Disrupting SDN via the Data Plane: A Low-Rate Flow Table Overflow Attack

Jiahao Cao[1,2], Mingwei Xu[1,2(✉)], Qi Li[1,3], Kun Sun[4], Yuan Yang[1,2], and Jing Zheng[1,3]

[1] Department of Computer Science and Technology, Tsinghua University, Beijing, China
{caojh15,zhengj14}@mails.tsinghua.edu.cn, xumw@tsinghua.edu.cn, qi.li@sz.tsinghua.edu.cn, yyang@csnet1.cs.tsinghua.edu.cn

[2] Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing, China

[3] Graduate School at Shenzhen, Tsinghua University, Beijing, China

[4] Department of Information Sciences and Technology, George Mason University, Fairfax, USA
ksun3@gmu.edu

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 data plane based attack that uses a *minimum* rate of attack packet to disrupt SDN. 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 with 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 experiment results show that LOFT can incur significant network performance degradation and potential network DoS at an attack rate of only tens of Kbps.

Keywords: Software-Defined Networking · Low-rate attack
Flow table overflow

1 Introduction

By decoupling the control plane and the data plane, Software-Defined Networking (SDN) emerges as a promising network architecture design that provides 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 amount 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. In particular, the SDN data plane (or SDN switches) is vulnerable to flow table overflow. First, it is “dumb”, i.e., for a flow that cannot match any installed flow rules in the switch flow table, the switch will generate packet-in messages to query a logically centralized controller for a new flow rule. Therefore, an attacker may abuse it to send crafted packets to trigger new rule installation. Second, most modern SDN-enabled hardware switches only support a small number of flow rules, e.g., thousands of rules [6–8], which are stored in power-hungry and expensive Ternary Content Addressable Memory (TCAM) to achieve high lookup performance [6,9]. The limited storage space of TCAM may be easily overflowed.

To effectively overflow SDN switches, existing attacks [10–12] normally generate a large number of random packets per second, but they can be easily captured by the existing defenses [10,11,13,14]. Shin and Gu [15] attempted to reduce the number of the required attack packets; however, since they did not systematically consider various configurations of flow rules (e.g., lifetime of flow rules), their attacks can fail in practice. Considering detailed network configurations in SDN, in this paper, we would like to ask:

- *Can we successfully construct a low-rate attack to SDN data plane and keep the flow table overflowed over time by generating a minimal rate of attack packets?*

Our answer is yes, though it is challenging. To decrease the attack packet rate, an attacker should craft packets so that each of them can trigger a new rule installation, which requires the attacker to know precisely what packets will trigger new rule installation. However, the rule installation logic is decided by the separated SDN controller, and the attacker usually has no access to those information. Moreover, flow rules are usually set with timeouts by the controller and will be removed when they expire. The attackers need to understand the timeout settings of the flow rules so as to choose the best attack strategies and decide the minimal attack rate.

To address the above challenges, we present a two-phase low-rate flow table overflow attack called LOFT, which consists of *probing phase* and *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 inferring 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 plane and data plane 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 using some specific match fields so that each packet can trigger a new flow rule installation. Meanwhile, based on the timeout configurations, it can compute the minimal packet rate to keep flow tables overflowed over time.

To demonstrate the feasibility and effectiveness of LOFT attack, we conduct experiments in a real SDN testbed that consists of commercial hardware switches. 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. During the attacking phase, it can successfully decrease the available maximum throughput from 850 Mbps to 10 Mbps for a new flow, and increase RTT from 0.1 ms to above 1000 ms. Moreover, it incurs a 69% degradation of network throughput at an attack rate of around 50 Kbps. To summarize, we make the following contributions:

- We propose a low-rate flow table overflow attack called LOFT, which can effectively degrade the network performance.
- We develop probing algorithms that can accurately infer network configurations of flow rules and compute the minimal feasible attack rate to successfully launch the LOFT attack.
- We conduct experiments in a real SDN testbed consisting of commercial hardware switches to verify the effectiveness of LOFT attack.

2 Background and Threat Model

2.1 Software-Defined Networking

SDN enables network innovations by decoupling the control plane and the data plane. The control plane contains a logically centralized controller that takes the full control of the network. Various applications can be developed atop the controller to offer complicated network functions, such as traffic engineering. The SDN data plane consists of SDN switches that conduct packets processing and forwarding according to the decisions made by the controller.

Nowadays the leading southbound protocol of SDN is OpenFlow [16]. OpenFlow allows a controller to define various forwarding behaviors of switches by installing related flow rules. There are two approaches to install flow rules, i.e., *proactively* and *reactively*. In proactive approach, flow rules are pre-installed before all the traffic comes. While in reactive approach, flow rules are installed dynamically. When an OpenFlow switch receives a new packet that can not match any installed flow rules, it generates a *packet-in* message to the controller to request a new forwarding rule. The controller may either send *packet-out* messages to the switch for one-time packet processing or send *flow-mod* messages to install flow rules in the switches that are among the calculated routing path.

In OpenFlow, each flow rule mainly consists of (i) match fields to match against incoming packets, (ii) a set of instructions that define how to process the matching packets, (iii) counters to get flow statistics and (iv) timeouts defining lifetime of the rule. Particularly, match fields of a flow rule specify what packets can be handled. According to the OpenFlow Switch Specification 1.3 [9], up to 39 match fields can be added in a rule to provide flexible flow control, such as MAC source/destination address, IPv4 source/destination address, and TCP source/destination port. Each match field in a rule can be exact value, wildcarded (i.e., matching any value) or in some cases with bitmasks. Note that except the default table-miss rule aiming at generating packet-in messages, each rule has at least one match field that conducts exact match with or without bitmasks. Moreover, each flow rule can be set with two types of timeout, namely, *idle timeout* and *hard timeout*. A flow rule will be automatically removed when either the hard timeout has passed or the idle timeout has passed without receiving a packet that matches the flow rule. Both timeouts can be set independently by a controller or applications on the controller.

