Hindawi
Security and Communication Networks
Volume 2018, Article ID 4760632, 15 pages
https://doi.org/10.1155/2018/4760632

WILEY | Hindawi

Research Article

Exploiting the Vulnerability of Flow Table Overflow in Software-Defined Network: Attack Model, Evaluation, and Defense

Yadong Zhou,[1] **Kaiyue Chen**,[1] **Junjie Zhang**,[2] **Junyuan Leng**,[1] **and Yazhe Tang**[1]

[1]*MOE Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, China*
[2]*Department of Computer Science and Engineering, Wright State University, Fairborn, OH, USA*

Correspondence should be addressed to Yadong Zhou; yadongzhou@gmail.com

Received 28 September 2017; Accepted 6 December 2017; Published 9 January 2018

Academic Editor: Zhiping Cai

As the most competitive solution for next-generation network, SDN and its dominant implementation OpenFlow are attracting more and more interests. But besides convenience and flexibility, SDN/OpenFlow also introduces new kinds of limitations and security issues. Of these limitations, the most obvious and maybe the most neglected one is the flow table capacity of SDN/OpenFlow switches. In this paper, we proposed a novel inference attack targeting at SDN/OpenFlow network, which is motivated by the limited flow table capacities of SDN/OpenFlow switches and the following measurable network performance decrease resulting from frequent interactions between data and control plane when the flow table is full. To the best of our knowledge, this is the first proposed inference attack model of this kind for SDN/OpenFlow. We implemented an inference attack framework according to our model and examined its efficiency and accuracy. The evaluation results demonstrate that our framework can infer the network parameters (flow table capacity and usage) with an accuracy of 80% or higher. We also proposed two possible defense strategies for the discovered vulnerability, including routing aggregation algorithm and multilevel flow table architecture. These findings give us a deeper understanding of SDN/OpenFlow limitations and serve as guidelines to future improvements of SDN/OpenFlow.

1. Introduction

By decoupling the control plane from the data plane, Software-Defined Network (SDN) makes programmability a built-in feature for networks, thereby introducing automaticity and flexibility to the networking management. SDN has therefore been foreseen as the key technology that enables the next generation of networking paradigm. Despite its promise, one of the most significant barriers towards SDN's wide practical deployment resides in overwhelming security concerns [1]. Therefore, proactively detecting, quantifying, and mitigating its security vulnerabilities become of fundamental importance.

In spite of its novelty, SDN indeed reuses various design and implementation elements ranging from architectures and protocols to systems from traditional network. It is not surprising that SDN inheres the vulnerabilities intrinsic to these elements. For example, similar to any networked service, secure channels between controllers and switches might be disrupted by DDoS attacks; like firewall rules, the flow entries may also conflict with each other, leaking unwanted traffic; malicious arp spoofing generated by attackers may poison the controller MAC table, disturbing the normal topology information gathering and packet forwarding; untrusted applications may instrument SDN controller to perform malicious behaviors without proper access control, which is one of the design objectives for modern operating systems. In response, existing research in the context of SDN security mainly focuses on detecting and mitigating these vulnerabilities. For example, [2] evaluates man-in-the-middle attacks that target at SDN/OpenFlow secure channels; FortNOX [3] brings security enforcement module into NOX [4] and enables real-time flow entry conflict check; VeriFlow [5] detects network-wide invariant violations by acting as a transparent layer between control plane and data plane.

In this paper, we introduce a novel SDN vulnerability. The novelty of this vulnerability stems from the feedback-loop nature of SDN, a fundamental difference compared

with traditional networks. Particularly, this vulnerability can be extremely severe in SDN-based networks where network traffic from different sources shares the same SDN switch's flow table, for example, different tenants in a SDN-based cloud computing network.

Specifically, most commercial SDN/OpenFlow switches have limited flow table capacities, ranging from hundreds to thousands [6]. Such capacity is usually insufficient to handle millions of flows that are typical for enterprise and data center networks [7]. Nevertheless, the flow table capacity was just considered as a potential bottleneck of resource consuming attacks in the past, motivating researches on flow caching systems like [8–10]. But according to our analysis, the flow table capacity can lead to inference attack and privacy leakage under certain circumstances.

As a consequence of flow table overflow, the SDN controller needs to dynamically maintain the flow table by inserting and deleting flow entries. The maintaining process typically includes packet information transferring, routing rule calculation, and flow entry deployment, which leads to measurable network performance decrease.

Particularly, once the flow table is full, extra interactions between controller and switch are needed to remove certain existing flow entries to make room for newly generated flow entries, resulting in further network performance decrease. An attacker can therefore leverage the perceived performance change to deduce the internal state of the SDN. To be more specific, we consider the scenario that an attacker resides in a network that is managed by a SDN. The attacker can then actively generate network traffic, triggering the interactions between the controller and switch with respect to flow entry insertion and deletion. The attacker can then measure the change of the network performance to estimate the internal state of the SDN including the flow table capacity and flow table usage. We have designed innovative algorithms to exploit this vulnerability and quantify their effectiveness on exploiting this vulnerability based on extensive evaluation.

Additionally, to mitigate this vulnerability, we have proposed two possible defense strategies. The first strategy is a new routing aggregation algorithm to compress the flow entries so they will consume less flow table space. The second strategy is building a multilevel flow table architecture. Multilevel flow table architecture can implement flow tables with larger capacities without introducing additional power assumption or charges.

To summarize, in this paper we made the following contributions:

(i) We have identified a novel vulnerability introduced by the limited flow table capacities of SDN/OpenFlow switches and formalized that threat.

(ii) We have designed effective algorithms that can successfully exploit this vulnerability to accurately infer the internal states of the SDN network including flow table capacity and flow table usage.

(iii) We have performed extensive evaluation to quantify the effectiveness of proposed algorithms. The experimental results have demonstrated that the discovered vulnerability indeed leads to significant security concerns: our algorithm can infer the network parameters with an accuracy of 80% or higher across various network settings.

(iv) We have proposed two possible defense strategies for the discovered vulnerability, including routing aggregation algorithm to compress the flow entries, and multilevel flow table architecture to implement flow tables with larger capacities.

The rest of this paper is organized as follows. Section 2 gives an overall statement of the inference attack problem. Section 3 gives detailed inference algorithms targeting at FIFO and LRU replacement algorithms, respectively. Section 4 gives a detailed evaluation of the simulation results. Section 5 proposes two possible defense methods against this kind of inference attack. Section 6 is a brief discussion about our findings and future research. Section 7 describes some related works in this area. Finally, Section 8 concludes this paper.

2. Problem Statement

The vulnerability of flow table overflow in SDN potentially exists in SDN-based cloud computing network and other important SDN-based networking systems [11, 12].

After analyzing current structure and implementation of SDN/OpenFlow, its decoupled nature gives us inspiration: the interactions between control plane and data plane will lead to network performance decrease, which can be measured through performance parameters like round trip time (RTT). If a flow matches one flow entry, the flow will be forwarded directly according to the matched entry. This process is fast and will cost little time. When the flow table is full, some flow entry will be removed, then the controller has to calculate the rule and send a new flow entry to the switch, and this process is more complex and has more interactions between controller and switch than the previous case, which will cost more time.

Figure 1 gives an overall flowchart of packet processing in an OpenFlow switch. The three rectangular regions surrounded by dotted line stand for three possible packet processing branches, respectively. When the switch encounters an incoming packet, it will parse it and send the parsed packet into the subsequent processing pipeline.

Then as the first step of the pipeline, the switch will look up its flow table to search flow entries matching the packet. When there is a match, the switch will directly forward the packet according to actions associated with the corresponding flow entry. This branch is illustrated in the innermost rectangle of Figure 1.

When there is no corresponding flow entry in the flow table, extra steps will be introduced into the procedure. Additional interactions between the switch and the controller will happen to acquire corresponding routing rules, including packet information transferring, routing rule calculation, and flow entry deployment. The middle rectangle of Figure 1 illustrates this process.

