive topology that will achieve an attack goal (e.g., evading a monitor, attracting traffic to an eavesdropping switch). When building the RL model, constraints are imposed to evade existing defenses [23] while limiting the differences between the poisoned and real topologies (§6.1). Once a target topology is determined, MARIONETTE automatically derives the required poisonous flow entries and sends them to the appropriate switches concealed as normal flow entries (§5.1, §5.2). Finally, while gaps in routes—introduced by differences between the real and fabricated topologies—can be fixed through reactive forwarding [63], we use a stealthier approach to patch these differences via additional flow entries that minimize suspicious `packet-in` packets sent to the benign controller (§5.3). Performing these three functions concurrently in a real network is not trivial. The end-to-end attack strategy is discussed in §6.

MARIONETTE is robust against encryption-based defenses [3, 7, 11, 29] because it does not rely on packet spoofing or fabrication. Software rootkit defenses [56] are also ineffective since the attack only relies on normal controller functionality. Moreover, the

---

[1] We use "poisonous" as the adjective of flow entries that poison the topology. Consequently, such topology is called a "poisoned topology".
[2] A link layer protocol used by IEEE 802 network devices to learn reachability and connection endpoint information from adjacent devices [22].
[3] The only two Open-Source controller projects with clustering implementation.

Table 1: Control Plane Vulnerabilities

Vulnerabilities	Security issues	CVEs
Malicious Controller	[35],[44],[54],[74],[3],[41]
Controller Impersonation	[43],[38],[26][64], [20]
Unauthenticated Access	[9],[18],[43]
Controller Vulnerabilities	CVE-2018-1000614, CVE-2023-30093, CVE-2017-1000081, CVE-2018-1132, CVE-2017-1000406
Malicious Application	[6],[28],[66],[74],[67],[56],[43],[16]
Northbound API abuse	[31],[65]

poisonous flow entries generated by MARIONETTE escape flow rule examinations [36, 53] and network policy checking [34] (§7.5). MARIONETTE overcomes scalability issues by computing its deceptive topologies offline and performing infrequent topology changes.

To validate the practicality of our approach, we implemented MARIONETTE attacks against nine OpenFlow-based discovery protocols and five SDN controllers, including attacks against OpenDaylight and ONOS *clusters*. We also used MARIONETTE's RL algorithm to compute deceptive topologies for two network topologies, *fat tree* [40], which is widely used in enterprise networks, and *Chinanet* [50], a backbone topology. Our experiments show that MARIONETTE generates a poisoned topology that is 92% similar to the original topology while attracting more than 60% additional flows to the eavesdropping point on a 36-node fat tree topology. We analyzed current state-of-the-art defenses against MARIONETTE, revealing that all are ineffective against our attack. We have ethically disclosed our attack details to the OpenDayLight community, which has acknowledged the vulnerability.

Our contributions can be summarized as follows:

- We introduce a novel method for topology poisoning initiated from the control plane which generates poisonous flow entries that manipulate link discovery packets' forwarding to precisely fabricate links toward a specific attack goal.
- We design a reinforcement learning algorithm that automatically computes network topologies that satisfy attack objectives (e.g., flow routing through an attacker-controlled node) and constraints (e.g., the deceptive topology has a certain graph similarity with the real topology).
- We successfully deployed MARIONETTE attacks on both ONOS and OpenDaylight clusters, and five open-source controllers, while systematically evading existing defenses. We also show that MARIONETTE can attack nine different SDN discovery protocols, and evaluate our RL model on two real topologies.

2 Overview

SDN provides dynamic, flexible, and programmable traffic management by decoupling the control plane from the data plane. Unlike legacy networks running a distributed routing algorithm on each switch, SDN employs a centralized controller to gather network information and orchestrate actions across switches. To enhance fault tolerance and scalability of the single-controller architecture, a multi-controller architecture was developed [30, 73]. However, the increased complexity and multiple interfaces expose many types of vulnerabilities, especially on the control plane, as listed in Table 1.

A malicious controller in a cluster or malicious application in the SDN platform can jeopardize the network by proactively injecting
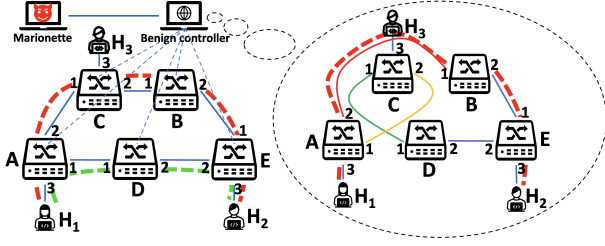
Figure 1: Motivating Example

malicious flow entries. As a result, defenses have been developed to detect malicious flow entries that cause routing attacks [6, 36, 53]; however, a heretofore overlooked attack inserts poisonous flow entries that *manipulate link discovery packet forwarding* to poison the topology view of benign controllers, evading existing defenses.

Motivating Example. Figure 1 exemplifies this attack. The left-hand side shows the *real* topology, while the right-hand side (i.e., the dotted circle) is the *deceptive topology* as it *appears* to the benign controller. Before the attack, the flow $f_1 : (src : H_1, dst : H_2)$ was routed through the shortest path of green dotted path $H_1 \rightarrow A \rightarrow D \rightarrow E \rightarrow H_2$ on the real topology. The deceptive link of $A \rightarrow B$ on the deceptive topology in Figure 1 can be fabricated by inserting a flow entry at switch C that instructs $LLDP_{A2}$ to be forwarded to Port 2 instead of feeding it back to the controller. Consequently, the benign controller thinks the shortest path of f_1 is the red dotted path $H_1 \rightarrow A \rightarrow B \rightarrow E \rightarrow H_2$ based on the deceptive topology. The absence of C on the fabricated link of $A \rightarrow B$ allows C to eavesdrop on f_1 without being noticed by the benign controller (red dotted path on the real topology). The details of precise link manipulation are in §5. Note that the deceptive topology in this work always has the same degree sequence as the real topology to remain stealthy. We show an implementation of this attack against the ONOS controller in §7.2 and §7.3.

2.1 OpenFlow and OpenFlow Link Discovery

OpenFlow [63] is a standard SDN protocol that allows a controller to gather information and instruct switches on how to route incoming packets by sending them flow entries[4]. When a packet arrives at a switch, the switch checks its flow table to determine whether there is a matching flow entry to determine where to forward the packet. If yes, the switch forwards the packet accordingly. Otherwise, it encapsulates the packet in a `packet-in` message and sends it to the controller due to a table-miss flow entry. A table-miss flow entry is the default flow entry instructing the unmatched packet to be sent to the controller. Once the controller determines the route, it sends the packet back to the switch encapsulated in a `packet-out` message and installs necessary flow entry[5] in the switch using `flow-mod` messages. The controller repeats this process until the packet arrives at its destination. This process is called *reactive forwarding*. The controller can also configure flow entries beforehand spontaneously, which is called *proactive forwarding*.

---

[4]The flow entry has a *match* field and a set of *actions* to apply to the matched packets. Entry types for matching include source/destination MAC and IP addresses, VLAN ID, input port, etc. Instructions associated with each entry may contain actions instructing packet forwarding, packet modification, group table processing, etc.

[5]When the controller fails to determine an output at a switch (because the destination host is unknown or the switch is not on the computed route), the controller sends the packet back to the switch and instructs it to flood this packet through all its ports.
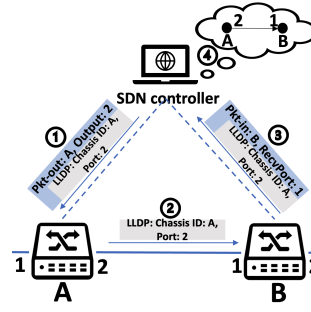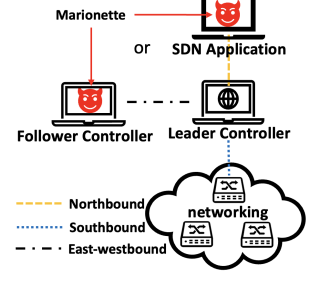

Figure 2: OFDP illustration

Figure 3: Threat Model

To maintain correct centralized control of modern, dynamically changing networks, controllers (re-)discover the network topology frequently (e.g., commonly every 100 ms), via the OpenFlow Discovery Protocol (OFDP). In Figure 2, we illustrate a single instance of OFDP discovering a unidirectional link from A to B. The OFDP process is initiated in step ① when the controller sends a `packet-out` message encapsulating an LLDP packet, which we denoted as $LLDP_{A2}$, to switch A and instructs A to forward the LLDP to port 2. In step ②, switch A receives the `packet-out` message, decapsulates the OpenFlow header, and sends the $LLDP_{A2}$ packet to port 2; switch B receives the $LLDP_{A2}$ packet sent from (A, Port: 2) (③). The LLDP packet matches the pre-configured table-miss /LLDP flow entry[6] in switch B which instructs the switch to send it to the controller using a `packet-in` message. Finally, in step ④, the controller receives the `packet-in` message. It learns that the $LLDP_{A2}$ packet was sent from (A, Port:2) by checking the $LLDP_{A2}$ payload, and received by (B, Port:1) by checking the `packet-in` payload. The controller writes a unidirectional link (A, Port:2) $\rightarrow$ (B, Port:1) to its data store. Hereafter, we use the notation $A2 \rightarrow 1B$ for simplicity.

2.2 MARIONETTE

MARIONETTE is a new topology poisoning attack that uses poisonous OpenFlow [63] flow entries to manipulate link discovery packet forwarding to induce benign controllers to discover a poisoned topology independently. Our key insight is that SDN discovery protocols such as the OpenFlow Discovery Protocol (OFDP) rely on a controller flooding the network with discovery packets to infer the topology. These packets are fed back to the controller from switches by either a table-miss or an LLDP flow entry. The controller knows the start and end points of a discovery packet's traversal; however, it cannot discern any intermediate points along the feedback loop, because that information is not stored in the LLDP packets. This creates an opportunity for a rogue controller to stealthily manipulate the path taken by the discovery packets which in turn manipulates the links that are discovered by the controller.