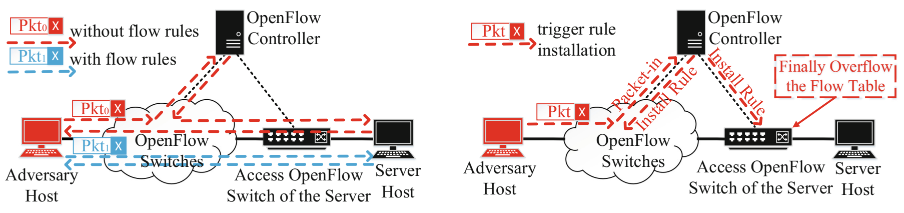
2.2 Threat Model

In our threat model, the attacker seeks to infer the network configurations and launch LOFT attack to effectively overflow the flow tables of victim switches in a stealthy way. To achieve it, the attacker needs to have (or control) a host that is attached to the victim network and can send packets to other hosts in the network. We do not require that the attacker has any prior knowledge on the network configurations or compromises any switches and controller. Moreover, we assume that the controller adopts *reactive rule installation*, which is widely used in most OpenFlow networks for flexible and dynamic flow control [13,14].

3 Overview of the LOFT Attack

We present an overview of LOFT attack that 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 the key observation that the small-sized flow tables in OpenFlow switches may be easily overflowed by malicious traffic flows and leave no space for normal traffic flows, since the centralized controller treat malicious flows and normal 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, for example, the access switch of a target network server.

LOFT consists of two phases: *the probing phase* and *the attacking phase*, as shown in Fig. 1. The probing phase prepares for the following attacking phase.



(a) Probing Phase (b) Attacking Phase

Fig. 1. Two phases of LOFT attack.

In the probing phase, 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 flow rules. This 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, the attacker can accurately infer what packets will trigger rule installation and the related timeouts configurations of flow rules. In the attacking phase, according to the inferred results on the network configurations, the attacker crafts the minimum number of attack packets to effectively trigger flow rule installation. Meanwhile, to keep flow tables continuously overflowed over time, the attacker carefully plots the attack strategies and calculates the minimal attack rate according to the timeouts configurations. In the next two sections, we detail the two phases of LOFT attack.

4 The Probing Phase

In this section, we present our probing schemes that aim to infer configurations of flow rules, particularly, the *match fields along with their bitmasks* and *timeouts* that have direct impact on attack strategies in the attack phase.

4.1 Probing Match Fields

In order to accurately infer what fields in a packet header can be used to trigger new rule installation, we generate and craft probing packets with various field values in the packet headers in the network to measure their RTTs. A probing packet can be any packet that can trigger 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 rule for the packet exists before inferring RTT. Second, we generate a new probing packet that *changes value of one field* of the previous packet to the same destination, and measure the RTT (denoted by RTT_0). Then, we send another probing packet with the same values of header fields and measure the RTT again (denoted by RTT_1). The RTT values of the later two packets meet the following conditions:

$$\begin{cases} RTT_0 \gg RTT_1, & \text{if the changed field triggers rule installation;} \\ RTT_0 \approx RTT_1, & \text{otherwise.} \end{cases}$$

The first equation indicates that the changed field is in the set of the match fields, while the second equation 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 protocol allows some match fields with bitmasks, which can interfere with the probing. For example,

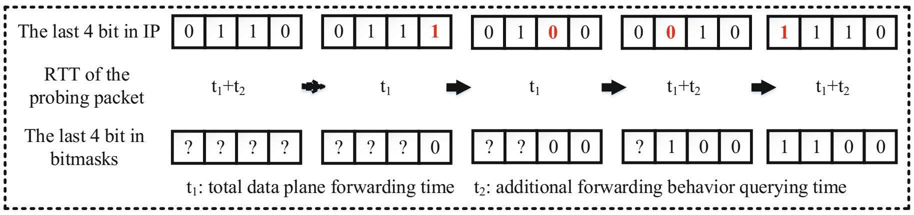

The last 4 bit in IP
0 1 1 0
0 1 1 1
0 1 0 0
0 0 1 0
1 1 1 0
RTT of the probing packet
t1+t2
t1
t1
t1+t2
t1+t2
The last 4 bit in bitmasks
? ? ? ?
? ? ? 0
? ? 0 0
? 1 0 0
1 1 0 0
t1: total data plane forwarding time
t2: additional forwarding behavior querying time

Fig. 2. An example of inferring bitmasks.

suppose that the match field is IP address with a bitmask "255.255.255.0". If we generate a probing packet with IP address "10.0.0.1" and produce another two probing packets with IP address "10.0.0.2". Then, the last two packets will not trigger new flow rule installation. In this case, we infer that IP address is not in the match fields by mistake, since the RTTs of the last two packets are close. To tackle this, we can generate additional probing packets by flipping the values of each bit in turn and reconstruct bitmasks. As shown in Fig. 2, a single bit in the bitmasks can be inferred as 1, if the RTT of the corresponding probing packet is close to the first probing packet. Otherwise, it can be inferred as 0.

Network Jitter Interference. Two RTT values may significantly deviate even if there is no new rule installation because of network jitter. We can apply the *t-test* method [17] to eliminate the impact of network jitters. In t-test, a significance level α is set with a predetermined value, and a p-value p is calculated according to the data, where p indicates 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-value 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 significant different from each other by calculating their p-value. Thereby, we can accurately infer if there exists new rule installation.

