

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

When Address Learning Goes Wrong: Inducing Forwarding Loops and DoS Amplification in SDN

Dezhang Kong, *Zhejiang University;* Yilun Zhang, *Purdue University;* Zekun Xie, *Xi'an University of Posts and Telecommunications;* Ningpeng Zheng, *Yancheng Institute of Technology;* Shi Lin, *Zhejiang Gongshang University;* Zhenhua Xu, *Zhejiang University;* Minghao Li, *Chongqing University;* Zhebo Wang, Xiang Chen, and Changting Lin, *Zhejiang University;* Dong Zhang, *Fuzhou University;* Xuan Liu, *Yangzhou University;* Chunming Wu and Meng Han, *Zhejiang University*

https://www.usenix.org/conference/usenixsecurity26/presentation/kong

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

When Address Learning Goes Wrong: Inducing Forwarding Loops and DoS Amplification in SDN

Dezhang Kong[1], Yilun Zhang[2], Zekun Xie[3], Ningpeng Zheng[4], Shi Lin[5], Zhenhua Xu[1], Minghao Li[6]
Zhebo Wang[1], Xiang Chen[1], Changting Lin[1], Dong Zhang[7], Xuan Liu[8], Chunming Wu[1], Meng Han[1]*
[1]*Zhejiang University,* [2]*Purdue University,* [3]*Xi'an University of Posts & Telecommunications*
[4]*Yancheng Institute of Technology,* [5]*Zhejiang Gongshang University,* [6]*Chongqing University*
[7]*Fuzhou University,* [8]*Yangzhou University*

Abstract

Address learning is a fundamental SDN service that maintains a dynamic mapping from host addresses to switch ports, supporting many critical network applications. However, the address learning's core position makes it an attractive attack target. Meanwhile, unfortunately, it lacks security protection from a global view, resulting in the fact that individual, normal events can collectively cause damage. Based on it, this paper proposes LoopGen, a new attack targeting the address learning mechanism. LoopGen shows that adversaries who compromised hosts can induce the controller to create a data-plane forwarding loop by only sending crafted packets in a specific order. This loop makes LoopGen a cost-effective DoS amplifier that can be combined with different DoS attacks. We conduct extensive experiments, evaluating LoopGen using diverse real-world topologies, different open-source controllers, and heterogeneous switches. The results show that LoopGen has non-trivial feasibility, significant amplification effect, low attack cost, and high stealthiness under existing defenses. Finally, we propose two countermeasures to mitigate this attack.

1 Introduction

Software-Defined Networking (SDN) has become a core building block of modern network architectures by decoupling the control plane from the data plane [51, 53]. This paradigm provides global visibility and flexible network management, and has been adopted in a wide range of scenarios [58, 91]. The precondition for such centralized control is that the controller should dynamically identify hosts' addresses and their locations, thereby making proper forwarding decisions.

To support this function, many SDN deployments rely on the *address learning* mechanism that builds a dynamic mapping between a host address and the ingress switch port. Concretely, when a new flow triggers a Packet-In event, the controller not only decides how to forward this packet to its destination, but also learns the source address by mapping it to the in_port. The learned location will be used to forward subsequent traffic destined to this source address. Address learning serves as the foundation for many critical applications, such as traffic engineering [92] and load balancing [115].

However, the core position of address learning makes it a good target for adversaries to damage the network. We find that the current address learning mechanism updates the address-location mapping database without a *global* security check, causing adversaries to inject a small, carefully ordered sequence of packets so that individual, normal address learning events can collectively trigger a malicious data-plane forwarding loop.

Based on this observation, this paper proposes a new attack, LoopGen. Specifically, adversaries compromise a small number of hosts and strategically inject crafted packets in a specific sequence, inducing the SDN controller to learn the locations of a target address. Each learning event for this address is normal, but can collectively form a forwarding loop. As a result, once related flow rules are installed, subsequent packets destined to the target address will circulate repeatedly without further controller involvement. This makes LoopGen a cost-effective DoS amplifier: the attacker pays a one-time setup cost to create the loop, after which even low-rate traffic can be amplified significantly. More importantly, LoopGen is not tied to a specific packet format, so it can be combined with different DoS attack methods.

We conduct extensive experiments to evaluate LoopGen in four dimensions. *Feasibility:* LoopGen is feasible across three open-source controllers, diverse real-world topologies, and two types of heterogeneous switches, possessing a non-trivial success probability. *Amplification effect:* LoopGen can induce a high amplification factor, causing severe latency increases and throughput degradation. *Attack cost:* compared with a baseline non-looping DoS, LoopGen achieves similar performance degradation while requiring orders-of-magnitude fewer packets (about 1.5% of the baseline). *Stealthiness:* even the simplest LoopGen can evade the detection from five popular defenses, significantly reducing the exposure risk. Finally,

*Corresponding author.

we propose two countermeasures with different properties and conduct an in-depth discussion of the limitations of LoopGen.

The contributions of this paper are as follows:

- We uncover a new vulnerability in SDN's address learning mechanism, revealing that it lacks a global security check so that normal events can collectively form a forwarding loop.
- We present an attack prototype, LoopGen, which can induce forwarding loops using only compromised hosts. We also give a detailed theoretical analysis to calculate the lower bound of the success probability.
- We conduct extensive experiments, showing LoopGen's non-trivial feasibility, significant amplification effect, low cost, and high stealthiness.
- We propose two different countermeasures applicable to different SDN scenarios, and conduct an in-depth discussion of LoopGen's limitations and future work.

2 Background

2.1 Software-Defined Networking

Software-Defined Networking fundamentally reshapes network architectures by decoupling the control plane from the data plane, granting operators centralized visibility and programmable management capabilities. In this paradigm, the controller acts as the network brain, orchestrating forwarding behaviors of the entire network via standard protocols (e.g., OpenFlow [85]). Due to its advantages in flexibility and scalability, SDN has been widely deployed in critical infrastructures. A prominent example is Software-Defined Wide Area Network (SD-WAN), which leverages SDN controllers to dynamically steer traffic across hybrid WAN links for geographically distributed enterprise branches [6,18,19,45,47,48], such as Google's B4 [63,67] and Espresso [107]. Similarly, SDN is also used in Cloud environments to manage the complex connectivity of network devices [2,7,46], such as Jupiter [93,95].

2.2 Redundant-Connection Design in SDN

SDN networks usually adopt *redundant connections* due to requirements like reliability, bandwidth, and fault tolerance. These redundant connections naturally introduce *a large number of loop structures* in the topology [93]. In this context, SDN does not rely on traditional distributed protocols like Spanning Tree Protocol (STP) to eliminate physical loops because these protocols inherently conflict with this redundant-connection design [4, 44, 50]. Instead, the centralized controller deploys global-visibility algorithms to determine forwarding policies and select proper paths to avoid loops [62]. However, this paradigm shifts the security boundary from

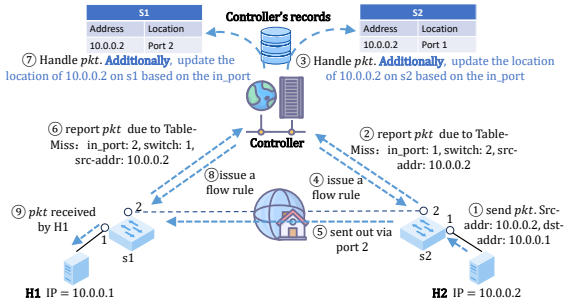


Figure 1: Address learning happens in a normal forwarding process. The controller stores the "addr-port" mapping for each switch.

"avoiding physical loops" to "ensuring that the behaviors of the controller are always secure". Once the controller's behaviors are influenced by adversaries, this ideal state will be disrupted.

2.3 SDN's Address Learning

In the above scenarios, to enable the controller to dynamically process network traffic, SDN adopts an *address learning* mechanism. It allows the SDN controller to maintain a real-time database of network addresses and their ingress points for each switch (i.e., the specific switch port via which a flow enters). The concrete workflow of address learning is reactive, as illustrated in Figure 1.

- *Packet-In*: When the first packet of a strange flow arrives at a switch (①), the switch reports it to the controller due to the lack of related flow rules, which is called Table-Miss (②).
- *Learning*: After receiving the packet, apart from issuing a flow rule to handle this packet (④), the controller additionally learns the source host address and stores it with the corresponding switch ID and port ID (③). This process not only enables the immediate discovery of newly connected hosts but also provides a key topological foundation for subsequent network communication. This packet will be forwarded to the next hop (⑤) and then get processed in the same way on the next hop (⑥, ⑦, ⑧).

As a result, when H1 sends packets to H2, the controller directly queries its location. The out_port on s1 is port 2, while that on s2 is port 1. In this way, the packet will reach its destination automatically.

3 Threat Model

Network. We assume that the network is an SDN network (e.g., an SD-WAN) following the standard SDN architecture.

A centralized controller C is responsible for managing the SDN data plane, which is a graph $G = (V, E)$, where $S \subseteq V$ is the set of switches and $H \subseteq V$ denotes the set of hosts. Each switch $s \in S$ has a port set $P(s)$. The controller maintains an address-learning database D that maps addresses A to ports P for each switch:

$$D(s,a) \to p, \text{where } s \in S, p \in P(s), a \in A \tag{1}$$

Flows are identified by the (a_{src}, a_{dst}) pair[1]. A Packet-In event is a tuple

$$e = (t, s, p_{\text{in}}, a_{src}, a_{dst}, pkt), \tag{2}$$

meaning that a packet pkt of flow (a_{src}, a_{dst}) arrives at switch s via in_port p_{in} at time t and triggers a Table-Miss report. For each e, the controller issues a flow rule if the flow (a_{src}, a_{dst}) is allowed to be forwarded. The output port is based on a_{dst}'s location on s, i.e., $D(s, a_{dst})$. Additionally, C updates the location of a_{src} on s

$$D(s, a_{src}) \leftarrow p_{\text{in}}. \tag{3}$$

When future traffic destined to a_{src} arrives at s, the controller can determine the output port by querying $D(s, a_{src})$.

Adversary's capabilities. We assume that adversaries can only compromise a set of hosts $H_c \subseteq H$. Switches S and the controller are perfectly protected, and the communication channels between the controller and switches are also secure. Adversaries cannot sniff or modify any packets to/from the controller, and cannot inject backdoors or malware to manipulate the behaviors of switches or the controller. In summary, adversaries are only allowed to send/receive packets using compromised hosts H_c, thereby inducing Packet-In events $\mathcal{E} = (e_1, e_2, \dots)$, which in turn influence the controller's partial behaviors.

Adversary's target. The address learning mechanism of SDN controllers is the attack target. Since in SDN architectures, data-plane switches primarily act as rule-enforcing components whose forwarding behaviors are completely determined by the controller, the feasibility of our attack is governed by the controller's address learning mechanism rather than by properties of the underlying switches.

