The LOFT Attack: Overflowing SDN Flow Tables at a Low Rate

Jiahao Cao, Mingwei Xu, *Senior Member, IEEE*, Qi Li, *Senior Member, IEEE*, Kun Sun, *Member, IEEE*, and Yuan Yang, *Member, IEEE*

***Abstract*— The emerging Software-Defined Networking (SDN) is being adopted by data centers and cloud service providers to enable flexible control. Meanwhile, the current SDN design brings new vulnerabilities. In this paper, we explore a stealthy attack that uses a minimum rate of attack packets to disrupt SDN data plane. To achieve this, we propose the LOFT attack that computes the lower bound of attack rate to overflow flow tables based on the inferred network configurations. Particularly, each attack packet always triggers or maintains consumption of one flow rule. LOFT can ensure the attack effect under various network configurations while reducing the possibility of being captured. We demonstrate its feasibility and effectiveness in a real SDN testbed consisting of commercial hardware switches. The experimental results show that LOFT incurs significant network performance degradation and potential network DoS at an attack rate of only tens of Kbps. To defeat the attack, we develop a data-to-control plane collaborative defense system named LOFTGuard, which is lightweight and transparent to SDN applications. Evaluations show that LOFTGuard effectively protects SDN against the attack and introduces a small overhead.**

***Index Terms*— Software-defined networking, low-rate attack, flow table overflow, defense system.**

I. Introduction

BY DECOUPLING the control and data planes, Software-Defined Networking (SDN) emerges as a promising network architecture design that provides the network with great programmability, flexible control, and agile management. Google data centers [1] and Microsoft Azure cloud platform [2] have deployed SDN to innovate their networks. A large number of SDN applications have been developed to enable various network functionalities, such as dynamic flow scheduling [3], holistic network monitoring and management [4], and security function deployment in large networks [5].

Unfortunately, the SDN design itself has serious security problems. Particularly, the SDN data plane (or SDN switch) is vulnerable to flow table overflow. In SDN, switches are "dumb", i.e., if a flow matches no rules in the switch flow table, the switch will generate packet-in messages to query a logically centralized controller for a new flow rule. However, an attacker may abuse such mechanism to consume the flow table by sending crafted packets to trigger new rule installation. More importantly, modern SDN-enabled hardware switches usually support a small number of flow rules, e.g., thousands of rules [6], [7], [8], which are stored in power-hungry and expensive Ternary Content Addressable Memory (TCAM) to achieve high lookup performance [6], [9]. Thus, the limited storage space of TCAM can be easily overflowed.

To effectively overflow SDN switches, existing attacks [10], [11], [12] normally generate a large number of short frequent flows per second. Since each attack flow only has one or two packets but the total rate of attack packets is high, those attacks can be easily captured and mitigated by existing defense solutions [10], [11], [13], [14]. Although a few studies [15], [16] attempted to reduce the rate of attack packets, they do not consider detailed configurations of flow rules, e.g., the lifetime of flow rules, and thus can fail in real SDN environments. Moreover, some attacks [16] require compromising thousands of hosts in SDN, which incurs expensive costs for an attacker.

In this paper, we aim to design a low-rate attack that can efficiently overflow flow tables of SDN switches by generating a minimum rate of attack packets. Moreover, it can be easily launched by one compromised host in real SDN environments with various network settings. However, designing such an attack is challenging. To decrease the attack rate as much as possible and efficiently consume the flow table, an attacker should carefully craft packets so that each of them can trigger new rule installation. It requires the attacker to know precisely what packets will trigger new rule installation. However, the rule installation logic is decided by the SDN controller and usually cannot be accessed by the attacker. Besides, flow rules will be removed after a certain time due to the timeout settings. Understanding the timeout settings is necessary for the attacker to choose the best attack strategy and the minimum attack rate.

To address the above challenges, we present a two-phase *LO*w-rate *F*low *T*able overflow attack called *LOFT*, which consists of a probing phase and an attacking phase. In the probing phase, it aims to accurately infer network configurations of flow rules by generating a small number of probing packets. These network configurations include the match fields along with their bitmasks that indicate what packets will trigger new rule installation and the timeouts that define the lifetime of the rules. The key insight behind our probing to

Manuscript received 21 May 2019; revised 31 March 2021, 24 January 2022, and 29 July 2022; accepted 27 October 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Guan. Date of publication 1 December 2022; date of current version 16 June 2023. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB2901303; in part by the NSFC under Grant 62202260, Grant 61832013, Grant 62221003, Grant 62132011, and Grant 61872209; in part by the NSF under Grant CNS-1815650; in part by the Office of Naval Research (ONR) under Grant N00014-20-1-2407; and in part by the Shuimu Tsinghua Scholar Program. *(Corresponding authors: Mingwei Xu; Qi Li; Yuan Yang.)*

Jiahao Cao, Mingwei Xu, Qi Li, and Yuan Yang are with the Department of Computer Science and Technology, Institute of Network Science and Technology, and the Beijing National Research Center for the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Quan Cheng Laboratory, Jinan, Shandong 250098, China, also with the Zhongguancun Laboratory, Beijing 100094, China, and also with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: caojh2021@tsinghua.edu.cn; xumw@tsinghua.edu.cn; qli01@tsinghua.edu.cn; yangyuan_thu@tsinghua.edu.cn).

Kun Sun is with the Department of Information Sciences and Technology, George Mason University, Fairfax, VA 22030 USA (e-mail: ksun3@gmu.edu).

Digital Object Identifier 10.1109/TNET.2022.3225211

infer configurations is that there exist remarkable forwarding delays for packets that cannot match any existing flow rules in the switches due to the separation of control and data planes in SDN. Thus, by measuring round-trip times (RTTs) of customized probing packets, an attacker can accurately infer the settings of the flow rules. In the attacking phase, LOFT generates low-rate attack traffic to overflow flow tables according to the inferred network configurations. It crafts different packets with specific packet headers so that each packet can trigger new flow rule installation. Meanwhile, based on the detailed timeout configurations, it computes the minimum packet rate to keep flow tables overflowed over time.

We evaluate the feasibility and effectiveness of LOFT attack in a real SDN testbed that consists of commercial hardware switches and a popular open source controller. We replay the real traffic trace from CAIDA [17] as background traffic in our experiments. The experimental results show that LOFT can accurately infer flow rule configurations using a small number of probing packets. In particular, by generating less than 10 probing packets per second, it can achieve more than 90% accuracy on probing the detailed timeout settings. In the attacking phase, the LOFT attack can successfully overflow the flow table with an attack rate of around 50 Kbps, which incurs a 53% degradation of network throughput. Besides, it introduces 10x forwarding delays for new flows. The results show that the LOFT attack is powerful and cost-effective.

Several countermeasures have been presented to protect SDN against flow table overflow attacks. Most defense solutions are *passive*, either sharing flow tables among switches to make SDN more resilient to table overflow [18], [19], [20] or adapting customized flow rule eviction strategies to reduce the impact of the attack [10], [16], [21]. However, such passive defenses cannot completely defend the LOFT attack since they lack the ability to identify and block the malicious flows. Malicious flows can still compete with benign flows to occupy scarce flow table resources. Besides, the passive defenses work only when the table is full, which fail to prevent malicious flows occupying flow rules in time. A few studies [13], [14], [22], [23], [24] adopt *active* defenses by detecting attacks and blocking identified malicious flows. Though they can effectively defeat the attacks that generate short frequent flows with a high rate, they cannot prevent the LOFT attack that crafts long infrequent flows with a low rate.

To defend against the attack, we present LOFTGuard, an active data-to-control plane collaborative security system. It can effectively prevent the attack from occupying flow table (or to be precise, TCAM) in time as well as accurately identify and block malicious flows. In the SDN data plane, LOFTGuard exploits the general processing powers of SDN switches, i.e., the CPU and Random-Access Memory (RAM), to create *cache region* of flow rules. Such a region is a software-implemented flow table compared to TCAM-implemented flow table. LOFTGuard dynamically migrates flow rules between the cache region and the TCAM region based on the liveness and rate of flows to effectively throttle the TCAM occupation of malicious flow rules. Meanwhile, LOFTGuard enforces an application in the SDN control plane to identify attack flows based on the features extracted from flow statistics. Once attack flows are identified, the application informs the data plane to remove their rules and block malicious requests of rule installation.

We implement a prototype of LOFTGuard on commercial hardware switches and the Floodlight controller. Experimental results show it effectively protects SDN against the LOFT attack. With LOFTGuard, TCAM space consumed by malicious flows is reduced from above 80% to under 9% and the degradation ratio of network throughput can be maintained to below 3%. Moreover, it introduces a small overhead, i.e., less than 5% RAM usage and less than 16% switch CPU usage.

In summary, this paper makes the following contributions:

- We propose a low-rate flow table overflow attack called LOFT that effectively degrades the network performance.
- We develop probing algorithms that accurately infer network configurations of flow rules and compute the minimum attack rate to successfully launch LOFT.
- We conduct extensive experiments in a real SDN testbed consisting of commercial hardware switches to verify the feasibility and effectiveness of the LOFT attack.
- We design LOFTGuard, which is a lightweight data-to-control plane collaborative security framework to effectively defend against the LOFT attack.
- We implement a prototype of LOFTGuard on commercial hardware switches and the Floodlight controller. And we systematically evaluate its effectiveness and overhead in a real SDN testbed.

This paper is an extended version of the work [25]. Compared to the previous version, we evaluate the effectiveness and feasibility of the LOFT attack in a more practical network environment, i.e., real CAIDA traffic trace is replayed. A number of experiments are added to explore the proportion of flow rules in TCAM, the degradation ratio of network throughput with different attack rates, and the probing accuracy. Furthermore, a countermeasure named LOFTGuard is designed to defeat the LOFT attack. It is also evaluated with extensive experiments in real SDN testbeds.

II. The LOFT Attack

Threat Model. The attacker seeks to infer the network configurations and launch the LOFT attack to effectively overflow the flow tables of victim switches in a stealthy way. The attacker controls a host that is attached to the victim network and can send packets to other hosts. We do not require that the attacker has any prior knowledge on the network configurations. Moreover, the attacker does not compromise any switches or controllers. We assume the controller adopts *reactive rule installation* that is widely used in most OpenFlow networks for flexible and dynamic flow control [13], [14].

Attack Overview. The LOFT attack can efficiently overflow flow tables of switches by generating a minimum number of packets, which can significantly degrade the network performance in a stealthy way. It is based on one key observation that the small-sized flow tables in OpenFlow switches may be easily overflowed by malicious flows and leave no space for normal traffic flows since the centralized controller treat malicious flows and benign flows equally. This attack can be launched to overflow flow rules of all switches in a network; however, in practice, we only need to overflow flow tables of specific switches, e.g., the access switch of a target server.

LOFT consists of two phases: *the probing phase* and *the attacking phase*, as shown in Figure 1. The probing phase prepares for the attacking phase, where an attacker infers the network configurations of flow rules with a small number of probing packets. The key insight behind our probing schemes is that packets matching no flow rules in SDN switches will experience longer forwarding delays than those matching

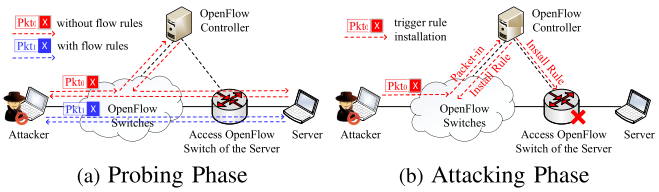


Fig. 1. Two Phases of LOFT Attack.