Algorithm 1. Probing Match Fields

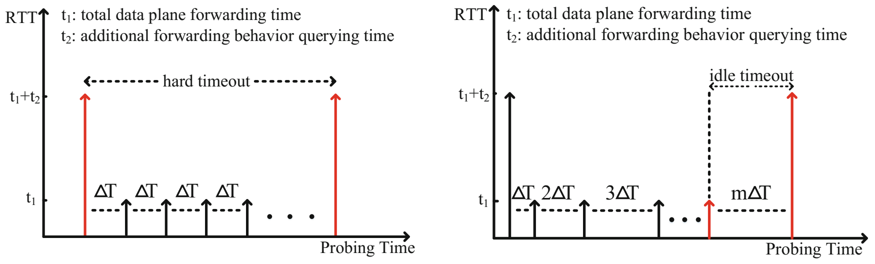
```
Input: $dst$, $F$, $n$, $\alpha$;
Output: a set of match fields $M$;
 1: $M \leftarrow \emptyset$;
 2: for each field $f \in F$ do
 3:     $pkt_0 \leftarrow build\_packet(dst, f)$;
 4:     for ($i = 0 \rightarrow n - 1$) do
 5:         $send\_packet(pkt_0)$;
 6:         $pkt \leftarrow modify\_field\_val(pkt_0, f)$;
 7:         $RTT_0[i] \leftarrow send\_packet(pkt)$;
 8:         $RTT_1[i] \leftarrow send\_packet(pkt)$;
 9:     end for
10:     $p \leftarrow t\_test(RTT_0, RTT_1)$;
11:     $b \leftarrow infer\_bitmask(f)$;
12:     if (($p < \alpha$) or ($b \neq 0$)) then
13:         $M.add(\{f, b\})$;
14:     end if
15: end for
```

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. The set of fields F includes typically fields used in flow rules, such as MAC addresses, IP addresses and port number. The significance level α

is set to 0.05, which is a typical value and widely used in t-test. As shown in the algorithm, by conducting several rounds of changing fields, we can infer match fields and the bitmasks (if they exist) used in the network configurations.

4.2 Probing Timeouts

We need to probe timeout values of flow rules so that we can calculate the minimal attack rate to keep flow tables overflowed. A packet will experience remarkable forwarding delay, if the rule matched by the packet is reinstalled by the controller after timeout expiration. Therefore, we can estimate the timeout values by measuring the elapsed time between two remarkable delays.


RTT
t_1: total data plane forwarding time
t_2: additional forwarding behavior querying time
hard timeout
t_1+t_2
t_1
ΔT ΔT ΔT ΔT
Probing Time
RTT
t_1: total data plane forwarding time
t_2: additional forwarding behavior querying time
idle timeout
t_1+t_2
t_1
ΔT 2ΔT 3ΔT mΔT
Probing Time

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

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

As shown in Fig. 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. Since there exists a remarkable RTT if a new rule is installed, we can periodically send the probing packets to the network 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 Fig. 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 successive probing packets. Here, we need to address the following issues of probing timeout values in practice.

Mutual Interference Between Timeouts. A flow rule may be configured with both hard timeout and idle timeout. In such cases, mutual interference may happen during probing, since a flow rule can be removed because of either hard timeout or idle timeout. Let us take an example. Suppose that the hard timeout of a flow rule is set to 15 s and the idle timeout is set to 10 s. In each round of probing idle timeout, we increase the time interval by 1s. After 15 s since we start the probing, a remarkable RTT will occur due to hard timeout. While the idle timeout has not taken into effect because it is reset by the probing packets. Thus, we evaluate the idle timeout as 5 s 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 successive packets in probing hard timeout values.

To overcome the problem, we probe hard timeout values first before probing idle timeout values. We note that all timeout values can only be set to an integer and the minimal valid value is 1 s. To eliminate the interference of idle timeout, we send the probing packets in a fixed interval less than 1 s, e.g., 0.5 s. Thereby, idle timeout will always be reset and will not take into effect. Thus, we can accurately infer hard timeout values. Moreover, the inferred hard timeout value is the upper bound of the idle timeout, since an idle timeout value greater than a hard timeout value in a rule is invalid. Therefore, to avoid the hard timeout interference during probing idle timeout, we enumerate all possible idle timeout values from the upper bound in a descending order. Different from the probing shown in Fig. 3(b), we decrease the time interval by 1 s from the upper bound in each round of probing idle timeout. The RTTs of two successive probing packets are close if the probing interval is larger than the idle timeout. They both experience remarkable delays, since flow rules will be removed due to the idle timeout. However, once the RTTs of two successive probing packets exhibit significant deviation, we can know that the idle timeout value is equal to the time interval between the two successive packets.

Probing Duration. It is time-consuming to probe 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 180 s and the idle timeout value is set to 10 s, the total probing time cost is 16,245 s, i.e., around 4.5 h. To effectively reduce the probing duration, we can apply binary search in probing 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 hard timeout in binary search. We can achieve this by waiting enough time to ensure removal of flow rules before sending packets in a new iteration. Thus, a flow rule will be reinstalled after a probing packet in each iteration. Therefore, the hard timeout will be reset and will not interfere with probing idle timeout in each iteration.

Network Jitter Interference. Network jitter can interfere with probing timeouts. To address this issue, we can simply send a group of packets in parallel in each iteration of probing, and apply t-test mentioned in Sect. 4.1 to determine if there is a significant deviation between two successive groups of RTTs.

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 which will be concurrently sent, the waiting interval t_{wait}, the maximal execution time of the algorithm t_{max}, and the significance level of the t-test α. Note that t_{wait} must be less than 1 s, which efficiently eliminates 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 4–9). 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 10–12). The total number of probing packets per second is $\frac{n}{t_{wait}}$. In our experiments, in order to well trade off between probing accuracy and cost, we set n to 5, t_{wait} to 0.5 s, which indicates the algorithm only requires ten packets per second to probe timeout. Moreover, the significance level α is set to 0.05 which is a typical value and widely used in t-test.