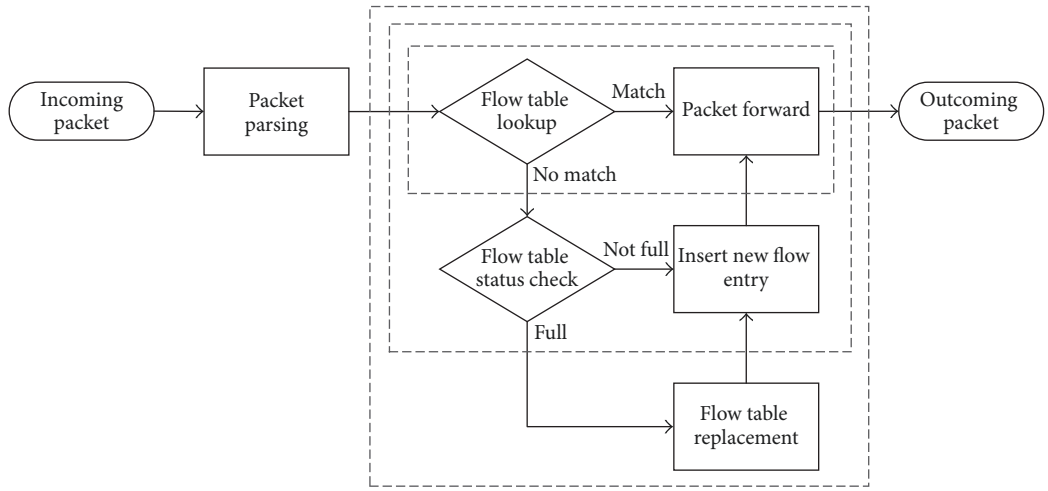

Figure 1: OpenFlow packet processing flowchart.

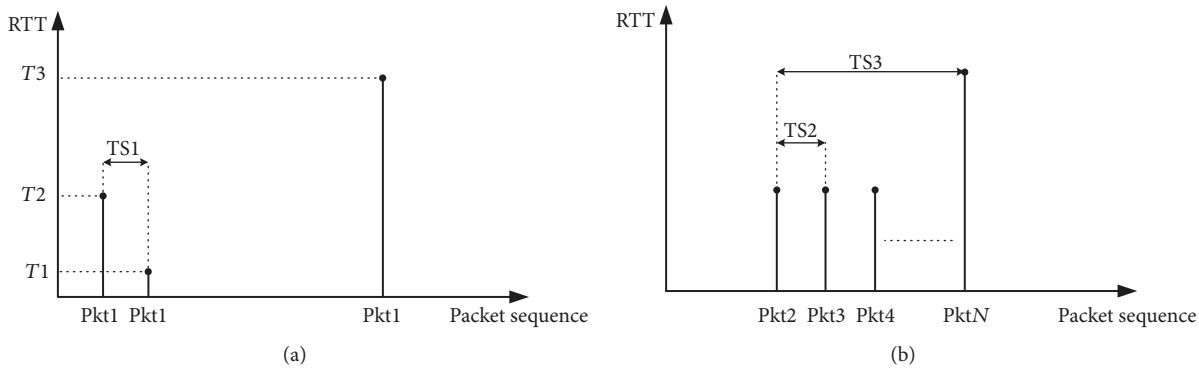

Figure 2: RTT measurement of different flow table state.

Before the switch inserts the newly generated flow entry, it has to check the flow table status to make sure that there is enough space in the flow table. When the flow table is full, the controller has to perform flow table replacement operations to make room for the upcoming flow entry. These operations include deciding which old flow entry to delete according to certain flow table replacement algorithm and flow entry deletion. The outermost rectangle in Figure 1 stands for this branch.

That is exactly where the vulnerability lies. In traditional networks, the switches and routers are autonomous, which means they can maintain their routing tables locally without interacting with an external device. But due to the decoupled nature of SDN/OpenFlow, maintaining switch flow tables needs frequent interactions between switches and controllers, making it possible for an attacker to leverage the perceived performance change to deduce the internal state of the SDN network.

As shown in Figure 1, the rectangular regions surrounded by dotted line correspond to different possible packet processing branches. The larger a rectangle is, the longer the processing time of that branch will be because of the extra steps that rectangle contains. When there is a match in the flow table, the processing time will be the shortest; when there is no match in the flow table and the flow table is not full, the processing time will be longer because of addition routing calculation and flow entry deployment; when there is no match in the flow table and the flow table is full, the processing time will be the longest because a flow table replacement operation has to be performed. So as a network parameter directly influenced by the processing time, the RTT of a packet can serve as an indicator of flow table state and flow entry state.

The process of deciding RTT thresholds for flow table state detection is shown in Figure 2.

Figures 2(a) and 2(b) represent two cooperating threads, the x-axis represents the packet sequence, and the y-axis represents the recorded RTT of every packet. Firstly, in the upper thread, we generate a packet with a specific {src_ip, dst_ip, src_mac, dst_mac} combination, calling it Pkt_1. Send Pkt_1 to the target OpenFlow switch and record the corresponding RTT as T_2. Currently there is no corresponding flow entry in the OpenFlow switch because Pkt_1 is a new packet. After a time span TS_1, send Pkt_1 to the target OpenFlow switch again and record the corresponding RTT as T_1. If TS_1 is chosen properly, the newly installed flow

entry matching Pkt_1 should still exist in the OpenFlow switch. Next, in the lower thread, we continuously generate packets $\text{Pkt}_2, \text{Pkt}_3, \ldots, \text{Pkt}_N$, each with a different combination of {src_ip, dst_ip, src_mac, dst_mac} and send these packets to the target OpenFlow switch with the time span of TS_2. Because there are no flow entries matching their packets in the OpenFlow switch, the recorded RTTs will be approximately the same as T_2. Keep generating and sending packets until we observe a sudden increase of the RTT, which indicates that the flow table is full. Then in the upper thread we send Pkt_1 again immediately and record the RTT as T_3. To achieve higher precision, we can repeat the process and use average values of T_1, T_2, and T_3 as final results.

From the process above we can see that T_1, T_2, and T_3 will serve as thresholds for flow table state detection: when the measured RTT is around T_1, we can infer that there is corresponding flow entry in the flow table; when the measured RTT is around T_2, we can infer that there is no corresponding flow entry in the flow table and the flow table is not full; when the measured RTT is around T_3, we can infer that there is no corresponding flow entry in the flow table and the flow table is full.

We model the SDN/OpenFlow network as a black box and observe its response (RTT) to different input (network packets), then we use the response to estimate the flow table state and flow entry state and perform further inference. The whole process comes in three steps.

Firstly, we send probing packets into the network to trigger the interaction. As there is still no mature routing aggregation algorithm or hierarchical routing rule solution, current SDN/OpenFlow switches typically use exact match rules. That means if we send n packets with different faked metainformation like src_ip and dst_ip, there will be n newly generated flow entries inserted into the flow table. If we send excessive probing packets in a short period of time, the flow table will overflow and then the interaction process will be triggered. Secondly, we measure RTTs of the responded packets and infer the flow table state and flow entry state. Thirdly, we use observed flow table states and flow table states as controlling signals in our inference algorithm and perform flow table capacity inference.

Having to achieve a hit rate as high as possible in a rather limited space, flow table serves like a "cache" in operating systems and web proxy servers. In this paper we choose FIFO and LRU because they are common and popular [13].

3. Inference Algorithm

The logical structure of our inference algorithm is shown in Figure 3. The inference algorithm consists of two main part: flow table state detection and flow table state control. For flow table state detection, we perform RTT measurement to classify the different states of flow table and specific flow entry. For flow table state control, we generate specific sequence of attacking network packets to manipulate the state of flow entries. For different flow table replacement algorithms, the relation between network traffic sequence and flow entry state will be different, so we will have different network traffic generation strategy for different flow table replacement algorithms like FIFO and LRU. We will introduce the inference algorithms for FIFO and LRU, respectively.