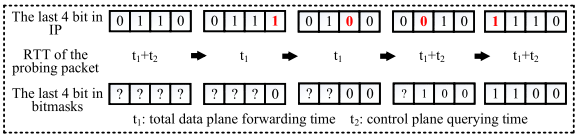


Fig. 2. An example of inferring the bitmask. Here, control plane querying time means the time waiting for new rule installation after triggering a *packet-in* message.

flow rules. It is because the switches need to query the controller for the forwarding decisions and rule installation. Therefore, by carefully crafting probing packets and analyzing the difference in RTTs between two hosts, an attacker can infer what packets may trigger rule installation and the related timeout configurations of flow rules. In the attacking phase, the attacker crafts the minimum number of attack packets to effectively trigger flow rule installation according to the inferred results on the network configurations. Meanwhile, to keep flow tables continuously overflowed over time, the attacker carefully plots the attack strategies and calculates the minimum attack rate based on the timeout configurations.

III. The Probing Phase

A. Probing Match Fields

In order to accurately infer what fields in a packet header can be used to trigger new rule installation, we craft probing packets with various field values in the packet headers to measure their RTTs. A probing packet can be any packet that triggers a response packet from a destination. We first send a probing packet to a destination to trigger possible flow rule installation in switches, which ensures that a matching rule exists before inferring RTT. Second, we generate a new probing packet that changes the value of one field of the previous packet to the same destination, and measure the RTT (denoted by RTT_0). Finally, we send another probing packet with the same values of header fields and measure the RTT again (denoted by RTT_1). The RTTs of the later two packets meet the following conditions: (1) $RTT_0 \gg RTT_1$, when a rule is installed; (2) $RTT_0 \approx RTT_1$, when no rule is installed. The first case indicates that the changed field is in the set of the match fields, while the second case denotes that the changed field is not in the set of the match fields. Based on this, we can enumerate all packet fields and then infer a complete set of match fields used in flow rules. However, there exist two challenges to accurately infer match fields:

Match Fields with Bitmasks Interference. OpenFlow allows some match fields with bitmasks, which can interfere with the probing. For example, suppose that the match field is the IP address with a bitmask "255.255.255.0". If we generate a probing packet with an IP "10.0.0.1" and produce another two probing packets with an IP "10.0.0.2", the last two packets will not trigger new flow rule installation. In this case, we mistakenly infer that the IP address is not in the match fields since the RTTs of the last two packets are close. To tackle this, we generate additional probing packets by flipping the values of each bit in turn and reconstruct bitmasks. As shown in Fig. 2, we flip a bit in the IP address of a probing packet and send the packet. If the RTT of the corresponding probing packet is close to the first probing packet, we can conclude that flipping this bit triggers new rule installation. Hence, the corresponding bit in the bitmask is 1. Otherwise, no rule installation is triggered, which means the corresponding bit in the bitmask is 0. Moreover, unlike traditional networks where only subnet masks in the prefix form can be used in IP addresses, SDN allows using arbitrary bitmasks for IP addresses [9]. For example, the match field of a flow rule can be the IP address with the bitmask "10101010.11000011.11111111.00001111". Hence, we have to go through all 32 bits to infer the bitmask of an IPv4 address. Similarly, the IPv6 address and the Ethernet address can also apply arbitrary bitmasks. Hence, we have to go through all 128 bits for the IPv6 address and 48 bits for the Ethernet address to infer their bitmasks. However, it does not take too much time to infer the match field along with its bitmask. We will show the completion time in Section V-D.

Network Jitter Interference. Two RTT values may significantly deviate even if there is no new rule installation because of network jitter. We apply the *t-test* method [26] to eliminate the impact of network jitters. In the t-test, a significance level α is set with a predetermined value and a p-value p is calculated to indicate the likelihood that the two groups of data share the same distribution. A significant difference between two groups of data is accepted if the calculated p is smaller than α. By changing the values of the same field several times, two groups of RTTs before and after the changes can be obtained. We then can evaluate if two groups of RTTs are significantly different from each other by calculating their p-value. Thereby, we can accurately infer if there exists new rule installation.

Algorithm 1 shows the pseudo-code of probing match fields. The inputs consist of the IP address of a probing destination dst, a set of fields F to be enumerated, the number of probing packets in a group n, and the significance level of the t-test α. Note that there are no general standards for using which match fields in flow rules to meet different network and application requirements. In reality, SDN controllers, applications, and network operators can configure which match fields are used in flow rules according to different network requirements. Hence, we feed all possible match fields in Algorithm 1 to accurately identify which match fields are actually used in the flow rules of the victim network. In other words, the input set of fields F includes all possible match fields defined in OpenFlow, e.g., MAC addresses, IP addresses, port numbers, etc. As shown in the algorithm, we can infer match fields and the bitmasks (if they exist) of flow rules by changing fields of probing packets.

B. Probing Timeouts

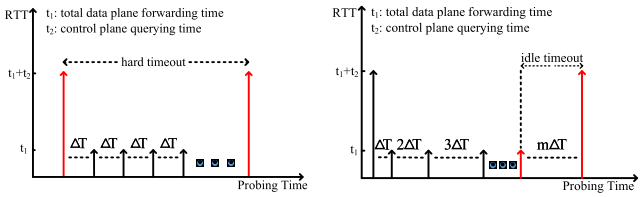
Probing timeout values of flow rules is necessary to calculate the minimum attack rate to keep flow tables overflowed. A packet will suddenly experience long forwarding delays when the rule matched by the packet is reinstalled by the controller after timeout expiration. Thus, we can estimate timeout values by measuring the elapsed time between two remarkable delays.

As shown in Figure 3(a), to infer hard timeout values, we first send a probing packet to trigger initial flow rule installation using the inferred match fields in Algorithm 1.

Algorithm 1 Probing Match Fields

Input: $dst,\ F,\ n,\ \alpha$;
Output: a set of match fields M;

$M \leftarrow \emptyset$;
for $each\ field\ f \in F$ **do**
  $pkt_0 \leftarrow build\_packet(dst, f)$;
  for $(i = 0 \rightarrow n-1)$ **do**
    $send\_packet(pkt_0)$;
    $pkt \leftarrow modify\_field\_val(pkt_0, f)$;
    $RTT_0[i] \leftarrow send\_packet(pkt)$;
    $RTT_1[i] \leftarrow send\_packet(pkt)$;
  $p \leftarrow t\_test(RTT_0, RTT_1)$;
  $b \leftarrow infer\_bitmask(f)$;
  if $((p < \alpha) or (b \neq 0))$ **then**
    $M.add(\{f, b\})$;



(a) Probing the Hard Timeout (b) Probing the Idle Timeout

Fig. 3. Inferring hard and idle timeout values. Note that mutual interference between timeouts is not considered in the figures.

Since there exists a remarkable RTT if a new rule is installed, we can periodically send the probing packets and measure their RTTs. If a remarkable RTT appears again, the hard timeout value can be inferred as the duration since the first probing packet. However, we cannot directly apply the same strategy to probe idle timeout values. The reason is that idle timeout of a flow rule will be reset once a packet matches the rule. Thus, as is shown in Figure 3(b), we need to generate and send probing packets with increasing time intervals. Once a remarkable RTT occurs again, we can infer that the idle timeout value is equal to the time interval between the two consecutive probing packets. Here, we need to address the following issues when probing timeout values in practice.

Mutual Interference Between Timeouts. A flow rule may be configured with both the hard timeout and the idle timeout. In such cases, mutual interference may happen during probing since a flow rule can be removed due to either the hard timeout or the idle timeout. For instance, suppose that the hard timeout of a flow rule is set to 15s and the idle timeout is set to 10s. In each round of probing the idle timeout, we increase the time interval by 1s. After 15s since we start the probing, a remarkable RTT will occur due to hard timeout. The idle timeout has not taken into effect because it is reset by the probing packets. Thus, we evaluate the idle timeout as 5s by mistake. Similarly, we may infer a wrong hard timeout value less than the configured value, if the configured idle value is smaller than the interval of two consecurtive packets in probing hard timeout values.

To overcome the problem, we probe hard timeout values before probing idle timeout values. We note that all timeout values can only be set to integers and the minimum valid value is 1s. To eliminate the interference of the idle timeout,

Algorithm 2 Hard Timeout Probing

Input: $dst,\ M,\ n,\ t_{wait},\ t_{max},\ \alpha,\ rand,\ e$;
Output: hard timeout t_{hard};

$pkts[] \leftarrow build\_packets(dst, M, n)$;
$t_{start} \leftarrow get\_clock\_time()$;
$RTT_0[] \leftarrow send\_packets(pkts, n)$;
repeat
  if $rand == true$ **then**
    $sleep(t_{wait} - random(0, 1))$;
  else
    $sleep(t_{wait})$;
  $t_{end} \leftarrow get\_clock\_time()$;
  $RTT_1[] \leftarrow send\_packets(pkts, n)$;
  $p \leftarrow t\_test(RTT_0, RTT_1)$;
  if $rand == true$ **then**
    $send\_extra\_packets(random(e))$;
until $((t_{end} - t_{start} > t_{max}) or (p > \alpha))$;
if $(t_{end} - t_{start} > t_{max})$ **then**
  $t_{hard} \leftarrow 0$;
else
  $t_{hard} \leftarrow round(t_{end} - t_{start})$;

we send the probing packets in a fixed interval less than 1s, e.g., 0.5s. Thus, the idle timeout will always be reset and will not influence probing a hard timeout value. Moreover, the inferred hard timeout value is the upper bound of the idle timeout since it is invalid to have an idle timeout value larger than a hard timeout value in a rule. Thereby, we enumerate all possible idle timeout values from the upper bound in a descending order to prevent hard timeout values influencing probing idle timeout values. Different from the probing shown in Figure 3(b), we decrease the time interval by 1s from the upper bound in each round of probing the idle timeout. The RTTs of two consecutive probing packets are close if the probing interval is larger than the idle timeout value. They both experience remarkable delays since flow rules will be removed due to the idle timeout. Once the RTTs of the two probing packets show significant deviation, we know that the idle timeout value is equal to the time interval between the two packets.

Probing Duration. It is time-consuming to probe the idle timeout, especially when a large hard timeout value is set. The total probing time is calculated as $\sum_{j=t_{idle}}^{t_{hard}} j$, where t_{idle} and t_{hard} are the configured idle timeout value and hard timeout value, respectively. For example, if the hard timeout value is set to 180s and the idle timeout value is set to 10s, the total probing time cost is 16,245s, i.e., around 4.5 hours. To effectively reduce the probing duration, we can apply the binary search in probing the idle timeout, since we can easily infer if an idle timeout value is smaller or larger than a given value by measuring RTTs of probing packet. Note that we also need to eliminate the interference of the hard timeout in the binary search. We can achieve this by waiting a long enough time to ensure the removal of flow rules before sending packets in a new iteration. Thus, a flow rule will be reinstalled after a probing packet during each iteration. The hard timeout will be reset and will not interfere with probing the idle timeout.

Moreover, network jitter can interfere with probing timeouts. To address this issue, we can simply send a group of

Algorithm 3 Idle Timeout Probing

Input: $dst,\ M,\ n,\ t_{sup},\ \alpha,\ rand,\ e,\ t_{rand}$;
Output: idle timeout t_{idle};
1: $pkts[] \leftarrow build\_packets(dst, M, n)$;
2: $l \leftarrow 0$, $r \leftarrow t_{sup}$;
3: **while** $(l < r)$ **do**
4: $RTT_0[] \leftarrow send\_packets(pkts, n)$;
5: $mid \leftarrow (l + r)/2$;
6: $sleep(mid)$;
7: $RTT_1[] \leftarrow send\_packets(pkts, n)$;
8: $p \leftarrow t\_test(RTT_0, RTT_1)$;
9: **if** $(p > \alpha)$ **then**
10: $r \leftarrow mid - 1$;
11: **else**
12: $l \leftarrow mid + 1$;
13: **if** $rand == true$ **then**
14: $send\_extra\_packets(random(e))$;
15: $sleep(r + random(0, t_{rand}))$;
16: **else**
17: $sleep(r)$;
18: **if** $(t_{idle} \geq t_{sup})$ **then**
19: $t_{idle} \leftarrow 0$;
20: **else**
21: $t_{idle} \leftarrow l$;

packets in parallel during each iteration of probing, and apply the t-test mentioned in Section III-A to determine if there is a significant deviation between two consecutive groups of RTTs.