Algorithm 2. Hard Timeout Probing

```
Input: $dst$, $M$, $n$, $t_{wait}$, $t_{max}$, $\alpha$;
Output: hard timeout $t_{hard}$;
 1: $pkts[] \leftarrow build\_packets(dst, M, n)$;
 2: $t_{start} \leftarrow get\_clock\_time()$;
 3: $RTT_0[] \leftarrow send\_packets(pkts, n)$;
 4: repeat
 5:     $sleep(t_{wait})$;
 6:     $t_{end} \leftarrow get\_clock\_time()$;
 7:     $RTT_1[] \leftarrow send\_packets(pkts, n)$;
 8:     $p \leftarrow t\_test(RTT_0, RTT_1)$;
 9: until $((t_{end} - t_{start} > t_{max})$ or $(p > \alpha))$;
10: if $(t_{end} - t_{start} > t_{max})$ then
11:     $t_{hard} \leftarrow 0$;
12: else
13:     $t_{hard} \leftarrow round(t_{end} - t_{start})$;
14: end if
```

Algorithm 3 shows the pseudocode of inferring the idle timeout values by applying binary search. The inputs are similar to these used in Algorithm 2, where t_{sup} denotes the upper bound of the algorithm execution time. 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 idle timeout value, such as 500 s[1]. We send two groups of packets in each iteration of binary search and measure their RTTs (see steps 3–15). In particular, step 14 aims to ensure that flow rules can be removed after each iteration. Thus, the interference of hard timeout can be eliminated. According to Algorithm 3, we can see that the execution time is equal to $O(t_{sup} \log t_{sup})$ seconds and each iteration of probing only generates $2 * n$ packets. Similar to inferring hard timeout, we set n to 5, which indicates it only generates 10 packets in each iteration.

5 The Attacking Phase

Now we can launch the attack in this phase according to the inferred results in the probing phase. In order to increase the attack effectiveness and keep it stealthy, we generate the minimum number of attack packets that can successfully overflow flow tables. Moreover, we carefully use various attack strategies to overflow flow tables and calculate the minimal attack rates to keep the tables overflowed over time.

[1] According to our observation, idle timeout is usually not set to a large value. Normally, 500 s is large enough to serve as the upper bound (see Table 1 in Sect. 5.2).

5.1 Crafting Attack Packets

The key point is to ensure that each attack packet can effectively trigger an unique rule installation in switches. Since we know the match fields along with their bitmasks in the probing phase, we can easily achieve it by carefully changing header field values of each packet. Thereby, the minimal 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 64 B, which is the minimum size of Ethernet packets. Thus, approximate 113 KB traffic can successfully overflow a switch with 1,800 rules. Hence, the volume of the total attack traffic is small. Note that, we can also use 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. In addition, we can generate payloads of different sizes in attack packets and then randomize the packet lengths.

5.2 Calculating the Minimal Attack Rate

Now we need to compute the minimal packet rate that can continuously overflow flow tables even after flow rules expire due to hard timeout or 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 hard timeout and idle timeout. Here, we assume x and y are integers, where $x > y$[2].

(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.

Algorithm 3. Idle Timeout Probing

```
Input: $dst$, $M$, $n$, $t_{sup}$, $\alpha$;
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:     end if
14:     $sleep(r)$;
15: end while
16: if ($t_{idle} \geq t_{sup}$) then
17:     $t_{idle} \leftarrow 0$;
18: else
19:     $t_{idle} \leftarrow l$;
20: end if
```

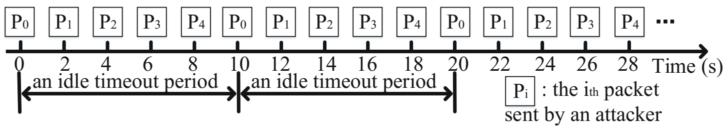
[2] As we discussed in Sect. 4, SDN does not set hard timeout values larger than idle timeout values.

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 category (III) and (IV) are widely used in default settings of different controllers (see Table 1). Thus, in this paper, we focus on developing two attack strategies that use minimal attack rate to overflow flow tables according to the settings in categories (III) and (IV).

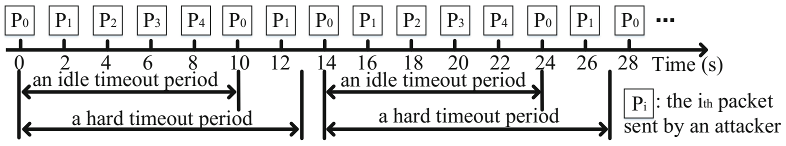
Table 1. Default timeout values in different controllers

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

Attack Strategy with Settings in Category (III). An attacker needs to fill in the flow table within an idle timeout period, and ensure consumption of entire flow table after the idle timeout expires. To achieve it, 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 Fig. 4(a)). Each packet will trigger a new rule installation if there is any available space, and the number of rules in the flow table can gradually increase.


P0 P1 P2 P3 P4 P0 P1 P2 P3 P4 P0 P1 P2 P3 P4 ···
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 Time (s)
an idle timeout period
an idle timeout period
Pi : the ith packet sent by an attacker

(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.


P0 P1 P2 P3 P4 P0 P1 P0 P1 P2 P3 P4 P0 P1 P0 ···
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 Time (s)
an idle timeout period
a hard timeout period
an idle timeout period
a hard timeout period
Pi : the ith packet sent by an attacker

(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.

Meanwhile, if the table is overflowed already, the idle timeout timer of each flow rule can be periodically refreshed within each idle timeout interval, which ensures flow rules are persistently stored in the flow table.

Now we calculate the average rate of sending attack packets within an idle timeout period. Here, C denotes is the maximum capacity of the flow table of a switch, L_i denotes the length of the i_{th} packet within an idle timeout period, and t_{idle} denotes the idle timeout. The average packet rate $\overline{v}$ can be calculated by:

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

Note that, in order to fully consume flow rules, we need to generate at least C packets, each of which triggers a new rule installation. Thus, Eq. (1) gives the minimal attack rate. Any attack rate less than $\overline{v}$ cannot fully consume the table and keep the table full over time because of expiration of flow rules incurred by the idle timeout. According to Eq. (1), we can conclude that the attack rate is small. For example, assuming the flow table capacity is 1,800 flow rules, the idle timeout value is set to 20 s, and the size of each packet is 64B, the minimal attack rate to overflow flow tables is only 46 Kbps. Such low attack rate ensures that no malicious rules will expire and the attack traffic can be effectively concealed in the benign traffic.