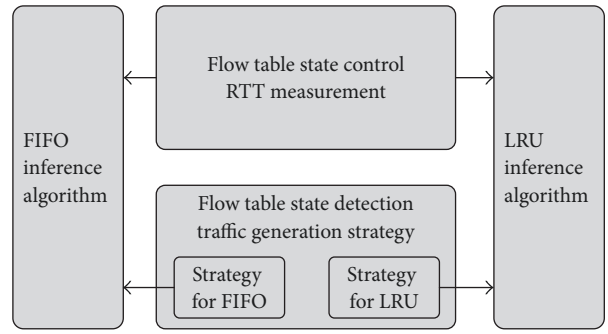

Figure 3: Inference algorithm.

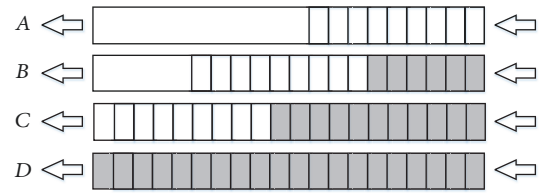

Figure 4: FIFO inference principle.

3.1. FIFO Inference Algorithm. As mentioned in Section 2, the inference process of FIFO algorithm will be as follows: we generate and send a huge amount of probing packets each with a different combination of src_ip, dst_ip, src_mac, and dst_mac, and the newly inserted flow entries matching the generated packets will "push" the other users' flow entries out of the flow table. We can detect if the flow table is full and the existence of our flow entries. Combined with the number of inserted flow entries we recorded, we can infer the flow table capacity and flow table usage. The process of flow table state transformation is shown in Figure 4.

We use F_{our} to represent the number of our inserted flow entries and use F_{other} to represent the number of flow entries from other users in the flow table. Both F_{our} and F_{other} are functions of time. We use T_A, T_B, T_C, and T_D to represent four time points corresponding to four subfigures, respectively, and use C to represent the flow table capacity.

Figure 4 (A) shows the flow table and the flow entries it contains just before the experiment starts. The rectangle items represent the flow entries from other users sharing the OpenFlow switch. The current number of other users' flow entries can be expressed as $F_{\text{other}}(T_A)$.

Figure 4 (B) illustrates the time when we start to send generated packets, inserting new flow entries into the flow table. The grey rectangles represent the flow entries inserted by us. As we can see, our flow entries keep pushing other users' flow entries to the front of the FIFO queue. During the experiment, we should keep a record of the generated packets, including their attributes and serial numbers.

Require:
(1) Packet-Sending Function: *SendPacket*();
(2) List of IP: *IP*;
Ensure:
(3) The flow table capacity: F_{capacity};
(4) The number of other users' flow entries: F_{other};
(5) $F_{\text{capacity}} \leftarrow 0$
(6) $F_{\text{other}} \leftarrow 0$
(7) $N \leftarrow 0$
(8) $N_1 \leftarrow 0$
(9) $N_2 \leftarrow 0$
(10) **while** $N <$ length(IP) **do**
(11) $ip \leftarrow IP[N]$
(12) SendPacket(ip)
(13) $N \leftarrow N + 1$
(14) **if** Flow table is full **then**
(15) $N_1 \leftarrow N$
(16) continue
(17) **end if**
(18) **if** One of our flow entries is deleted **then**
(19) $N_2 \leftarrow N$
(20) break
(21) **end if**
(22) **end while**
(23) $F_{\text{capacity}} \leftarrow N_2$
(24) $F_{\text{other}} \leftarrow N_2 - N_1$
(25) **return** F_{capacity}, F_{other}

Algorithm 1: FIFO inference algorithm.

Figure 4 (C) shows the time when we detect the flow table is full. At this point of time, flow entries from us and other users add up to fill the whole flow table precisely. We have

$$F_{\text{our}}(T_C) + F_{\text{other}}(T_C) = C. \tag{1}$$

Figure 4 (D) shows the time when we detect that one of our inserted flow entries has been deleted. That means the flow table is now full of our flow entries, without any flow entries from other users. We have

$$F_{\text{our}}(T_D) = C. \tag{2}$$

Combine the two equations above; we have

$$\begin{aligned} F_{\text{other}}(T_A) = F_{\text{other}}(T_C) &= C - F_{\text{our}}(T_C) \\ &= F_{\text{our}}(T_D) - F_{\text{our}}(T_C). \end{aligned} \tag{3}$$

According to the analysis above, we describe the inference process for FIFO algorithm as shown in Algorithm 1.

The main error of the inference comes from the flow entries inserted by other users when our insertion is in progress. We assume that our flow entry insertion speed is fast enough so that, during the period of experiment, the newly inserted flow entries are all from us. But that is not always the truth. Ignoring the possible flow entries inserted by other users will make our inference result smaller than the actual value.

Considering the flow entries inserted by other users, the actual equations are listed below.

When we detect the flow table is full, if we use $E(A, B)$ to represent the number of just inserted flow entries from other users from time point A to time point B, the equation becomes

$$F_{\text{our}}(T_C) + F_{\text{other}}(T_C) + E(T_A, T_C) = C. \tag{4}$$

And when we detect that one of our inserted flow entries is deleted, the equation becomes

$$F_{\text{our}}(T_D) + E(T_A, T_C) + E(T_C, T_D) = C. \tag{5}$$

Combine the two equations above; we have

$$F_{\text{other}}(T_C) = F_{\text{our}}(T_D) - F_{\text{our}}(T_C) + E(T_C, T_D). \tag{6}$$

So the actual equation considering flow entry insertions during inference should be

$$\begin{aligned} C &= F_{\text{our}}(T_D) + E(T_A, T_C) + E(T_C, T_D) \\ F_{\text{other}}(T_C) &= F_{\text{our}}(T_D) - F_{\text{our}}(T_C) + E(T_C, T_D). \end{aligned} \tag{7}$$

Compared with our former equation ignoring flow entry insertions,

$$\begin{aligned} C &= F_{\text{our}}(T_D) \\ F_{\text{other}}(T_C) &= F_{\text{our}}(T_D) - F_{\text{our}}(T_C). \end{aligned} \tag{8}$$

We can see that the inferred flow table usage F_{other} and the inferred flow table capacity F_{capacity} will both be smaller than the actual value.

3.2. LRU Inference Algorithm. The experiment principle of LRU algorithm has something in common with that of FIFO algorithm, because under these two circumstances we can both keep our flow entries stay in the back of the cache queue using certain operations. However, there are still differences lies in the flow entry maintaining process.

The nature of FIFO algorithm ensures that the position of the flow entries only depends on the time they are inserted. The earlier inserted flow entries are sure to be nearer to the front of the cache queue compared with the later inserted flow entries. But in LRU algorithm, the positions of the flow entries depend not only on the time they are inserted, but also on the last time they are accessed. In order to keep our flow entries stay in the back of the cache queue, we need to continuously access the previously inserted flow entries.

During the maintain process, every time we insert a new flow entry, we need to access all previously inserted flow entries for one time to "lift" them to the back of the cache queue. The access history may be like $\{P_1\}, \{P_1, P_2\}, \{P_1, P_2, P_3\}, \{P_1, P_2, P_3, P_4\}, \ldots$, and we call it a "rolling" maintaining process. The maintaining algorithm is shown in Algorithm 2. According to the analysis above, we describe the inference process for LRU in Algorithm 3.

The feasibility and error analysis of LRU algorithm is similar to that of FIFO algorithm. The inferred flow table usage F_{other} and the inferred flow table capacity F_{capacity} will both be smaller than the actual value because of ignoring the flow entries inserted by other users during the experiment.

```
Require
 (1) Packet-Sending Function: SendPacket();
 (2) List of Inserted IP: IP_inserted;
 (3) function RollingPacketSender(IP_inserted)
 (4)     i ← 1
 (5)     while i < length(IP_inserted) do
 (6)         for j ← 0; j < i; j + + do
 (7)             ip ← IP_inserted[j]
 (8)             SendPacket(ip)
 (9)         end for
(10)         i ← i + 1
(11)     end while
(12) end function
```

Algorithm 2: Rolling maintaining algorithm.

```
Require:
 (1) Packet-Sending Function: SendPacket();
 (2) List of IP: IP;
Ensure:
 (3) The flow table capacity: F_capacity;
 (4) The number of other users' flow entries: F_other;
 (5) F_capacity ← 0
 (6) F_other ← 0
 (7) N ← 0
 (8) N_1 ← 0
 (9) N_2 ← 0
(10) IP_inserted ← []
(11) while N < length(IP) do
(12)     ip ← IP[N]
(13)     IP_inserted ← IP_inserted + ip
(14)     RollingPacketSender(IP_inserted)
(15)     N ← N + 1
(16)     if Flow table is full then
(17)         N_1 ← N
(18)         continue
(19)     end if
(20)     if One of our flow entries is deleted then
(21)         N_2 ← N
(22)         break
(23)     end if
(24) end while
(25) F_capacity ← N_2
(26) F_other ← N_2 − N_1
(27) return F_capacity, F_other
```

Algorithm 3: LRU inference algorithm.