Probing Stealthiness. As shown in Fig. 3, the probing activities may generate certain patterns such as fixed frequency and intervals. Hence, we generate random patterns in our probing activities to effectively increase the probing stealthiness. First, we randomly send extra packets during probing to perturb frequency and intervals. Particularly, we randomly inject extra packets within each interval when probing the timeout. Hence, there are no certain frequency and intervals during probing. Second, we carefully control the sending time of probing packets to proactively vary frequency and intervals during probing. For example, we randomly change the interval between sending two probing packets in Fig. 3a to a value less than one second when probing the timeout. We will show the implementation details in Algorithm 2 and 3.

Additionally, our probing activities cannot be easily detected by existing countermeasures [10], [11], [12], [13], [14] since they are designed to detect high-rate flooding attacks. Typically, they identify attacks by checking whether packet-in rate [10], [11], [12], [13], [14], controller CPU utilization [13], [14], or control delay [12] exceeds a certain value. For example, SPHINX [11] considers there is an attack when the packet-in rate exceeds 50 Mbps. However, our probing activities only generate a small number of packet-in packets per second, which is far below the threshold. Besides, they do not cause remarkable controller CPU utilization increase and control delay increase. Hence, it is difficult for the existing countermeasures to detect our probing activities. It is also challenging to identify our probing activities by setting up a low threshold. As benign network activities such as network diagnosis also generate a large number of packet-in packets per second, it can cause a high false positive rate with a low threshold.

Algorithm. The pseudo-code of probing hard timeout values is shown in Algorithm 2. The inputs consist of the IP address of a destination dst, the match fields M inferred by Algorithm 1, n packets to be concurrently sent, the waiting interval t_{wait}, the maximal execution time of the algorithm t_{max}, the significance level of the t-test α, the flag of random $rand$, and the number of extra noise packets. Note that t_{wait} must be set to less than 1s to effectively eliminate the interference of the idle timeout. As shown in Algorithm 2, we generate a group of packets in each iteration to probe the hard timeout value (see steps 4-14). The hard timeout value will be inferred as 0 when the execution time reaches to t_{max}, which indicates the hard timeout is not set in the flow rule (see steps 15-17). If $rand$ is set to be $true$, the algorithm randomly varies waiting intervals (see step 6) and randomly injects at most e extra noise packets (see step 13) to generate random probing patterns. Hence, the probing stealthiness is effectively increased.

Algorithm 3 shows the pseudo-code of inferring the idle timeout values by applying the binary search. The inputs are similar to those used in Algorithm 2, where t_{sup} denotes the upper bound of the algorithm execution time and t_{rand} denotes the upper bound of maximum waiting time during each round of probing. If the hard timeout value is not equal to zero, t_{sup} is set to t_{hard}. Otherwise, it is set to a value larger than the possible maximum idle timeout value. We send two groups of packets in each iteration of binary search and measure their RTTs (see steps 3-17). In particular, step 13-17 aims to ensure that flow rules can be removed after each iteration. Thus, the interference of the hard timeout can be eliminated. Similar to Algorithm 2, If $rand$ is set to be $true$, the algorithm randomly varies waiting intervals (see step 15) and randomly injects at most e extra noise packets (see step 14) to generate random probing patterns.

IV. The Attacking Phase

A. Crafting Attack Packets

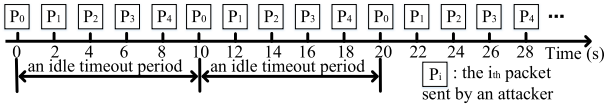
The key point is to ensure that each attack packet can effectively trigger unique rule installation in switches. Because we know the match fields along with their bitmasks in the probing phase, we can achieve it by carefully changing header field values of each packet. Thus, the minimum number of packets to overflow flow tables of a switch is equal to the flow table size. Moreover, attack packets do not need to include any real payload. We can generate a packet with 64B, which is the minimum size of Ethernet packets. Thus, approximate 113 KB traffic can successfully overflow a switch with 1,800 rules. Note that, we can further set multiple match fields with different values in the attack packets to disguise the attack packets as benign packets. For example, if match fields of a flow rule are set with the IP source address, the IP destination address, and the TCP source port, we can change the IP source address in some packets while change the TCP source port in other packets. Besides, we can generate payloads of various sizes in attack packets to increase the stealthiness.

B. Calculating the Minimum Attack Rate

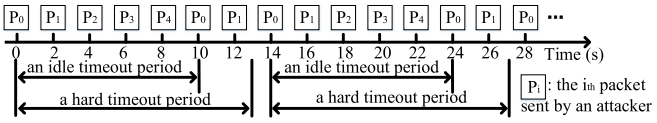
We need to compute the minimum packet rate that can continuously overflow flow tables even after flow rules expire

TABLE I
DEFAULT TIMEOUT VALUES IN DIFFERENT CONTROLLERS

Controller	Beacon	Floodlight	Maestro	NOX	ONOS	OpenDaylight	POX	Trema
Hard Timeout	0	0	180s	0	0	600s	30s	0
Idle Timeout	5s	5s	30s	5s	10s	300s	10s	60s



(a) An example to illustrate the attack strategy of Category III. Assume that the flow table can support up to 5 rules, and each rule is configured with 0s hard timeout and 10s idle timeout.



(b) An example to illustrate the attack strategy of Category IV. Assume that the flow table can support up to 5 rules, and each rule is configured with 13s hard timeout and 10s idle timeout.

Fig. 4. Examples of different attack strategies.

due to the hard timeout or the idle timeout. Normally, LOFT generates different attack packet rates with respect to different timeout settings. We classify the timeout settings into four categories according to the values of the hard and idle timeout. Here, we assume x and y are integers, where $x > y$.

(I) $t_{hard} = 0, t_{idle} = 0$: a flow rule will permanently exist in flow tables until the controller actively removes it;

(II) $t_{hard} = x, t_{idle} = 0$: a flow rule will be removed from flow tables after x seconds;

(III) $t_{hard} = 0, t_{idle} = y$: a flow rule will be removed from flow tables if the switch does not receive any packet matching the rule within y seconds;

(VI) $t_{hard} = x, t_{idle} = y$: a flow rule will be removed from flow tables either after x seconds or after y seconds without any received packet.

Among the above four categories, the settings in Categories (I) and (II) are rarely used. If the settings in Category (I) are applied, a significant amount of resources in the controller are required to actively monitor all flow rules such that flow rules will be removed when there are no matching packets. While the settings in Category (II) cannot ensure that flow rules can be removed in time if the network does not generate any packets matching the rules, resulting in the waste of the scarce flow table resources. According to our studies, we find that the settings in Categories (III) and (IV) are widely used in default settings of different controllers (see Table II). Thus, in this paper, we focus on developing two attack strategies that use minimum attack rate to overflow flow tables according to the settings in Categories (III) and (IV).

Attack Strategy with Settings in Category (III). An attacker needs to fill in the flow table within an idle timeout period since a rule will be removed after an idle timeout period. To achieve this goal, the attacker can periodically generate C attack packets, where C is the maximum capacity of the flow table, and evenly distribute them within each idle timeout period (see Figure 4(a)). Each packet will trigger new rule installation if there is any available space, and the number of rules in the flow table can gradually increase. Meanwhile, to ensure flow rules are persistently stored in the flow table, each flow rule needs to be periodically refreshed by repeatly generating identical packets iduring each idle timeout interval.

Formally, we use C to denote the maximum capacity of the flow table of a switch, L_i to denote the length of the i_{th} packet within an idle timeout period, and t_{idle} to denote the idle timeout. The average packet rate of sending attack packets within an idle timeout period $\overline{p}$ can be calculated by:

$$\overline{p} = \frac{\sum_{i=0}^{C-1} L_i}{t_{idle}}. \tag{1}$$

Note that we need to generate at least C packets in order to fully consume flow rules, each of which triggers new rule installation. Thus, Equation (1) gives the minimum attack rate. Any attack rate less than $\overline{p}$ cannot fully consume the table and keep the table full over time due to the expiration of flow rules incurred by the idle timeout. According to Equation (1), we can conclude that the attack rate is small. For example, assuming the flow table supports up to 1,800 flow rules and there are no other flows consuming the flow table except our attacking flows, the idle timeout value is set to 20s, and the size of each packet is 64B, the attack rate is only 46 Kbps.

We should note that Equation (1) only denotes the upper limit of the packet rate for an attacker to saturate the flow table of an SDN switch without other flows. In reality, the required packet rate to overflow the flow table is much less than the upper limit since free flow table space is far smaller than the table capability. Most space of the flow table has been occupied by many flow rules matching a large number of benign flows in real networks. Hence, the available number of flow rules in SDN is limited and scarce [6], [27], [28], [29], [30], [31]. Furthermore, fine-grained and flexible control in SDN requires more rules for flows [6], [27], [28], [29], [31]. Therefore, the attacker only needs to generate a small number of packets to saturate the remaining small flow table space. Hence, the packet rate for an attacker is quite low.

Although most SDN switches can only support up to 750–20,000 flow rules due to power-hungry and expensive TCAM [6], [27], [28], [29], [31], we may need to increase the attack rate to hundreds of packets per second for high-end SDN switches with a large table size. However, compared to brute-force flooding attacks [10], [11], [12], [13], [14] that generate tens of thousands of packets per second, our attack rate is still much smaller. Moreover, our attack rate can be further reduced like traditional distributed attacks [32], i.e., controlling multiple hosts to simultaneously saturate the same switch so that the attack rate for each attacking host is remarkably reduced. Furthermore, an attacker can also change the target to the switch that can be easily overflowed in the target networks with a small number of packets.

Attack Strategy with Settings in Category (IV). Given the settings in Category IV, flow rules will be removed when either the hard timeout or the idle timeout expires. Thus, besides sending packets to gradually overflow flow tables within an idle timeout period and periodically refreshing the flow rules, an attacker needs to make a rule reinstalled in time once it is removed due to the hard timeout. Since the timeout settings have been known, an attacker can easily achieve it. Furthermore, to achieve a constant attack rate, we manage to delay the sending time when a rule needs to be reinstalled. As is shown in Figure 4(b), the rule triggered by p_0 is removed at 13s due to the hard timeout. We reinstall the rule at 14s

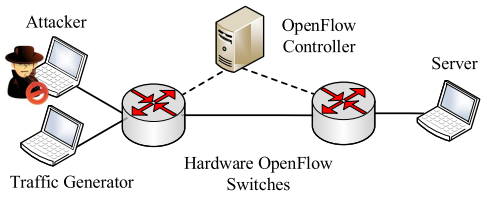


Fig. 5. The Hardware SDN Testbed.

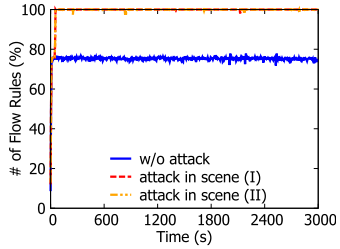
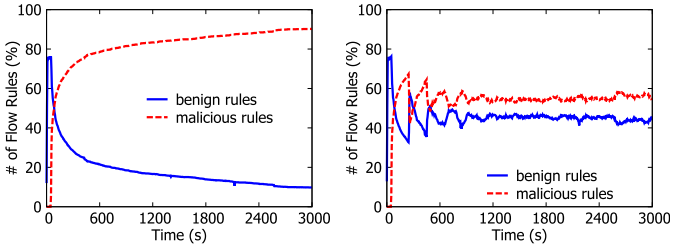


Fig. 6. The number of total flow rules in the switch's TCAM.