Attack Strategy with Setting 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 hard timeout. Since the timeout settings have been known, an attacker can easily achieve it. However, to make the attack have a constant attack rate and easy to be launched, we properly delay the sending time when a rule needs to be reinstalled. As is shown in Fig. 4(b), the rule triggered by p_0 is removed at 13 s due to the hard timeout. We reinstall the rule at 14 s rather than at 13 s so as to keep the time interval between two successive attacking packets equal. In this way, the average attack rate is same with that in category III, which can be calculated by Eq. (1).

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 can construct the attack to occupy n flow rules. After this, we can infer if the flow tables are full by sending some 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 insertion of a new flow rule but none of them have been successfully installed. Otherwise, we can launch another round of probing to occupy n' flow rules. Note that we can not accurately infer the size of flow table since the flow rules used by benign traffic always change. However, in practice, we need not to know the accurate table capacity. We can increase n using a larger number, such as 2,000. By repeating the procedure for several times, we can gradually occupy the flow tables until they are all consumed.

6 Attack Evaluation

6.1 Experiment Setup

Figure 5 shows the topology of our hardware SDN testbed. We use Floodlight [18] OpenFlow controller running in a Intel Xeon Quad-Core CPU E5504 and 12 GB RAM machine. The *Forwarding* application [19] that provides topology discovery and basic forwarding services runs on the controller by default. Two commercial hardware OpenFlow switches, EdgeCore AS4610-54T [20], are deployed in the testbed. Each switch allows 1,800 TCAM-based flow rules and infinite software flow rules[3]. An attacker host controlled by an adversary is attached to one of the switches and generates packets to attack both switches. We implement LOFT attack program in approximate 2,000 lines of C code. Moreover, to simulate real network conditions, we deploy one client host that generates background traffic and one server host that receives the traffic. We use hping3 [21] to generate 200 different benign flows and the rate of a flow is 500 Kbps. Thus, there are total 1,600 flows and 800 Mbps benign traffic in the network.

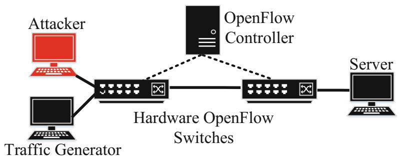


Fig. 5. Hardware SDN testbed.

6.2 Measuring Attack Rate

In this experiment, we measure the attack rate of LOFT and demonstrate that it really generates the minimal rate of attack packets. Note that, in order to accurately measure the attack rate, background traffic is not generated in the experiments. As shown in Eq. (1), the minimal attack rate to overflow the flow table is impacted by the values of idle timeout. Figure 6 shows the theoretical packet rate we computed and real packet rate with respect to different idle timemout values. We can observe that the minimal attack rates are below 100 Kbps, which is consistent with the theoretical values. Moreover, we measure the average packet-in rates at different attack rates before flow table overflow. As is shown in Fig. 7, the packet-in rate is less than 300 Kbps even when the attack rate is 100 Kbps. Note that compared to existing overflow attacks [10–13] that can generate tens

[3] The performance is not given by EdgeCore but measured in our experiments.

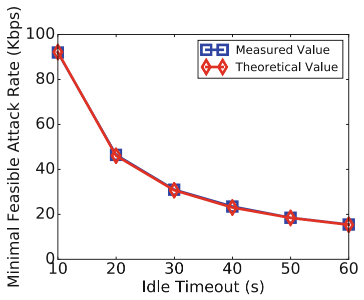


Fig. 6. The minimal attack rate with different idle timeout values.

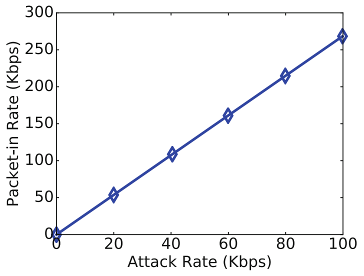


Fig. 7. Packet-in rate with different attack rate.

of Mbps attack traffic and packet-in traffic, our attack rate is relatively low and does not incur high packet-in rate. These features increase the stealthiness of the attack[4].

6.3 Evaluation of Attack Effectiveness

We conduct our attack experiments in two typical scenarios to demonstrate the effectiveness of LOFT: (I) only idle timeout is set; (II) both hard timeout and idle timeout are set. The idle timeout value is set to 20 s in both two scenarios, and the hard timeout value is set to 200 s in the second scenario. According to Eq. (1), we launch LOFT with the average attack rate at 46 Kbps in both scenarios. Since the attacker does not know the timeouts and match fields of flow rules in advance, they need to probe the configurations before launching the attack. We will evaluate the accuracy of probing in Sect. 6.4.

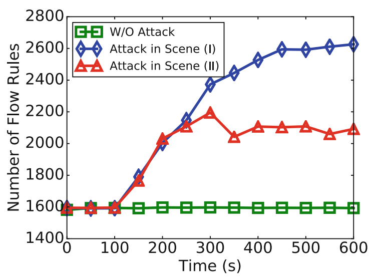


Fig. 8. The number of switch flow rules under LOFT attack.

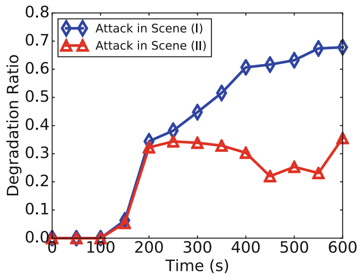


Fig. 9. Background traffic throughput degradation ratio under LOFT attack.

[4] We note that lots of benign traffic will be sent to the controller when the table is overflowed and thus the packet-in rate will significantly increase. However, the attack has successfully caused remarkable damage when it has some obvious features.