4. Evaluation

4.1. Implementation. The emulation environment of our experiment consists of three parts: a network prototyping system used to emulate host and switch, a network controller, and our inference attack toolkit.

We choose Mininet [14] as the network prototyping system because it encapsulates host and switch emulation and thus easy to use. Our emulated network prototype for evaluation uses a star topology, consisting of 20 hosts connected to a single OpenFlow switch. We build FIFO and LRU controller applications using Python on the basis of POX [15] OpenFlow controller. As for the inference attack toolkit, we use libnet [16] to generate probing packets, and libpcap [17] to capture replied packets. To simulate the background traffic in real network, we built a SDN testbed using Mininet and POX. On the SDN testbed, we performed a series of basic SDN operations. These operations include building a customized SDN network topology, setting up the link between SDN switches and performing the ping test between all SDN nodes. We captured the network traffic generated during these operations and use them as the background network traffic sample.

4.2. RTT Measurement. As we have mentioned in Section 2, the difference between traditional network and SDN/OpenFlow network in handling previously unseen packets gives us a possible indicator of the flow table state and the flow entry living state, RTT. When there is not corresponding flow entry existing in the flow table, the RTT of a packet will significantly increase due to the interactions between controller and switch in order to acquire new flow entries. That is the case when there is still space in the flow table. Once the flow table is full, the RTT of a packet will further increase as a result of extra flow table replacement operations. To prove the effectiveness of using RTT as the flow table state and flow entry state indicator, we measured packet RTTs corresponding to different flow table state and flow entry state.

Figure 5 gives the RTT measurement result showing the difference. The points with different symbols represent the total 300 times of RTT measurements we have conducted, 100 times of measurement for each combination of flow table state and flow entry state. The square points stand for RTTs when flow entry exists in flow table. The circle points and triangle points both stand for RTTs when flow entry does not exist in flow table; the circle points are measured when the flow table is full, and the triangle points are measured when the flow table is not full.

As can be seen from the figure, when flow entry exists in flow table, the packet RTTs are highly concentrated in the

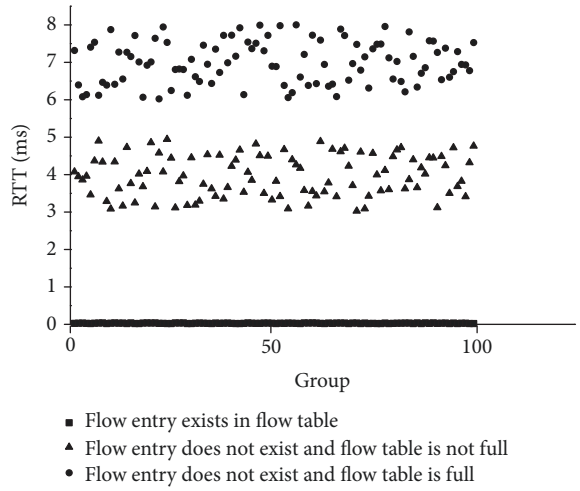

Figure 5: RTT measurement.

Table 1: Default timeout values.

Controller	Hard_timeout	Idle_timeout
Ryu	0	0
Beacon	0	5 s
Floodlight	0	5 s
NOX	0	5 s
POX	30 s	10 s
Trema	0	60 s
Maestro	180 s	30 s

range of 0.2~0.3 ms; when flow entry does not exist in flow table and flow table is not full, the packet RTTs will increase to about 3~5 ms; when flow entry does not exist in flow table and flow table is full, the packet RTTs will be the highest, ranging from 6 ms to 8 ms. These three groups of RTTs all distribute intensively in a small range without overlapping other groups, showing the excellent discrimination of using RTT as a flow table state and flow entry state indicator.

4.3. Timeout

4.3.1. Default Timeout Values. According to our previous analysis, the feasibility of our inference attack depends on whether we can generate enough flow entries to fulfill the flow table within a single timeout cycle. That means we must have the ability to generate as many flow entries as the flow entry can hold during a timeout period. So we analyze several popular open-source controllers and search for their default timeout values in the built-in applications. The result is presented in Table 1. The zero values in the table mean the corresponding timeout will not take effect, or in other words the timeout value is "permanent." As can be seen from the table, most available controllers have timeout values in the range of 5 s to 30 s.

If we take the flow table capacity of 2000 flow entries as an example, the minimum packet generating speed required will be 2000/5 = 400 packets per second, while libnet can generate tens of thousand packets per second. So the default timeout values ensure the feasibility of our inference attack.

4.3.2. Timeout Measurement. Though default timeout values of mainstream OpenFlow controllers can be read from their source codes, it is still possible for SDN network administrators to manually change the default timeout values. In order to handle nondefault timeout values and provide basis for adjusting packet generating speed, it is essential to examine the accuracy of passive timeout measurement.

Figure 6 illustrates relative errors (see equation (9)) of hard_timeout and idle_timeout measurement, respectively. We manually modify hard_timeout and idle_timeout values of POX OpenFlow controller to 5 s, 10 s, 15 s, 20 s, 25 s, and 30 s, and then we use timeout measurement algorithm mentioned in Section 2 to measure these timeout values and calculate relative errors:

$$\text{relative\_error} = \frac{\left|\text{value}_{\text{detected}} - \text{value}_{\text{true}}\right|}{\text{value}_{\text{true}}} * 100\%. \quad (9)$$

Every line in Figures 6(a) and 6(b) corresponds to 10 times of repeated measurements conducted under a certain timeout setting from 5 s to 30 s. The margin stays in the range of plus-or-minus 10 percent, showing the effectiveness and high accuracy of our timeout measurement algorithm.

4.4. Flow Table Capacity. Flow capacity is the primary target of our inference attack. It reflects the hardware specification of an OpenFlow switch. Figure 7 illustrates the flow table capacity measurement result when controller adopts FIFO replacement algorithm. We manually limited the switch flow table capacity to 10 different values from 100 flow entries to 1000 flow entries and used our framework to perform the inference.

The dark bars represent the manually set flow table capacities or *real* capacities. The light bars represent the measured flow table capacities. For every manually set flow table capacity, we conduct 10 times of repeated measurements and take their mean value as the final result. From the figure we can see that the measured capacities are quite close to the real capacities, indicating the high accuracy of our inference framework. For example, when the real capacity is 400 flow entries, our measured capacity is 408 flow entries with an error of only 8 flow entries. As the real capacity grows, the packet generating speed required becomes faster, placing higher requirements on packet sending, receiving synchronization and accurate timing. But our inference algorithm shows unbelievable stability and accuracy: when the real capacity is 1000 flow entries, our measured capacity is 973 flow entries with an error of just 27 flow entries.

Like Figure 7, Figure 8 also illustrates the flow table capacity measurement results, with the only difference of being performed under LRU replacement algorithm instead of FIFO.