(a) Scene (I) (b) Scene (II)

Fig. 7. The proportion of flow rules in TCAM with the attack.

rather than at 13s so as to keep the time interval between two consecutive attack packets equal. In this way, the average attack rate is the same with that in Category III.

We also need to predict the table capacity of the switch to construct the LOFT attack. We develop an online scheme to infer the table size by gradually increasing the number of attack packets and checking if the tables are full. At first, we construct the attack to occupy n flow rules. After this, we infer if the flow tables are full by sending additional packets to measure the RTT differences. If the RTTs of these packets significantly deviate, it indicates that the flow tables are full since each packet triggers the insertion of a new flow rule but none of them have been successfully installed. Otherwise, we launch another round of probing to occupy n' flow rules. Note that we can not accurately infer the size of a flow table since the flow rules used by benign traffic always change. However, in practice, we do not need to know the accurate table capacity. We can increase n using a larger number, such as 2,000. By repeating the procedure several times, we can gradually occupy the flow tables until they are all consumed.

V. Attack Evaluation

A. Experiment Setup

As shown in Fig. 5, a real SDN testbed is built to conduct our experiments. The OpenFlow controller Floodlight [33] runs in a host with an Intel Xeon Quad-Core CPU and 12GB RAM. The *Forwarding* application [34] providing basic forwarding services runs on the controller by default. Two commercial hardware OpenFlow switches, Pica8 P-3290, are deployed in the testbed. Each switch can store up to 2,000 flow rules in TCAM. A host compromised by an adversary is attached to one of the switches. The LOFT attack program contains appoximate 2000 lines of C code. The malicious host runs the attack program to overflow the flow tables of switches by sending crafted packets to the server. Moreover, to emulate real network conditions, we deploy one client host that replays real traffic trace from CAIDA [35] [1] with the server host.

B. Evaluation of Attack Effectiveness

We conduct our experiments in two typical scenarios to demonstrate the effectiveness of LOFT: (I) only the idle timeout is set; (II) both the hard timeout and idle timeout are set. The idle timeout value is set to 20s in both two scenarios, and the hard timeout value is set to 200s in the second scenario. According to Equation (1), we launch LOFT with an average attack rate of 51 Kbps in both scenarios. Note that the attacker does not know the timeouts or the match fields of flow rules in advance. Such information needs to be probed before launching the attack. We evaluate the probing accuracy and efficiency in Section V-C and V-D, respectively.

Impacts on the Number of Flow Rules. We measure the number of flow rules in the TCAM of the switch that connects to the server in the two scenarios. Fig. 6 shows that the number of flow rules occupies about 77% TCAM capacity on average without the attack. However, the number of flow rules quickly increases to 100% with the attack. Such results demonstrate that the attack can successfully saturate the flow table with a low rate (51 Kbps). Moreover, benign flows occupy fewer flow rules with the attack along with time. As Fig. 7a shows, the number of benign rules decreases from 77% to 10% and the number of malicious rules increases from 0 to 90% at 3600s in scenario (I). The number of benign rules decreases to 45% and the number of malicious rules increases to 55% at 900s.

There are some differences on the number of rules in the two scenarios. The number of malicious rules in scenario (I) shows sustainable growth over time. However, in scenario (II), it shows continuous jitter. The reason is that malicious flows persistently occupy flow rules by periodically sending packets to refresh the idle timeout in scenario (I), but such rules will be periodically removed due to the hard timeout in scenario (II). Besides, malicious flows can hardly compete with benign flows for the space of removed rules along with less space left in TCAM. Therefore, the growth rate of malicious rules becomes slower over time in scenario (I) and the number of malicious rules converges to a stable value at scenario (II).

Impacts on the Network Throughput. Fig. 8 shows the network throughput affected by the attack in the two scenarios. The network throughput can achieve 650 Mbps on average in absence of the attack. However, it continually decreases with the attack. At 3000s, the network throughput decreases to 300 Mbps in scenario (I) and 420 Mbps in scenario (II), respectively. Moreover, to quantify the impacts of the attack on the network throughput, we measure the throughput

[1]There are a huge number of concurrent flows in the trace, which is far beyond the flow table capacity of our switches. Thus, we randomly choose flows in the trace to ensure the number of flow rules is under the table capacity.

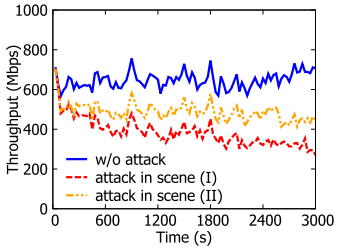


Fig. 8. The network throughput.

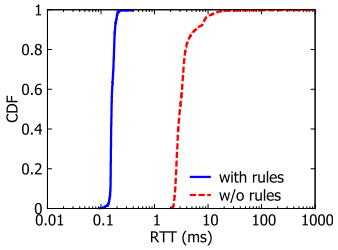
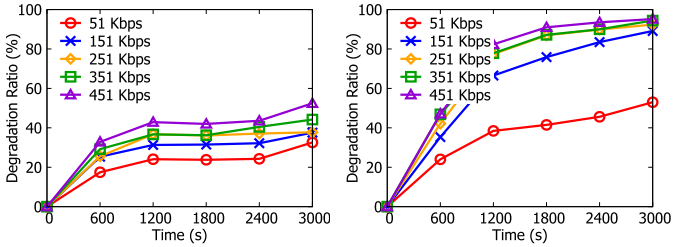


Fig. 9. RTT of a flow.



(a) Scene (I) (b) Scene (II)

Fig. 10. The degradation ratio of throughput with attack rates.

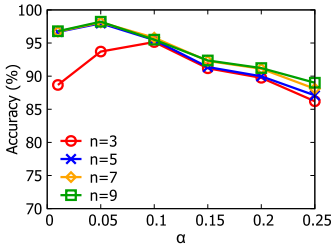


Fig. 11. The probing accuracy with different parameters.

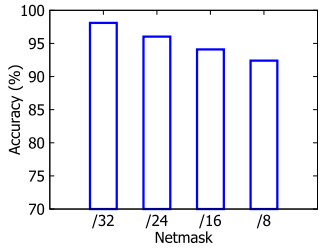


Fig. 12. The probing accuracy with different netmasks.

degradation ratio of the network traffic with different attack rate in each attack scenario. The *degradation ratio* is the fraction of the traffic decreased by the attack over the total traffic without the attack within a period. Here, for simplicity, we set the period to 600s. The degradation ratio with different attack rates is shown in Figure 10. From both scenarios, we can see that the quicker the attack sends the traffic, the faster the degradation ratio increases. For example, sending traffic with 51 Kbps, 151 Kbps and 251 Kbps results in a 22%, 29% and 36% degradation ratio at 1200s in scenario (I) and a 38%, 63% and 77% degradation ratio at 1200s in scenario (II), respectively. Such results demonstrate that an attacker can cause more damage with a faster attack rate. However, it also becomes easier for the attack to be exposed. There is a tradeoff between the damage and stealthiness of the attack.

Impacts on the Forwarding Delays. We measure the RTTs between the server and the client to show the impacts on the forwarding delays of flows winning no rules due to the attack. Each packet of such flows has to be delivered to the controller for forwarding, which significantly increases the forwarding delays. As is shown in Fig. 9, the average RTT is 0.17 ms and 100% RTTs are smaller than 0.5 ms when a flow match related flow rules. However, the average RTT increases to 5.9 ms when the flow wins no flow rules with the attack. Some RTTs even exceeds 100 ms, which are about 1000x longer than the RTTs of flows with rules.

C. Evaluation of Probing Accuracy

We evaluate the accuracy of probing match fields η_m, the accuracy of probing bitmasks η_b, the accuracy of probing hard timeouts η_h, and the accuracy of probing idle timeouts η_i. They are calculated as follows:

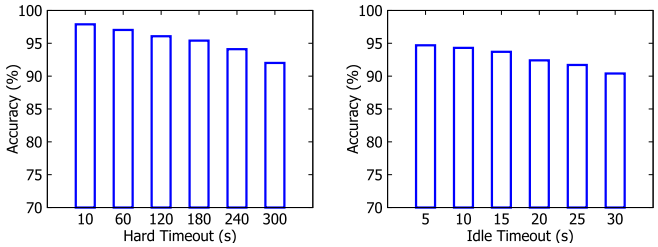
$$\left\{\eta_m = \frac{C_m}{N_m},\ \eta_b = \frac{C_b}{N_b},\ \eta_h = \frac{C_h}{N_h},\ \eta_i = \frac{C_i}{N_i},\right. \tag{2}$$

Here, N_m, N_b, N_h, and N_i denote the number of experiments on probing match fields, bitmasks, hard timeouts, and idle timeouts, respectively. C_m, C_b, C_h, and C_i denote the number of experiments where match fields, bitmasks, hard timeouts, and idle timeouts are correctly inferred, respectively.

Accuracy of Probing Match Fields. By default, the match fields of the *Forwarding* application in Floodlight are $\langle src\_mac,\ dst\_mac,\ src\_ip,\ dst\_ip,\ src\_port,\ dst\_port\rangle$ without bitmasks. We conduct Algorithm 1 to measure the accuracy of probing match fields with a different number of probing packets in a group n and various significance level α. From Fig. 11, we can see that our algorithm accurately identifies the match fields that are actually used in the flow rules at more than 98% accuracy when $n = 5$ and $\alpha = 0.05$. When n is further increased from 5, the probing accuracy is only slightly improved. Thus, we configure n as 5 and α as 0.05 in all our probing algorithms.

Accuracy of Probing Bitmasks. To evaluate the accuracy of probing bitmasks, we configure the match fields of the *Forwarding* application as $\langle src\_ip,\ dst\_ip\rangle$ with 8-bit, 16-bit, 24-bit and 32-bit netmasks, respectively. As is shown in Fig. 12, the accuracy of probing can achieve more than 90% accuracy with different bitmasks. Besides, the accuracy shows a slight decrease along with smaller length of bitmasks since more times of probing are needed to reconstruct bitmasks.

Accuracy of Probing Timeout Values. We systematically measure the probing accuracy with different timeout settings. Fig. 13a shows the accuracy of probing the hard timeout values when conducting Algorithm 2. We observe that the probing



(a) Hard Timeout (b) Idle Timeout

Fig. 13. The accuracy of probing timeouts.

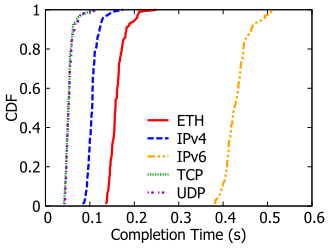


Fig. 14. The completion time on probing different match fields with bitmasks. Note that the bitmask of each field is set with all ones. The Ethernet address, the IPv4 address, and the IPv6 address are set with the 48-bit bitmask, the 32-bit bitmask, and the 128-bit bitmask, respectively. No bitmasks are set for the TCP and UDP ports since they cannot have bitmasks [9].

can reach more than 90% accuracy rate with different hard timeout values. Similarly, Fig. 13b shows that the probing can also reach more than 90% accuracy when conducting Algorithm 3 to probe different idle timeout values. Moreover, the accuracy of both the probing decreases slightly when the timeout value increases since more times of probing are needed. However, such accuracy is enough to construct the LOFT attack. We may not be able to infer correct timeout values with one round of probing, but instead we can obtain the correct results by performing multiple rounds of probing.