Real-world risk sources and examples. Overall, there are many real-world vulnerabilities revealed, which enable adversaries to invade hosts. These vulnerabilities include but not limited to OS vulnerabilities [24–27, 29, 30, 32], application program defects [33, 35, 39, 40], network firewall vulnerabilities [20, 21, 28, 34, 36, 37], and malware [22, 23, 31, 38]. These vulnerabilities also introduced real-world damage. For instance, in 2017, Nayana, a web hosting enterprise, disclosed that more than 150 of its servers had been compromised [16]. In 2023, Resecurity uncovered that two of the largest data centers in Asia (GDS Holdings in Shanghai and ST Telemedia Global Data Centers in Singapore) had been compromised [5]. Cisco also reported that over 200,000 switches, which are deployed globally across enterprises and data centers, were hacked [1].

[1] Other fields like the protocol field may also be used to identify a flow. Since these fields do not influence the attack logic, we omit these irrelevant fields for simplification.

Table 1: Threat sources in SDN and representative studies.

Risk source	Difficulty	Representative works
From controller apps	relatively high	[57, 98, 99]
From switches	middle	[49, 60, 66, 78, 84]
From hosts (this paper)	relatively low	[60, 64, 68, 76–78, 84, 96]

Comparison with existing studies. Security threats in SDN can be broadly categorized into three classes based on from which they are conducted: (1) controller applications, (2) switches, and (3) hosts. Representative studies are summarized in Table 1. Overall, malicious/compromised hosts are the most common and may be the relatively easiest assumption compared to the others. This is because (1) not only hosts themselves expose a wider attack surface (e.g., OS, application, and firewall bugs), (2) adversaries can also directly become legitimate host users through normal leasing in some scenarios. Besides, (3) adversaries can further compromise other hosts in the network from the already compromised/leased ones, thereby controlling a considerable set of hosts [60].

4 LoopGen

4.1 Goal

Let $a_{\text{target}} \in A$ denote an address chosen by the adversary. Let

$$L = (s_1, s_2, \dots, s_\ell), \quad \ell \geq 3^2, \ s_i \in S \tag{4}$$

be a forwarding loop in the data plane. For each $i \in \{1, \dots, \ell\}$, let $p_i^{\to} \in P(s_i)$ denote the output port on switch s_i that leads to the next hop s_{i+1} (with $s_{\ell+1} = s_1$). Adversaries aim to achieve:

$$\forall i \in \{1, \dots, \ell\}, \quad D(s_i, a_{\text{target}}) = p_i^{\to}. \tag{5}$$

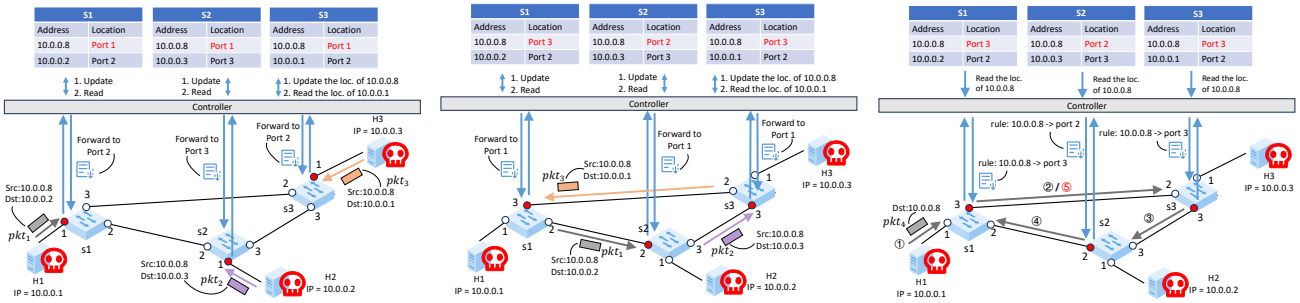
Under this state, packets destined to a_{target} will traverse L continuously.

4.2 Attack Process

Figure 2 illustrates an instance of LoopGen ($\ell = 3$). The adversary controls three compromised hosts, denoted by h_1, h_2, and h_3. The attack proceeds in three stages.

Stage 1: Initial learning at the edge switch ports. The three compromised hosts first negotiate a coordinated start time t_0. At time t_0, they send three forged packets simultaneously,

[2] Interestingly, we find that $\ell = 2$ applies to some cases, which we will show in Section 5 and Appendix B. This variant only succeeds under certain conditions, so we do not regard it as a general example.



(a) Stage 1. (b) Stage 2. (c) Stage 3.

Figure 2: The procedure of LoopGen.

denoted as pkt_1, pkt_2, and pkt_3, respectively. As shown in Figure 2(a), pkt_1 (sent by h_1) uses the IP address of h_2 (10.0.0.2) as its destination, and hence will be forwarded along the path $s_1{:}1 \Rightarrow s_1{:}2 \Rightarrow s_2{:}2 \Rightarrow s_2{:}1$, where $s_1{:}1$ denotes port 1 on switch s_1. Similarly, pkt_2 will be sent from h_2 to h_3 (will traverse $s_2{:}1 \Rightarrow s_2{:}3 \Rightarrow s_3{:}3 \Rightarrow s_3{:}1$), and pkt_3 will be sent from h_3 to h_1 (will traverse $s_3{:}1 \Rightarrow s_3{:}2 \Rightarrow s_1{:}3 \Rightarrow s_1{:}1$).

Crucially, all three packets share the same source address a_{target} (e.g., 10.0.0.8 in Figure 2), which should not be currently used by any host. This ensures that these packets are treated as strange flows and thus trigger Packet-In events at their ingress switches. Upon handling these events, the controller learns the mapping for a_{target}, recording $D(s_1, a_{\text{target}}) \leftarrow 1$, $D(s_2, a_{\text{target}}) \leftarrow 1$, $D(s_3, a_{\text{target}}) \leftarrow 1$.

Stage 2: Overwriting learned locations to align with the loop. After the packets leave their ingress switches, they reach the next-hop switches and again trigger Packet-In events. As shown in Figure 2(b), when such Packet-In events are processed, the controller issues the required forwarding rules and, importantly, *updates* the learned location of a_{target} on each involved switch according to the new in_port.

As a result, the address-learning entries for a_{target} become overwritten as follows: $D(s_1, a_{\text{target}}) \leftarrow 3$, $D(s_2, a_{\text{target}}) \leftarrow 2$, and $D(s_3, a_{\text{target}}) \leftarrow 3$.

Stage 3: Triggering persistent looping forwarding. Once the address learning database has been steered into the above state, a forwarding loop ($s_1{:}3 \Rightarrow s_3{:}3 \Rightarrow s_2{:}2 \Rightarrow s_1{:}3$) is created in the data plane (as shown in Figure 2(c)). The adversary then sends an additional packet pkt_4 whose destination is a_{target} (step ①). Since this packet is unknown at s_1, s_1 reports it to the controller. The controller subsequently reads the location of a_{target} on s1 and then installs a rule that forwards this packet to port 3 (step ②). Similarly, when pkt_4 reaches s_3 and s_2, it will be forwarded to port 3 and port 2, respectively (③, ④). After these rules are installed, a loop is created. Subsequent packets destined to 10.0.0.8 are forwarded entirely in the data plane without further controller involvement, repeatedly traversing the loop (⑤).

4.3 Challenges and Solutions

The above three-stage procedure shows that LoopGen can be realized by steering the controller's address-learning mechanism. However, turning this procedure into a practical attack requires the adversary to address two key challenges: (1) *how to obtain one/multiple a_{target} that are not currently in use*, and (2) *how compromised hosts can reliably confirm whether the loop has been successfully established given their limited network visibility*.

4.3.1 Challenge 1: Probing Unused IP

There are multiple mature methods to solve this challenge.

Probing. Finding unused addresses is generally not a difficult task for an adversary with access to the data plane, because host liveness probing has been extensively studied and is widely supported by mature tools. Concretely, the adversary can pre-select a candidate set of addresses and send standard probing packets (e.g., ICMP, UDP, TCP, or ARP requests). If a response is observed, the corresponding address is used and excluded from a_{target}. There are also mature tools to achieve this goal, such as NMAP [10]. Additionally, Richard et al. proposed four methods for liveness probing in SDN [96], and Li et al. also proposed programmable switch-based in-network scanning methods [81].

Collaboration and registration. If probing fails, the adversary can adopt a strategy by "sacrificing" addresses it already controls. For example, in Figure 2, if four hosts are compromised, the adversary can temporarily silence one compromised host (no responses to probes and no proactive traffic), allowing the network's forwarding state associated with that address to expire due to timeouts. After this period (typically tens of seconds [77]), other compromised hosts can use this address as a_{target}. Besides, adversaries can register new addresses. Since flexible and lenient address registration is one of the features of SDN [89], adversaries usually do not need to worry about a_{target}.

Usurping. Suppose that in extreme cases where adversaries cannot probe or register any usable address, adversaries can

Algorithm 1 Automation Algorithm on Compromised Hosts

Input: Target address a_{target}; $K_{\max}$; packet list $pkts = \{pkt_1, pkt_2, pkt_3\}$; waiting threshold T;

```
1:  for k = 1 to K_max do
2:      t_0 ← Negotiate()
3:      Wait until Current_time()== t_0
4:      SendPacket(src= a_target)                          ▷ Stage 1
5:      Wait(T_1)
6:      SendPacket(dst= a_target)                          ▷ Stage 3
7:      if Received(pkt.dst == a_target) in T_2 then       ▷ Wait for T_2, if this packet is
        successfully looped, it will not be received by any host
8:          Broadcast this failure
9:          Continue                                       ▷ Prepare for the next round
10:     else
11:         Break
```

usurp the legal addresses of normal hosts in SDN, which has been demonstrated by previous studies [60, 64, 68, 96]. This kind of method can be the last choice.

4.3.2 Challenge 2: Confirming Loops

Compromised hosts do not have a global visibility of the entire network, so they cannot easily confirm whether the attack was successful or not. To this end, we design an automation algorithm that allows each host to launch LoopGen automatically. Algorithm 1 is the automatic attack process within each compromised host. Before attempting, compromised hosts negotiate a start time t_0 (line 2). At t_0, each host send a packet whose source address is a_{target}, e.g., 10.0.0.8 in Figure 2. Then, they wait for T_1 time to leave enough time for the construction of the loop. After that, hosts send a packet whose destination is a_{target} (stage 3). If the loop is not created, one of these compromised hosts will receive this packet, and it will broadcast this message to other compromised hosts. Otherwise, if this packet is not received by any host after T_2, the attack succeeded. The number of trials $K_{\max}$ can be estimated based on the success probability, which we will analyze in Section 4.4.

4.4 Theoretical Analysis

After solving the above challenges, there is still a question: *what is the approximate probability of a successful attack*? It is critical for LoopGen to really make sense in reality.

4.4.1 Success Condition Derivation

LoopGen may not succeed in every attempt because network delays exhibit non-negligible jitters. Although compromised hosts start sending packets at the same scheduled time t_0, the actual time for transmitting a packet is affected by different factors (e.g., CPU scheduling and queuing). Therefore, Packet-In events may arrive at the controller in an unexpected order, breaking the loop construction. Figure 3 shows one such failure case. Ideally, the controller should first receive the packet from h_1 (stage 1), then receive the packet from h_3 (stage 2). However, if the sending time for h_1 is delayed,

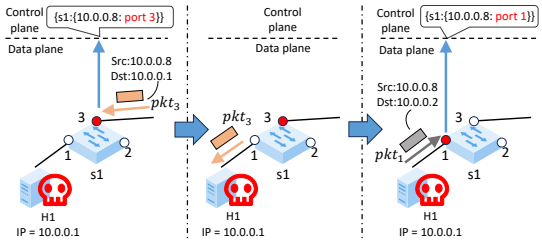


Figure 3: The failed case of attack.

it may first receive the packet from h_3. Consequently, the controller first updates $D(s_1, a_{\text{target}}) \leftarrow 3$ and later overwrites it $D(s_1, a_{\text{target}}) \leftarrow 1$ (triggered by h_1's packet). This disorder breaks the condition of LoopGen.