Impacts on the Number of Flow Rules. We measure the number of flow rules in the switch that connects to the server with and without the attacks in two scenarios. Figure 8 shows that the number of flow rules is around 1,600 in absence of the attack. When the attack is launched at 100 s in both attack scenarios, the number of flow rules starts to increase. At 600 s, the number of flow rules reaches to 2,610 and 2,090 in scenario (I) and scenario (II), respectively. Since the switch can store up to only 1,800 rules in TCAM, these results demonstrate that our attack can effectively overflow the scarce TCAM resources with low-rate attack traffic. In addition, we can observe that the number of rules continuously increases over time in scenario (I). However, the number of rules in scenario (II) drops at 300 s and tends to convergence after that time. The reason is that hard timeout is configured as 200 s in scenario (II) and the flow rules that are installed by attack flows always expire after the hard timeout. These rules are periodically removed and reinstalled in the switch and the number of them tends to converge.

Impacts on Throughput Degradation. To quantify the impacts of the attack on network throughput, we measure the throughput degradation ratio of the total background traffic 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 50 s. The degradation ratio is shown in Fig. 9. In scenario (I), the ratio continuously increases and reaches to 69% at 600 s. The results demonstrate that the attack can significantly degrade the throughput of the network and have accumulative damage effect on the network over time. In scenario (II), the degradation ratio reaches 36% at 600 s and the throughput degradation is less than that in scenario (I). Moreover, it decreases at 300 s and increases again at 450 s. The reason is that flow rules installed by attack packets will be periodically removed and reinstalled due to hard timeout.

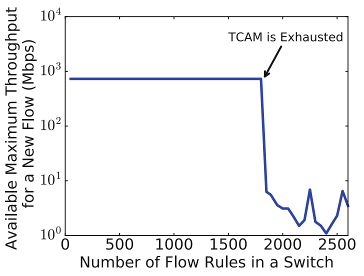


Fig. 10. Maximum throughput of a new flow with different numbers of flow rules.

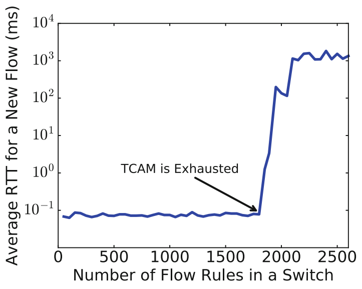


Fig. 11. Average RTT of a new flow with different numbers of flow rules.

Impacts on Maximum Throughput of a New Flow. We use *iperf* [22] to measure the available maximum throughput for a new flow with different flow rules in our attack. As is shown in Fig. 10, the available maximum throughput for a new flow significantly decreases to below 10 Mbps from 850 Mbps when the number of flow rules exceeds 1,800. The reason is that TCAM is overflowed and extra new flow rules are stored in software. Note that storing flow rules in software can not ensure high and stable forwarding performance. These results demonstrate that our attack can significantly degrade the maximum throughput of a new flow when the TCAM is overflowed.

Impacts on Forwarding Delay of a New Flow. We use *ping* to measure the average RTT of a new flow with different numbers of installed rules in our attack. 100 rounds of pings are performed for each different numbers of rules to compute the average RTT. As is shown in Fig. 11, the average RTT of a new flow significantly increases when TCAM is overflowed. We can see that the average RTT reaches to approximate 1,000 ms when the number of rules reaches to 2,100. Compared to forwarding by TCAM, software forwarding introduces remarkable delay. Moreover, the RTT does not tend to increase at the end. The possible reason is that we ignore the ICMP packets that are dropped in calculating the average RTT. Actually, when the number of rules exceeds 2,000, more ICMP packets are dropped along with the increase of the number of flow rules in our measurement. These results demonstrate that our attack can significantly increase forwarding delay of a new flow when the TCAM is overflowed.

6.4 Evaluation of Probing Accuracy

Accuracy of Probing Match Fields. The *Forwarding* application in Floodlight controller conducts fine-grained forwarding. By default, the match fields of it are configured as $\langle src\_mac, dst\_mac, src\_ip, dst\_ip, src\_port, dst\_port\rangle$ without bitmasks. To better evaluate match fields probing accuracy, we configure the match fields as $\langle src\_ip, dst\_ip\rangle$ with different bitmasks. Algorithm 1 are conducted to measure the accuracy of probing match fields. The results are summarized in Tables 2 and 3. As is shown in Table 2, when we change MAC source address, TCP source port or UDP source port in the probed packets headers, both p-values are less than 0.05. However, when we change the IP source address in the packets headers, the p-value increases and reaches 0.92, which is significantly large. Thus, we can easily infer that the IP address field is used in the forwarding rules by evaluating p-value. Table 3 shows the probing accuracy of different bitmasks. The results demonstrate that Algorithm 1 can infer the bitmasks with more than 90% accuracy. The accuracy is enough for an attacker to effectively launch LOFT attack.

Accuracy of Probing Timeout Values. We systematically measure the probing accuracy with different hard timeout and idle timeout settings. 100 rounds of probing are performed for each different setting to compute the average accuracy. Figure 12(a) shows the accuracy rate for various hard timeout values. We observe that hard timeout probing can reach more than 90% accuracy rate with different

Table 2. p-value for each changed packet header field.

Changed header field	p-value
MAC source address	0.01
TCP source port	0.03
UDP source port	0.01
IP source address	0.92

Table 3. Probing accuracy of different bitmasks.

configured bitmasks	Accuracy
255.0.0.0	91%
255.255.0.0	91%
255.255.255.0	92%
255.255.255.255	94%

hard timeout values. Similarly, Fig. 12(b) shows that idle timeout probing can also reach more than 90% accuracy rate with different idle timeout values. Note that the accuracy rate is enough to construct LOFT. We may not be able to infer correct timeout values with one round of probing. However, we can obtain the correct results by performing multiple rounds of probing.

7 Possible Defenses

In this section, we discuss possible countermeasures against the LOFT attack. We can throttle the attack at two phases, i.e., interfering with the probing and dismissing attack packets.