D. Evaluation of Probing Efficiency

Completion Time on Probing Match Fields with Bitmasks. We measure the completion time for Algorithm 1 to probe different match fields with bitmasks. We set the bitmask of each field with all ones. Hence, we can measure the completion time on probing match fields with bitmasks at the worst case. Moreover, as network delays vary with time, we repeat our experiments 100 times to draw the CDF of the completion time. As shown in Fig. 14, the completion time on probing different match fields with bitmasks is different. As the bitmask for the IPv6 address has 128 bits, it spends the longest time to probe the IPv6 address. However, the longest completion time is still below 0.6s. For the 32-bit IPv4 address, the completion time is always below 0.2s. The completion time on probing the 16-bit UDP/TCP ports achieves the shortest, i.e., less than 0.15s.

Completion Time on Probing Timeouts. We conduct experiments to measure the completion time for Algorithm 2 and Algorithm 3 to probe different timeouts. As network delays vary with time, we repeat our experiments 100 times to obtain the average completion time. As shown in Table II, the completion time on probing hard timeouts is close to the hard timeout values. For example, it spends 50.05s to complete probing the hard timeout of 50s. It is reasonable since Algorithm 2 continually runs and sends probing packets until a flow rule is removed due to the timeout value. However, the completion time on probing idle timeouts is much longer than the hard timeout value. It is because Algorithm 3 waits for a long time in each round of probing in order to eliminate the mutual interference of the hard timeout.

TABLE II
THE AVERAGE COMPLETION TIME ON PROBING TIMEOUTS

Hard Timeout (s)	50	100	150	200	250	300
Completion Time (s)	50.05	100.05	150.11	200.04	250.19	300.06
Idle Timeout (s)	5	10	15	20	25	30
Completion Time (s)	113.03	141.04	211.04	223.04	253.04	265.03

VI. LOFTGUARD

To defend against the attack, we provide LOFTGuard, a data-to-control plane collaborative defense system. It is lightweight and transparent to applications on SDN controllers.

A. Design Challenges

To defeat the LOFT attack, we meet three challenges:

(1) How to accurately detect the LOFT attack with a low false positive rate and a high recall rate? Compared to existing high-rate flooding attacks [10], [11], [12], [13], [14] that generate thousands of different new flows per second to incur the controller DoS, the LOFT attack only generates a small number of packets and new flows per second, which is more stealthy. One possible countermeasure is to set up a new flow generation threshold to detect the LOFT attack. However, benign hosts in real networks can also generate a number of different flows per second due to real network applications [36]. Thus, this countermeasure may incur a high false positive rate[2] Another countermeasure may be based on port security to detect the attack. However, as benign hosts may scan ports to diagnose networks and also generate a large number of new flows per second [37], the false positive rate of this countermeasure may be still high. Besides, an attacker can control multiple hosts to attack the same switch like traditional distributed attacks [32]. Thus, the attack rate and the number of new flows for a host can be remarkably reduced, which may cause a low recall rate for the two countermeasures above.

(2) How to effectively thwart malicious flows from occupying TCAM before knowing which flows are malicious? As an attacker can easily and quickly craft different values in packet headers, different and endless attacking flows can be continuously generated. However, any methods on accurately identifying malicious flows take a time and hence are always lagging. Continuous malicious flows may have already occupied a number of flow rules before we identify them and clear the corresponding flow rules. Moreover, as a sophisticated attacker can control multiple hosts to attack the same switch with varying flows, the TCAM of the switch can still be quickly saturated even we continuously delete malicious flow

[2]In our experiments, the new flow generation threshold must be set less than about 90 flows per second so that the LOFT attack can be detected. However, this incorrectly identifies many benign flows as attacking flows, e.g., about 67% false positive rate even with the threshold of 90 flows per second. Besides, the false positive rate becomes higher with a smaller threshold.

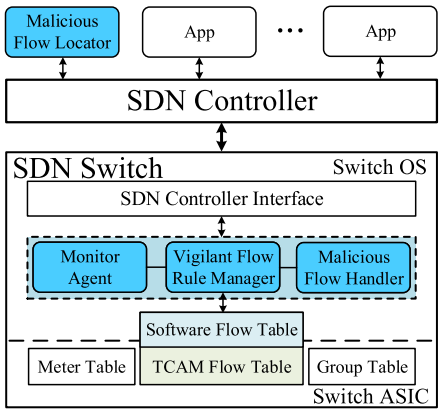


Fig. 15. The architecture of LOFTGuard.

rules. Thus, we may require a method that can effectively prevent malicious flows from occupying TCAM even without knowing them. Meanwhile, it can maintain normal forwarding for benign flows as much as possible.

(3) How to continuously block malicious flows in a fine-grained manner without consuming a number of flow rules? By crafting different header values in packet headers, the LOFT attack can generate endless malicious flows. If we continuously block the malicious flows in a fine-grained manner, a number of flow rules matching these flows with drop actions must be installed to the switch. Hence, most available flow table space will be consumed with these blocking rules, which instead satisfies the attackers' goal. However, if we directly block the ingress SDN switch ports of the malicious flows in a coarse-grained manner, benign flows will also be blocked. This is because the same ingress SDN switch port can be connected by multiple hosts, e.g., several virtual hosts located in the same physical host, or a group of physical hosts connecting the SDN network through a hub or traditional layer-2 switch.

B. System Architecture

To overcome the challenges above, we present LOFTGuard that can effectively defeat LOFT attacks. Instead of using new flow generation rate, it applies long-term flow features and flow table utilization features to accurately detect attacks and identify malicious flows. Meanwhile, it dynamically migrates flow rules among TCAM and software regions based on flow liveness and rate, which can effectively thwart malicious flows from occupying TCAM even without knowing them. It also maintains normal forwarding for benign flows as much as possible. To continuously block malicious flows in a fine-grained manner without consuming many flow rules, LOFTGuard maintains the blacklist of malicious flows in switches, and directly filters out packet-in messages of malicious flows with general processing powers of SDN switches.

The architecture of LOFTGuard is shown in Fig. 15. In the data plane, it introduces three modules in SDN switches: *monitor agent*, *vigilant flow rule manager*, and *malicious flow handler*. The modules exploit the general processing powers of SDN switches to assist an SDN controller in defending against the LOFT attack. They are deployed as software in the switch. In the control plane, a *malicious flow locator* is implemented as an application on the controller and can flexibly apply advanced algorithms to accurately identify malicious flows.

Initially, the monitor agent periodically monitors the flow table utilization of switches and the other modules remain idle. When abnormal flow table utilization is detected, it activates other modules to defend possible LOFT attacks. The vigilant rule manager creates the cache region of flow rules, which is software-implemented flow table space. It dynamically migrates flow rules between the TCAM region and the cache region based on the liveness and rate of flows to effectively throttle the occupation of TCAM for malicious flows. Meanwhile, the malicious flow locator collects statistics of flows and identifies malicious flows. Once malicious flows are identified, it informs the malicious flow handler to add such flows to a blacklist and delete related flow rules. To prevent malicious flows from occupying rules again, the malicious flow handler filters packet-in messages based on the blacklist.

C. Data Plane Design

1) Monitor Agent: The monitor agent aims to monitor abnormal flow table utilization and thus to detect the LOFT attack in time. Once the attack is detected, it awakes other modules to collaboratively defend against the attack. The monitor agent can detect the potential LOFT attack when the flow table utilization is high. However, when the attack slowly occupies the flow table, it will last for a long time before a high utilization is detected in the flow table. A timely method to detect the attack is to observe growth trend of flow table utilization. In normal cases, the flow table utilization is usually stable or fluctuant due to the expiration of old flows and creation of new flows, as shown in Fig. 6. When the flow table utilization continues to show an upward tendency, a potential attack may occur. It may mistakenly assert a normal case as an attack case in a few cases. Nevertheless, we can detect the mistake later since the flow table will not be overflowed and the malicious flow locator cannot identify malicious flows.

2) Vigilant Flow Rule Manager: The vigilant flow rule manager works when the LOFT attack is detected. It prevents the occupation of TCAM against suspicious flows and maintains normal forwarding for benign flows before the controller finishes identifying malicious flows. Note that it takes some time to collect enough statistics of flows in order to accurately identifying the low-rate malicious flows. Specifically, the vigilant flow rule manager divides flow tables into different regions and dynamically migrate rules among these regions to effectively utilize TCAM resources as well as throttle occupation of TCAM for suspicious flows. As is shown in Fig. 16, the flow table is divided into three regions: *cache region*, *pending region*, and *confirmed region*. The cache region is implemented as software flow table using the processing powers of the general CPUs of switches. The other two regions are TCAM-based flow table. These regions of flow table are transparent to SDN applications. Initially, all installed flow rules are stored into the pending region. Flow rules that are later been identified as benign flows are migrated into the confirmed region, as depicted in step ④ in Fig. 16. The vigilant rule manager applies two strategies to dynamically migrate rules among these three regions.

The first strategy is to migrate rules according to the liveness of flows. According to our investigation (See Table II in Section IV), SDN controllers set fixed timeouts for all flows by default. It allows an attacker to send one packet every timeout

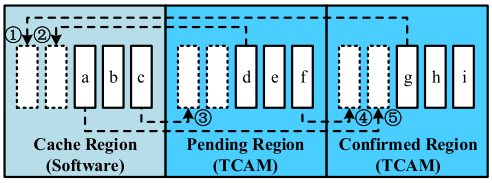


Fig. 16. The vigilant flow rule manager dynamically migrates rules among different regions. ① and ⑤: rules are migrated among the confirmed region and the cache region according to the liveness; ② and ③: rules are migrated among the pending region and the cache region according to the liveness or flow rate; ④: rules are migrated into the confirmed region when they are identified as benign rules.

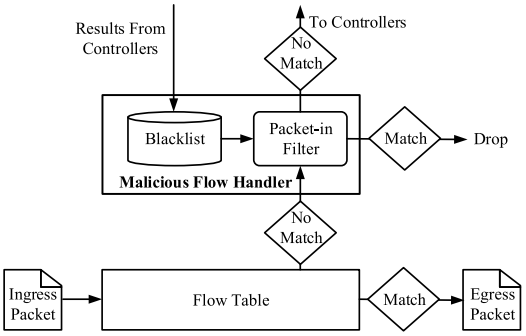


Fig. 17. The malicious flow handler.

period to effectively occupy a rule in TCAM. Moreover, for benign flows that only last for a short time, the rules matching them in TCAM cannot be removed in time to save the space. In order to throttle malicious flows from occupying TCAM space and effectively use the TCAM space of flow table, the vigilant rule manager monitors the liveness of rules in pending and confirmed regions. When no more packets match the rule for some time beyond a threshold but the rule is not expired, the vigilant rule manager migrates the rule into the cache region to save the scarce TCAM space, as depicted in step ① or ② in Fig. 16. When a packet matches the rule again, it will be migrated into its corresponding pending or confirmed region again, as depicted in step ③ or ⑤ in Fig. 16.

The second strategy is to migrate rules according to the flow rate. Although migrating inactive rules into the cache region can throttle malicious flows from occupying TCAM, an armored attacker might prevent installed flow rules from migrating into the cache region by periodically sending packets with intervals below the threshold. Thus, when TCAM is unfortunately saturated due to the smart attacker, the vigilant rule manager will dynamically migrate rules among the cache region and the pending region based on flow rate to maintain normal forwarding of potential benign flows. Specifically, a new rule is first installed into the cache region when the TCAM has saturated. Meanwhile, the vigilant rule manager periodically checks the rate of flows in the cache region and the pending region according to the counters and duration information in flow rules. When the highest flow rate of a rule in the cache region (denoted by rule A) has exceeded the lowest flow rate of a rule in the pending region (denoted by rule B), it migrates rule B into the cache region and migrates rule A into the pending region. Steps ② and ③ in Fig. 16 show the process of migrating the rules according to their flow rate.