To reason about the conditions required by LoopGen, we consider three classes of delays: *host-related*, *switch-related*, and *controller-related* delays. For simplicity and to avoid over-fragmented notation, we define each class as a *compound delay* that includes (1) the processing/queuing time at the current device itself and (2) the transmission time to the next-hop device.

- *Host-related delay (Δt_{h_i}).* Let Δt_{h_i} denote the delay contributed by host h_i when it sends a packet. Concretely, Δt_{h_i} covers not only the host-side operations from packet generation until the packet leaves the NIC, but also the time for the packet to reach the ingress switch.
- *Switch-related delay (Δt_{l_i}).* Δt_{l_i} includes the processing time at switch s_i (e.g., pipeline processing and buffering) as well as the transmission time to the next-hop device (a switch, host, or the controller).
- *Controller-related delay (Δt_c).* Let Δt_c denote the controller-side delay for handling a Packet-In and sending the corresponding control response, as well as the transmission time to the next hop (switches).

We intentionally aggregate these components because, in practice, the dominant variability of delays mainly comes from processing/queuing rather than propagation on physical links [90].

Consider switch s_1 in Figure 3. The packet sent by h_1 reaches the controller at $T_1 = t_0 + \Delta t_{h_1} + \Delta t_{l_1}$, and the packet sent by h_3 reaches the controller (reported by s_1) at $T_3 = t_0 + \Delta t_{h3} + \Delta t_{l_3} + \Delta t_c + \Delta t_{l_3} + \Delta t_{l_1}$. To ensure that h_1's packet is processed by the controller before h_3's packet, it requires $T_1 < T_3$:

$$\Delta t_{h_1} - \Delta t_{h_3} < 2\Delta t_{l_3} + \Delta t_c \tag{6}$$

This condition is also suitable for the other two switches. Therefore, the success condition is:

$$\begin{cases} \Delta t_{h_1} - \Delta t_{h_3} < 2\Delta t_{l_3} + \Delta t_c \\ \Delta t_{h_2} - \Delta t_{h_1} < 2\Delta t_{l_1} + \Delta t_c \\ \Delta t_{h_3} - \Delta t_{h_2} < 2\Delta t_{l_2} + \Delta t_c \end{cases} \tag{7}$$

These three conditions should be satisfied simultaneously.

4.4.2 Data Distribution Fitting

From Equation 7, we can observe that it is critical to **figure out the distribution of Δt_h, Δt_l, and Δt_c**. We first collect enough samples of Δt_h, Δt_l, and Δt_c, respectively. The detailed collection process is introduced in Appendix A.1. Then, we use the `distfit` Python library [41] to fit the distribution for them. The detailed configuration we choose is shown in Appendix A.2. The results show that Normal Distribution [88] fits best for Δt_h, Δt_l, and Δt_c. The concrete estimated parameters are shown in Table 2.

Table 2: Distribution Fitting Results

Variable	Distribution	Parameters
Δt_h	Normal Distribution	$\mu_h = 0.0000099, \sigma_h = 0.0000068$
Δt_l	Normal Distribution	$\mu_l = 0.000025, \sigma_l = 0.0000013$
Δt_c	Normal Distribution	$\mu_c = 0.0007, \sigma_c = 0.000039$

4.4.3 Success Probability Estimation

For simplification, we use μ_h and σ_h to denote the mean and standard deviation for all host-related delays, respectively. Similarly, μ_l and σ_l are the mean and standard deviation for all switch-related delays, respectively. Based on the distribution fitting results, we can get that $\Delta t_{h_i} \sim \mathcal{N}(\mu_h, \sigma_h^2)$, $\Delta t_{l_i} \sim \mathcal{N}(\mu_l, \sigma_l^2)$, $\Delta t_c \sim \mathcal{N}(\mu_c, \sigma_c^2)$. Let $X = 2\Delta t_{l_j} - \Delta t_{h_i} + \Delta t_{h_j} + \Delta t_c$ Then, we have

$$X \sim \mathcal{N}\left(2\mu_l + \mu_c,\ 2\sigma_l^2 + 2\sigma_h^2 + \sigma_c^2\right) \tag{8}$$

Let p_{link} be the probability that a single constraint in Equation 7 holds. We obtain

$$\begin{aligned} p_{\text{link}} &= \Pr(X > 0) \\ &= \Phi\left(\frac{2\mu_l + \mu_c}{\sqrt{2\sigma_l^2 + 2\sigma_h^2 + \sigma_c^2}}\right), \end{aligned} \tag{9}$$

where $\Phi(\cdot)$ is the Cumulative Distribution Function (CDF) of the *standard* normal distribution. The detailed derivation of Equation 9 is shown in Appendix A.3. Using it, we can directly compute the probability: $p_{\text{link}} = \Phi(\frac{2\times 0.000025 + 0.0007}{\sqrt{2\times 0.0000013^2 + 2\times 0.0000068^2 + 0.000039^2}}) = 1.0000$, which is a high probability for attacks.

LoopGen succeeds only if all three constraints in Equation 7 are satisfied simultaneously. For simplification, we assume the three constraint events are independent and identically distributed; thus,

$$P_{\text{total}} = (p_{\text{link}})^3 = 1.0000. \tag{10}$$

If we extend LoopGen to more nodes, the attack success probability is $P_{\text{total}} = (p_{\text{link}})^\ell$. Since $0 < p_{\text{link}} \leq 1$, we can get that $\ell = 3$ is the best choice for adversaries. Based on Equation 10, we can calculate $K_{\max}$ in Algorithm 1. For example, if we want to achieve an overall success rate of 99% after $K_{\max}$ attempts, it should satisfy that $(1 - P_{\text{total}})^{K_{\max}} < 0.01$.

5 Evaluation

5.1 Environment Setup

The experiments are conducted on an Ubuntu 20.04 operating system equipped with 8 CPU cores and 16 GB of memory. The controllers we use are ONOS 3.0.0 [11], Ryu 4.34 [14], and POX 0.3.0 [13]. Topologies are from the real-world dataset Topology Zoo [74].

Packet generation, storage, and capture are performed using Scapy [15], whereas Tcpreplay [17] is utilized to inject traffic into the network. Bandwidth measurements are obtained using Iperf [42]. We used two types of switches: OpenFlow switches (versions 1.0 to 1.5) [85] and BMv2 [3] switches. When we use BMv2, network functionalities are implemented in P4-16 [12]. The hardware switch used is a Barefoot Tofino [8] programmable switch with 12 stages per pipeline. The network topology is created using Mininet 2.3.1 [9]. Background traffic traces are obtained from the MAWI dataset [43] (`202201011400.pcap`).

The experiments consider two types of packets: IP and ARP. For IP packets, a flow is defined by the tuple (src-IP, dst-IP), whereas for ARP packets, a flow is defined by the tuple (src-MAC, dst-MAC). The thresholds T_1 and T_2 in Algorithm 1 are set to 3 s and 5 s, respectively, and the parameter $K_{\max}$ is set to 30.

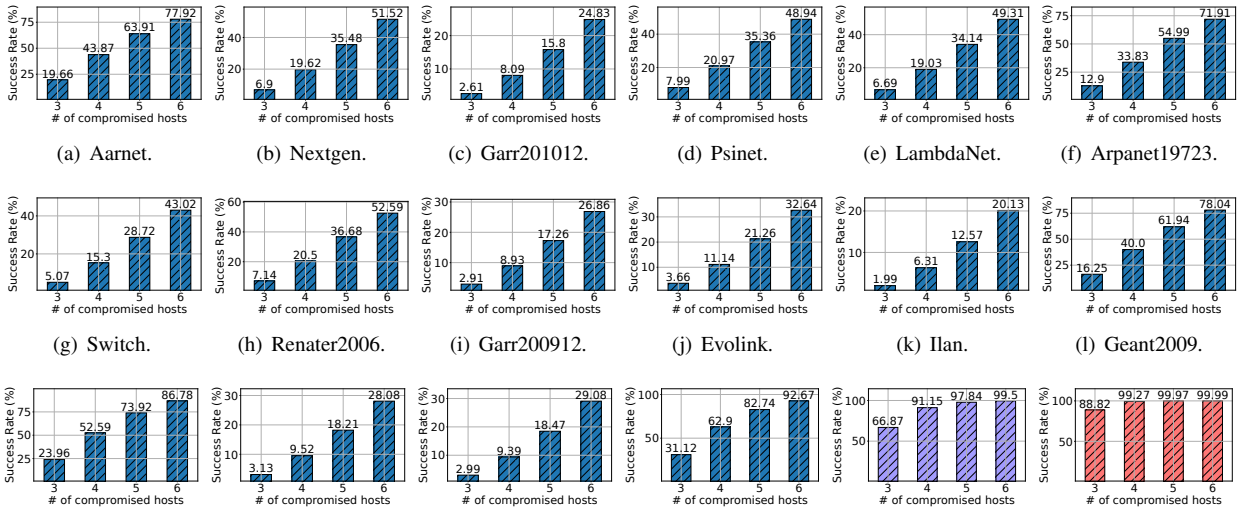
5.2 Feasibility Evaluation

This section assesses the practical feasibility of LoopGen in five dimensions: (1) the feasibility of the prerequisite required by LoopGen, (2) the feasibility across different controllers, (3) the feasibility across diverse network topologies, (4) the feasibility across heterogeneous switches, and (5) the empirical probability of successful attacks in practice.

5.2.1 Feasibility of LoopGen's Prerequisite

We first answer a fundamental question: *what is the probability that the compromised hosts can form a loop structure*?[3] It is the prerequisite for LoopGen. For instance, if the compromised hosts are located along a linear portion of the network, LoopGen cannot establish a closed forwarding loop. Since this condition is primarily determined by the underlying topology and is largely beyond the adversary's capabilities, we estimate this probability empirically using real-world WAN topologies randomly selected from the Topology Zoo

[3] Note that this is not the probability analyzed in Section 4.4. Only when compromised hosts are placed on a loop structure can they attempt LoopGen.



(a) Aarnet. (b) Nextgen. (c) Garr201012. (d) Psinet. (e) LambdaNet. (f) Arpanet19723.

(g) Switch. (h) Renater2006. (i) Garr200912. (j) Evolink. (k) Ilan. (l) Geant2009.

(m) Arpanet19719. (n) Harnet. (o) BTN. (p) Ibm. (q) Leaf-Spine. (r) Fat-Tree.

Figure 4: The probability that hosts can form a loop structure in the topology.

dataset [74] and two data-center topologies (Leaf-Spine (8 leaf, 4 spine) and Fat-Tree (pod=6)).