The attack requires two types of flow entries to be installed in a switch. The first type, which we call *poisonous flow entries*, is used to misdirect topology discovery packets so that legitimate controllers learn the poisoned topology independently. The second type, which we call *gap-patching flow entries*, is used to repair the updated paths (based on the poisoned topology) that are disjoint in

---

[6]match: "ether-type:0x88cc", where "0x88cc" is LLDP packet type

Mingming Chen, Thomas La Porta, Teryl Taylor, Frederico Araujo, and Trent Jaeger

Table 2: Related Threats and State-of-the-art Defences

Attack Surface	Attack Methodology	Impact	Defense		
Malicious Host(s)/ Switch	**(A.1) Link Fabrication:** Fabricate a deceptive link A1 -> B2 on the controller's view. Methods: 1. Relay LLDP packet from A1(switch A port 1) to B2 by: (1) malicious hosts [29, 60] (2) malicious switch [8] ("Match: ether-type: 0x88cc, inport:1 Action: 2" [5]) 2. Fabricate LLDP packet with "src: A1" and send it to B2 by: (3) malicious hosts [29, 60] (4) malicious switch [12]	Localized Persistent	**(D.1)** SPV: With benign controller assumption, it sends a probe to go through the target link. If the link is fabricated, the probe forwarding through A1 will be dropped or fed back to the controller by a switch port other than B2 [8].		
			(D.2) Latency-based Detection: A link is considered fabricated if its latency is greater than a threshold value [5, 27, 32, 61].		
			(D.3) Sphinx: With benign controller assumption, it builds the flow graph incrementally from Flow-mod OF-message, comparing it with actual flow routings using switch metadata. If a discrepancy is found, fabricating links are on the differing points [23].		
			(D.4) Port-based Detection: If an active port connects more than one link, either switch-host link or switch-switch link, one of the switch-switch links is fabricated [12, 23].		
			(D.5) TopoGuard(+): If an active port connects with both host and switch, the link connects switch is fabricated; Authenticating LLDP packets to prevent LLDP packet fabrication [29, 60].		
Malicious Application/ Controller in Cluster	**(A.2) Wrong Information Sharing:** Malicious controller shares wrong (topology) information in the controller cluster to cause topology poisoning [35, 43, 56].	Globalized Ephemeral	**(D.6)** Dynamic Cluster: Leader controller is dynamically assigned [30, 62, 73] and it periodically (e.g., every 100 milliseconds [68]) re-discovers the topology from the network independently [11, 23, 68].	**(D.7)** Monitoring-based Detection: Defenses [56, 66, 67, 74] audit the controller's or applications' behavior to detect anomalies by backup controller, information flow model, root causes, or comparison with real network state.	**(D.8)** Voting-based Defense: Fleet [44], TopoTrust [3], and Mcad-SA[54] depend on the majority to vote out a malicious controller based on odd behaviors around reactive forwarding or spoofing.
	(A.3) Malicious Routing: Insert malicious flow entries to conflict with existing traffic-routing flow entries to detriment flow routings, network policy, or other applications [6, 28, 34, 36, 53, 66].	Globalized Persistent	**(D.9)** Flow Rule Checker: FlowChecker [6], Veriflow [36], Netplumber [34], Eirene [28], and FortNOX [53] build the flow graph/logic with header information (e.g.src/dst IPs/ports) of flow entries to detect flow rule conflicts and policy violations.		
	MARIONETTE: Insert poisonous flow entries to bypass table-miss/LLDP flow entries to poison the topology.	Globalized Persistent	**No existing defense against MARIONETTE** because: existing defenses have overlooked the vulnerability where regular flow entries conflict with table-miss and LLDP flow entries (which is called priority-bypassing attack more specifically) to manipulate link discovery results, as detailed in Table 3.		

reality but are seen as connected by the legitimate controllers, so that packets on these paths can still be routed end-to-end.

To illustrate, using compact notation, given a segment $A2-1C2-1B$ in the real topology in our motivating example, our goal is to make $(A, Port : 2)$ appear to connect to $(B, Port : 1)$ directly as $A2-1B$. Note that since we are removing a link from the topology to connect A directly to B, we need to add additional links back to the idle ports to go undetected. As a result, the links $A1-2C$ and $C1-1D$ are also fabricated. The example assumes LLDP packets are used in the topology discovery protocol, but our attack works on most existing OpenFlow-based discovery packets without any required changes.

3 Related Work

Table 2 summarizes previously developed topology poisoning attacks and the defenses that have been proposed to defend against them. Table 3 highlights the features in MARIONETTE that allow it to evade these defenses. We refer to the table entries in our discussion below. Prior attacks can be categorized as those that fabricate links (A.1), share false topologies (A.2), and maliciously route packets (A.3). Our attack is unique in that it exploits a latent vulnerability that has been overlooked by prior attacks and defenses: *flow entries intended for traffic forwarding can impact link discovery results.*

Link Fabrication Attacks (A.1). The first set of attacks originate from malicious hosts or switches and are typically localized in their network impact. Research in this area focuses on link fabrication attacks [29][60][8] and includes two key techniques: (1) Fabricating LLDP packets; (2) Relaying LLDP packets. Sungmin, et al. [29] and Richard, et al. [60] show that a malicious host connecting with

a switch can fabricate a link by analyzing the received discovery packets, fabricating a discovery packet, and sending it back to the switch. Another approach to link fabrication is relaying legitimate LLDP packets to a wrong endpoint using malicious hosts [60] or switches [8]. MARIONETTE is different from existing works because it is initiated from the control plane and does not need to spoof or fabricate packets. By contrast, MARIONETTE does not fabricate or relay LLDP packets, but uses poisonous flow entries to deceive controllers in a cluster into accepting a new poisoned topology. Our attack is global, persistent, and stealthy.

Link Fabrication Defenses (D.1-5). TOPOGUARD [29] and TO-POGUARD+ (D.5) [60] record and verify the identity of connected switches to mitigate host-involved link fabrication attacks. These methods do not detect MARIONETTE because it is not launched from the edge of the network. Defenses that use port-based detection (D.4) [12, 23] verify that active ports are connected to only one other active port. We maintain this property by designing the poisonous topology with the same degree sequence as the real topology. Latency-based detection methods (D.2) [32, 61] analyze link latencies and consider a link fabricated if its latency is greater than a threshold value. This method suffers from a high error rate due to link latency fluctuation. MARIONETTE only uses switches that have undiscernible incremental latency, especially when the network latency is high [5, 27], to relay LLDP packets.

SPHINX (D.3) [23] assumes controllers are benign and trusts them to construct a flow graph database to detect abnormal flows. Similarly, stealthy probe-based verification SPV (D.1) [8] trusts the control plane and uses it to probe target links. As a result, it cannot detect our attack as MARIONETTE is initiated from the control

Table 3: MARIONETTE Features and Evasion of Existing Detection Mechanisms

Feature	Related Defense	Evading Detection Mechanism
Fabricates links without spoofing, hiding, or fabricating packets	(D.5) TopoGuard(+), (D.7) Monitoring-based Detection	(D.5) detects fabricated packets, and (D.7) identifies spoofing and hiding behaviors. **MARIONETTE does not engage in any such activities.**
Explore the vulnerability that flow entries can manipulate link discovery precisely	(D.7) Monitoring-based Detection	(D.7) analyzes control plane events to identify suspicious behaviors but overlooks **the vulnerability explored by MARIONETTE that flow entries meant for traffic routing can manipulate link discovery.**
Originates at malicious controller/application	(D.1) SPV, (D.3) Sphinx	**MARIONETTE is initiated from control plane** that is trusted by (D.1)(D.3) to detect anomalies initiated from data plane.
Utilizes flow entries matching MAC address (with in-port) with higher priority to bypass table-miss/LLDP flow entries	(D.9) Flow Rule Checker	(D.9) examines flow entry conflict with existing flow entries that routes data plane traffic. However, (D.9) fails to detect MARIONETTE because **the poisonous flow entries by MARIONETTE conflict with table-miss/LLDP flow entries, which are not monitored by (D.9).**
Proactively poisons topology with stealthy flow entries	(D.8) Voting-based Defense	**MARIONETTE leaves benign controllers in (D.8) to incorrectly process reactive forwarding independently** due to altered topology.
Instructs switches to relay link discovery packets	(D.2) Latency-based Detection	(D.2) has a high rate of misjudgment, which may fail to detect the **low-latency switch-based relay of MARIONETTE** very often [5, 27].
Maintains the same degree sequence and high graph similarity	(D.4) Port-based Detection, (D.5) TopoGuard(+)	**Preserved degree sequence by MARIONETTE** ensures each active port connects to only one link, making **(D.4)(D.5) ineffective in detection.**
Any controller in cluster discovers deceptive topology independently	(D.6) Dynamic Cluster	**MARIONETTE allows any dynamically assigned leader controllers in (D.6) to consistently discover the same deceptive topology.**

plane, which both solutions trust, and is able to access the probe information to proactively patch gaps.

Malicious Controller/Application Attacks (A.2-3). The second set of attacks in the literature assumes a malicious controller in a cluster, or a rogue SDN application. As a result, these attacks affect the network globally. Some malicious controllers will share incorrect topology information [35, 43, 56] with the other controllers in the cluster. While these attacks wreak havoc, they do not last long because controllers periodically re-discover the topology such that the wrong topology is replaced with the correct one. Another class of attacks inserts malicious flow entries that impact the routing of network traffic [6, 28, 34, 36, 53, 66]. Such attacks are persistent; however, there are many systems available to detect them.

Malicious Controller/Application Defenses (D.6-9). Monitoring-based detections [56, 59, 74] rely on tracking the controller/application behaviors by the backup controller or intercepted OpenFlow messages and comparing them with real network state to detect anomalies. SDN-RDCD (D.7) [74] detects real-time tampering-based attacks from malicious network elements (switches and controllers) by utilizing a backup controller to audit network events to detect inconsistency among them. TopoTrust (D.8) [3] detects any spoofing and tampering-based attacks with a blockchain technique. MARIONETTE does not cause the type of inconsistencies these systems detect because it does not use tampering, spoofing, or hiding.

More recently, ProvSDN [66] tracks the information flow on the controller to capture cross-app attacks that cause unwanted flow entries to be installed by other applications. PicoSDN enhances ProvSDN by adding a data plane model to achieve fine-grained analysis. Nevertheless, these systems overlook the vulnerability explored by MARIONETTE: flow entries meant for traffic routing can also influence topology discovery, thus failing to establish a causal connection and leaving MARIONETTE undetected.