The migration strategy is simple but effective to maintain normal forwarding of benign flows as well as thwart malicious flows from occupying TCAM. According to the migration strategy, an attacker has to make the rate of malicious flows higher than that of benign flows in order to permanently occupy more TCAM. For example, suppose that there are 5 benign rules with different flow rate: $[rule_1 : 100Mbps,\ rule_2 : 200Mbps,\ rule_3 : 300Mbps,\ rule_4 : 400Mbps,\ rule_5 : 500Mbps]$. If an attacker wants to occupy 3 rules in TCAM competing with these five benign flows, it is required to generate three malicious flows with different rates at least higher than $100Mbps$, $200Mbps$ and $300Mbps$, respectively. Otherwise, malicious rules will be stored into the cache region due to the low rate. Obviously, occupying more rules requires the newly generated malicious flows to have a higher rate. Thus, the migration strategy significantly improves the costs for an attacker to occupy TCAM space. An attacker cannot easily occupy much TCAM space with a low rate. Meanwhile, high-rate attacks can easily be detected and defended by DoS defenses [13], [38]. Under the migration strategy, benign rules matching high-rate flows are stored in TCAM to maintain normal forwarding at line rate. A few benign rules matching low-rate flows may be mistakenly migrated into the cache region. However, they will be migrated into the confirmed region later once they are identified as benign flows.

3) Malicious Flow Handler: It receives the identified results from SDN controllers and prevents identified malicious flows from occupying flow rules. It first informs the vigilant rule manager to delete the malicious rules and to migrate the lively benign flows from the cache region to the confirmed region (TCAM space) to conduct forwarding at line rate. Meanwhile, it updates the blacklist of malicious flows and filters packet-in messages according to the blacklist in order to prevent the identified malicious flows from competing rules with benign flows again. As shown in Fig. 17, it intercepts the packet-in messages generated from flows for querying flow rule installation. If the packet-in messages are generated from malicious flows in the blacklist, it drops the messages. Otherwise, it transparently delivers the packet-in messages to controllers. As the malicious packet-in messages are dropped within switches, the bandwidth of control channels are saved.

D. Control Plane Design

The malicious flow locator is designed as an application on SDN controllers. When the LOFT attack is detected, it extracts the features of flows based on flow statistics and identifies malicious flows. As mentioned in Section IV, to stealthily overflow the flow table, the LOFT attack generates low-rate flows and periodically sends packets with a long time interval to maintain existing flow rules. As a result, the malicious flows usually have much idle time sending no packets. Thus, based on the *low rate* and *much idle time* of the attack, we extract features for identifying attack flows:

- Sequences of Average Flow Rate $[v_1, v_2, \ldots, v_n]$. Here, $v_i, i \in [1, n]$, describes the average flow rate in the i_{th} time interval. Malicious flows have relatively low rates.
- Sequences of Average Packet Length $[l_1, l_2, \ldots, l_n]$. Here, $l_i, i \in [1, n]$, describes the average packet length of a flow in the i_{th} time interval. Note that a malicious flow may use small packets to occupy flow rules in order to decrease its attack rate.

- Sequences of Idle Status of Rules $[\beta_1, \beta_2, \ldots, \beta_n]$. Here, $\beta_i, i \in [1, n]$, describes the idle status of a flow rule in the i_{th} time interval. Note that malicious rules typically remain idle, i.e., matching no packets for a long time.

Based on the flow statistics, we can extract above features according to the following equations:

$$\begin{cases} v_i = \dfrac{b_i - b_{i-1}}{\delta}, \\ l_i = \dfrac{b_i - b_{i-1}}{p_i - p_{i-1}}, \\ \beta_i = \begin{cases} 1 & b_i = b_{i-1} \\ 0 & b_i \neq b_{i-1} \end{cases} \end{cases} \tag{3}$$

Here, $i \in [1, n]$. b_i and p_i denotes the counter values of passed bytes and passed packets of the corresponding flow rule obtained at the i_{th} time, respectively. δ denotes the time interval between collecting flow statistics. After extracting the features, identifying malicious flows from benign flows is a classification problem. We apply Random Forests [39] as our classifier, which is one of the most effective machine learning models for classification and performs well in our experiments.

VII. Defense Evaluation

A. Implementation

We implement LOFTGuard with the Floodlight controller and the hardware switch named EdgeCore AS4610-54T [40]. The switch has a Dual-core ARM Cortex A9 1 GHz CPU, 2 GB DDR3 SDRAM, and 8 GB NAND Flash. The operating system running in it is PICOS v2.11.22 [41]. All the data plane parts of LOFTGuard, i.e., the monitor agent, vigilant flow rule manager, and malicious flow handler, are simply implemented in C as daemons running in switches together with the original OpenFlow agent. We require neither modifying hardware of SDN switches nor recompiling the original OpenFlow agent to add new functionalities. Similar to installing new software in switches, adding them into SDN switches does not remarkably increase the complexity of the switches. The control plane part of LOFTGuard, i.e., the malicious flow locator, is implemented as an application running in the controller. We next detail the implementation for each part.

Monitor Agent. It periodically obtains the number of installed rules in TCAM by invoking a shell command named *ovs-ofctl dump-tables* [42] provided by PICOS. Moreover, we measure the maximum number of flow rules supported by TCAM in advance. Hence, the TCAM utilization can be calculated. We set the period of executing the command as 0.2s in our implementation since it incurs low CPU usage for switches (shown in Fig. 19). The period enables the monitor agent to detect abnormal flow table utilization since it takes at least several seconds for the LOFT attack to saturate the flow table.

Vigilant Flow Rule Manager. It actively manages the switch TCAM by invoking the shell commands provided by PICOS. Instead of physically partitioning the TCAM space into two regions, the rule manager maintains a location table that records the logical location of each flow rule. When flow rules are migrated between the pending region and the confirmed region, it just updates the corresponding logical location in the table without physically moving the flow rules. However, when flow rule migration happens between TCAM and the cache region, the rule manager invokes *ovs-ofctl del-flows* [42] or *ovs-ofctl add-flow* [42] to delete or create corresponding flow rules in TCAM. Meanwhile, it creates or deletes corresponding flow rules in software flow tables, i.e., the cache region. The location table will also be updated.

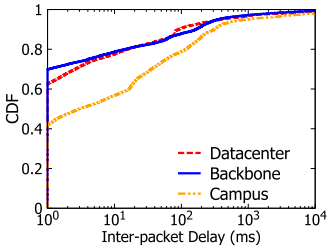


Fig. 18. Inter-packet delays in different scenarios.

In our implementation, we set a liveness threshold as the experimental value 200 ms to migrate flow rules among different regions based on their liveness. As shown in Fig. 18, more than 80% packets of benign flows have less than 200 ms inter-packet delays.[3] Hence, the threshold is enough for most flows to efficiently use the TCAM space. We do not dynamically change the threshold based on the real-time measurement on inter-packet delays since OpenFlow-enabled SDN switches lack the capabilities to measure the inter-packet delays in real time. Moreover, the vigilant rule manager collects the counters and duration information in flow rules to calculate flow rate every 0.2s by invoking the shell command *ovs-ofctl dump-flows* [42]. We use the quicksort algorithm to sort flows and determine which flows go to TCAM or cache region based on flow rate. Furthermore, we limit the maximum number of software flow rules in the cache region to 10,000 in order to prevent running out of memory. If the cache region is saturated, we evict the oldest rules based on the creation time.

Malicious Flow Handler. It creates a separated TCP socket with the malicious flow locator running in the controller to report flow statistics. Particularly, it invokes *ovs-ofctl dump-flows* [42] in switches every 50 ms to extract two parts of each flow rule, i.e., the 8-byte cookie value indicting the rule identity and the two 8-byte counter values showing the passed packets and bytes. These values are all put into the payload of TCP packets that will be sent to the malicious flow locator. We do not use the OpenFlow protocol since it takes much bandwidth to frequently report statistics of all flow rules and cannot obtain the flow rule statistics in the cache region. The malicious flow handler also receives the cookie values of identified malicious rules from the TCP socket and informs the vigilant rule manager to delete the malicious rules.

To prevent identified attack flows from leveraging packet-in packets to query flow rule installation again, the malicious flow handler applies NFQUEUE [45] to intercept and filter packet-in packets generated by the openflow agent when a new flow arrives. The packet-in packets triggered by attack flows will be dropped in switches according to the identified results before they are sent to the control plane.

Malicious Flow Locator. In our implementation, it continually collects flow statistics for each flow every 50 ms from the TCP socket built with the malicious flow handler. It identifies a malicious flow based on the flow statistics of the last 100 collections. Note that even if a flow rule is migrated into the confirmed region, the malicious flow locator still gets

[3]We extract the inter-packet delays from real traffic traces coming from a Datacenter [43], an Internet Backbone [35] and a Campus [44].

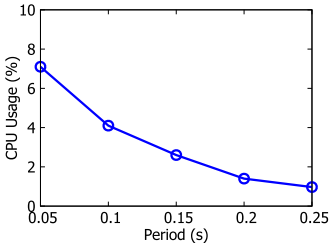


Fig. 19. Average CPU usage with different collection periods.

its statistics. Hence, we can identify malicious flows that are launched by attackers after their corresponding flow rules have been migrated into the confirmed region.

We collect benign and malicious flows to train the malicious flow locator. Specifically, we replay the real traffic trace from CAIDA [35] as benign flows in our SDN testbed. Meanwhile, we launch the LOFT attack to generate attack flows. As an attacker may change their packet size to increase its stealthiness rather than only send attack packets with 64B, we randomly change the size of attack packets between 64B and 1500B. We first collect the statistics of 10,000 benign flows and 10,000 malicious flows and extract their features based on Equation 3. We use the data to train a random forests classifier model with *scikit-learn*. Based on our empirical study, the training hyperparameters that can make our model achieve the best location performance are selected as follows: *n_estimators = 89, oob_score = true, criterion = gini, min_samples_split = 10, min_samples_leaf = 20, max_depth = 8, max_features = sqrt, random_state = 10*. To evaluate the effectiveness of the model on identifying various attack flows, we build three types of testing datasets as follows:

- **Fixed.** This testing dataset is based on the assumption that attackers launch the LOFT attack with 64-B attack packets. It contains statistics of 5,000 attack flows with 64-B packet size and 5,000 benign flows from CAIDA [35].
- **Random.** This testing dataset is based on the assumption that attackers launch the LOFT attack with random packet size to increase its stealthiness. It contains statistics of 5,000 attack flows with random packet size and 5,000 benign flows from CAIDA [35].
- **Advanced.** This testing dataset is based on the assumption that advanced attackers launch the LOFT attack with packet size following the size of normal flows, which may greatly increase the stealthiness. It contains statistics of 5,000 attack flows with packet size following the size of normal flows and 5,000 benign flows from CAIDA [35].

B. Effectiveness

Number of Malicious Flow Rules in TCAM. As shown in Fig. 20, malicious flows consume much TCAM without LOFTGuard, i.e., more than 50% in scenario (I) and more than 80% in scenario (II). The higher the attack rate is, the more TCAM the malicious flows consume. However, the number of malicious flow rules in TCAM significantly drops with LOFTGuard. The malicious flows consume less than 10% even with a high attack rate of 451 Kbps in both scenarios. Such results benefit from the rule migration adopted in vigilant flow rule manager. As malicious flows in the LOFT attack have lower flow rates compared to benign flows and corresponding

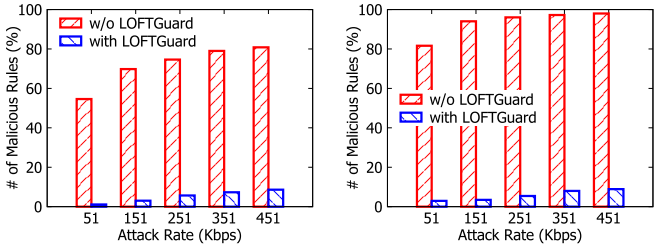


Fig. 20. The average number of malicious flow rules in the switch's TCAM within 3000s.