For each topology, we attach 20 hosts to each access switch (for Topology Zoo topologies, each switch; for Leaf-Spine, each leaf switch; for Fat-Tree, each edge switch). We then calculate the probability that k randomly compromised hosts are positioned on *loop-feasible switches*. First, we enumerate all $\binom{|V|}{m}$ distinct sets, each set has m switches. For each set, we enumerate all $\binom{m}{3}$ triples. A set is labeled as *loop-feasible* if it has a triple in which the three switches can form a loop structure in the topology[4]; otherwise, we label it as not loop-feasible. In the experiment, we chose $m = 3,4,5,6$, respectively.

Figure 4 exhibits the ratio of loop-feasible sets. Across all tested networks, the ratio increases monotonically with the number of m. We can also see that data-center topologies (Figures 4(q) and 4(r)) show much higher probabilities than WAN topologoies. Although these experiments are only a rough estimation of the probability that the prerequisite is satisfied, they can provide solid evidence that the prerequisite is *not* a rare case in practice, especially in topologies with richer path diversity and redundant connectivity. Therefore, LoopGen merits attention as a practical risk rather than a purely theoretical case.

5.2.2 Feasibility Across Different Controllers

We evaluate LoopGen's feasibility across several widely used open-source controllers. The network is a triangular topology created by Mininet. Table 3 summarizes the vulnerable controllers and the corresponding applications that we have identified. Our results show that LoopGen is not specific to any single controller; instead, it arises from a common design pattern in address learning: the unconditional overwriting of address-location mapping based on the in_port, without an effective loop detection mechanism.

[4]To align with reality, we use a strict criterion to determine whether switches form a loop structure, rather than simply considering a triangle formed by switches as a loop. The details are introduced in Appendix C.

Table 3: Vulnerable Open-source Controllers

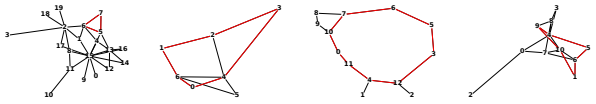
Controller	Vulnerability Location
Ryu	`simple_switch.py`, `simple_switch_12.py` `simple_switch_13.py`, `simple_switch_14.py` `simple_switch_15.py`
ONOS	`ReactiveForwarding.java`
POX	`12_learning.py`, `l3_learning.py`

Table 4: Used topologies and attack points. An attack point is the switch to which a compromised host is connected.

Network	Abbreviation	Δt_l	Attack points
BtAsiaPac	BAP	4.5 ms	5, 6, 7
Nettrail	NT	2.3 ms	0, 1, 3
HiberniaCanada	HC	2.1 ms	0, 12, 6
Sprint	ST	2.6 ms	1, 9, 5

5.2.3 Feasibility across Diverse Topologies

We also evaluate the feasibility of LoopGen across multiple network topologies. As shown in Table 4, we randomly select four topologies and report their critical characteristics. The specific forwarding paths constructed by LoopGen in each topology are depicted in red in Figure 5. In the BtAsiaPac topology, the attack points are switches 5, 6, and 7; in Nettrail, the attack points are switches 0, 1, and 3; in HiberniaCanada, they are switches 0, 12, and 6; and in Sprint, we select



(a) BtAsiaPac. (b) Nettrail. (c) HC. (d) Sprint.

Figure 5: The topologies used. Each node is a switch. Each attack point is connected to a host (not drawn in the figure).

switches 1, 9, and 5. The switches are BMv2 switches created using Mininet. Link delays are configured to match the real latency values reported in the dataset. LoopGen is successfully executed in all topologies, thereby providing additional evidence of its practicality and robustness in real-world network environments.

5.2.4 Feasibility Across Heterogeneous Switches

We evaluated whether LoopGen depends on specific switch implementations. We first conducted experiments on OpenFlow switches spanning protocol versions 1.0 to 1.5, then verified the attack on BMv2 switches and Barefoot Tofino switches. In all of these scenarios, we successfully induced persistent forwarding loops with $\ell = 3$, indicating that LoopGen primarily stems from the controller's address learning mechanism, rather than relying on switch-specific properties.

In addition, we observed a special case $\ell = 2$, where packets oscillate between two adjacent switches. This variant requires that a switch can forward a packet back to its in_port. In our BMv2 experiments, this behavior was supported. In contrast, in our OpenFlow experiments, the rule forwarding a packet back to its in_port was installed on switches, but we did not observe any packets actually forwarded. We infer that the inherent OpenFlow design does not support this logic. These results indicate that in SDN with such programmable switches, if the developers do not deploy correct underlying logic, LoopGen will be conducted more easily (because adversaries only need two compromised hosts). Note that $\ell = 2$ is a special case, while the general LoopGen attack with $\ell = 3$ does not rely on such an assumption.

5.2.5 Success Probability: Theory vs. Practice

Section 4.4 theoretically estimates the success probability of LoopGen (about 100%). To validate this, we conducted practical experiments in a triangular topology using the Ryu controller and OpenFlow13 switches. Across 10 independent trials, LoopGen consistently succeeds on the first attempt in every case, achieving a 100% success rate.

5.3 Amplification Effect Evaluation

Although LoopGen is a DoS amplifier that can be combined with existing DoS attacks, it is necessary to evaluate LoopGen as an independent DoS attack to quantify its amplification effect. This allows a fair comparison with a baseline non-looping DoS in Section 5.4. The network is created using Mininet. We do not set any additional delay on links, and the link bandwidth is set to 10 Mbps. All results in this section are collected after a forwarding loop is created.

Table 5: Amplification Factor α. ARP packets are only observed using Wireshark for one minute. We find that they are forwarded infinitely during this time and mark their α as ∞.

Pkt Type	BtAsiaPac	Nettrail	HiberniaCanada	Sprint
IP	84	42	28	51
ARP	∞ (1 min)	∞ (1 min)	∞ (1 min)	∞ (1 min)

5.3.1 Amplification Factor

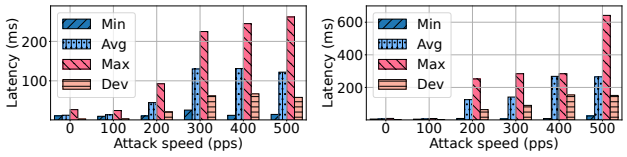
To quantify the amplification effect of LoopGen, we define the amplification factor as $\alpha = \frac{N_{rcv}}{N_{snd}}$. It means how many repeated receptions one attack packet can trigger on the switch port. We evaluated α using IP and ARP packets, respectively. The IP packet represents packets that have a Time-To-Live (TTL) value, which means that such packets will be dropped when TTL is zero. The ARP packet represents packets that do not have a TTL field. For IP packets, we set the initial TTL value to 255. Table 5 shows the results. It can be seen that for IP packets, the amplification factor depends on the number of devices that packets pass through. The fewer devices, the higher the amplification factor. For ARP packets, we only observe for one minute and find that the sent packet is continuously forwarded without being dropped.

5.3.2 Link Latency Growth

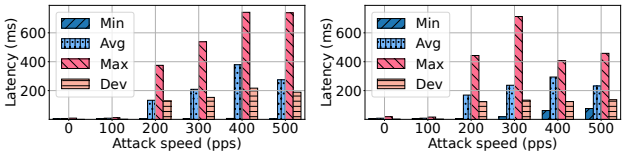
We also evaluated the latency caused by LoopGen. Round-trip time (RTT) is used as the metric. We chose 5 different attack rates (100, 200, 300, 400, and 500 pps). The results are compared to a normal case without any attack. Results for IP packets are shown in Figure 6, while those for ARP are shown in Figure 7.

From Figure 6, we can observe that the attack effects start to become obvious when the attack rate reaches 200 pps for all networks. At 200 pps, the average latency under LoopGen increases by 11× in BtAsiaPac, and by 37×, 55×, and 33× in Nettrail, HiberniaCanada, and Sprint, respectively. Combined with Table 4, we can get that the smaller the Δt_l, the higher the latency caused by LoopGen. This is because a smaller Δt_l means that attack traffic will reach a link faster, which is equivalent to enlarging the amplification factor.

From Figure 7, we can observe that when the attack rate is 100 pps, the latency caused by LoopGen already starts to show a significant increase. This is because ARP packets will not be deleted due to TTL, causing the accumulation of attack traffic in the network. As a result, even at 100 pps, LoopGen already causes significant latency inflation. The ratio between

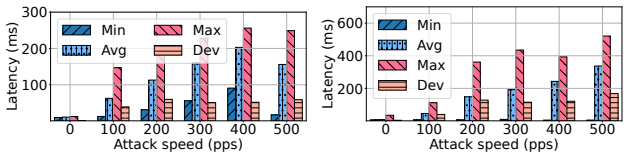


(a) BtAsiaPac. (b) Nettrail.

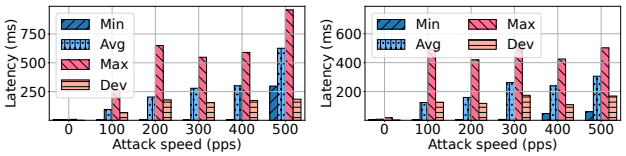


(c) HiberniaCanada. (d) Sprint.

Figure 6: Latency caused by LoopGen (IP packets).



(a) BtAsiaPac. (b) Nettrail.



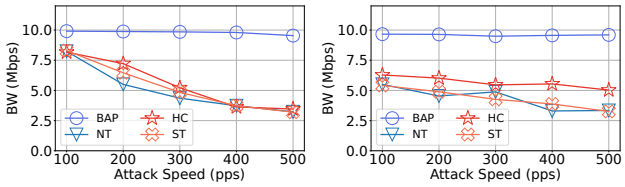
(c) HiberniaCanada. (d) Sprint.

Figure 7: Latency caused by LoopGen (ARP packets).

the attacked latency and the baseline latency reaches 18×, 41×, 89×, and 35× for BtAsiaPac, Nettrail, HiberniaCanada, and Sprint, respectively, which is substantially higher than the corresponding IP-packet results.

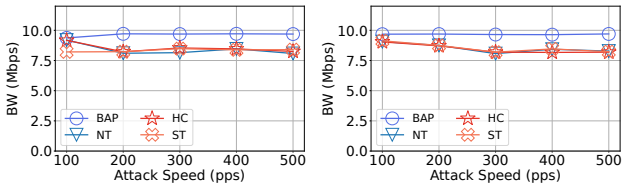
5.3.3 Throughput Degradation

We further evaluated the throughput degradation caused by LoopGen under different attack rates. Figure 8 illustrates the TCP throughput. When LoopGen uses IP packets, a clear decline starts around 200 pps. When the attack packets are ARP, the decline occurs significantly at 100 pps. Figure 9 illustrates the UDP throughput. Differently, UDP bandwidth degradation is significantly smaller than the TCP bandwidth for all topologies. This is because TCP reacts to congestion via loss recovery and congestion control, while UDP does not reduce its sending rate in response to loss. To capture the actual service impact on UDP traffic, we additionally measure packet loss rates, as shown in Figure 10. The results show that even at 100 pps, the loss rate already exceeds 5% in all tested topologies, and it approaches 20% at 500 pps, indicating substantial degradation in delivery reliability despite the seemingly higher UDP throughput.



(a) Use IP packets. (b) Use ARP packets.