According to our previous analysis, the inference principle of LRU replacement algorithm is more complex because of the unavoidable mixed nature of flow entries in the flow table and the rolling maintaining process. But our inference

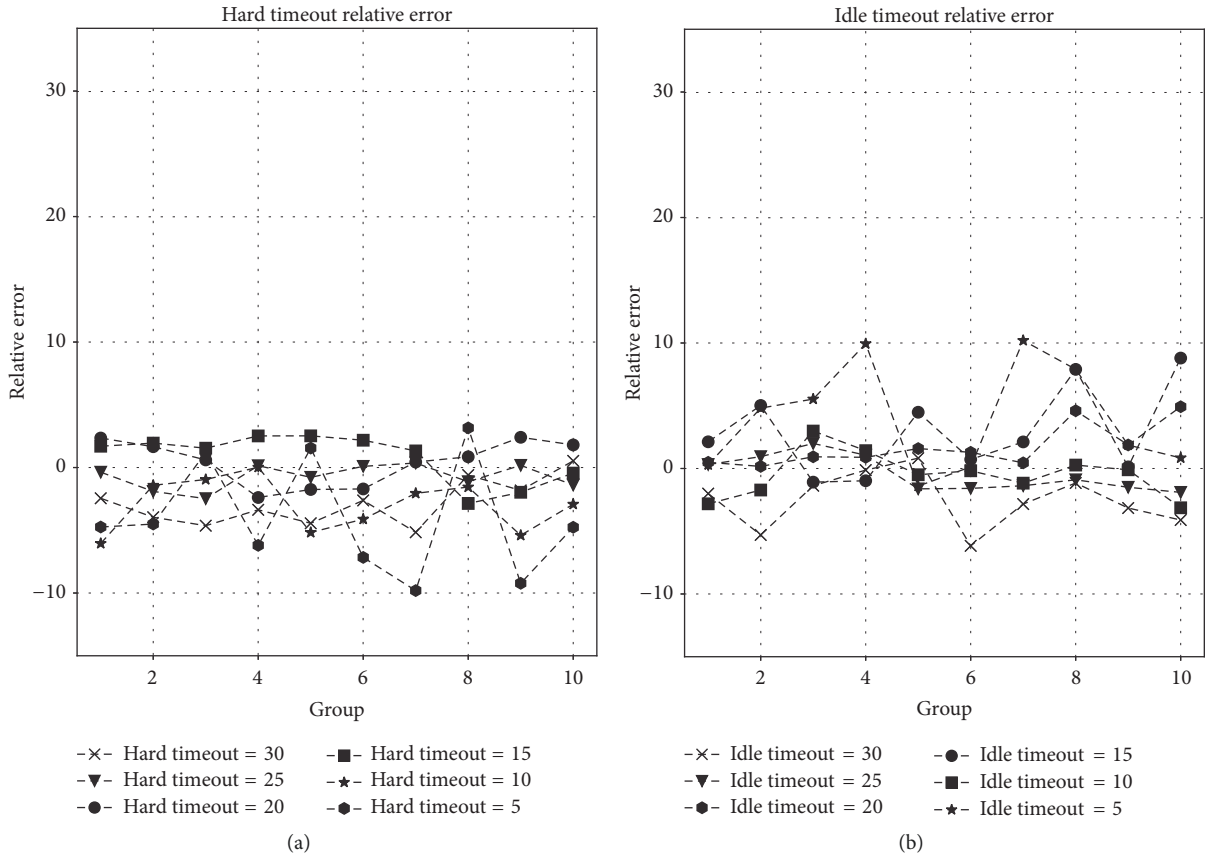

Figure 6: Timeout relative error.

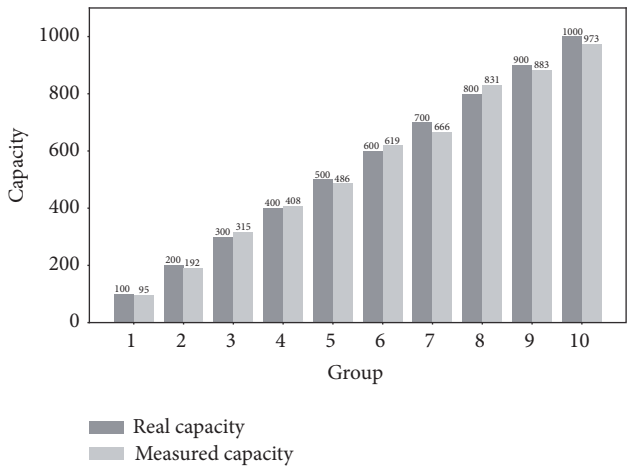

Figure 7: FIFO flow table capacity.

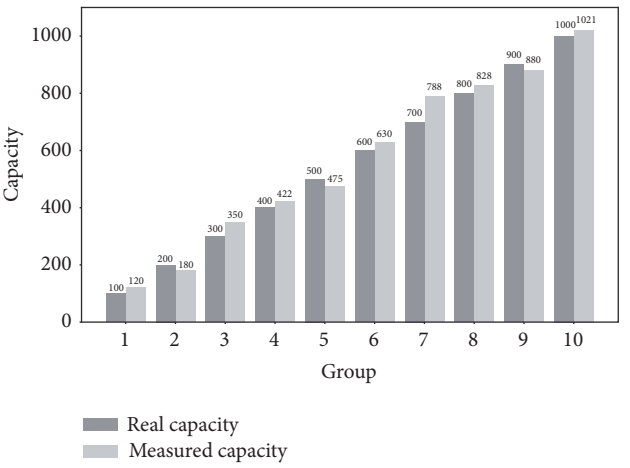

Figure 8: LRU flow table capacity.

framework still shows high accuracy and reliability. Even when the real flow table capacities are set to be rather large values like 900 and 1000, the errors of our measure capacities are just around 20 flow entries.

Only illustrating the mean value of measured flow table capacities may not be enough: the mean value may be the result of error compensations and hide the detailed measurement errors of every separate experiment. So in Figure 9 we illustrate the relative error of every single flow table capacity measurement.

We choose 5 groups of different flow table capacities from 200 flow entries to 1000 flow entries and perform 10 times of measurements under every single flow table capacity value. Figure 9(a) stands for relative error of flow table capacity measurements conducted under FIFO replacement

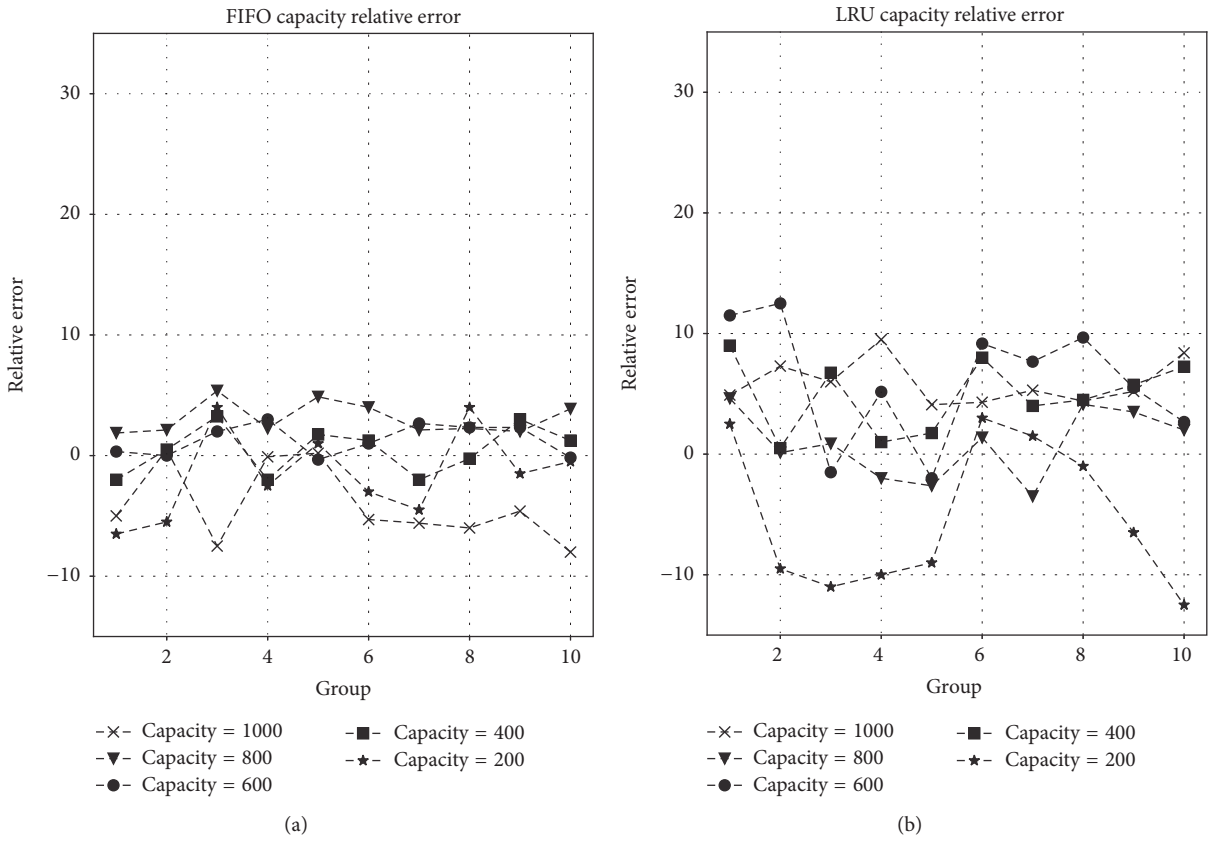

Figure 9: Flow table capacity relative error.