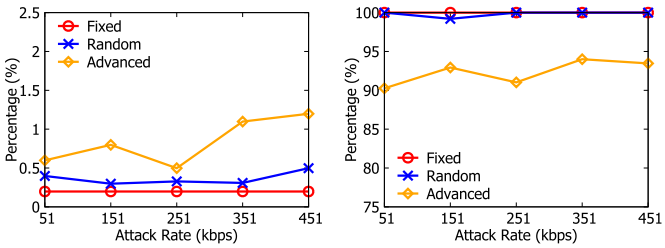


Fig. 21. The effectiveness of identifying three types of attack flows: (I) fixed packet size with 64B, (II) random packet size, and (III) advanced packet size following the size of benign flows.

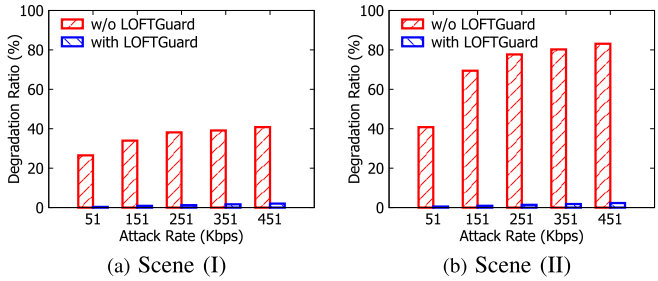


Fig. 22. The average degradation ratio of throughput within 3000s.

malicious flow rules are inactive for most of the time, most malicious rules are migrated to the cache region to prevent possible consumption on TCAM.

Identification of Malicious Flows. The effectiveness of identifying malicious flows is depicted in Fig. 21. When attackers generate malicious flows with fixed or random packet size, our system can achieve excellent results, e.g., less than 0.5% false positive rate and more than 99.8% recall rate. However, advanced attackers may generate attack packets following the size of benign packets to increase its stealthiness. In such case, our system can still identify the malicious flows with no more than 1.5% false positive rate and more than 90% recall rate. It is reasonable that the average recall rate drops by about 7.0%. As the packet size of malicious flows follows that of benign flows, it indeed gets much harder to filter them out.

Degradation Ratio of Throughput. Fig. 22 shows the degradation ratio of throughput with various attack rates. In both scenarios, the average degradation ratio can be maintained less than 3% with LOFTGuard. Such results demonstrate that our system can effectively defend against the LOFT attack.

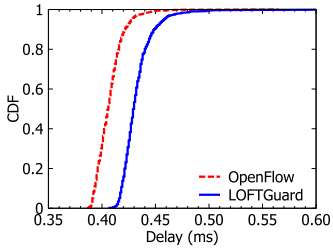


Fig. 23. The delay of control traffic.

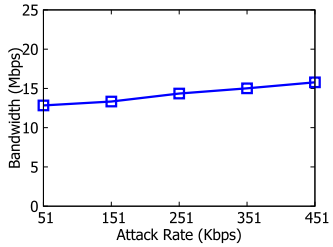


Fig. 24. The control channel bandwidth usage of LOFTGuard.

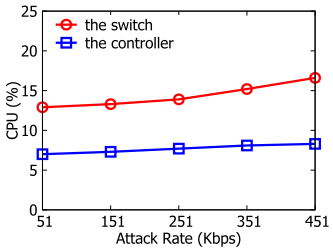


Fig. 25. The CPU usage of LOFTGuard.

C. Overhead

Delay of Control Traffic. Since *NFQUEUE* is adopted to intercept control messages, i.e., packet-in messages, LOFTGuard inevitably introduces delays for communication between the controller and switches. However, as shown in Fig. 23, LOFT only incurs 0.03s delay in average, which is negligible compared to 0.41s average delay of OpenFlow.

Control Channel Bandwidth Usage. Fig. 24 shows the average control channel bandwidth usage. The average bandwidth usage is calculated within the period from the begin of the attack to the end of attack identification. It slightly increases with a higher attack rate due to statistics collection of more flow rules in switches. However, LOFTGuard only consumes less than 16.0 Mbps bandwidth, which is small compared to the available 1000 Mbps bandwidth of control channel.

CPU Usage. We evaluate the average CPU usage of LOFTGuard on both the switch and the controller. The average CPU usage is calculated within the period from the begin of the attack to the end of identifying the attack. Fig. 25 shows that LOFTGuard slightly increases the CPU usage with a higher attack rate in both the switch and the controller. The results are reasonable since more packets will be processed by the CPU of the switch and more rules will be inspected by the controller when the attack rate goes high. Generally speaking, LOFTGuard consumes less than 17% CPU usage in the switch and less than 9% CPU usage in the controller. Note that the overall CPU usage of an SDN switch typically is under 30% since the CPU only processes the control messages. Our LOFTGuard makes use of the free CPU resources.

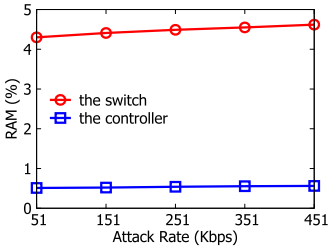


Fig. 26. The RAM usage of LOFTGuard.

RAM Usage. Fig. 26 shows the average RAM usage of LOFTGuard, which is calculated within the period from the begin of the attack to the end of identifying the attack. We can see that LOFTGuard consumes less than 5% RAM usage in the switch and 1% RAM in the controller.

VIII. Related Work

SDN Probing Techniques. Several SDN probing approaches have been proposed [12], [15], [46], [47], [48], [49], [50]. Shin et al. [15] present an SDN scanner to infer whether or not a network is using SDN by observing response time of packets. Cui et al. [46] further analyze the feasibility of SDN fingerprint in practical SDN deployments. Achleitner et al. [47] introduce SDNMap to infer the composition of flow rules in a network. Klöti et al. [12] identify whether or not there are aggregation flow rules in SDN by timing the TCP setup. Liu et al. [48] build a Markov model of an SDN switch to infer whether a target flow has recently occurred. Sonchack et al. [49] learn host communication patterns, ACL entries and network monitoring settings by injecting lots of timing pings. Leng et al. [50] design an inference attack that can learn the approximate table size of an SDN switch. Yu et al. [51] leverage cache-like behaviors of flow tables to accurately infer the internal configuration and state. Above work motivates the probing phase of our LOFT attack. However, we enable probing to accurately infer detailed timeout settings of flow rules and bitmasks in the match fields, which are essential to quantitatively analyze the minimum attack rate and construct LOFT. Particularly, we can accurately infer the timeout values even if both the idle timeout and the hard timeout are set in a flow rule, which is not addressed in [50].

Security for SDN Data Plane. There exist several studies on SDN data plane security [10], [11], [12], [52]. Antikainen et. al [52] study a wide range of attacks with a compromised switch, such as eavesdropping network traffic and man-in-the-middle attacks. Zhang et al. [53] study reflection attacks that generate massive control messages to consume processing capability of switches. Moreover, prior work [10], [11], [12] studies flow tale overflow threats that are brute-force and high-rate attacks. They generate massive random packets per second and can be easily detected by existing defenses [10], [11], [13]. Although Shin et al. [15] and Pascoal et al. [16] attempt to reduce the attack rate, they either lack detailed configurations of flow rules or require thousands of hosts in SDN, which make attacks difficult to be successfully launched in real world. Different from existing overflow attacks, LOFT is a practical attack that infers detailed SDN network settings in advance and efficiently overflows flow tables with low-rate traffic based on the probed settings that ensure the effectiveness of the attack.

Several countermeasures have been provided to protect SDN against flow table overflow. Most defense systems share flow

tables among switches to make SDN more resilient to table overflow [18], [19], [20] or apply customized flow rule eviction strategies to reduce the attack impacts [10], [16], [21]. Such passive defenses cannot completely solve the LOFT attack due to the lack of the ability to identify and block attack flows. Attack flows can still compete with benign flows to occupy flow table. A few countermeasures detect attacks and block identified attack flows [13], [14], [23], [24]. However, they cannot effectively defeat the LOFT attack since they are based on the underline assumption that attacks generate short frequent flows at a high rate, where LOFT crafts long infrequent flows at a low rate. We present LOFTGuard to protect SDN against this new attack.

Other Security Work for SDN. Existing studies have focused on the security of SDN for many aspects. Xu et al. [54] present harmful race conditions in SDN controllers. Dixit et al. [55] identify novel vulnerabilities in SDN datastores. Jero et al. [56] provide binding attacks to break the bindings of all layers of the SDN stack. Hong et al. [57] study SDN topology poisoning attacks. Xiao et al. [58] provide data dependency creation and chaining attacks that allow attackers to abuse and poison previously unreachable target methods in SDN controllers. Cao et al. [59] identify a new vulnerability named buffered packet hijacking that allows malicious applications to bypass many defense systems and launch various attacks. SDN-Rootkits [60] provides rootkit techniques to subvert SDN controllers. Tok et al. [61] identify several attacks resulting from abused DHCP services in SDN and design DHCPguard to mitigate them. FloodGuard [14], LineSwitch [62], and FloodDefender [13], [22] mainly prevent SDN from high-rate saturation attacks against the controller. Our defense aims to prevent SDN from low-rate saturation attacks against the data plane. Furthermore, a number of studies focus on automatic vulnerability discovery [63], [64], [65], [66], network security forensics [67], [68], and legacy DoS attack mitigation [69] in SDN. Our paper presents a data plane attack to significantly degrade the network performance with low-rate attack traffic, which is orthogonal to those work.

IX. Conclusion

In this paper, we provide the LOFT attack that seriously challenges the security of SDN data plane. By accurately inferring the detailed settings of flow rules and plotting the attack strategies, LOFT can efficiently overflow flow tables of switches at a minimum attack rate. It significantly degrades the network performance and incurs potential network DoS with an attack rate of only tens of Kbps. Experiments in a real SDN testbed demonstrate its feasibility and effectiveness. We also propose and evaluate LOFTGuard, a data-to-control plane defense system that can effectively defeat the attack with a small overhead.

References

[1] S. Jain et al., "B4: Experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 3–14.

[2] *Microsoft Azure and Software Defined Networking*. Accessed: Jan. 17, 2019. [Online]. Available: https://technet.microsoft.com/en-us/windows-server-docs/networking/sdn/%azure_and_sdn

[3] S. Jia, X. Jin, G. Ghasemiesfeh, J. Ding, and J. Gao, "Competitive analysis for online scheduling in software-defined optical WAN," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2017, pp. 1–9.

[4] R. Jang, D. Cho, Y. Noh, and D. Nyang, "RFlow+: An SDN-based WLAN monitoring and management framework," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2017, pp. 1–9.

[5] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Enabling practical software-defined networking security applications with OFX," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[6] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "CacheFlow: Dependency-aware rule-caching for software-defined networks," in *Proc. Symp. SDN Res.*, Mar. 2016, pp. 1–12.