VeriFlow [36], FlowChecker [6], and FortNOX [53] (D.9) depend on exploring the source/destination address or port information of existing traffic-routing flow entries to construct forwarding graphs

to detect flow entry conflicts. However, the table-miss flow entry (with no match) and the LLDP flow entry (matching protocol: $0x88cc$) are not traffic-routing flow entries, and are thus not monitored by them. As a result, MARIONETTE poisonous flow entries will not be detected as a conflict because they do not conflict with the monitored traffic-routing flow entries but the unmonitored table-miss and LLDP flow entries. Similarly, NetPlumber [34] employs header space analysis to construct a plumbing graph capturing all possible flow paths to ensure incoming flow entries comply with predefined network flow policies. MARIONETTE uses poisonous flow entries to impact link discovery results and has no signature yet.

Unlike existing malicious routing attacks **(A.3)** which insert malicious flow entries to alter data plane traffic routing directly [6, 34, 36, 66] [7], the flow entries inserted by MARIONETTE match `ether-src` (with `in-port`) to supersede table-miss/LLDP flow entries thus manipulating LLDP packet forwarding. This results in the benign controller independently routing traffic incorrectly due to the false topology. This attack is understudied and is undetectable by existing defenses.

Table 3 summarizes MARIONETTE's features and justifies why these features enable MARIONETTE to evade existing detections.

4 Threat Model

The goal of this work is to persistently poison the topology view of SDN controllers to alter some routes to enable traffic eavesdropping or flow exclusion from monitoring at a switch. The control plane topology poisoning attack is an indirect data plane attack that can be initiated from either a malicious controller in a fault-tolerant multi-controller scenario or a malicious SDN application above the control plane (see Figure 3). Unlike existing link fabrication attacks [8, 29, 60], our approach does not require any switches

---

[7]Because each flow entry has a priority value such that the highest priority match takes precedence when a packet may match multiple flow entries, a malicious flow entry with higher priority can conflict/override existing flow entries to manipulate flow routing.

or hosts to be compromised or malicious. The victims are benign controllers and the rest of the impacted network.

Application-Controller Scenario. In this scenario, a malicious application runs on top of a secure control plane—an attack vector popular in existing works [16, 37, 66, 67]. SDN is designed to support third-party applications. These applications may originate from third-party developers[8] or be a REST client [37] using the northbound interface[9] and are thus untrusted and potentially malicious [66]. Attackers may also phish to repackage or redistribute malicious installers that infect controllers at runtime [72]. Configuring flow entries is fundamental for many SDN applications (e.g. routing applications). Both OpenDaylight and ONOS provide northbound RESTful APIs for configuring flow entries. Attacks originating from SDN applications necessitate basic privileges, including read permissions to topology, nodes (switches), and flow table information, and write permission to the flow table. Notably, the malicious SDN application does not exploit northbound API vulnerabilities or attempt to exhaust controller resources. Instead, it utilizes these privileges normally to evade existing malicious SDN application defense strategies [39, 56].

Multi-Controller Scenario. This scenario comprises a fault-tolerant multi-controller SDN architecture in which one of the controllers is malicious. Attackers have leveraged rogue package managers or controller source code repositories [72] to compromise a controller, and controller impersonation attacks [20, 38, 43] to introduce a malicious controller in the cluster. The minimum required privileges of the malicious controller include *read* permissions to the topology, nodes (switches), and flow table data store, along with *write* permissions to its own flow table data store. These requirements align with the fundamental capabilities of OpenDaylight's and ONOS's fault-tolerant replication implementation and can be fulfilled without necessitating the leader role within the fault-tolerant cluster. This is because a malicious controller can induce changes in the network topology via (1) the direct manipulation of flow entries in the OpenFlow switches when it has an equal role [10], or through (2) a confused deputy attack when it has a follower role, in which flow entries are altered in the malicious controller's datastore, causing a leader controller to learn and deploy these entries through consensus. Consequently, the malicious controller can operate with any (leader, equal, or follower/none) role to launch the MARIONETTE attack.

5 MARIONETTE Flow Entry Design

This section details MARIONETTE's flow entry design using the example presented in §2.

---

[8]The open-source SDN controller projects (e.g. OpenDaylight and ONOS) allow third-party developers to submit their applications to be included in the official repositories. What's worse, some commercial products (e.g. Samsung SDN solution [48], Cisco ACI [21], Comcast [69]) are developed based on open-source controllers. Network solution companies also provide applications to run on their SDN controllers (e.g. Junos Space [33], Cisco ACI [21]).
[9]The Restful northbound API is supported by both OpenDaylight and ONOS controller to support diverse applications even written in diverse languages.
[10]OpenFlow v1.3 [63] defines three roles for a controller: leader, follower, and equal. A follower controller has read-only access to the switch and does not receive asynchronous messages(e.g. packet-in). Both leader and equal controllers can modify the switch state and receive asynchronous messages from the switch.

5.1 Precise Link Manipulation

The flow entries that instruct data plane traffic forwarding can also be used for discovery packet forwarding. Existing discovery protocols typically utilize the default table-miss flow entry or a flow entry matching the LLDP packet type (which we call LLDP flow entry) to cause incoming LLDP packets to be forwarded to the controller in a *packet-in* message. Similarly, it is possible to use flow entries to manipulate discovery packet forwarding to fabricate links. To do this, poisonous flow entries must be inserted into the switch at a higher priority than the table-miss/LLDP flow entry. The addition of poisonous flow entries at a higher priority than existing entries will not affect the hit rate metrics of existing entries as discussed in §5.2.

Link fabrication requires the following:

RULE 1. *Given port x of switch A and port y of switch B that are not directly connected, to fabricate a link $Ax \rightarrow yB$, an attacker must force a discovery packet sourced from (A, Port:x) to be received at (B, Port:y).*

To execute the attack, we take advantage of the fact that LLDP packets do not encode intermediate link information. For this reason, the controller only knows the node and port to which it sends the discovery packet and the node and port from which the discovery packet is returned. It uses these two pieces of information from all of the discovery packets to map the network. The controller expects that each switch will send the discovery packet back to it at each hop allowing it to learn every link. To make switch C appear invisible on link $A2 \rightarrow 1B$ to the controller in our example, we must therefore force switch C to forward the LLDP packet to B, rather than pass it back to the controller. We can do so by creating a poisonous flow entry with a higher priority than the table-miss/LLDP flow entry in Switch C. For example, it is easy to construct a naïve poisonous flow entry in C that forwards all packets of type LLDP (i.e., ethernet-type:0x88cc) to switch B. So, we can simply insert flow entry e_1^C on C:

e_1^C	**match**:	ether-type: 0x88cc	**action**:	output: 2

However, this flow entry is easily detectable because "0x88cc" can be recognized as LLDP and this flow entry's output is atypical for an LLDP-related flow entry. A normal LLDP flow entry has the actions as the to_controller rather than a out_port_num. Moreover, it is problematic because links are bidirectional ($A2 \leftrightarrow 1C2 \leftrightarrow 1B$). The poisonous flow entry e_1^C on C influences all LLDP packets received by C, so the entry can fabricate links but fails to precisely affect the bidirectional link. This may result in one port being part of two links which can be detected by existing port-based detection [12, 23].

To manipulate links through a stealthy set of poisonous flow entries, we rely on the MAC addresses of the source switch port to fabricate links shown as flow entries e_2^C and e_3^C on C. These entries achieve our goal without conflicting with other flow entries because all MAC addresses are unique, and only discovery packets (of all types) use the switch port MAC address as the source MAC in their header.

e_2^C	**match**:	ether-src: A2_mac	**action**:	output: 2
e_3^C	**match**:	ether-src: B1_mac	**action**:	output: 1

Stealthiness. Note that e_2^C and e_3^C are stealthy because nothing easily detectable is contained in the "match" field of the flow entry.

The MAC address of a switch port is nothing but a string of 12 hexadecimal digits as any host. We call the design of e_2^C and e_3^C a *vanilla poisonous flow entry design*, and it directly attacks most of the existing SDN discovery protocols because the standard ethernet header is mandatory on any discovery packet.

5.2 VLAN-based Poisonous Flow Entries

However, ONOS controller uses the `ether-src` field of LLDP packets to store a unified fixed fingerprint (FFP) to identify clusters [10]. This limits the attack effectiveness when matching the `ether-src` to distinguish LLDP packets from different sources; therefore, we need another approach.

To realize the attack, we first configure e_4^C and e_5^C matching `ether-src` with `in-port` on C to distinguish $LLDP_{A2}$ and $LLDP_{B1}$ to fabricate links $A2 \rightarrow 1B$ and $A2 \leftarrow 1B$:

e_4^C	**match**:	ether-src: FFP, in-port: 1	**action**:	output:2
e_5^C	**match**:	ether-src: FFP, in-port: 2	**action**:	output:1

However, e_4^C and e_5^C are problematic for later link fabrications. For example, in order to fabricate $A1 \rightarrow 2C$, we need to route the $LLDP_{A1}$ to be received at (C, Port:2) ($A1 \rightarrow 1D2 \rightarrow 2E1 \rightarrow 2B1 \rightarrow 2C$) based on the real topology. Irrespective of the configuration on D, E and B, suppose that the $LLDP_{A1}$ has been forwarded to (C, Port:2); C should forward it to the controller to complete link $A1 \rightarrow 2C$ discovery, but it will forward $LLDP_{A1}$ to $Port : 1$ due to e_5^C. To address this problem, we can use virtual local area network (VLAN) tagging to distinguish the sources of LLDP packets. Each VLAN has its own ID (*vlan_id*), and we use this attribute to tag a set of flows passing through the switch, and then match on them. To tag flows, we use the action `push-vlan` in our flow entry as shown in e_6^D. This entry is configured on switch D and is used to tag the sources of $LLDP_{A1}$.

e_6^D	**match**:	ether-src:FFP, in-port: 1	**action**:	push-vlan:1, output:2

By pushing `vlan_ids` to the LLDP packets at their first hop, the sources are marked precisely without any flow entry conflicting issues. The `push-vlan` action is therefore also added to e_4^C and e_5^C. As the last step of fabricating $A1 \rightarrow 2C$, the flow entries e_7^E on E, e_8^B on B and e_9^C on C listed below are needed to collaborate with e_6^D on D, and e_9^C's priority must be higher than e_5^C on C to function correctly. e_7^E matches on the `vlan_id` and forward to B through ($Port : 1$). Similarly, e_8^B matches on the `vlan_id` and forward to C through ($Port : 1$). e_9^C removes the `vlan_id` using the `pop-vlan` action and sends the original $LLDP_{A1}$ to the controller as desired.

e_7^E	**match**:	vlan-id: 1	**action**:	output:1
e_8^B	**match**:	vlan-id: 1	**action**:	output:1
e_9^C	**match**:	vlan-id: 1	**action**:	pop-vlan, to_controller

We refer to these types of entries as *VLAN poisonous flow entry design* (e_6^D and e_9^C are called E_{VStart} and E_{VEnd} respectively, e_7^E and e_8^B are called E_{VBody}) and define the path $1D2 \rightarrow 2E1 \rightarrow 2B1 \rightarrow 2C$ for fabricating $A1 \rightarrow 2C$ as the *poisoning path*. The VLAN poisonous flow entry design is also important even when `ether-src` is available to distinguish the source of the LLDP packet because the VLAN tunnel can be reused for gap-patching (described in Section 5.3).