Figure 8: TCP Bandwidth caused by LoopGen.



(a) Use IP packets. (b) Use ARP packets.

Figure 9: UDP Bandwidth caused by LoopGen.

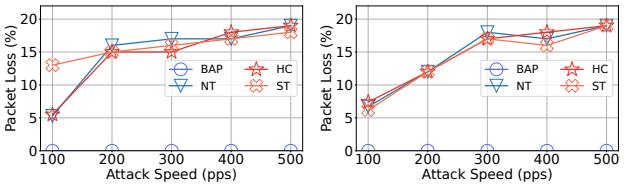
5.4 Cost Evaluation

Although the experiments in Section 5.3 quantify the impact of LoopGen, they are not fully fair for comparison. Therefore, we need to evaluate a baseline (non-looping) DoS attack *under the same experimental settings* and compare it with LoopGen, so as to further demonstrate that LoopGen is a cost-effective DoS amplifier. The metric of cost is the number of attack packets. Tcpreplay is used to inject IP traffic whose TTL is 255 into the network at specified rates. The attack rates for non-looping DoS are 5,000, 10,000, 15,000, 20,000, and 25,000 pps, respectively, while LoopGen only sends packets at 300 pps. The results are shown in Figure 11.

We use Figure 11(a) to illustrate. The baseline DoS causes an average latency of about 200 ms at the attack rate of 20,000 pps. Notably, 200 ms is the first time at which the baseline DoS matches the latency achieved by LoopGen at only 300 pps. It means that to achieve the same attack effect, the non-looping DoS needs to generate 19700 more packets, which is 65.7× that of LoopGen. Besides, the packets required to create a forwarding loop are negligible (three packets per attempt). Therefore, we can conclude that the attack cost of LoopGen is tiny compared to non-looping DoS.

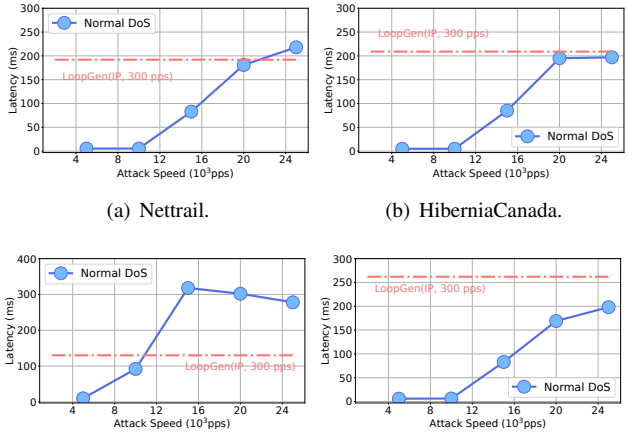
5.5 Stealthiness Evaluation

We evaluate the stealthiness of LoopGen against different SDN defenses. Although LoopGen can serve as an amplifier and be combined with different DoS strategies, in this



(a) Use IP packets. (b) Use ARP packets.

Figure 10: Loss rate when evaluating UDP bandwidth.



(a) Nettrail. (b) HiberniaCanada.

(c) BtAsiaPac. (d) Sprint.

Figure 11: LoopGen vs. non-looping DoS on diverse topologies.

section, we focus on evaluating the intrinsic stealthiness of LoopGen itself. Therefore, in our experiments, we do not combine LoopGen with any additional DoS techniques; instead, the attack dataset is intentionally configured in a simple and conservative manner: we use only a single a_{target}, and all attack packets share the same format.

We think this setup makes the evaluation more convincing, because it provides a *lower bound* of LoopGen's stealthiness. In real attacks, adversaries can further improve stealthiness by diversifying their strategies, e.g., occupying multiple a_{target} addresses, varying packet lengths, source addresses, and header/payload formats, and tailoring the traffic patterns to specific network conditions and defense assumptions. Such adaptations can make the attack harder to distinguish from benign traffic while preserving the amplification benefit, but are out of the scope of this paper.

Defenses on the controller. We consider three controller-side defenses: identity verification, EventScope [100], and SVHunter [103]. Identity verification is commonly used by existing defenses (such as SecureBinder [68] and RPV [76]) to defend against spoofed addresses, which requires a host to prove ownership of an address before the controller accepts location updates. To achieve it, we configure the controller to assign a unique password for each address, and any update of D needs the host to provide the correct password for the claimed source address. Specifically, the controller sends a verification packet to the host, and the host generates a response, which contains the password. As we introduced in Section 4.3.1, since a_{target} can be registered or shared by adversaries, they also possess the password for a_{target}. As a result, when receiving the verification packet for a_{target}, they can provide the correct password. The controller we chose is Ryu, the switches are OpenFlow13 switches created by Mininet. The results show that identity verification cannot detect LoopGen in our setting. EventScope and SVHunter also did not detect LoopGen because they conduct static code analysis on ONOS, while LoopGen does not need to modify the controller's logic.

Defenses on programmable switches. We also deployed two state-of-the-art, open-source DoS defenses running on programmable switches: Lemon [102] and SISTAR [65]. Lemon surpasses Jaqen [82] and Couper [97]. We deployed its BMv2 version on a triangular topology. The benign traffic dataset is from MAWI, while the attack traffic consists of uniformly formatted IP packets injected at 3,200 pps (about 10% of the benign rate). Although Lemon detected LoopGen, it misclassified a large number of benign flows (11 false positives), resulting in the F1 score of only 0.15 ($2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = 2 \times \frac{\frac{1}{12} \times 1}{\frac{1}{12} + 1} = \frac{2}{13}$), which significantly decreases compared to the value on other DoS attacks in the paper (> 0.9). We infer that this is because Lemon lacks enough inspection into internal links due to its unique design to avoid repeated counting.

We then evaluated SISTAR using the parameters provided by its authors. SISTAR surpasses Soter [104], IIsy [110], NetBeacon [112], and Planter [111]. Since SISTAR drops the packets identified as malicious, we use Wireshark to monitor the internal switch port to judge whether LoopGen is detected or not. The background traffic is the same as when evaluating Lemon. We choose three attack rates: 1,000, 2,000, and 3,000 pps. The results show that none of these attacks were detected by SISTAR. It indicates that LoopGen does not possess similar features to existing attacks, making it comparatively stealthy.

6 Countermeasures

We discuss potential countermeasures to mitigate LoopGen.

Loop detection. A key root of LoopGen is that the controller updates the address-learning database D based on in_port without a defense from a global view. As a result, even if each individual update event passes existing defenses (e.g., identity verification), they can collectively form an attack. We therefore propose to design a loop check based on D. Specifically, for each address a, the controller can continuously *trace* the next hop implied by the address-learning

database D: starting from a switch s_0, it reads the output port $p_0 = D(s_0, a)$, uses the topology to locate the next switch s_1 connected to p_0, and repeats this process to obtain a sequence (s_i, p_i) where $p_i = D(s_i, a)$. If any previously visited location (s_j, p_j) reappears, the controller concludes that a forwarding loop is detected. If the tracing length exceeds a threshold H, the controller treats a as unreachable (or the mapping as inconsistent).

The advantage of this method is that it directly checks whether the learned address-location mapping can collectively form a hidden data-plane forwarding loop, which is the core condition exploited by LoopGen. However, it requires accurate topology information and introduces extra controller-side computation. Besides, given the large number of addresses in the network, how to balance the detection frequency and security trade-off is also a problem. Overall, this method suits for security- or correctness-critical SDN scenarios that can afford additional control-plane overhead to proactively prevent loop formation.

Time Constraints on Location Updates. Another mitigation is to limit consecutive location updates for the same address a in the address-learning database D. Specifically, when the controller processes a Packet-In event $e = (t, s, p_{in}, a_{src}, a_{dst}, pkt)$ and is about to perform the learning update $D(s, a_{src}) \leftarrow p_{in}$, it additionally checks the time elapsed since the last accepted update of a_{src}; if $t - t_{\text{last}}(a_{src}) < \delta$ (a minimum window, e.g., 50 ms, depending on the concrete network scenarios), the controller raises an alert. This prevents adversaries from rapidly steering $D(\cdot, a)$ via bursts of crafted packets to induce conflicting forwarding decisions and loops. Overall, this defense is most suitable for SDN scenarios with low host mobility; otherwise, an overly large δ may interfere with legitimate frequent host migrations.

The advantage of this method is that it is simple and has relatively low overhead, and it mitigates the core feature of LoopGen that the location of a_{target} changes frequently. However, it may disrupt normal operation when legitimate host migration is also frequent; thus, δ must be chosen carefully and may need dynamic tuning. Overall, this defense is suitable for networks with low host mobility; it is explicitly noted to be unsuitable for environments with frequent migration, such as data centers.

Experiments. We deployed the prototypes of the above two countermeasures on the Ryu controller. The switch is OpenFlow13 switch. The results show that they detected LoopGen($\ell = 3$) successfully and did not have false positives. On average, Loop Detection consumes about 0.17ms per detection, while Time Constraint consumes 0.0012ms (the threshold of Time Constraint is 50ms).

7 Limitations and Discussion

Restrictions on host positions. LoopGen requires that compromised hosts attach to switches that form a loop structure. Our topology study in Section 5.2.1 shows that, in real-world topologies, the fraction of loop-feasible switch combinations can be non-trivial and increases significantly as the number of switches grows. However, two important limitations remain. First, even if a network contains loop structures, the prerequisite is not always satisfied (i.e., the probability is not 100%). Second, compromised hosts may not be connected to the correct switches exactly. The mapping from compromised hosts to their access switches is outside the attacker's capabilities.

Stealthiness vs. destructiveness trade-off. The amplification effect depends on how long looped packets can persist in the data plane. For IP packets, TTL bounds the number of loop traversals and thus has finite amplification factors; for non-TTL frames (e.g., ARP-like Ethernet frames), packets can circulate continuously until dropped due to congestion, which maximizes amplification but may also increase the exposure risk. As a result, attackers face a trade-off: using bounded-TTL traffic can reduce long-lived signatures while still causing substantial performance degradation, whereas non-TTL traffic provides stronger amplification but is potentially easier to detect.

Environment and generalization. Our evaluation is conducted in a controlled simulated testbed with Mininet. While LoopGen ($\ell = 3$) targets the controller's address-learning logic instead of specific switch properties, the performance can vary with real deployments due to hardware speed, link capacities, and background traffic. For example, if real-world controllers run at a high-speed and stable state, μ_c will be smaller than we observed. As a result, the success probability p_{link} may decrease. These uncertainties may impact the final performance of LoopGen.

Future work: attack variation. Note that LoopGen is not a completely stealthy attack that will never be detected. Its major advantage is the amplification effect that reduces attack cost significantly. In our evaluation, we only use a fixed packet format and one a_{target}, which is actually very simple and rare in practice. Our goal is to use this configuration to evaluate the lower bound of LoopGen's stealthiness. We believe that it can be further improved by changing the packet formats and combing with other DoS techniques. Besides, we think the created forwarding loop may also be used in other ways, instead of only for DoS.