algorithm, showing that the margin is no larger than plus-or-minus 10 percent. Figure 9(b) stands for relative error of flow table capacity measurements conducted under LRU replacement algorithm. Due to the more complex inference principle and the rolling maintaining process, the margin becomes larger but still has not exceeded 15 percent even in the worst case.

The above inference attacks are performed without any background network traffic. When performing inference attack in real networks, the impact of background network traffic cannot be ignored. So it is necessary to examine the efficiency of our inference algorithm under these circumstances.

In this evaluation, we choose the background network traffic dataset from a SDN testbed. Figures 10 and 11 have the same experiment setting with Figures 7 and 8, with the only difference of replaying background traffic captured from SDN testbed during the inference attack process. Even with the impact of background traffic, our inference algorithm still shows high accuracy.

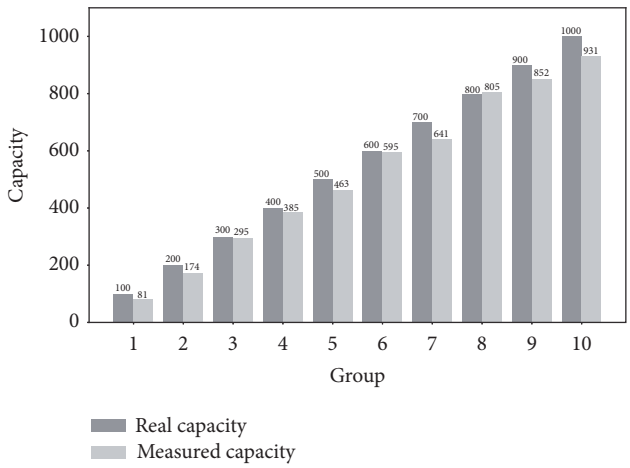

Figure 10: FIFO flow table capacity with testbed background traffic.

4.5. Flow Table Usage. In this section we evaluated our framework's efficiency of inferring the number of flow entries from other users sharing the same flow table or the flow table usage. Flow table usage is our secondary inference target, and it reflects the network resource consuming condition of other tenants in the same SDN network. Figures 12 and 13 illustrate the flow table usage measurement results conducted under FIFO and LRU replacement algorithm, respectively.

Again we manually set 10 different flow table usage values from 100 to 1000 flow entries by manually generating and inserting corresponding number of flow entries into the flow table beforehand. Then we use our inference algorithm to infer the flow table usage and take mean values of every 10 times of measurements as the final results. The errors of all these measurements show the high accuracy, stability and reliability of our inference algorithm.

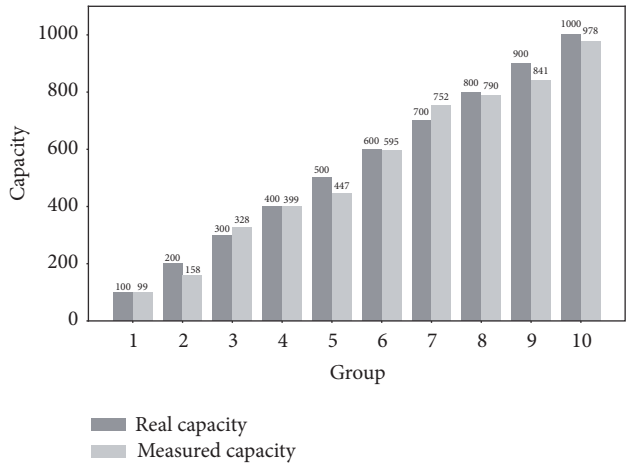

Figure 11: LRU flow table capacity with testbed background traffic.

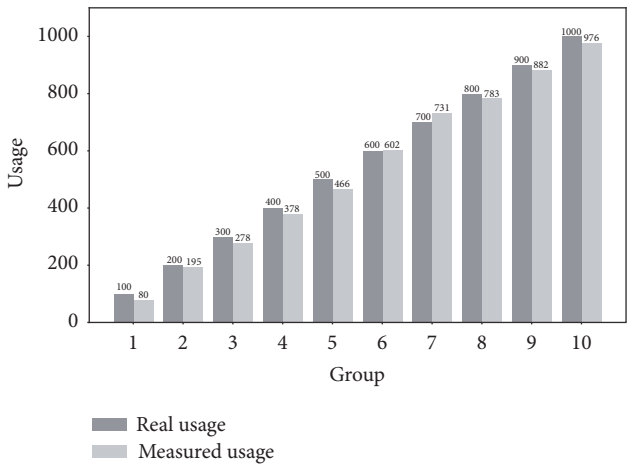

Figure 13: LRU flow table usage.

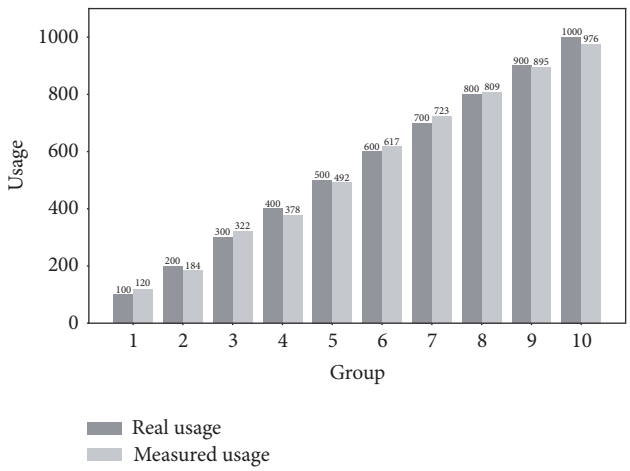

Figure 12: FIFO flow table usage.

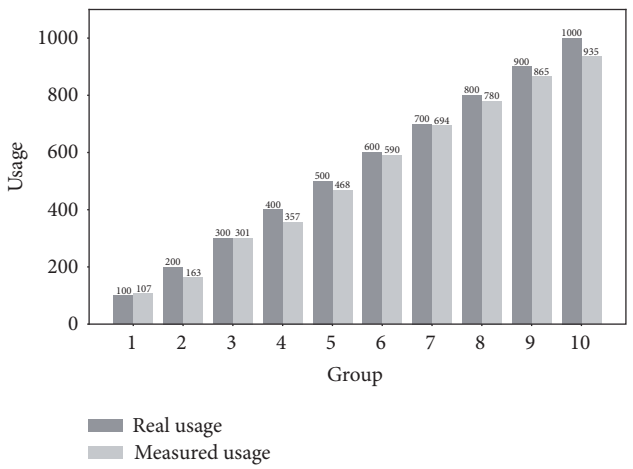

Figure 14: FIFO flow table usage with testbed background traffic.

We also conducted the flow table usage inference attack with background traffic. Experiment results with testbed background traffic are shown in Figures 14 and 15. Our inference algorithm can smoothly handle the impact of background traffic, which ensures the stability and robustness demonstrated in the experiment results.

The relative errors are shown in Figure 16. We emulate 5 groups of different flow table usage values and conducted 10 times of flow table usage inference for every single value. For both FIFO and LRU replacement algorithm, the relative errors of flow table usage inference stay in a quite small range. The results prove that our algorithm can infer other tenants' flow table usage condition in high accuracy.