To fabricate the same link, the number of poisonous flow entries matching on `ether-src` is less than the case of matching on [`ether-src`, `in-port`] because (1) VLAN is not mandatory for the single-hop case $A2 - 1B$, thus no `vlan_id` is needed and we can use e_2^C and e_3^C safely without flow entry conflicts, and (2) the $LLDP_{A1}$ can be forwarded to the controller with a default table-miss flow entry instead of configuring e_9^C to avoid flow entry conflicts. Thus, the `vlan_id` can be safely popped at B by adding `pop-vlan` to the e_8^B.

$e_8^{B'}$	**match**:	vlan-id: 1	**action**:	pop-vlan, output:1

Stealthiness. Each flow entry in a flow table has a corresponding metric, which increments each time a particular entry routes a packet. As a result, to make deceptive flow entries as stealthy as possible, we must ensure that the new entries do not interfere with the counting rates of existing ones. Our distinguishable `ether-src` VLAN approach keeps these counts consistent because the original table-miss/LLDP flow entries are always used to forward (original or relayed) LLDP packets back to the controller to maintain the same degree sequence of the topology. But when we have to use [`ether-src`, `in-port`] to fabricate links precisely, the metric of original table-miss/LLDP flow entries cannot remain consistent because the mis-forwarded LLDP packet has to be fed back to the controller by our flow entries as e_9^C, instead of original table-miss/LLDP flow entry.

Note that the distinguishable `ether-src` VLAN poisonous flow entry design is the *default* setting for the following discussion unless specified differently. The route of this motivating example is simple with only one hop length. In real network scenarios, there could exist multiple switches along a route $Ax \rightarrow yB$ that need to be configured. This poses a non-trivial challenge and is discussed in Section 6.2.

5.3 Gap Patching Flow Entries

Once the poisonous flow entries are set, the controllers believe that switch A directly connects to switch B, when in reality switch C is in the middle. As a result, when inserting flow entries for paths that include the deceptive link between A and B, the controller will not configure a corresponding flow entry on C to complete the path because it is unaware that C is on the path. This results in a gap in the path configuration.

We fix this issue through a *gap patching* process. The idea is that for each fabricated link setup, we must ensure that the first hop on the fabricated link will route packets along the poisoning path of that link. To avoid flow rule conflict with the legitimate flow entries, we combine the `in-port` with the original match as the match of the gap-patching flow entries to prevent any other normal forwarding packets from hitting this flow entry. For the single-hop gap of flow $f_1 : (H1, H2)$ caused by the single-hop poisoned path, this is done through a flow entry like e_{10}^C. For a longer gap of flow $f_2 : (H1, H3)$, which is caused by a longer poisoning path, we can take advantage of the existing VLAN poisonous flow entries by simply adding a flow entry that pushes the corresponding VLAN tag to the f_2 packets on the VLAN start switch D as shown in e_{11}^D. In either case, only one flow entry is required to patch the gap.

e_{10}^C	**match**:	ether-dst: H2, in-port:1	**action**:	output: 2
e_{11}^D	**match**:	ether-dst: H3, in-port:1	**action**:	push-vlan: 1, output:2

Stealthiness. Note that we could get away without gap patching if the controller has the *reactive forwarding feature* enabled. This is because the controller will send an unknown packet back to the switch with a `packet-out` message and instruct the switch to flood the packet across all ports, filling in the gap. However, we still patch gaps even in reactive forwarding because (1) too many `packet-in` message requests sent to the benign controller at a time are suspicious; and, (2) switch flooding unnecessarily wastes network resources.

Another note is that the gap patching is undetectable by the flow rule checker (i.e., **(D.9)** in Table 3). Because gap patching happens after the topology is poisoned, the switches having gap patching flow entries will not be on the flow paths computed based on the deceptive topology. As a result, the flow rule detection fails to correlate the gap-patching flow entries with the flows based on its logic graph.

6　System Design

To automate and scale up the attack, we designed MARIONETTE, a framework that automatically learns an existing topology, generates a poisoned topology from an attacker's goal (e.g., evading a network monitor, conducting a man-in-the-middle attack), and generates flow entries to instantiate the poisoned topology. The framework comprises four modules: Information Collection, Poisonous Topology Computation, Poison Computation and Setup, and Gap Patching, as shown in Figure 4. The interaction follows a sequence of eight steps. Note that the framework assumes the presence of a malicious controller within the controller cluster, operating in the follower/none role and without direct connection to the underlying network. When the MARIONETTE attack is initiated from a malicious application in an application-controller scenario, it follows similar steps. More details on the individual steps are provided in the full version paper [20].

In Steps ① and ②, MARIONETTE learns the topology and switch information of a network from the replication controller's data store, and then uses this information as the basis for a reinforcement learning (RL) model to compute a deceptive poisonous topology. Step ③ involves MARIONETTE composing stealthy poisonous flow entries to manipulate links based on the information gathered in Step ① to construct the computed target topology in Step ②. In Step ④, the Gap Patching module identifies and computes the required gap-patching flow entries for existing flows. In Step ⑤ and Step ⑥, the poisonous and gap-patching flow entries are configured to the passive replication controller's data store and propagated in the controller cluster by the data store consistency mechanism. The leader controller accepts these data store updates and sends the poisonous and gap-patching flow entries to the network on behalf of MARIONETTE. In Step ⑦, the benign leader controller in the cluster independently discovers the topology which is the one designed by MARIONETTE from Step ③. As a result of the altered topology, the leader controller calculates and sends new routes over the deceptive topology in Step ⑧. The flows will arrive at their proper destinations since gap patching occurs beforehand
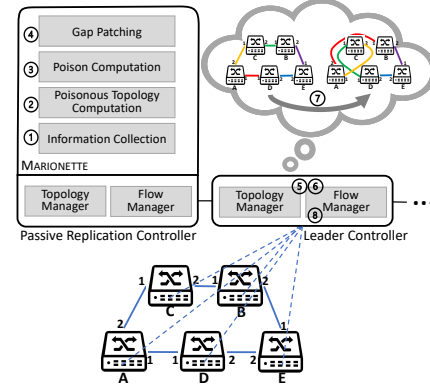

Figure 4: MARIONETTE framework

in Step ⑥ and the Gap Patching module keeps monitoring the flow table to patch when necessary.

In a controller cluster with an active replication design, where the malicious controller assumes an equal role, MARIONETTE can independently send poisonous and gap-patching flow entries while concealing them by not updating the flow table data store.

6.1　Poisonous Topology Computation

The specific attack goal determines the desired deceptive topology. We focus on two attack goals due to space considerations but can support others: (1) routing traffic to an eavesdropping switch point; (2) routing traffic away from a monitor at a given switch point. To create a deceptive topology to satisfy these attack goals, MARIONETTE learns the existing topology through an information-gathering step. With the real topology and the node targeted for an attack (i.e., for eavesdropping or evasion), it employs an RL algorithm taking the collected information and the attack goal to output the desired topology.

RL uses machine learning training methods to train an agent to accomplish some task or behavior based on trial-and-error by executing a set of actions to attain a goal. Training is done through a feedback loop, where the RL agent is given a reward for desirable actions that lead it closer to the goal, and a penalty for actions that are counter to the goal. The agent is effectively a state machine, and after each action, it receives a transition to the next state, and feedback from the environment in the form of a reward or penalty.

We choose RL over traditional methods because:

- *RL adaptation*: The reward component of the RL algorithm can be easily adapted to meet different goals, without modifying the algorithm.
- *Flexibility*: RL provides flexibility in incorporating specific constraints relevant to the attack goals.
- *Feasibility*: The traditional brute force exploration space can be huge: $S = \binom{n}{m} * (\binom{2*m}{2} * \binom{2*(m-1)}{2}... * \binom{4}{2} - 1)$ where n is the total number of edges, m is the number of edges allowed to be changed with a similarity constraint.
- *Best effort*: Unlike traditional optimization formulations that may fail to find the optimal solution within a polynomial running time and return no solution, RL provides its best effort result, which is often satisfactory.
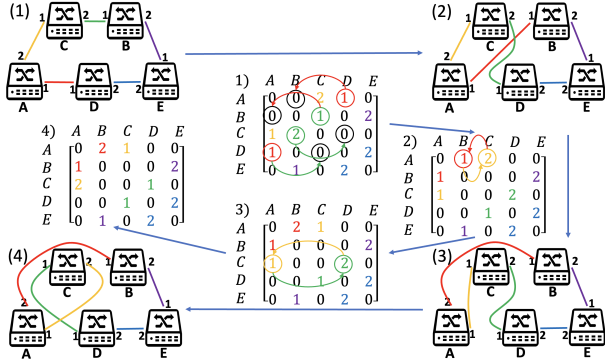
Figure 5: 2-switch actions on PA matrix

To create a realistic and stealthy topology using RL, we enforce two constraints: (1) maintaining the same degree sequence as the real topology, and (2) ensuring a threshold level of similarity between the poisoned and real topologies. Constraint (1) is important because distinct degree sequences are easily detected through the collection of network information via the OpenFlow protocol. Additionally, as our attack utilizes original discovery packets, but does not fabricate any discovery packet, the number of links in the poisoned topology cannot exceed the number of links in the real topology. Constraint (2) aims to prevent suspicion by avoiding sudden and significant alterations in the topology.

To accelerate the learning process, we analyze the topology characteristics and attack goal to guide the agent by certain actions [52] which will be discussed after introducing actions of our RL model.