8 Related Work

Topology security in SDN. Hong et al. [64] propose attacks against host tracking and link discovery and propose a targeted defense, TopoGuard. Skowyra et al. [96] make improvements based on the work of Hong et al., proposing Port Amnesia Port Probing attacks, and design corresponding countermeasures, TopoGuard+. Jero et al. [68] show that malicious attackers can conduct identifier binding attacks to make other hosts believe their forged identities, and design SecureBinder to mitigate it. Alimohammadifar et al. [49] propose SPV, which detects

fake links injected by malicious switches. Marin et al. [84] conduct an in-depth analysis of topology attacks and related defenses, pointing out methods to evade these defenses. Kong et al. [76] propose collaboration attacks to enable attackers to inject fake links while bypassing related defenses, and design RPV to detect this attack. Chen et al. [59] propose Marionette, which manipulates LLDP packets and combines reinforcement learning to achieve persistent and covert global topology poisoning.

Flow table security in SDN. DoS attacks against SDN mainly focus on the limited switch flow table capacity and the centralized controller. Kandoi et al. propose high-rate flow table DoS attacks that can rapidly deplete the available resources of flow tables [69]. Cao et al. propose low-rate DoS attacks [55] to circumvent relevant detection mechanisms by reducing the attack rate while maintaining effects. Kong et al. point out that flow table security should strictly satisfy the time requirement [75].

Controller security in SDN. Xu et al. propose a state manipulation attack that exploits harmful race conditions in SDN controllers and design ConGuard, a targeted detection framework [106]. AIM-SDN [61] conducts the security analysis of NMDA-defined datastores in SDN controllers, proposes targeted attacks, and puts forward corresponding mitigation measures. ForenGuard [101] is a fine-grained SDN security forensics and diagnosis system that records runtime activities and their causal dependencies via static analysis and dynamic instrumentation, pinpointing root causes of forwarding anomalies with minor runtime overhead. Cao et al. [56] propose the CrossPath attack, which disrupts SDN control channels via shared links between control and data paths by leveraging adversarial path reconnaissance to identify target paths and low-rate TCP-targeted DoS traffic, causing severe network anomalies like routing blackholes and flow table resets. Cao et al. [57] identify a buffered packet hijacking vulnerability in SDN, design four attacks (cross-app poisoning, policy bypass, TCP handshake disruption, control traffic amplification) to disrupt all SDN layers, and propose a lightweight defense, ConCheck, to mitigate the vulnerability with negligible overhead. EventScope [100] is an automated vulnerability detection tool that identifies cross-plane event-based vulnerabilities in SDN by clustering apps via event use similarity and constructing event flow graphs. D^2C^2 (Data Dependency Creation and Chaining) [103] is a novel SDN attack that abuses SDN protocol custom fields to create malicious data dependencies, and develops SVHunter that detects 18 unknown vulnerabilities across four SDN controllers, enabling remote attacks like arbitrary command execution and data leakage. PICOSDN [99] is a provenance-informed causal analysis tool for SDN attacks, implementing fine-grained data/execution partitioning and a unified control-data plane model on the ONOS controller to enable precise root cause identification. Rashidi et al. [94] identify Network Data Remanence, a side-channel vulnerability in SDN-implemented secret sharing-based multi-path switching schemes (e.g., MSSS on ONOS controller), propose NDR Blind/Planned attacks to exploit it, and present a time-space share distribution countermeasure. ProvGuard [83] is a provenance graph-based detection framework for SDN Control Policy Manipulation attacks.

Defenses using programmable data planes. Poseidon [109] is a DDoS defense system based on programmable switches, which achieves efficient protection against DDoS attacks through modular strategy abstraction, resource orchestration, and runtime state replication. Poise [70] is a programmable in-network security system for BYOD scenarios, enabling line-rate enforcement of context-aware policies via a high-level declarative language (compiled to P4), outperforming OpenFlow-based SDN in agility and resilience to control plane saturation attacks. Ripple [105] is a programmable and decentralized link-flooding defense system based on programmable switches, constructing a network-wide defense panorama through distributed synchronization protocols, and achieving fast response to adaptive/rolling attacks with low latency and hardware overhead. FlowLens [54] is a programmable switch-based system that enables efficient ML-based flow classification (covert channel detection, website fingerprinting, botnet detection) with minimal accuracy loss and significant memory savings. Jaqen [82] is a switch-native DDoS defense system on programmable switches, achieving high line-rate mitigation with rapid response to dynamic attacks. Mew [114] is a resource-efficient and runtime-adaptable link-flooding defense system on programmable switches, enabling effective mitigation of large-scale/dynamic link flooding attacks with low overhead and zero service interruption. NetBeacon [112] proposes an intelligent data plane design for programmable switches, achieving line-speed and high-accuracy traffic analysis while reducing table entry consumption. Cerberus [113] is an efficient and adaptable in-network monitoring system for programmable switches, which supports concurrent high-volume INM tasks (e.g., multi-vector DDoS defense) with low overhead and no service interruption. ZAPDOS [87] is a prefix-level volumetric DDoS signature detection system for edge networks, leveraging programmable switches to monitor aggregate prefix features and achieving high accuracy with minimal hardware resource overhead. P4Control [52] is a line-rate cross-host attack prevention system that integrates programmable switches (P4) and eBPF, enabling in-network decentralized information flow control via label propagation, multi-table policy enforcement, and the expressive NETCL policy language, achieving real-time defense with minimal network/host overhead and facilitating zero-trust architecture. SmartCookie [108] is a split-proxy defense system against large-scale SYN floods, leveraging programmable switches (P4-implemented) for secure HalfSipHash-based SYN cookie checks and eBPF-based server agents for sequence number translation. ISDC [86] is a distributed in-network serverless data collection system using programmable data planes, enhancing flow cov-

erage and feature accuracy for ML/DL-based network attack detection while mitigating model self-poisoning from fragmented measurements. SketchFeature [73] is a sketch-based per-flow 3rd-order feature extractor for programmable switch data planes, enabling High-resolution, All-flow, Full-range measurement via sketch virtualization and Bloom filter-based membership testing. Lemon [102] is a routing-oblivious network-wide DDoS detection system that employs a multi-layer Lemon sketch, enabling accurate per-flow estimation and robust DDoS detection on programmable switches. SISTAR [65] is an efficient DDoS detection and mitigation framework for programmable data planes, integrating optimized DT-CTS/RF-CTS models, distributed deployment across switches, and an alert pushback mechanism, achieving high accuracy and low hardware overhead. SecureINT supports a line-rate encryption and hash function on programmable switches, supporting hop-by-hop packet protection [78, 79].

Fuzzing on SDN. DELTA is an automated SDN security assessment framework that integrates protocol-aware blackbox fuzzing, supports three types of agents (application, channel, and host) [80]. Intender is a semantically-aware black-box fuzzing framework for intent-based networking, which discovers 12 unknown bugs (11 CVE-assigned security-critical vulnerabilities) in ONOS [72]. Chimera is a multi-plane fuzzing framework for P4 network infrastructure that leverages concolic execution, parser-aware packet mutation, and header-guided rule generation to simultaneously test control and data planes [71].

9 Conclusion

This paper identifies a security weakness in SDN address learning: the controller lacks global security checks when updating the address-location mapping, causing individual, normal events to collectively form an attack. Leveraging it, we present LoopGen, a host-only attack in which adversaries strategically inject a small, ordered sequence of packets to drive benign-looking learning updates that collectively cause a data-plane forwarding loop. LoopGen is a cost-effective DoS amplifier with a low setup cost. Our evaluation across multiple open-source controllers, real-world topologies, and heterogeneous switches demonstrates that LoopGen is broadly feasible, can induce substantial latency inflation and throughput degradation, and can achieve comparable effect to conventional non-looping DoS with orders-of-magnitude fewer attack packets. Moreover, it can evade several representative SDN defenses. Then, we discuss defenses and outline the practical trade-offs of them across different SDN scenarios. Finally, we conduct an in-depth discussion of limitations and future work.

Acknowledgments

We are grateful to our shepherd, the anonymous reviewers, and the anonymous artifact reviewers for their valuable guidance and insightful comments. This research is supported by the National Key R&D Program of China (Grant No. 2024YFB2906503) and the National Natural Science Foundation of China (Grant No. 62376246).

Ethical Considerations

We acknowledge the ethical concerns that may arise from this study, particularly regarding the potential for misuse. To address these concerns responsibly, we adhere to several key principles.

Stakeholders and Impacts

Network operators and organizations deploying SDN. They may be affected because the vulnerability can be exploited to degrade network performance, potentially causing operational disruption and financial loss.

End users whose traffic traverses affected networks. They may experience degraded service quality and loss of connectivity if forwarding loops are triggered.

SDN controller/switch developers and open-source communities. Our findings apply across multiple open-source controllers (e.g., Ryu, ONOS, POX); maintainers may need to revise the address learning logic and add safety checks, which can impose engineering burden but improve robustness.

Researchers and the broader security community. The community benefits from understanding an under-protected design pattern in SDN and from the concrete mitigation directions we propose, while facing dual-use risk if the technique is misused.

Attackers and other malicious actors. Adversaries may use the technique as an amplification primitive that reduces attack cost and potentially increases stealthiness under some existing defenses, which raises misuse concerns.

Mitigations

Claim. Our work does not involve human subjects, deception, or the collection of personal data. We conducted evaluation in controlled, isolated test environments using open-source controllers and hardware/software switches, and relied on public traces for background traffic, rather than probing or attacking real-world networks.

Mitigations proposed. To improve safety for SDN, we propose two countermeasure directions: (1) controller-side loop detection, and (2) time constraints on rapid consecutive location updates. We also discuss practical trade-offs (e.g., overhead, reliance on accurate topology information, and potential

interference with legitimate mobility) to support responsible deployment decisions.

Responsible risks exposure. We believe that publicly disclosing and rigorously analyzing security weaknesses in SDN is crucial to building more resilient and secure networks. Our work is intended for the research community and network developers, with the goal of increasing awareness of potential threats and helping guide the development of effective defenses. As a result, **we are in the process of applying for CVE identifiers** for the identified vulnerability in the SDN address-learning mechanism. Since the CVE assignment timeline is outside our control, the identifiers may not be issued in time; in that case, **we will update our project page[5] with the CVE IDs promptly once they are assigned**.

Commitment Against Misuse. We do not condone, encourage, or assist in deploying attacks against SDN for harmful or malicious use. Instead, by demonstrating how the address learning service can be used to create forwarding loops, our work aims to help the community better understand this risk and to motivate stronger detection and mitigation strategies, ultimately improving the security and reliability of SDN.

Open Science

The code of LoopGen. Our code can be accessed via the DOI https://doi.org/10.5281/zenodo.20307092 or our GitHub https://github.com/JiangYingEr/LoopGen.

Other public code/data/tool used in this study. We have correctly cited the public defenses, network traces, topologies, and tools used in this paper. We have confirmed that each reference either directs to a public repository or a paper that contains the repository link.

References

[1] 200,000 cisco network switches reportedly hacked. https://www.bankinfosecurity.com/200000-cisco-network-switches-reportedly-hacked-a-10788, 2024.

[2] Adopting a software-defined network fabric in aws. https://pages.awscloud.com/rs/112-TZM-766/images/SANS-and-AWS-Marketplace_SDN_Whitepaper.pdf, 2024.

[3] Bmv2 switch. http://bmv2.org/index.html, 2024.