Thwarting Probing. We could interfere with RTT measurement to thwart the probing. An SDN controller can generate artificial jitter during delivery of the very first few packets of flows. For example, the controller does not generate a flow rule for a new flow immediately once it receives packet-in messages. Instead, it deliberately waits for a random delay before sending the packets back to the switches. And it installs the flow rules after receiving several packet-in messages for the new flow. Therefore, an attacker cannot accurately infer whether there are new rules installed or not for probing packets. The potential disadvantage is that the approach also incurs extra forwarding delays for benign packets and requires the controller to process more packet-in messages.

Another approach could possibly adopt dynamic timeout values. According to our investigations, we find that almost all applications atop of the same controllers set flow rules with fixed values. We suggest that the applications set different timeout values once there is a new rule installed or a rule is reinstalled due to rule expiration. Thereby, an attacker could not easily infer the timeout values set by the controller. In this case, overflowing the flow table at low-rate is not likely to succeed due to lacking accurate information of timeout values.

Dismissing Attacks. Significant work exists on taming flow table overflow, which falls in two categories: mitigating normal flow table overflow triggered by many benign flows [23,24], and defending against malicious flow table overflow attacks [10,11,13]. Solutions of the first category assume that there are no overflow attacks. They cannot effectively throttle persistent and malicious flow table

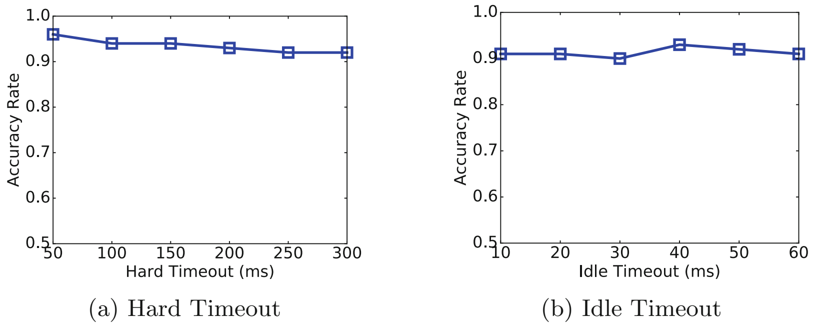


Fig. 12. Average timeout probing accuracy. (a) shows the probing accuracy of hard timeout with Algorithm 2; (b) shows the probing accuracy of idle timeout with Algorithm 3.

overflow. Solutions of the second category can effectively resist malicious flow table overflow. However, they are based on the underlying assumption that the attack rate is high. These defenses are not complete in terms of resisting the low-rate overflow attack, as (i) they may not detect the attack until the flow table is overflowed[5], and (ii) they lack the ability to accurately identify low-rate malicious flows so as to throttle them.

In order to specifically defend against the low-rate flow table overflow attack, a possible countermeasure is to monitor and identify flow table consumption patterns generated by the attack and then flush suspicious flow rules in real time. As shown in Fig. 8, under the LOFT attack, we can observe that the number of the installed flow rules continually increase before the table is full, and the increase rate is slow. These features could be used to capture the attack. Once the attack is detected, the SDN controller could actively delete such suspicious flow rules. A suspicious rule could be the rule that is always in the flow table but forwards very few number of packets per second. Besides, since the attack periodically generates packets to refresh the rule, the forwarding rate of a suspicious rule could show an periodicity pattern, which could further help to locate and flush a rule created by the attack packets.

8 Related Work

SDN Probing Techniques. Several SDN probing approaches have been proposed [12,15,25–29]. Shin and Gu [15] present an SDN scanner to infer whether or not a network is using SDN by observing response time of packets. Cui et al. [25] further analyze the feasibility of SDN fingerprint in practical SDN deployments. Achleitner et al. [26] introduce SDNMap to infer the composition of flow

[5] The defenses will be enabled only when there are lots of packet-in packets per second. However, our attack does not trigger high-rate packet-in packets before overflowing the flow table.

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. [27] build a Markov model of an SDN switch which allows attackers to select the best probes to infer whether a target flow has recently occurred. Sonchack et al. [28] learn host communication patterns, ACL entries and network monitoring settings by injecting lots of timing pings. Leng et al. [29] design an inference attack that can learn the approximate table size of an SDN switch, by estimating the significant changes in response time of requests when flow tables are overflowed. Above work motivates the probing phase of our LOFT attack. However, different from them, we enable probing to accurately infer detailed timeout configurations of flow rules and bitmasks in the match fields, which are essential to quantitatively analyze the minimal attack rate and construct the LOFT attack. 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 [29].

SDN Data Plane Security. There exist several studies on SDN data plane security [10–12,30]. Antikainen et. al [30] study a wide range of attacks, such as eavesdropping network traffic and man-in-the-middle attacks. They require a strong assumption that an attacker can compromise SDN switches. Prior work [10–12] also studies flow tale overflow threats, which are brute-force and high-rate attacks. They generate many random packets per second and can be easily detected by existing defenses [10,11,13]. Different from them, LOFT is a sophisticated attack that infers SDN network configurations in advance and then efficiently overflows flow tables with low-rate traffic in a stealthy way. LOFT may seem similar to the attack proposed by Shin and Gu [15] that constructs packets according to the probed configurations. However, their attack can fail in practice because it does not consider detailed settings of the flow rules, e.g., lifetime values and bitmasks in the match fields that significantly impacts the effectiveness of the attack. LOFT systematically measures configurations of flow rules and generates packets with minimal feasible attack rate according to the probed configurations such that it ensures the effectiveness of the attack.

SDN Control Plane Security. The security issues of SDN control plane have been widely studied recently. SDN-Rootkits [31] provides rootkit techniques to subvert SDN controllers. SDNShield [32] and SE-Floodlight [33] focus on the application-level security on SDN controllers. These security extensions prevent SDN against malicious or buggy applications. FortNox [34] introduces the dynamic tunneling attacks that violate security policies and provides role-based authorization to defend against those attacks. VeriFlow [35] investigates the correctness of flow rules. AvantGuard [36] and FloodGuard [14] prevent SDN from saturation attacks against the controller. TopoGuard [37] studied SDN topology poisoning attacks. Our paper presents a data plane attack to significantly degrade the network performance with low-rate attack traffic, which is orthogonal to these previous work.