State. The state of the model is the topology after changes caused by actions. The initial state is the real topology. Mathematically, we define a port-based adjacency (PA) matrix to depict the topology:

DEFINITION 1. *Given a topology with n switches and m bidirectional links, $p_{i,j}$ is the port number on switch s_i such that switch s_i connects to switch s_j through port $p_{i,j}$, $n_{i,j} = (s_i, p_{i,j})$ is called the neighbor of switch s_j. A port-based adjacency matrix is a $n \times n$ matrix A such that:*

$$A_{i,j} = \begin{cases} p_{i,j}, & \text{if there is a link from switch } s_i \text{ to switch } s_j \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Action. A well-known result is that graphs with the same degree sequence can convert to each other via a sequence of *2-switches* [13]. A typical 2-switch is deleting any two edges and reconnecting the idle ports differently [13]. There are various ways to change the graph while maintaining the same degree sequence and they can be converted by several 2-switches. Another method to change the graph while maintaining the degree sequence is what we call *node-reallocation*. The node-reallocation is a series of 2-switches as shown in Figure 5 which is essentially only one node-reallocation (reallocating C between A and D) but three 2-switches. Mathematically, a 2-switch is a sequence of element switching on the PA matrix shown in Figure 5. The color of the matrix elements matches the color of the bidirectional fabricated links:

Action Priors. Both 2-switch and node-reallocation lead to any graph that has the same degree sequence. However, certain actions may work faster than others on different topologies. We use this observation as the action priors [52] to accelerate the learning

process. Intuitively, node-reallocation is well suited to force a different route to be chosen because it is the unit action that directly changes the routing distance (e.g. $A2 - 1B2 - 1E$ seems shorter than $A1 - 1C2 - 1D2 - 2E$ after reallocating C between $A1 - 1D$). However, the 2-switch action is more efficient than node-reallocation on a tree(-like) topology because there are no (less) other routes that can be forced on a tree topology due to its acyclic nature. In our evaluation, we choose an action of either 2-switch or node-reallocation based on their suitability for different topologies. The action space size is equal to $\binom{m}{2}$, where m is the number of edges. Note that a link re-connecting to a different port on the same node is also considered as changing the adjacencies.

The number of actions n_a in each training episode depends on the network size n and similarity requirement $s \in [0, 1]$ because an increasing number of actions in each episode results in a fabricated topology that diverges more from the real topology. We choose the number of actions roughly by $n_a \approx n \cdot (1 - s)$. We must also preserve the connectivity of the altered topology realistically after each action.

Reward. We define two dimensions to measuring a reward. First, our goal is to either divert as many flows as possible to a given eavesdropping switch (s_{vul}) or drive as many flows as possible away from a monitoring switch point (s_{mon}). We refer to this maximization of flow diversions to or away from a node as our *flow coverage goal*. Second, we wish to maintain a graph similarity beyond a threshold value to be stealthy. For each step, when the flow coverage on s_{vul} (s_{mon}) is greater (less) than the threshold value and the graph similarity is beyond the threshold value, a reward of 1 is granted. Otherwise, the reward is -1 (penalty). Episode training is complete when a step achieves the goal. Here we prioritize the flow coverage goal and guarantee a certain graph similarity value.

Flow Coverage: After the topology has been altered, the flow routing will be re-calculated based on the poisoned topology. We must check whether this altered topology can trigger enough flow routing updates that meet the flow coverage goal on node s_{vul}/s_{mon}. Nodes such as s_{vul} may not appear on the paths in the poisoned topology; however, they are still on those paths in the real topology. It is problematic to calculate flow coverage by directly checking whether the s_{vul} is on the updated paths based on the poisoned topology. The correct approach is to check whether the *real neighbors* of s_{vul} are on the updated paths. For the example in Figure 1, the updated path $P_{update} : H_1- > 3A2- > 1B2- > 1E3- > H_2$ does not include C based on the deceptive topology. But C is actually between B and E in the real topology. We can infer this result by checking whether the real neighbors of C are on P_{update}. $A : 2$ and $B : 1$ are C's real neighbors and they are on P_{update}, so we know this deceptive topology meets our goal to eavesdrop flow f_1 on C. Finally, we count the number of such paths to get the flow coverage result. Note that the neighbor node is associated with a port number as defined in Definition 1.

Graph Similarity: There are various ways to evaluate the similarity between graphs. We choose a simple method called vertex/edge overlap (VEO) [51] as it fits our scenario well. VEO measures the overlapped vertices and edges between graphs.

Since we maintain the degree sequences of the topology graph, the numbers of vertices and edges do not change. As a result, the

higher the number of vertices, the less sensitive the similarity score is to changes in the edges. To exclude this negative influence caused by the number of nodes, we derive the edge overlap (EO) method from VEO to evaluate graph similarity for our case and compute it with the following formula:

$$sim_{EO}(G, G') = \frac{|E \cap E'|}{|E|} \quad (2)$$

Where G and G' are the real topology and poisoned topology, and E and E' are the edge sets for both graphs.

Stealthiness. MARIONETTE modifies only a few links in a large network while maintaining the degree of each network node to evade (D.4) port-based detection. The similarity constraint is for avoiding manual detection, implementing the intuition that topologies should not change dramatically. Because some goals on certain topologies are impossible to meet while maintaining the same degree sequence regardless of the similarity, this RL model is a best-effort algorithm. The strictness of the constraints (similarity) and the effectiveness of goals (flow coverage) have a tradeoff that needs adjusting based on the situation.

6.2 Poison Computation and Setup

Once the RL workflow creates a poisonous topology, MARIONETTE generates and installs the poisonous flow entries to realize the deceptive topology. In §5, we introduced a topology poisoning attack using poisonous flow entries and provided examples of simple link poisoning. However, in large network scenarios, the setup of poisonous flow entries can become complex due to the sheer number of possible links that can be manipulated with possible intermediate nodes along the path. To scale our approach, we designed a general algorithm for computing the necessary poisonous flow entries for each hop, taking into account the network topology, switch nodes, and a deceptive link as input. The algorithm of the precise link manipulation implements Rule 1, focusing on how to ensure that the discovery packet sent from $(A, Port : x)$ is received at $(B, Port : y)$.

6.3 Gap Patching Computation and Setup

Once the topology has been poisoned, the routes calculated by the benign leader controller based on the deceptive topology may contain gaps as described in §5.3. We address this proactively and reactively. For proactive patching, we set the gap-patching and poisonous flow entries together because we can predict the new routes for existing flows. In reactive forwarding mode, the controller instructs the switch to flood this unexpected packet [11] with `packet-out` message to all its ports. This behavior provides MARIONETTE with ample time to patch the gaps. In this context, the latency performance of the gap-patching model primarily affects stealthiness rather than persistence. Reducing the number of unexpected packets sent to benign controllers results in a lower level of suspicion.

---

[11]When a packet is forwarded to the controller by a switch, but the switch's position contradicts the packet's routing path based on the recent topology, we designate it as an unexpected packet to the switch.

Table 4: MARIONETTE attack level towards SDN Controllers

Controller	Protocol	Security	MAC LLDP/BDDP	Attack
Floodlight [55]	HOFDP	Hash	src switch port	Complete
OpenDaylight [25]	OFDP	Hash	src switch port	Complete
ONOS [10]	HOFDP	No	Fingerprints	Complete
Ryu [1]	OFDP	No	src switch port	Complete
Pox [45]	OFDP	No	src switch port	Complete

7 Evaluation

We evaluate MARIONETTE against 10 SDN discovery protocols that use LLDP or variants for topology discovery (§7.1), and 5 open-source SDN controllers (§7.2). We demonstrate MARIONETTE attacks starting with a controller impersonation attack on both the OpenDaylight and ONOS clusters (§7.3). We then evaluate our RL agent on two use cases: (1) traffic eavesdropping and (2) monitoring evasion (§7.4). Lastly, we measure MARIONETTE's stealthiness against the current state-of-the-art in SDN attack detection – PicoSDN [67].

7.1 Attacking discovery protocols

MARIONETTE can attack nine different SDN discovery protocols [20]. There are various versions of SDN Discovery Protocols aiming to enhance performance by reducing unnecessary discovery packet transmissions. These protocols use customized discovery packets while relying on the same or similar discovery mechanism as OFDP, making them vulnerable to our attack. Hybrid OFDP (HOFDP) uses LLDP packets along with Broadcast Domain Discovery Protocol (BDDP) packets to discover a hybrid network (both legacy and SDN devices), following OFDP's mechanism. The BDDP packet has a standard ethernet header with `ether-src`, making it vulnerable to our poisonous flow entries, too.

Discussion. Among the SDN discovery protocols, only SLDP [46], TILAK [47], and sOFTP [11] address security issues related to discovery. However, they focus on the data plane, as existing topology poisoning attacks are typically initiated from the data plane. For example, SLDP randomizes the `ether-src`, which inhibits a successful attack unless we use VLAN-based poisonous flow entries to regain full control. Only sOFTP provides a partial defense against MARIONETTE because sOFTP only partially relies on OpenFlow for link discovery.

7.2 Attacking SDN controllers

We evaluate 5 popular open-source SDN controllers (Table 4) using the illustrative example introduced in §5 assuming a topology poisoning attack initiated by a malicious application. We deploy Mininet-v2.2.2, Floodlight-v1.2, OpenDaylight-v0.15.3, ONOS-2.2.0, Ryu-v4.34, and pox-v_eel on six connected VMs with Linux systems on the CyberVAN testbed [17].

Among the five controllers, only OpenDaylight and Floodlight have implemented security enhancements by introducing hash checks on LLDP packets [68]. However, the hash check mechanism does not defend against MARIONETTE as our technique does not tamper with LLDP packets but modifies the LLDP packets' path. Indeed, we achieved precise topology poisoning on all controllers.

(a) Original topology on ONOS
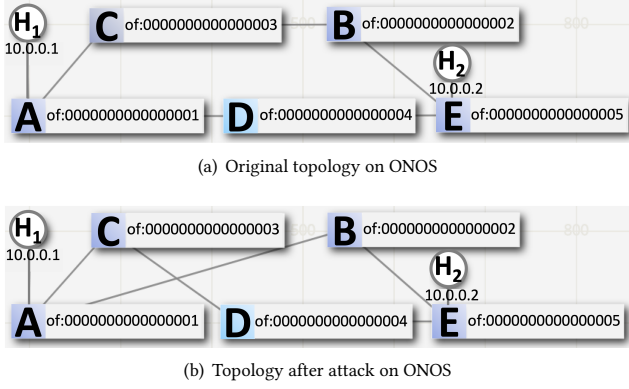

(b) Topology after attack on ONOS

Figure 6: MARIONETTE attack on ONOS cluster

Interestingly, while Ryu and ONOS lack security enhancements in discovering the topology, they present additional challenges in launching a successful attack. Ryu sets its flow entries for forwarding LLDP packets back to the Ryu controller with the highest priority in the flow table. As a result, MARIONETTE must lower the priority of the entries set by the Ryu controller before initiating the attack. ONOS employs per-logical-plane fingerprints encoded on the `ether-src` field, resulting in a unified value of `ether-src` per cluster. As a result, VLAN-based poisonous flow entry matching [`ether-src`, `in-port`] is mandatory to identify the source of discovery packets for all three deceptive links. To conclude, MARIONETTE can attack all 5 open-source controllers.