[4] Data center design models. https://learningnetwork.cisco.com/s/question/0D53i00000KsqtxCAB/data-center-design-models, 2024.

[5] Hackers compromised two large data centers in asia and leaked major tech giants' login credentials. https://www.cpomagazine.com/cyber-security/hackers-compromised-two-large-data-centers-in-asia-and-leaked-major-tech-giants-login-credentials/, 2024.

[6] How microsoft builds its fast and reliable global network. https://azure.microsoft.com/en-us/blog/how-microsoft-builds-its-fast-and-reliable-global-network/, 2024.

[7] Huawei software defined network (sdn) solution. https://actfornet.com/huawei-cloud/sdn.html, 2024.

[8] Intel tofino. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html, 2024.

[9] Mininet. https://mininet.org/, 2024.

[10] Nmap. https://nmap.org/, 2024.

[11] Onos controller. https://opennetworking.org/onos/, 2024.

[12] P4 tutorial. https://github.com/p4lang/tutorials, 2024.

[13] Pox controller. https://noxrepo.github.io/pox-doc/html/, 2024.

[14] Ryu controller. https://ryu-sdn.org/, 2024.

[15] Scapy. https://scapy.net/, 2024.

[16] South korean web host pays historical ransom of $1m. https://www.bitdefender.co.uk/blog/businessinsights/hystorical-ransom-ransomware/, 2024.

[17] Tcpreplay. https://tcpreplay.appneta.com/, 2024.

[18] Cisco catalyst sd-wan solution overview. https://www.cisco.com/c/en/us/solutions/collateral/enterprise-networks/sd-wan/nb-06-sd-wan-sol-overview-cte-en.html, 2025.

[19] Critical capabilities for sd-wan. https://www.gartner.com/reviews/market/sd-wan, 2025.

[20] Cve-2025-22280. https://www.cve.org/CVERecord?id=CVE-2025-22280, 2025.

[21] Cve-2025-29914. https://www.cve.org/CVERecord?id=CVE-2025-29914, 2025.

[5] https://github.com/JiangYingEr/LoopGen

[22] Cve-2025-30259. https://www.cve.org/CVERecord?id=CVE-2025-30259, 2025.

[23] Cve-2025-33053. https://www.cve.org/CVERecord?id=CVE-2025-33053, 2025.

[24] Cve-2025-39778. https://www.cve.org/CVERecord?id=CVE-2025-39778, 2025.

[25] Cve-2025-39930. https://www.cve.org/CVERecord?id=CVE-2025-39930, 2025.

[26] Cve-2025-39989. https://www.cve.org/CVERecord?id=CVE-2025-39989, 2025.

[27] Cve-2025-40014. https://www.cve.org/CVERecord?id=CVE-2025-40014, 2025.

[28] Cve-2025-48866. https://www.cve.org/CVERecord?id=CVE-2025-48866, 2025.

[29] Cve-2025-52454. https://www.cve.org/CVERecord?id=CVE-2025-52454, 2025.

[30] Cve-2025-53013. https://www.cve.org/CVERecord?id=CVE-2025-53013, 2025.

[31] Cve-2025-54313. https://www.cve.org/CVERecord?id=CVE-2025-54313, 2025.

[32] Cve-2025-6018. https://www.cve.org/CVERecord?id=CVE-2025-6018, 2025.

[33] Cve-2025-6443. https://www.cve.org/CVERecord?id=CVE-2025-6443, 2025.

[34] Cve-2025-6704. https://www.cve.org/CVERecord?id=CVE-2025-6704, 2025.

[35] Cve-2025-7145. https://www.cve.org/CVERecord?id=CVE-2025-7145, 2025.

[36] Cve-2025-7382. https://www.cve.org/CVERecord?id=CVE-2025-7382, 2025.

[37] Cve-2025-7624. https://www.cve.org/CVERecord?id=CVE-2025-7624, 2025.

[38] Cve-2025-7772. https://www.cve.org/CVERecord?id=CVE-2025-7772, 2025.

[39] Cve-2025-7921. https://www.cve.org/CVERecord?id=CVE-2025-7921, 2025.

[40] Cve-2025-8259. https://www.cve.org/CVERecord?id=CVE-2025-8259, 2025.

[41] distfit 2.0.1. https://pypi.org/project/distfit/, 2025.

[42] iperf. https://iperf.fr/, 2025.

[43] Mawi working group traffic archive. https://mawi.wide.ad.jp/mawi/, 2025.

[44] Scaling data centers with fabricpath and the cisco fabricpath switching system. https://www.cisco.com/c/dam/global/en_id/solutions/midmarket/assets/pdf/white_paper_c11605488.pdf, 2025.

[45] Sd-wan reviews and ratings. https://www.gartner.com/reviews/market/sd-wan, 2025.

[46] Software defined networking (sdn) managed by on-premises tools in azure local. https://learn.microsoft.com/en-us/azure/azure-local/concepts/software-defined-networking-23h2?view=azloc-2512, 2025.

[47] Software-defined wide area network (sd-wan) market worth $21.67 billion by 2030. https://www.marketsandmarkets.com/PressReleases/software-defined-wan.asp, 2025.

[48] Vmware sd-wan enables managed wan for service providers. https://www.vmware.com/docs/solution-overview-service-providers, 2025.

[49] Amir Alimohammadifar et al. Stealthy probing-based verification (spv): An active approach to defending software defined networks against topology poisoning attacks. In *ESORICS*, pages 463–484, 2018.

[50] J. Alvarez-Horcajo, D. Lopez-Pajares, Jose M. Arco, J. Antonio Carral, and I. Martinez-Yelmo. Tcp-path: Improving load balance by network exploration. In *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, pages 1–6, 2017.

[51] Rashid Amin et al. Hybrid sdn networks: A survey of existing approaches. *IEEE Communications Surveys & Tutorials*, 20(4):3259–3306, 2018.

[52] Osama Bajaber, Bo Ji, and Peng Gao. P4control: Line-rate cross-host attack prevention via in-network information flow control enabled by programmable switches and ebpf. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4610–4628. IEEE, 2024.

[53] Fetia Bannour et al. Distributed sdn control: Survey, taxonomy, and challenges. *IEEE Communications Surveys & Tutorials*, 20(1):333–354, 2017.

[54] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. Flowlens: Enabling efficient flow classification for ml-based network security applications. In *NDSS*, 2021.

[55] Jiahao Cao et al. Disrupting sdn via the data plane: A low-rate flow table overflow attack. In *SecureComm*, pages 356–376, 2018.

[56] Jiahao Cao, Qi Li, Renjie Xie, Kun Sun, Guofei Gu, Mingwei Xu, and Yuan Yang. The {CrossPath} attack: Disrupting the {SDN} control channel via shared links. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 19–36, 2019.

[57] 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.

[58] Jasmeen Kaur Chahal et al. Ddos attacks & defense mechanisms in sdn-enabled cloud: Taxonomy, review and research challenges. *Computer Science Review*, 53:100644, 2024.

[59] Mingming Chen et al. Manipulating openflow link discovery packet forwarding for topology poisoning. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3704–3718, 2024.

[60] Mohan Dhawan et al. Sphinx: detecting security attacks in software-defined networks. In *Ndss*, volume 15, pages 8–11, 2015.

[61] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. Aim-sdn: Attacking information mismanagement in sdn-datastores. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 664–676, 2018.

[62] Klaus-Tycho Foerster, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Loop-free route updates for software-defined networks. *Ieee/acm Transactions on Networking*, 26(1):328–341, 2017.

[63] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.

[64] Sungmin Hong et al. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS*, volume 15, pages 8–11, 2015.

[65] Junjie Hu, Feng Guo, Qihang Zhou, Yixin Zhang, Zibo Gao, Yinglong Han, and Zhiqiang Lv. Sistar: An efficient ddos detection and mitigation framework utilizing programmable data planes. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 4589–4603, 2025.

[66] Lilima Jain et al. Sectopo: Efficient hybrid model for detecting lldp topology poisoning attack in programmable data plane. *Computer Networks*, page 111510, 2025.

[67] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[68] Samuel Jero et al. Identifier binding attacks and defenses in software-defined networks. In *USENIX Security*, pages 415–432, 2017.

[69] Rajat Kandoi and Markku Antikainen. Denial-of-service attacks in openflow sdn networks. In *IM*, pages 1322–1326, 2015.

[70] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable {In-Network} security for context-aware {BYOD} policies. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 595–612, 2020.

[71] Jiwon Kim, Dave Jing Tian, and Benjamin E Ujcich. Chimera: Fuzzing p4 network infrastructure for multi-plane bug detection and vulnerability discovery. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3088–3106. IEEE, 2025.

[72] 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.

[73] Sian Kim, Seyed Mohammad Mehdi Mirnajafizadeh, Bara Kim, Rhongho Jang, and DaeHun Nyang. Sketch-feature: High-quality per-flow feature extractor towards security-aware data plane. In *Proc. of ISOC NDSS*, 2025.

[74] Simon Knight et al. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[75] Dezhang Kong, Xiang Chen, Chunming Wu, et al. Rdefender: A lightweight and robust defense against flow table overflow attacks in sdn. *IEEE Transactions on*

Information Forensics and Security, 19:9436–9451, 2024.

[76] Dezhang Kong et al. Combination attacks and defenses on sdn topology discovery. *IEEE/ACM Transactions on Networking*, 31(2):904–919, 2022.

[77] Dezhang Kong et al. Tableguard: A novel security mechanism against flow table overflow attacks in SDN. In *GLOBECOM*, pages 4167–4172, 2022.

[78] Dezhang Kong et al. In-band network telemetry manipulation attacks and countermeasures in programmable networks. In *IWQoS*, pages 1–10, 2023.

[79] Dezhang Kong et al. Toward security-enhanced in-band network telemetry in programmable networks. *IEEE Transactions on Network and Service Management*, 2024.

[80] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, and Vinod Yegneswaran. Delta: A security assessment framework for software-defined networks. 2017.

[81] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. {IMap}: Fast and scalable {In-Network} scanning with programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 667–681, 2022.

[82] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3829–3846, 2021.

[83] Ziwen Liu, Jian Mao, Jun Zeng, Jiawei Li, Qixiao Lin, Jiahao Liu, Jianwei Zhuge, and Zhenkai Liang. Provguard: Detecting sdn control policy manipulation via contextual semantics of provenance graphs. In *NDSS*, 2025.

[84] Eduard Marin et al. An in-depth look into sdn topology discovery mechanisms: Novel attacks and practical countermeasures. In *CCS*, pages 1101–1114, 2019.

[85] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[86] Seyed Mohammad Mehdi Mirnajafizadeh, Ashwin Raam Sethuram, David Mohaisen, DaeHun Nyang, and Rhongho Jang. Enhancing network attack detection with distributed and {In-Network} data collection system. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5161–5178, 2024.

[87] Chris Misa, Ramakrishnan Durairajan, Arpit Gupta, Reza Rejaie, and Walter Willinger. Leveraging prefix structure to detect volumetric ddos attack signatures with programmable switches. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4535–4553. IEEE, 2024.

[88] Saralees Nadarajah. A generalized normal distribution. *Journal of Applied Statistics*, 32(7):685–694, 2005.