9 Conclusion

In this paper, we design and implement a data plane attack called LOFT that seriously challenges the security of SDN. By accurately inferring the network configurations of flow rules and plotting the attack strategies in advance, LOFT can efficiently overflow the flow tables of switches at minimal feasible attack rate. It can significantly degrade the network performance and incur potential network DoS at an attack rate of only tens of Kbps. Experiments in a real SDN testbed consisting of commercial hardware switches demonstrate the feasibility and effectiveness of the attack.

Acknowledgment. The research is partially supported by the National Natural Science Foundation of China under Grant 61572278 and 61625203, the National Key Research and Development Program of China under Grant 2016YFB0800102 and 2016YFC0901605, and U.S. Office of Naval Research under Grant N00014-16-1-3214 and N00014-16-1-3216.

References

1. Jain, S., et al.: B4: experience with a globally-deployed software defined WAN. In: SIGCOMM. ACM (2013)
2. Microsoft Azure and Software Defined Networking. https://technet.microsoft.com/en-us/windows-server-docs/networking/sdn/azure_and_sdn
3. Jia, S., et al.: Competitive analysis for online scheduling in software-defined optical WAN. In: INFOCOM, pp. 1–9. IEEE (2017)
4. Jang, R.H., et al.: Rflow$^+$: an SDN-based WLAN monitoring and management framework. In: INFOCOM, pp. 1–9. IEEE (2017)
5. Sonchack, J., et al.: Enabling practical software-defined networking security applications with OFX. In: NDSS (2016)
6. Katta, N., et al.: Infinite cacheflow in software-defined networks. In: HotSDN, pp. 175–180. ACM (2014)
7. Cisco Plug-in for OpenFlow Configuration Guide 1.3. http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus/openflow/b_openflow_agent_nxos_1_3.pdf
8. IBM Networking OS? 7.4 ISCLI-Industry Standard CLI for the RackSwitch G8264. http://www-01.ibm.com/support/docview.wss?uid=isg3T7000580&aid=1
9. OpenFlow Switch Specification v1.3.4. https://www.opennetworking.org
10. Qian, Y., et al.: Openflow flow table overflow attacks and countermeasures. In: European Conference on Networks and Communications, pp. 205–209. IEEE (2016)
11. Dhawan, M., et al.: SPHINX: detecting security attacks in software-defined networks. In: NDSS (2015)
12. Klöti, R., et al.: OpenFlow: a security analysis. In: ICNP, pp. 1–6. IEEE (2013)
13. Shang, G., et al.: Flooddefender: protecting data and control plane resources under SDN-aimed DoS attacks. In: INFOCOM, pp. 1–9. IEEE (2017)
14. Wang, H., et al.: Floodguard: a DoS attack prevention extension in software-defined networks. In: DSN, pp. 239–250. IEEE (2015)
15. Shin, S., Gu, G.: Attacking software-defined networks: a first feasibility study. In: HotSDN, pp. 165–166. ACM (2013)
16. McKeown, N., et al.: OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)

17. Box, J.F.: Guinness, Gosset, Fisher, and small samples. Stat. Sci. **2**, 45–52 (1987)
18. Floodlight SDN Controller. http://www.projectfloodlight.org/floodlight/
19. Floodlight Forwarding Application. https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/forwarding
20. AS4610-54T Data Center Switch. http://www.edge-core.com
21. hping3. http://tools.kali.org/information-gathering/hping3
22. iperf. https://iperf.fr/
23. Qiao, S., et al.: Taming the flow table overflow in OpenFlow switch. In: SIGCOMM, pp. 591–592. ACM (2016)
24. Zhu, H., et al.: MDTC: an efficient approach to TCAM-based multidimensional table compression. In: IFIP Networking, 2015, pp. 1–9. IEEE (2015)
25. Cui, H., et al.: On the fingerprinting of software-defined networks. IEEE Trans. Inf. Forensics Secur. **11**(10), 2160–2173 (2016)
26. Achleitner, S., et al.: Adversarial network forensics in software defined networking. In: SIGCOMM SOSR, pp. 1–13. ACM (2017)
27. Liu, S., et al.: Flow reconnaissance via timing attacks on SDN switches. In: ICDCS, pp. 1–11. IEEE (2017)
28. Sonchack, J., et al.: Timing-based reconnaissance and defense in software-defined networks. In: ACSAC, pp. 89–100. ACM (2016)
29. Leng, J., et al.: An inference attack model for flow table capacity and usage: exploiting the vulnerability of flow table overflow in software-defined network. arXiv preprint arXiv:1504.03095 (2015)
30. Antikainen, M., Aura, T., Särelä, M.: Spook in your network: attacking an SDN with a compromised OpenFlow switch. In: Bernsmed, K., Fischer-Hübner, S. (eds.) NordSec 2014. LNCS, vol. 8788, pp. 229–244. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11599-3_14
31. Röpke, C., Holz, T.: SDN rootkits: subverting network operating systems of software-defined networks. In: Bos, H., Monrose, F., Blanc, G. (eds.) RAID 2015. LNCS, vol. 9404, pp. 339–356. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26362-5_16
32. Wen, X., et al.: SDNShield: reconciliating configurable application permissions for SDN app markets. In: DSN, pp. 121–132. IEEE (2016)
33. Porras, P.A., et al.: Securing the software defined network control layer. In: NDSS (2015)
34. Porras, P., et al.: A security enforcement kernel for OpenFlow networks. In: HotSDN, pp. 121–126. ACM (2012)
35. Khurshid, A., et al.: Veriflow: verifying network-wide invariants in real time. In: NSDI 2013, pp.15–27 (2013)
36. Shin, S., et al.: Avant-guard: scalable and vigilant switch flow management in software-defined networks. In: CCS, pp. 413–424. ACM (2013)
37. Hong, S., et al.: Poisoning network visibility in software-defined networks: new attacks and countermeasures. In: NDSS (2015)