Discussion. Discovery packet flow entries are static and specifically serve the control plane. As a result, they should not be changed or overridden easily by arbitrary controllers. Unfortunately, there is currently no access control mechanism for flow entry enforcement implemented in any of the SDN controllers examined.

7.3 Attacking SDN Controller Clusters

Among all open-source controllers, only OpenDaylight and ONOS support controller clustering [25] [10]. To assess the attack in both OpenDaylight and ONOS cluster scenarios, we first initiate a controller impersonation attack occurring when a controller goes offline, and a malicious machine located in the same subnet exploits the opportunity to join the cluster by impersonating the offline controller [20]. The experiment on the ONOS cluster is described below.

We construct a three-node ONOS cluster. The cluster is built with ONOS docker `onosproject/onos:2.2.2` and atomic docker `atomix/atomix:3.1.5` on the host machine with Ubuntu 20.04.6 LTS system. Its network is simulated with `Mininet-v2.3.0` on the host machine. The third controller ONOS-3 is in passive replication mode without a direct OpenFlow connection to the Mininet.

We recreate our example scenario (§5) on the ONOS cluster, with attack results shown in Figure 6. We use VLAN-based poisonous flow entries to poison the topology view of the controllers.

In this experiment, we define a topology on Mininet as shown in Figure 6(a) and connect it with ONOS-1 and ONOS-2; ONOS-3 is not connected, serving as a passive replication of the control plane. The malicious ONOS-mal takes ONOS-3's identity in the cluster and sets up computed poisonous flow entries regardless of

Table 5: Networking Scenario and Randomized Flows

Attack Goal	Topology	# Nodes	# Edges	# Flows	Max Degree
Eavesdrop Node 6	FatTree	36	48	120	4
Evade Node 8	Chinanet	42	66	89	20

not connecting with Mininet, either. After that, the topology view changes on all the controllers and settles as shown in Figure 6(b). Nodes B and D switched their locations.

ONOS has reactive forwarding enabled so it fixes the gap automatically when we start the ping from H_1 to H_2 after the topology poisoning. Because the version of ODL we use does not have an available forwarding application, we wrote a proactive flow rule installation application on it. To show the gap, we only sent gap-patching flow entries after the controller changed the routing based on the deceptive topology.

Discussion. In both OpenDaylight and ONOS clusters, any controller, regardless of role, can set up flow entries even without a direct OpenFlow connection due to the data store consistency mechanism. When a passive replication controller sets up flow entries, it actually updates its flow table data store. This update is captured and accepted without evaluation by other controllers including the leaders. As a result, the leader controllers set up the flow entries to the network on behalf of it.

7.4 Poisonous Topology Computation by RL

Enterprise networks and backbone networks are both common SDN scenarios [42]. To validate the practicality of our attack, we evaluate our RL algorithm on an enterprise *fat tree* topology [40] and a backbone network with centralized nodes named *Chinanet* [50]. Table 5 summarizes the topologies and flows used in our evaluation.

We implemented the RL algorithm with Python3.9 using the Stable-Baselines3 [14]. The 2-switch and node reallocation actions are used because these are the unit actions to change a graph while maintaining the same degree sequence, and they are efficient actions for training on the tree topology and cyclic topology, respectively. Because evading monitoring is harder to achieve, we set 10 actions per episode for Chinanet case and 5 for FatTree. With the equation of $n_a = n(1 - s)$, where n is the number of links, s is the similarity, and n_a is the number of actions per episode (§6.1), the similarity threshold for FatTree and Chinanet are roughly set to 0.9 and 0.8, respectively. For each step, a reward of 1 is granted to the RL agent if: (1) the number of the shortest path routes covered by the eavesdropping/monitoring node is greater/smaller than the flow coverage goal and (2) the similarity is above the threshold value. Otherwise, a -1 is granted. Because a failed action causes a -1 reward, the acceptable Episode Reward Mean of FatTree and Chinanet topologies are the ones larger than -5 and -10, respectively, meaning at least one success is made.

Eavesdropping on Enterprise Network. We depict a general fat tree topology as an enterprise network. The goal of this example (Figure 7) is to determine a poisonous topology that will route four additional flows through Node 6 using shortest path routing. Node 6 has 52 out of 120 randomly generated flows covered originally. We use the 2-switch action to train the model since this is a tree-like topology.
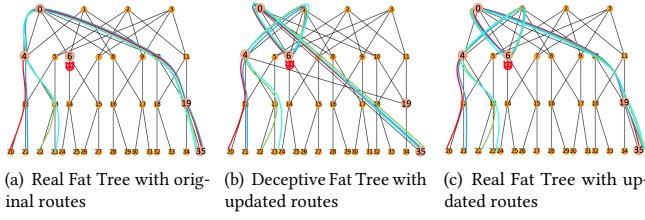
(a) Real Fat Tree with orig-
inal routes

(b) Deceptive Fat Tree with
updated routes

(c) Real Fat Tree with up-
dated routes

Figure 7: Eavesdrop Node 6 (Fat Tree) covering 4 extra flows

(a) RL achieved metrics

(b) RL training log

Figure 8: RL for different eavesdropping goals (Fat Tree)

(a) Real Chinanet with
original routes

(b) Deceptive Chinanet
with updated routes

(c) Real Chinanet with up-
dated routes

Figure 9: Goal: Remove 4 flows from Node 8 (Chinanet)

(a) RL achieved metrics

(b) RL training log

Figure 10: RL for different evasion goals (Chinanet)

Figure 7(a) shows the real topology with the four flows on their original routes. Figure 7(b) shows the deceptive topology computed by RL. The routes of the four flows are changed based on the deceptive topology. They all traverse Node 6 according to the shortest path routing. In Figure 7(c), the induced routes of those four flows still traverse Node 6 on the real topology, thus achieving the goal. The induced shortest path routes of those four flows have two gaps, one at Node 10 and one at Node 19, that will be patched by the gap patching module discussed in §6.3.

Figure 8 shows the details of the RL algorithm results for five different flow coverage goals and their corresponding episode reward mean (ERM) during the training. In Figure 8(a), the graph similarity decreases with an increase in the flow coverage goal which means more changes are required to eavesdrop on more flows. We achieved up to 32 additional flows being routed through Node 6 as a result of the deceptive topology while still maintaining the same degree sequence as the original topology and achieving a similarity larger than 0.9 when compared with the real topology. Figure 8(b) shows that the ERM of the three lowest coverage goals stops at around 50,000 steps (indicating a convergence) because the callback function to stop training when the reward meets threshold -4 is set. However, the ERM for covering 84 flows at Node 6 fails to converge (maintained -5 except for 1 timepoint) and never stops before the end of the total steps, meaning this is a hard goal to achieve. Despite this, the RL still found a solution because of the 1 timepoint (around Step 30000) with ERM greater than -5.

Evading Monitoring on Backbone Network. The attack goal on the Chinanet topology (Figure 9) is to drive four flows away from Node 8 which may be a monitoring point that we wish to avoid. Node 8 originally covers 51 out of 89 randomly generated flows. We use node-reallocation action to train the model since it is a mesh-like topology.

Figure 9(a) shows the real topology with the four flows on their original routes, all traversing Node 8. Figure 9(b) shows the deceptive topology learned by our RL model. The routes of the four
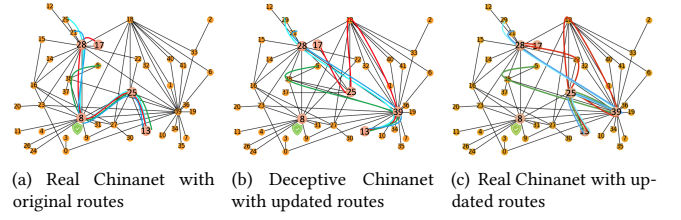
flows are changed based on the deceptive topology and no longer go through Node 8. Instead, as shown in Figure 9(c), they all go through Node 39. Figure 10 shows the details of the RL algorithm results and the ERM for five different goals for evading Node 8. When the flow coverage is smaller than our flow coverage upper bound, we grant a reward of 1. Otherwise, the reward is -1.

Figure 10 shows the RL results and corresponding ERM during the training of five different flow coverage goals. We successfully divert up to 17 flows away from being monitored at Node 8 by computing a deceptive topology with the same degree sequence and a similarity score larger than 0.8 when compared with the real topology. Figure 10(a) shows that the graph similarity generally decreases with the decreased expected flow coverage upper bound which means more changes are required to drive more flows away from being monitored. Figure 10(b) shows that the ERM is higher for easier training goals, indicating fewer actions per episode. Although the ERM values never converge, our goal is a one-time solution rather than model convergence. An ERM greater than -10 suggests that the RL has successfully found a solution.

Discussion. The RL-based poisonous topology computation is scalable because it computes the deceptive topology offline and infrequently to maintain stealth. We do not need a stable agent returning a solution with real-time flows as input. Instead, we observe and select representative flows as input for the RL training. During the training, any reward greater than the minimum ERM means a deceptive topology meeting our goal has been captured. Moreover, the topology only has hundreds of nodes, and changes should correspond to a small segment (e.g. around the eavesdropping node) of the large production networks.

To validate the RL acceleration of action priors technique in Section 6.1, we also used a 2-switch action to train the model to evade monitoring on the Chinanet topology to compare it with the node reallocation. In this case, ERM remains -10 for all 50,000 steps even with the simplest goal of driving one flow away from Node 8. Intuitively, that is because a single 2-switch action cannot achieve

the goal of evading a node on the mesh-like Chinanet topology and the exploration space is huge $\binom{66}{2}^{10}$. On the other hand, the node reallocation action returns a successful result quickly because very few node reallocations are necessary on the cyclic Chinanet topology to change the length of some paths leading to driving traffic away from a node.

7.5 Evading Existing Detection

(D.7) Monitor-based detection is the only detection relevant to Marionette. Among all the systems listed in (D.7), only the code of PicoSDN has been made available. The other types of defenses described in the paper focus on characteristics of the network that are not altered by Marionette, so they cannot detect it by design. These characteristics include degree sequence, LLDP packet integrity, information integrity, and traffic routing integrity. Note that because there are defenses (D.3-5) that measure degree sequence, one of the core components of Marionette is to ensure the degree sequence remains unchanged. (D.8) Voting-based detection can address spoof-based attacks and suspicious activity involving multiple entities with respect to decision-making (e.g. reactive forwarding decisions among controllers), but the Marionette does not spoof; it unilaterally inserts malicious flow entries with no other controllers involved to evaluate its behavior.