[89] Bruno Astuto A Nunes et al. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications surveys & tutorials*, 16(3):1617–1634, 2014.

[90] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[91] Md Naimul Pathan, Maisha Muntaha, Selina Sharmin, Sajeeb Saha, Md Ashraf Uddin, Fernaz Narin Nur, and Sunil Aryal. Priority based energy and load aware routing algorithms for sdn enabled data center network. *Computer Networks*, 240:110166, 2024.

[92] Xinglong Pei et al. Enabling efficient routing for traffic engineering in sdn with deep reinforcement learning. *Computer Networks*, 241:110220, 2024.

[93] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 66–85, 2022.

[94] Leila Rashidi, Daniel Kostecki, Alexander James, Anthony Peterson, Majid Ghaderi, Samuel Jero, Cristina Nita-Rotaru, Hamed Okhravi, and Reihaneh Safavi-Naini. More than a fair share: Network data remanence attacks against secret sharing-based schemes. In *NDSS*, 2021.

[95] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter

rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[96] Richard Skowyra et al. Effective topology tampering attacks and defenses in software-defined networks. In *DSN*, pages 374–385, 2018.

[97] Xun Song, Jiaqi Zheng, Hao Qian, Shiju Zhao, Hongxuan Zhang, Xuntao Pan, and Guihai Chen. Couper: Memory-efficient cardinality estimation under unbalanced distribution. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 2753–2765. IEEE, 2023.

[98] 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.

[99] 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.

[100] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Steven R Gomez, Adam Bates, William H Sanders, and Hamed Okhravi. Automated discovery of cross-plane event-based vulnerabilities in software-defined networking. In *Network and Distributed System Security Symposium*, 2020.

[101] Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. Towards fine-grained network security forensics and diagnosis in the sdn era. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 3–16, 2018.

[102] Wenhao Wu, Zhenyu Li, Xilai Liu, Zhaohua Wang, Heng Pan, Guangxing Zhang, and Gaogang Xie. Lemon:{Network-Wide}{DDoS} detection with {Routing-Oblivious}{Per-Flow} measurement. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3577–3594, 2025.

[103] Feng Xiao, Jinquan Zhang, Jianwei Huang, Guofei Gu, Dinghao Wu, and Peng Liu. Unexpected data dependency creation and chaining: A new attack to sdn. In *2020 IEEE symposium on security and privacy (SP)*, pages 1512–1526. IEEE, 2020.

[104] Guorui Xie, Qing Li, Chupeng Cui, Peican Zhu, Dan Zhao, Wanxin Shi, Zhuyun Qi, Yong Jiang, and Xi Xiao. Soter: Deep learning enhanced in-network attack detection based on programmable switches. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 225–236. IEEE, 2022.

[105] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A programmable, decentralized {Link-Flooding} defense against adaptive adversaries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3865–3881, 2021.

[106] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the brain: Races in the {SDN} control plane. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 451–468, 2017.

[107] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445, 2017.

[108] Sophia Yoo, Xiaoqi Chen, and Jennifer Rexford. {SmartCookie}: Blocking {Large-Scale}{SYN} floods with a {Split-Proxy} defense on programmable data planes. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 217–234, 2024.

[109] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[110] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Iisy: Hybrid in-network classification using programmable switches. *IEEE/ACM Transactions on Networking*, 32(3):2555–2570, 2024.

[111] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Planter: Rapid prototyping of in-network machine learning inference. *ACM SIGCOMM Computer Communication Review*, 54(1):2–21, 2024.

[112] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An efficient design of intelligent network data plane. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6203–6220, 2023.

[113] Huancheng Zhou and Guofei Gu. Cerberus: Enabling efficient and effective in-network monitoring on programmable switches. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4424–4439. IEEE, 2024.

[114] Huancheng Zhou, Sungmin Hong, Yangyang Liu, Xiapu Luo, Weichao Li, and Guofei Gu. Mew: Enabling large-scale and dynamic link-flooding defenses on programmable switches. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3178–3192. IEEE, 2023.

[115] Xu Zhou et al. Deep reinforcement learning-based resource scheduling for energy optimization and load balancing in sdn-driven edge computing. *Computer Communications*, 226:107925, 2024.

A Theoretical Analysis Details

A.1 Data Collection

As discussed in Section 4.4, the success of LoopGen depends on three delays: the host-related delay Δt_h, the switch-related delay Δt_l, and the controller-related delay Δt_c. To figure out their distribution, we collect independent samples for each delay type via repeated trials. Each variable was sampled 1,000 times. The environment is the same as Section 5.

A.1.1 Host-related delay Δt_h.

Δt_h captures the host-side sending delay, i.e., the compound delay from when the user-space program initiates a send operation until the packet is emitted toward the ingress switch. We measure Δt_h on the compromised host using Linux kernel's SO_TIMESTAMPING.

A.1.2 Switch-related delay Δt_l

Δt_l models the per-hop data-plane delay contributed by switches and links, including switch processing/queuing and transmission to the next hop. We first establish a stable forwarding loop of length $\ell = 3$ in the data plane. We then inject a probe packet into the loop and timestamp two consecutive receptions of this packet at the same switch port (i.e., the packet reappears after completing one full loop traversal). Let t_1 and t_2 denote these two adjacent reception timestamps. Since the packet traverses exactly three data-plane hops per loop cycle, we compute Δt_l as $\frac{t_2-t_1}{3}$.

A.1.3 Controller-related delay Δt_c

Δt_c denotes the controller-side handling delay for a reactive event. We measure Δt_c by monitoring the controller's control-plane message stream and recording the time gap between a Packet-In and its adjacent Packet-Out response. For each trial, we compute $t_{\text{out}} - t_{\text{in}}$, where t_{in} is the timestamp when the controller receives the Packet-In and t_{out} is the timestamp when the controller sends the corresponding Packet-Out. To reduce ambiguity in matching Packet-In and Packet-Out messages (e.g., due to concurrent triggers), the host sends probes at a fixed, large interval, and each probe uses a fake source address to intentionally trigger a Packet-In. This pacing ensures that consecutive control-plane events are well separated in time and thus can be reliably paired for Δt_c measurement.

A.2 Distribution Fitting Configuration

The candidate distributions are: Normal Distribution, Student's t-distribution, Generalized Extreme Value Distribution, Gamma Distribution, Lognormal Distribution, Beta Distribution, Log-Gamma Distribution, Exponential Distribution, and Continuous Uniform Distribution. To quantify how well each fitted candidate distribution matches the empirical data, we select the Kolmogorov-Smirnov (KS) statistic. We choose the K-S test because it does not rely on assumptions about the distribution shape, is applicable to any continuous distribution, and is sensitive to changes in location and shape.

Note that the standard KS test assumes the null distribution is fully specified *a priori*; in our setting, parameters are estimated from the same data, which can cause bias if we use the standard KS reference distribution. To address this, we apply a parametric bootstrap procedure to obtain calibrated p-values for each candidate F_m. The round is set to 100.

A.3 Derivation of Equation 9

For simplification, we denote $X \sim \mathcal{N}(\mu_X, \sigma_X)$. X can be transformed to the standard normal distribution:

$$Z = \frac{X - \mu_X}{\sigma_X} \sim \mathcal{N}(0,1).$$

Then

$$\begin{aligned} p_{\text{link}} &= \Pr(X > 0) \\ &= \Pr\left(\frac{X-\mu_X}{\sigma_X} > \frac{0-\mu_X}{\sigma_X}\right) \\ &= 1 - \Phi\left(\frac{-\mu_X}{\sigma_X}\right) \\ &= \Phi\left(\frac{\mu_X}{\sigma_X}\right), \end{aligned}$$

where $\Phi(\cdot)$ is the CDF of the standard normal distribution, and the last step uses the symmetry feature of the standard normal distribution: $1 - \Phi(-a) = \Phi(a)$. Substituting $\mu_X = 2\mu_l + \mu_c$ and $\sigma_X = \sqrt{2\sigma_l^2 + 2\sigma_h^2 + \sigma_c^2}$:

$$p_{\text{link}} = \Phi\left(\frac{2\mu_l + \mu_c}{\sqrt{2\sigma_l^2 + 2\sigma_h^2 + \sigma_c^2}}\right).$$

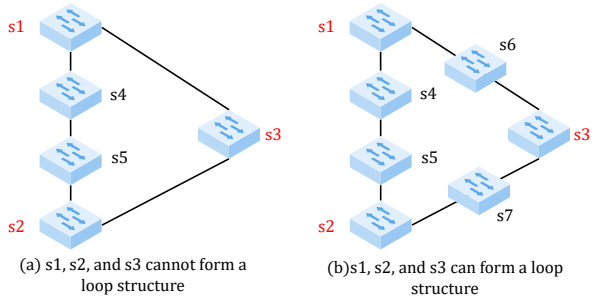


Figure 12: Shortest path is the criterion for loop structures.

B A special form $\ell = 2$

When $\ell = 2$, LoopGen becomes a *two-switch oscillation* between two adjacent switches. Consider two adjacent switches $s_1, s_2 \in S$ connected by a bidirectional link. Let $p_{1\to2} \in P(s_1)$ denote the port on s_1 leading to s_2, and $p_{2\to1} \in P(s_2)$ denote the port on s_2 leading to s_1. The adversary still selects an unused address $a_{target} \in A$ and aims to steer the controller's address-learning database into the following state:

$$D(s_1, a_{target}) = p_{1\to2}, D(s_2, a_{target}) = p_{2\to1}. \quad (11)$$

Under Equation (11), any packet with destination $a_{dst} = a_{target}$ arriving at s_1 will be forwarded to $p_{1\to2}$ (towards s_2), and upon arriving at s_2 will be forwarded to $p_{2\to1}$ (back towards s_1), thus bouncing between s_1 and s_2 repeatedly at the data plane. Compared to the $\ell \geq 3$ case where the adversary must identify a directed loop L among multiple switches, the $\ell = 2$ oscillation lowers the condition significantly. However, although LoopGen is only governed by the controller's address learning logic instead of the underlying switches' properties, the $\ell = 2$ oscillation becomes an exception: its success depends on whether the underlying switch allows forwarding a packet back to the in_port. Some switches, especially traditional switches, may enforce a policy that rejects actions that output a packet to the same port it arrived on.

C Detect Loop Structures in Topology

In Section 5.2.1, we label a set of ℓ switches as *loop-feasible* if it contains at least one triple of switches that can form a loop structure in the network topology. Concretely, for any candidate triple (a, b, c), we test whether there exist three *disjoint shortest paths* between each two-switch pair: $a \leftrightarrow b$, $b \leftrightarrow c$, and $c \leftrightarrow a$. It is *not* sufficient that three switches only form a triangle topology.

Figure 12 illustrates this distinction. In case (a), although there exists a physical cycle that goes through (s_1, s_2, s_3) via the chain $(s_1 - s_4 - s_5 - s_2 - s_3 - s_1)$, this cycle is not likely to become a real routing path. This is because the path $s_1 - s_3 - s_2$ is shorter than $s_1 - s_4 - s_5 - s_2$. Consequently, (s_1, s_2, s_3) is labeled *not loop-feasible*. In contrast, in case (b), the three shortest paths are disjoint, so we label the sets containing (s_1, s_2, s_3) as loop-feasible.