5. Defense

From the previous sections we can conclude that the inference attack is rooted in the flow table overflow. To defend that kind of inference attack, we have to prevent flow table overflow in two aspects: one aspect is to compress the flow entries to save flow table space, and the other one is to implement a larger flow table to store more flow entries.

5.1. Routing Aggregation. Routing aggregation is to combine multiple entries in the flow table without changing the next hops for packet forwarding. This approach is particularly appealing because it can be done by a software upgrade at the OpenFlow switch and its impact is limited within that switch. Routing aggregation has already been used in traditional networks, but it has not been deployed in SDN/OpenFlow networks. To fully utilize the flexibility of SDN/OpenFlow network under certain scenarios like load balancing, we proposed a global routing schedule using packing optimization algorithms.

Traditional routing aggregation algorithms [18] can be used to compress the flow table, but their effectiveness cannot be ensured. If the matching fields and next hops are dispersed enough, chances are that we may not be able to perform any routing aggregation because we cannot find flow entries sharing common matching fields and next hops. This is often

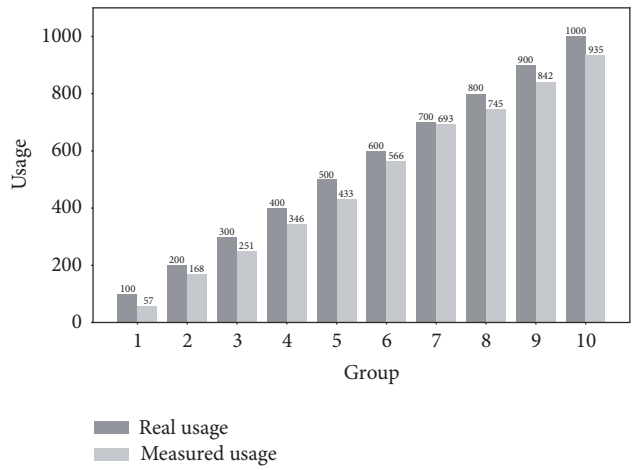

Figure 15: LRU flow table usage with testbed background traffic.

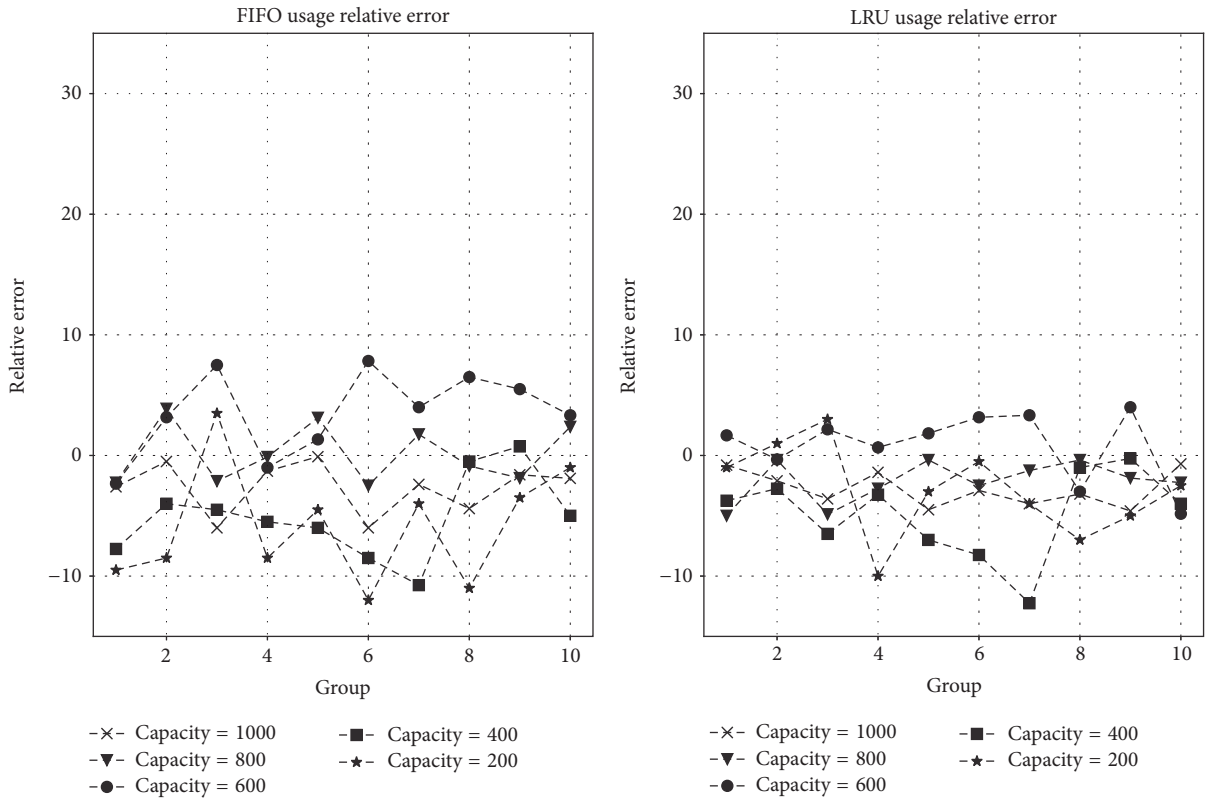

Figure 16: Flow table usage relative error.

the case when dealing with web traffic, for example, load balancing services. For that reason, we introduced an extra stage of routing aggregation: global routing schedule optimization. First we model this routing aggregation problem as a packing optimization problem and solve it, then we perform global routing reschedule by rewriting the flow entries according to the optimization result, and finally we perform another time of traditional routing aggregation on these new flow entries. After the global routing reschedule, there will be much more aggregatable flow entries, so the effectiveness of routing aggregation is ensured.

5.2. Multilevel Flow Table Architecture. It is important to note that routing aggregation is not a replacement for the long-term architectural solutions because it does not address the root causes of the flow table scalability problem and the following inference attack. To eliminate the inference attack vulnerability, a flow table architecture with larger capacity is

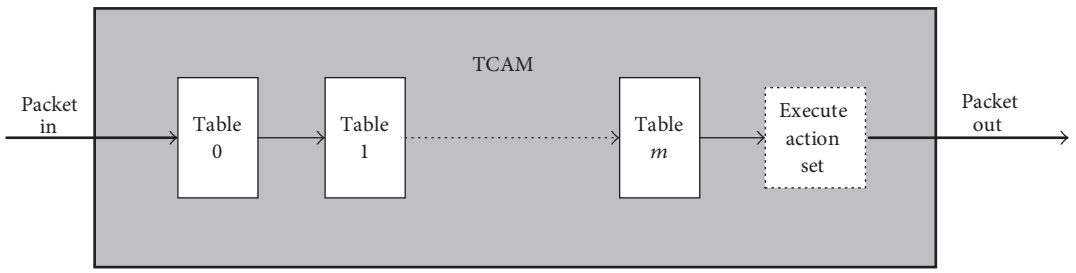

Figure 17: Single-level flow table architecture.