We have comprehensively analyzed the existing related defenses (Table 2) and their ineffectiveness against Marionette (Table 3). We provide an in-depth evaluation of PicoSDN [67]. PicoSDN [67] is a provenance-informed detection system that analyzes network event logs and uses topology information to enable packet tracing on the data plane, which in turn allows the construction of causal relationships between control and data plane models. This approach enables precise identification of the roots of some attacks.

The GitHub repository of PicoSDN [15] does not provide any algorithm implementation to build the provenance graph, but only the classes and methods to operate on a provenance graph (confirmed with the author). Thus we manually construct a snippet of the provenance graphs of fabricating link $A2 \rightarrow 1B$ in Figure 1 and the consequent gap patching for $H_1 \rightarrow H_2$ traffic [20], following the PicoSDN algorithm [67].

PicoSDN fails to detect the topology poisoning attack of fabricating link $A2 \rightarrow 1B$ caused by poisonous flow entries f_1, f_2 of Marionette because it overlooks that flow entries can attack the topology discovery result. Whenever the topology is changed, either legitimately or through a Marionette attack, PicoSDN starts a new epoch to construct a provenance graph independently [Algorithm 1(line 1-2) [67]]. In Epoch 1, Marionette attacks the topology view based on real topology. As a result of the poisonous flow entries, PicoSDN starts Epoch 2 due to the changed topology.

PicoSDN fails to detect the gap patching flow entries f_3 because it fails to recognize that Switch C is between Switch A and Switch B due to the undetected poisoned topology. When traffic starts from H_1 to H_2, PicoSDN constructs a data plane model based on the deceptive link of $A1 \rightarrow 1B$ so it links Packet-Out p_2 at A with Packet-In p_5 at B [Algorithm 1(line 7-10) [67]] which misses Packet-In p_3 and Packet-Out p_4 at C in the gap [12]. Consequently, the gap

patching flow entry f_3, set to prevent unnecessary flooding of reactive forwarding, is not linked to the traffic from H_1 to H_2, evading the anomaly detection successfully. We omit the Reactive Forwarding on E which is in the same pattern as Reactive Forwarding on B. To conclude, the causal analysis by PicoSDN based on its fine-grained provenance graph fails to detect Marionette.

Discussion. PicoSDN and existing related defenses overlook the vulnerability of the SDN topology discovery process that certain flow entries can impact the link discovery outcome. The detection of this attack is impossible without monitoring flow rule conflicts against table-miss/LLDP flow entries, which has not been studied. Failing to detect the control plane topology poisoning is troublesome. A direct consequence is that the gap patching flow entries also evade PicoSDN's detection because causal analysis based on a false topology fails to link the gap patching flow entries with the flows that these gap patching flow entries are patching.

8 Discussion

Defending against Marionette: Detecting a Marionette attack involves recognizing that flow entries intended for traffic forwarding can fabricate links. Therefore, monitoring flow rule conflicts on table-miss/LLDP flow entries becomes crucial. More importantly, the discovery protocol should be included in the OpenFlow protocol so that it can be regulated and implemented to a set of standards.

Monitoring-based detection is an option to detect Marionette's attack if the attack signature is known. However, augmenting a monitor on the control plane is cumbersome and the monitoring-based detection can only detect the attack, not defend it. It is more effective to make slight modifications to the discovery packets or discovery process to secure the link discovery result.

According to the OpenFlow protocol, if a packet with an invalid TTL is received at a switch, this switch must drop the packet or send it to the controller. A naive defense against Marionette is to make TTL=1 for the discovery packet to force the switch to send it back to the controller instead of allowing it to be forwarded. However, this method is not robust for two reasons: (1) the TTL implementation on OpenFlow switches may vary; we found that the OpenVSwitch in Mininet will drop the packet with an invalid TTL only when it has a corresponding flow entry that has an action of decreasing the TTL (dec_nw_ttl), otherwise, this packet will still be forwarded in Mininet environment. The process of confirming TTL==0 on dec_ttl action is specified in the Open vSwitch Manual [24]. As a result, this process may also apply to some physical OpenFlow switches because white box OpenFlow switches are just x86 machines running OpenVSwitch with some variants. For example, the OpenFlow switches with the PICOS system by Pica8 have a default dec_ttl action but it can be overwritten by ECMP select group flow or set-l3-egress-keep-fields [70]. They also depend on flow entries with the action of to_controller (reason = invalid_ttl) to send the invalid-TTL packet to the controller [71]. All of the above can be manipulated by Marionette. (2) Marionette can set flow entries to adjust the TTL (set_nw_ttl) to make the discovery packet's TTL valid again.

Although the LLDP has been widely used by SDN to discover links, the LLDP packet is not designed for SDN but for the traditional layer 2 networks. As a result, many fields of the LLDP

---

[12]The controller *flood*s the p_4 due to not knowing a path to forward it on the altered topology, so B can still receive p_5.

Mingming Chen, Thomas La Porta, Teryl Taylor, Frederico Araujo, and Trent Jaeger

payload are redundant and expose vulnerabilities. Our patching plan is to let OpenDaylight randomize the `ether-src`, `ether-dst`, and `ether-type` of the discovery packets each time they are sent. Thus, there will be no way for a flow entry to match these discovery packets, guaranteeing that they are a miss and will be sent back to the controller due to the default table-miss flow entry. This method brings complexity to the controllers because the controller needs to maintain and update a table of matching the randomized `ether-src` with the real `ether-src` to know the source of the received discovery packet. The discovery packet's destination can be known from the `packet-in` message without any changes. Because the table needs to be updated with the frequency of the topology discovery process, maintaining its data consistency is nontrivial.

Limitations: In a single-controller setting, Marionette may need to frequently query the SDN flow table to patch the gaps for long-lasting flows, potentially raising suspicion. However, in a controller cluster setup, all controllers get flow table update notifications, eliminating the need for querying the flow table. Moreover, hit rate statistics on original flow entries will not be kept consistent when it is mandatory to use an LLDP signature field (e.g. `ether-src`) combined with `in-port` to distinguish LLDP packets from different sources to fabricate links precisely.

Ethical disclosure and open-sourcing: We have gone through an ethical disclosure with the OpenDaylight maintainers, who acknowledged the vulnerability and CVE-2024-37018 has been assigned. We will patch this vulnerability and contribute to the OpenDaylight source code. We have published the source code with documentation to reproduce our attack work [19].

9 Conclusion

We describe a new SDN link fabrication attack that is global, stealthy, and persistent. Launched from the control plane, a single compromised controller or a malicious application can manipulate all other controllers in a cluster into learning a poisoned topology by influencing the paths of link layer discovery packets. Link fabrication attacks can route traffic in nefarious ways, to eavesdrop on a set of devices, or avoid a network monitoring device.

To scale the attack to large networks, we present a framework based on reinforcement learning called Marionette, which, given the network topology, and an attacker goal, can automatically generate the set of poisonous OpenFlow flow table entries required to launch the attack. Results show that our approach can successfully attack 9 discovery protocols, and 5 controllers we tested, including controller clusters. Our fabricated links also go undetected by current state-of-the-art defenses.

10 Acknowledgements

References

[1] Build sdn agilely. https://ryu-sdn.org/, 2011. Accessed on 2023-06-19.
[2] AbdelRahman Abdou, Paul C Van Oorschot, and Tao Wan. Comparative analysis of control plane security of sdn and conventional networks. *IEEE Communications Surveys & Tutorials*, 20(4):3542–3559, 2018.
[3] Mohamed Lamine Adjou, Chafika Benzaïd, and Tarik Taleb. Topotrust: A blockchain-based trustless and secure topology discovery in sdns. In *2022 International Wireless Communications and Mobile Computing (IWCMC)*, pages 1107–1112. IEEE, 2022.
[4] Ijaz Ahmad, Suneth Namal, Mika Ylianttila, and Andrei Gurtov. Security in software defined networks: A survey. *IEEE Communications Surveys & Tutorials*, 17(4):2317–2346, 2015.
[5] Ismail Al Salti and Ning Zhang. Link-guard: an effective and scalable security framework for link discovery in sdn networks. *IEEE Access*, 10:130233–130252, 2022.
[6] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44, 2010.
[7] Talal Alharbi, Marius Portmann, and Farzaneh Pakzad. The (in) security of topology discovery in software defined networks. In *2015 IEEE 40th Conference on Local Computer Networks (LCN)*, pages 502–505. IEEE, 2015.
[8] Amir Alimohammadifar, Suryadipta Majumdar, Taous Madi, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad 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*, pages 463–484. Springer, 2018.
[9] Liviu ARSENE. Apic vulnerability in cisco's sdn controller allows unauthenticated remote root access. https://www.bitdefender.com/blog/hotforsecurity/apic-vulnerability-in-ciscos-sdn-controller-allows-unauthenticated-remote-root-access/, 2015.
[10] Shaoyong Wu Ayaka Koshibe. Onos network discovery. https://wiki.onosproject.org/display/ONOS/Network+Discovery, 2016. Accessed on 2023-05-25.
[11] Abdelhadi Azzouni, Raouf Boutaba, Nguyen Thi Mai Trang, and Guy Pujolle. softdp: Secure and efficient topology discovery protocol for sdn. *arXiv preprint arXiv:1705.04527*, 2017.
[12] Sonali Sen Baidya and Rattikorn Hewett. Link discovery attacks in software-defined networks: Topology poisoning and impact analysis. *J. Commun.*, 15(8):596–606, 2020.
[13] Michael D Barrus. On 2-switches and isomorphism classes. *Discrete Mathematics*, 312(15):2217–2222, 2012.
[14] Stable Baselines3. Stable-baselines3 docs - reliable reinforcement learning implementations. https://stable-baselines3.readthedocs.io/en/master/, 2022.
[15] Samuel Jero Benjamin E. Ujcich. Provenance for software-defined networking (sdn). https://github.com/bujcich/PicoSDN/tree/main, 2021.
[16] Jiahao Cao, Renjie Xie, Kun Sun, Qi Li, Guofei Gu, and Mingwei Xu. When match fields do not need to match: Buffered packets hijacking in sdn. In *Proc. of the Network and Distributed System Security Symposium (NDSS'20)*, 2020.
[17] Ritu Chadha, Thomas Bowen, Cho-Yu J Chiang, Yitzchak M Gottlieb, Alex Poylisher, Angello Sapello, Constantin Serban, Shridatt Sugrim, Gary Walther, Lisa M Marvel, et al. Cybervan: A cyber security virtual assured network testbed. In *MILCOM 2016-2016 IEEE Military Communications Conference*, pages 1125–1130. IEEE, 2016.
[18] Sang-Yoon Chang, Younghee Park, and Bhavana Babu Ashok Babu. Fast ip hopping randomization to secure hop-by-hop access in sdn. *IEEE Transactions on Network and Service Management*, 16(1):308–320, 2018.
[19] Mingming Chen. Manipulating OpenFlow link discovery packet forwarding for topology poisoning. https://zenodo.org/doi/10.5281/zenodo.12786197, 2024.
[20] Mingming Chen, Thomas La Porta, Teryl Taylor, Frederico Araujo, and Trent Jaeger. Manipulating openflow link discovery packet forwarding for topology poisoning. https://doi.org/10.48550/arXiv.2408.16940, 2024.
[21] Cisco. Open source used in cisco application policy infrastructure controller (apic) 1.2(1). https://www.cisco.com/c/dam/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/3rd-party/Cisco_ACI_Open_Source_1_1_3_v1_0.pdf.
[22] Paul Congdon. Link layer discovery protocol and mib v2.0. https://www.ieee802.org/1/files/public/docs2002/lldp-protocol-02.pdf, 2002. Accessed on 2023-06-21.
[23] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. Sphinx: detecting security attacks in software-defined networks. In *Ndss*, volume 15, pages 8–11, 2015.
[24] Linux Foundation. Open vswitch manual. https://www.openvswitch.org/support/dist-docs/ovs-ofctl.8.html.
[25] Linux Foundation. Opendaylight. https://www.opendaylight.org/.