[7] *Cisco Plug-in for OpenFlow Configuration Guide 1.3*. Accessed: Jan. 17, 2019. [Online]. Available: http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus/openflow/b_op%enflow_agent_nxos_1_3.pdf

[8] *IBM Networking OS? 7.4 ISCLI-Industry Standard CLI for the Rack-Switch G8264*. Accessed: Jan. 17, 2019. [Online]. Available: http://www-01.ibm.com/support/docview.wss?uid=isg3T7000580&aid=1

[9] *OpenFlow Switch Specification v1.3.4*. Accessed: Jan. 17, 2019. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.%3.0.pdf

[10] Y. Qian, W. You, and K. Qian, "OpenFlow flow table overflow attacks and countermeasures," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, Jun. 2016, pp. 205–209.

[11] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[12] R. Kloti, V. Kotronis, and P. Smith, "OpenFlow: A security analysis," in *Proc. 21st IEEE Int. Conf. Netw. Protocols (ICNP)*, Oct. 2013, pp. 1–6.

[13] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "FloodDefender: Protecting data and control plane resources under SDN-aimed DoS attacks," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2017, pp. 1–9.

[14] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS attack prevention extension in software-defined networks," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 239–250.

[15] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 165–166.

[16] T. A. Pascoal, Y. G. Dantas, I. E. Fonseca, and V. Nigam, "Slow TCAM exhaustion DDoS attack," in *IFIP SEC*. Cham, Switzerland: Springer, 2017, pp. 17–31.

[17] *Caida Passive Monitor: Chicago B*. Accessed: Jan. 17, 2019. [Online]. Available: http://www.caida.org/data/passive/trace_stats/chicago-B/2015/?monitor=201502%19-130000.UTC

[18] B. Yuan, D. Zou, S. Yu, H. Jin, W. Qiang, and J. Shen, "Defending against flow table overloading attack in software-defined networks," *IEEE Trans. Services Comput.*, vol. 12, no. 2, pp. 231–246, Mar. 2019.

[19] Z. Guo et al., "STAR: Preventing flow-table overflow in software-defined networks," *Comput. Netw.*, vol. 125, pp. 15–25, Oct. 2017.

[20] S. Qiao, C. Hu, X. Guan, and J. Zou, "Taming the flow table overflow in OpenFlow switch," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 591–592.

[21] M. Zhang, J. Bi, J. Bai, Z. Dong, Y. Li, and Z. Li, "FTGuard: A priority-aware strategy against the flow table overflow attack in SDN," in *Proc. SIGCOMM Posters Demos*, Aug. 2017, pp. 141–143.

[22] S. Gao, Z. Peng, B. Xiao, A. Hu, Y. Song, and K. Ren, "Detection and mitigation of DoS attacks in software defined networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1419–1433, Jun. 2020.

[23] R. Durner, C. Lorenz, M. Wiedemann, and W. Kellerer, "Detecting and mitigating denial of service attacks against the data plane in software defined networks," in *Proc. IEEE Conf. Netw. Softwarization (NetSoft)*, Jul. 2017, pp. 1–6.

[24] T. Xu, D. Gao, P. Dong, C. H. Foh, and H. Zhang, "Mitigating the table-overflow attack in software-defined networking," *IEEE Trans. Netw. Service Manage.*, vol. 14, no. 4, pp. 1086–1097, Dec. 2017.

[25] J. Cao, M. Xu, Q. Li, K. Sun, Y. Yang, and J. Zheng, "Disrupting SDN via the data plane: A low-rate flow table overflow attack," in *SecureComm*. Cham, Switzerland: Springer, 2017, pp. 356–376.

[26] J. F. Box, "Guinness, Gosset, Fisher, and small samples," *Stat. Sci.*, vol. 2, no. 1, pp. 45–52, Feb. 1987.

[27] G. Zhao, H. Xu, S. Chen, L. Huang, and P. Wang, "Joint optimization of flow table and group table for default paths in SDNs," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1837–1850, Aug. 2018.

[28] Y. Zhai, H. Xu, H. Wang, Z. Meng, and H. Huang, "Joint routing and sketch configuration in software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 28, no. 5, pp. 2092–2105, Oct. 2020.

[29] B. Isyaku, M. S. M. Zahid, M. B. Kamat, K. A. Bakar, and F. A. Ghaleb, "Software defined networking flow table management of OpenFlow switches performance and security challenges: A survey," *Future Internet*, vol. 12, no. 9, p. 147, Aug. 2020.

[30] Y. Xue, J. Peng, K. Han, and Z. Zhu, "On table resource virtualization and network slicing in programmable data plane," *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 1, pp. 319–331, Mar. 2020.
[31] L. Zhang, R. Lin, S. Xu, and S. Wang, "AHTM: Achieving efficient flow table utilization in software defined networks," in *Proc. IEEE Global Commun. Conf.*, Dec. 2014, pp. 1897–1902.
[32] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "Botnet in DDoS attacks: Trends and challenges," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2242–2270, 4th Quart., 2015.
[33] *Floodlight SDN Controller*. Accessed: Jan. 17, 2019. [Online]. Available: http://www.projectfloodlight.org/floodlight/
[34] *Floodlight Forwarding Application*. Accessed: Jan. 17, 2019. [Online]. Available: https://github.com/floodlight/floodlight/tree/master/src/main/java/net/flood%lightcontroller/forwarding
[35] *CAIDA Passive Monitor: Chicago B*. Accessed: Jan. 17, 2019. [Online]. Available: http://www.caida.org/data/passive/trace_stats/chicago-B/2015/?monitor=201502%19-130000.UTC
[36] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th Annu. Conf. Internet Meas. (IMC)*, 2010, pp. 267–280.
[37] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf. (IMC)*, 2009, pp. 202–208.
[38] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDoS defense," in *Proc. USENIX Secur.*, 2015, pp. 817–832.
[39] V. Vapnik, *Statistical Learning Theory*, vol. 3. New York, NY, USA: Wiley, 1998.
[40] *AS4610-54T Data Center Switch*. Accessed: Jan. 17, 2019. [Online]. Available: https://www.edge-core.com/productsInfo.php?cls=&cls2=&cls3=46&id=21
[41] *Pica8 PicOS*. Accessed: Jan. 17, 2019. [Online]. Available: https://www.pica8.com/wp-content/uploads/Pica8-Datasheet.pdf
[42] *PICOS Command Reference*. Accessed: Jan. 17, 2019. [Online]. Available: https://docs.pica8.com/display/picos2102cg/ovs-ofctl+Common+Commands
[43] (2010). *Data Set for IMC 2010 Data Center Measurement*. [Online]. Available: http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html
[44] (2018). *Traffic and Tools*. [Online]. Available: http://traffic.comics.unina.it/Traces/ttraces.php
[45] *Using NFQUEUE and Libnetfilter_Queue*. [Online]. Available: https://home.regit.org/netfilter-en/using-nfqueue-and-libnetfilter_queu%e/
[46] H. Cui, G. O. Karame, F. Klaedtke, and R. Bifulco, "On the fingerprinting of software-defined networks," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 10, pp. 2160–2173, May 2016.
[47] S. Achleitner, T. L. Porta, T. Jaeger, and P. McDaniel, "Adversarial network forensics in software defined networking," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 1–13.
[48] S. Liu, M. K. Reiter, and V. Sekar, "Flow reconnaissance via timing attacks on SDN switches," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 1–11.
[49] J. Sonchack, A. Dubey, A. J. Aviv, J. M. Smith, and E. Keller, "Timing-based reconnaissance and defense in software-defined networks," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 89–100.
[50] J. Leng, Y. Zhou, J. Zhang, and C. Hu, "An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network," 2015, *arXiv:1504.03095*.
[51] M. Yu, T. Xie, T. He, P. McDaniel, and Q. K. Burke, "Flow table security in SDN: Adversarial reconnaissance and intelligent attacks," *IEEE/ACM Trans. Netw.*, vol. 29, no. 6, pp. 2793–2806, Dec. 2021.
[52] M. Antikainen, T. Aura, K. M. Särelä, and S. Fischer-Hübner, "Spook in your network: Attacking an SDN with a compromised openflow switch," in *NordSec*. Cham, Switzerland: Springer, 2014, pp. 229–244.
[53] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai, "Control plane reflection attacks in SDNs: New attacks and countermeasures," in *RAID*. Cham, Switzerland: Springer, 2018, pp. 161–183.
[54] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN control plane," in *Proc. USENIX Secur.*, 2017, pp. 451–468.
[55] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, "AIM-SDN: Attacking information mismanagement in SDN-datastores," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 664–676.
[56] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *Proc. USENIX Secur.*, 2017, pp. 415–432.
[57] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.
[58] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu, "Unexpected data dependency creation and chaining: A new attack to SDN," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1512–1526.
[59] J. Cao, R. Xie, K. Sun, Q. Li, G. Gu, and M. Xu, "When match fields do not need to match: Buffered packets hijacking in SDN," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–33.
[60] C. Röpke and T. Holz, "SDN rootkits: Subverting network operating systems of software-defined networks," in *RAID*. Cham, Switzerland: Springer, 2015, pp. 339–356.
[61] M. S. Tok and M. Demirci, "Security analysis of SDN controller-based DHCP services and attack mitigation with DHCPguard," *Comput. Secur.*, vol. 109, Oct. 2021, Art. no. 102394.
[62] M. Ambrosin, M. Conti, F. D. Gaspari, and R. Poovendran, "LineSwitch: Tackling control plane saturation attacks in software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1206–1219, Apr. 2017.
[63] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.
[64] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "Beads: Automated attack discovery in openflow-based SDN systems," in *RAID*. Cham, Switzerland: Springer, 2017, pp. 311–333.
[65] B. E. Ujcich, U. Thakore, and W. H. Sanders, "ATTAIN: An attack injection framework for software-defined networking," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 567–578.
[66] B. E. Ujcich et al., "Automated discovery of cross-plane event-based vulnerabilities in software-defined networking," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.
[67] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards fine-grained network security forensics and diagnosis in the SDN era," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 3–16.
[68] H. Li, H. Hu, G. Gu, G.-J. Ahn, and F. Zhang, "VNIDS: Towards elastic security with safe and efficient virtualization of network intrusion detection systems," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 17–34.
[69] D. Tang, Y. Yan, S. Zhang, J. Chen, and Z. Qin, "Performance and features: Mitigating the low-rate TCP-targeted DoS attack via SDN," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 1, pp. 428–444, Jan. 2022.

Jiahao Cao received the B.Eng. degree from the Beijing University of Posts and Telecommunications in 2015 and the Ph.D. degree from Tsinghua University, Beijing, China, in 2020. He has been a Visiting Scholar at George Mason University. He is currently a Post-Doctoral Researcher with the Department of Computer Science and Technology, Tsinghua University. His current research interests include SDN security, routing security, and network traffic analysis.

Mingwei Xu (Senior Member, IEEE) received the B.Sc. and Ph.D. degrees from Tsinghua University. He is currently a Full Professor with the Department of Computer Science, Tsinghua University. His research interests include computer network architecture, high-speed router architecture, and network security.

Qi Li (Senior Member, IEEE) received the Ph.D. degree from Tsinghua University, Beijing, China. He is currently an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network and system security, particularly in internet and cloud security, mobile security, and big data security. He is also an Editorial Board Member of IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING and *ACM DTRAP*.

Kun Sun (Member, IEEE) received the Ph.D. degree from the Department of Computer Science, North Carolina State University. He is currently an Associate Professor with the Department of Information Sciences and Technology, George Mason University. He is also the Associate Director of the Center for Secure Information Systems and the Director of the Sun Security Laboratory, George Mason University. He has more than 15 years working experience in both industry and academia on systems and network security.

Yuan Yang (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees from Tsinghua University. He was a Visiting Ph.D. Student with The Hong Kong Polytechnic University from 2012 to 2013. He is currently a Research Assistant Professor with the Department of Computer Science and Technology, Tsinghua University. His major research interests include computer network architecture, routing protocol, and green networking.