[26] Open Networking Foundation. Threat analysis for the sdn architecture. pages 1–21, 2016.

[27] Yang Gao and Mingdi Xu. Defense against software-defined network topology poisoning attacks. *Tsinghua Science and Technology*, 28(1):39–46, 2022.

[28] Sana Habib, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Mitigating threats emerging from the interaction between sdn apps and sdn (configuration) datastore. In *Proceedings of the 2022 on Cloud Computing Security Workshop*, pages 23–39, 2022.

[29] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Ndss*, volume 15, pages 8–11, 2015.

[30] Tao Hu, Zehua Guo, Peng Yi, Thar Baker, and Julong Lan. Multi-controller based software-defined networking: A survey. *IEEE access*, 6:15980–15996, 2018.

[31] Tao Hu, Zhen Zhang, Peng Yi, Dong Liang, Ziyong Li, Quan Ren, Yuxiang Hu, and Julong Lan. Seapp: A secure application management framework based on rest api access control in sdn-enabled cloud environment. *Journal of Parallel and Distributed Computing*, 147:108–123, 2021.

[32] Xinli Huang, Peng Shi, Yufei Liu, and Fei Xu. Towards trusted and efficient sdn topology discovery: A lightweight topology verification scheme. *Computer Networks*, 170:107119, 2020.

[33] Juniper. Junos space datasheet. https://www.juniper.net/us/en/products/sdn-and-orchestration/junos-space-datasheet.html.

[34] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, 2013.

[35] Suleman Khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, Mohsen Guizani, and Muhammad Khurram Khan. Topology discovery in software defined networks: Threats, taxonomy, and state-of-the-art. *IEEE Communications Surveys & Tutorials*, 19(1):303–324, 2016.

[36] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, April 2013. USENIX Association.

[37] Jiwon Kim, Benjamin E Ujcich, and Dave Jing Tian. Intender: Fuzzing {Intent-Based} networking with {Intent-State} transition guidance. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4463–4480, 2023.

[38] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[39] Chanhee Lee, Changhoon Yoon, Seungwon Shin, and Sang Kil Cha. Indago: A new framework for detecting malicious sdn applications. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 220–230. IEEE, 2018.

[40] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10), 1985.

[41] He Li, Peng Li, Song Guo, and Shui Yu. Byzantine-resilient secure software-defined networks with multiple controllers. In *2014 IEEE International Conference on Communications (ICC)*, pages 695–700. IEEE, 2014.

[42] Tong Li, Jinqiang Chen, and Hongyong Fu. Application scenarios based on sdn: an overview. In *Journal of Physics: Conference Series*, volume 1187, page 052067. IOP Publishing, 2019.

[43] Yassine Maleh, Youssef Qasmaoui, Khalid El Gholami, Yassine Sadqi, and Soufyane Mounir. A comprehensive survey on sdn security: threats, mitigations, and future directions. *Journal of Reliable Intelligent Environments*, 9(2):201–239, 2023.

[44] Stephanos Matsumoto, Samuel Hitz, and Adrian Perrig. Fleet: Defending sdns from malicious administrators. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 103–108, 2014.

[45] Colin Scott Murphy. The pox network software platform. https://github.com/noxrepo/pox, 2013. Accessed on 2023-06-19.

[46] Ajay Nehra, Meenakshi Tripathi, Manoj Singh Gaur, Ramesh Babu Battula, and Chhagan Lal. Sldp: A secure and lightweight link discovery protocol for software defined networking. *Computer Networks*, 150:102–116, 2019.

[47] Ajay Nehra, Meenakshi Tripathi, Manoj Singh Gaur, Ramesh Babu Battula, and Chhagan Lal. Tilak: A token-based prevention approach for topology discovery threats in sdn. *International Journal of Communication Systems*, 32(17):e3781, 2019.

[48] Samsung Newsroom. Samsung expands its lineup of sdn solutions. https://news.samsung.com/global/samsung-expands-its-lineup-of-sdn-solutions, 2021.

[49] Tri-Hai Nguyen and Myungsik Yoo. Analysis of link discovery service attacks in sdn controller. In *2017 International Conference on Information Networking (ICOIN)*, pages 259–261. IEEE, 2017.

[50] University of Adelaide. Topology zoo. http://www.topology-zoo.org/dataset.html.

[51] Panagiotis Papadimitriou, Ali Dasdan, and Hector Garcia-Molina. Web graph similarity for anomaly detection. *Journal of Internet Services and Applications*, 1:19–30, 2010.

[52] Karl Pertsch, Youngwoon Lee, and Joseph Lim. Accelerating reinforcement learning with learned skill priors. In *Conference on robot learning*, pages 188–204. PMLR, 2021.

[53] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 121–126, 2012.

[54] Chao Qi, Jiangxing Wu, Hongchao Hu, Guozhen Cheng, Wenyan Liu, Jianjian Ai, and Chao Yang. An intensive security architecture with multi-controller for sdn. In *2016 ieee conference on computer communications workshops (infocom wkshps)*, pages 401–402. IEEE, 2016.

[55] Ryan Izard Qing Wang, Geddings Barrineau. Floodlight sdn openflow controller. https://github.com/floodlight/floodlight, 2016. Accessed on 2023-06-19.

[56] Christian Röpke and Thosten Holz. Preventing malicious sdn applications from hiding adverse network manipulations. In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges*, pages 40–45, 2018.

[57] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[58] Arash Shaghaghi, Mohamed Ali Kaafar, Rajkumar Buyya, and Sanjay Jha. Software-defined network (sdn) data plane security: issues, solutions, and future directions. *Handbook of Computer Networks and Cyber Security*, pages 341–387, 2020.

[59] Arash Shaghaghi, Salil S. Kanhere, Mohamed Ali Kaafar, and Sanjay Jha. Gwardar: Towards protecting a software-defined network from malicious network operating systems. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–5, 2018.

[60] Richard Skowyra, Lei Xu, Guofei Gu, Veer Dedhia, Thomas Hobson, Hamed Okhravi, and James 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)*, pages 374–385. IEEE, 2018.

[61] Dylan Smyth, Sean McSweeney, Donna O'Shea, and Victor Cionca. Detecting link fabrication attacks in software-defined networks. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8. IEEE, 2017.

[62] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, volume 3, 2010.

[63] ONF TS-009. Openflow switch specification version 1.3.2. https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.2.pdf.

[64] Yuchia Tseng, Farid Naït-Abdesselam, and Ashfaq Khokhar. A comprehensive 3-dimensional security analysis of a controller in software-defined networking. *Security and Privacy*, 1(2):e21, 2018.

[65] Yuchia Tseng, Montida Pattaranantakul, Ruan He, Zonghua Zhang, and Farid Naït-Abdesselam. Controller dac: Securing sdn controller with dynamic access control. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[66] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-app poisoning in software-defined networking. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 648–663, 2018.

[67] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Adam Bates, William H Sanders, and Hamed Okhravi. Causal analysis for {Software-Defined} networking attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3183–3200, 2021.

[68] Raniyah Wazirali, Rami Ahmad, and Suheib Alhiyari. Sdn-openflow topology discovery: an overview of performance issues. *Applied Sciences*, 11(15):6999, 2021.

[69] Alan Weissberger. Comcast: Onf trellis software is in production together with l2/l3 white box switches. https://techblog.comsoc.org/2019/09/14/comcast-puts-onf-trellis-software-into-production/.

[70] Tracy Yang. Picos 4.4.4 configuration guide (special release): Ecmp select group. https://pica8-fs.atlassian.net/wiki/spaces/PicOS4443beta/pages/115898390/Ecmp+Select+Group.

[71] Tracy Yang. Picos 4.4.4 configuration guide (special release): ovs-ofctl add-flow <bridge> <flow>. https://pica8-fs.atlassian.net/wiki/spaces/PicOS4443beta/pages/115900326/ovs-ofctl+add-flow+bridge+flow.

[72] Ch Yoon and S Lee. Attacking sdn infrastructure: Are we ready for the next-gen networking? *BlackHat-USA-2016*, pages 17–18, 2016.

[73] Yuan Zhang, Lin Cui, Wei Wang, and Yuxiang Zhang. A survey on software defined networking with multiple controllers. *Journal of Network and Computer Applications*, 103:101–118, 2018.

[74] Haifeng Zhou, Chunming Wu, Chengyu Yang, Pengfei Wang, Qi Yang, Zhouhao Lu, and Qiumei Cheng. Sdn-rdcd: A real-time and reliable method for detecting compromised sdn devices. *IEEE/ACM transactions on networking*, 26(5):2048–2061, 2018.