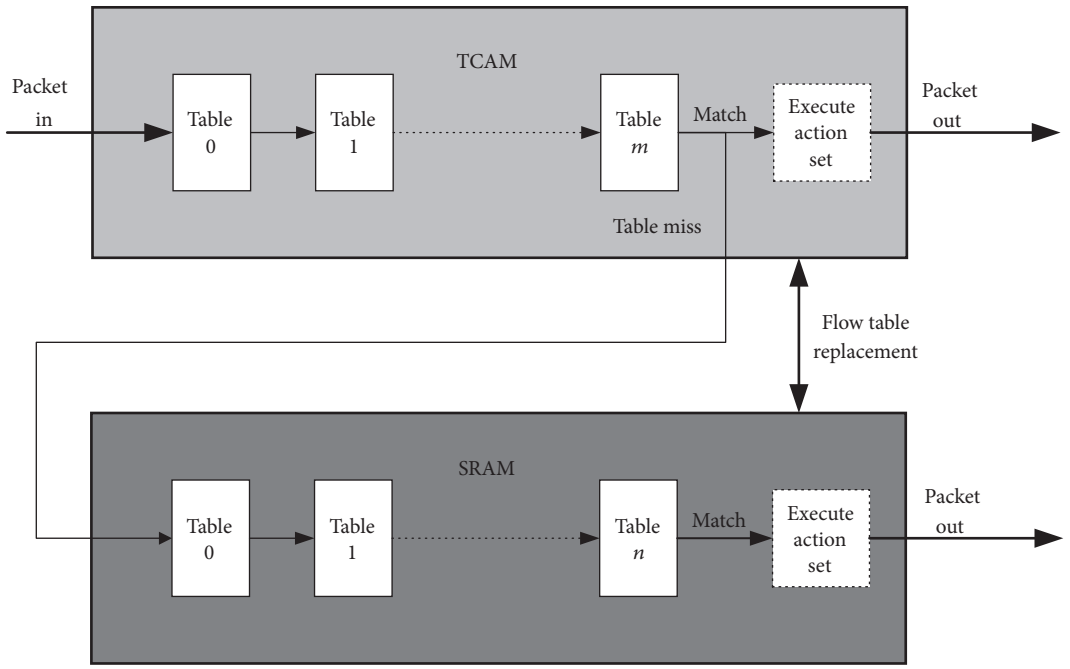

Figure 18: Multilevel flow table architecture.

required, which can be achieved through multilevel flow table consisting of both TCAM and SRAM.

The original single-level flow table architecture is shown in Figure 17. In this architecture, the flow table is completely implemented using TCAM. An input packet will traverse from table 0 to table m and add corresponding actions to the action set. Then all actions in the action set are executed and the packet is forwarded according to these actions.

Our proposed multilevel flow table architecture is shown in Figure 18. Besides flow table implemented using TCAM, we add another flow table implemented using SRAM, which is cheaper and can provide larger flow table space. Under this multilevel flow table architecture, the packet processing pipeline will be different: first an input packet will find matching flow entries in TCAM flow table, just like in the original single-level flow table architecture. If there is a match, the packet will execute the corresponding actions and get forwarded. If there is no match, the packet will continue its lookup in the SRAM flow table. If there is a match in the SRAM flow table, the packet can then be forwarded; otherwise it will be sent to the controller.

From the process above, we can see that if the capacity of TCAM flow table is m, the capacity of SRAM flow table is n, and then the multilevel flow table will have a capacity of $m + n$. Actually n is far more larger than m, so this approach can greatly increase the flow table capacity, thus preventing flow table overflow.

6. Discussion

SDN/OpenFlow has become a competitive solution for next-generation network and is being more and more widely used in modern datacenters. But considering its key role as the fundamental infrastructure, we have to admit that the security issues of SDN/OpenFlow have not been explored to a large extent. Particularly, the flow table capacity of SDN/OpenFlow switch is only considered as a vulnerable part for DDoS and flooding attacks in published researches. But according to our analysis in previous sections, the flow table capacity can lead to potential inference attack if combined with reasonable assumptions and RTT measurements.

Firstly, we found in Section 2 that exact match flow entries as well as the lack of route aggregation would consume a lot

of flow table space, making it impossible to process millions of flows per seconding using SDN/OpenFlow. Secondly, we found in Section 4 that assigning the decision making job of flow table replacement to the controller would lead to significant network performance decrease, which had to be changed in time. Thirdly, there is currently no mature attack detection mechanism for SDN/OpenFlow network, so it is quite easy for criminals to exploit system vulnerabilities or invoke DDoS attacks.

The inference method proposed in this paper just uses some basic elements and parameters of OpenFlow, such as idle timeout and hard timeout, which are significant for the implementation of SDN. These features will not be removed except very huge changes made. On the other hand, although some security frameworks [3] were proposed to detect the malicious insertion of flow rules, attackers can also bypass the detection by some well-designed insertion strategies.

All these security issues call for improvements to current OpenFlow switch and flow table design. The improvements should at least contain the following aspects: (1) New OpenFlow switch architecture, like embedding local caches in the switch or implementing multilevel flow table to achieve a much larger flow table capacity. With larger flow table capacity, the switch will not have to query the controller for flow entries, which will reduce the interaction latency to a large extent. (2) New flow table maintaining mechanism, like transferring the flow entry deleting workload from controller to switch. Switch itself can decide which flow entry to delete and then sync state with controller, and during the flow entry deleting process, the controller's intervention is not needed. In the widely used OpenFlow Switch Specification 1.4.0 [19], this mechanism has been added as an optional feature, but without any mature implementation so far. (3) Routing aggregation. Routing aggregation can match a group of flows using one flow entry, which will reduce the flow table consuming significantly compared with exact match. (4) Inference attack detection. Administrators can develop inference attack detecting applications and then perform defenses like portspeed limiting or network address validation.

From the discussion above, we can see that there is still a long road to go before SDN/OpenFlow becomes a truly mature and reliable network paradigm. There are still urgent and severe issues to solve, which have been neglected in the past. Only by solving these security issues and architectural vulnerabilities can SDN/OpenFlow be widely deployed in real-world commercial datacenters and fully demonstrate its revolutionary flexibility and intelligence.

7. Related Work

The inference attack proposed in this paper is motivated by the limited flow table capacity of SDN/OpenFlow switches. The flow table capacity issue has been presented in many previous works like [20–24]. They all point out the limitation of switch flow table memory and potential scalability and security issue. However, these works do not give further analysis on the inference attack and information leakage caused by the limited flow table capacity.

Klöti et al. [25] present potentially problematic issues in SDN/OpenFlow including information disclosure through timing analysis. However, this information disclosure requires disclosing existing flows with side channel attack, which is hard to perform in real world. Compared with their approach, our inference attack is self-contained and requires no prior knowledge.

Gong et al. [26] present a kind of inference attack using RTT measurement to infer which website the victim is browsing. They recover victims' network traffic patterns based on the queuing side channel happened at the Internet router. However, the scenario of their work is in the public Internet, while our approach focuses on SDN/OpenFlow infrastructures in cloud computing network. Compared with public Internet and website inference, the inference attack and information leakage in modern data centers are more sensitive and valuable.

Shin and Gu [27] demonstrate a novel attack targeting at SDN networks. This attack includes fingerprinting SDN networks and further flooding the data plane flow table by sending specifically crafted fake flow requests in high speed. In the fingerprinting phase, header field change scanning is used to collect the different response time (RTT) for new flow and existing flow. The fingerprinting result is then analyzed to estimate if the target network used SDN technology. The RTT measurement and analysis they used in fingerprinting are similar to our approach. But they just perform DoS attacks to the SDN network, without performing any further information leakage or network parameter inference.

As for flow table overflow defending strategy, Shelly et al. [28] and Katta et al. [29] introduce flow entry caching mechanism into SDN/Openflow network by inserting a transparent intermediate layer between controller and switch. Yan et al. [9] use CAB to generate wildcard flow entries dynamically and reactively to handle bursting network traffic. Kannan and Banerjee [30] present a flow entry compaction algorithm to save TCAM flow table space. This algorithm uses flow entry tags instead of matching fields as forwarding rules. Kim et al. [31] develop a new flow entry management scheme to reduce the controller overhead.

8. Conclusion

In this paper, we have explored the structure of SDN/OpenFlow network and some of the possible security issues it brings. After our detailed analysis of the SDN/OpenFlow network, we proposed a novel inference attack model targeting at the SDN/OpenFlow network, which is the first proposed inference attack model of this kind in the SDN/OpenFlow area. This inference attack is introduced by the OpenFlow switch, especially by its limited flow table capacity. The inference attack can be done in a completely passive way, making it hard to detect and defend. We also implemented the inference attack framework and examined the efficiency and accuracy of it using network traffic data from different sources. The simulation results show that the inference attack framework can infer the network parameter (flow table capacity and flow table usage) with an accuracy of up to 80% or higher. We also proposed two possible defense strategies for the discovered

vulnerability, including routing aggregation algorithm and multilevel flow table architecture.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The research presented in this paper is supported in part by the National Key Research and Development Program of China (no. 2016YFB0800100), the Fund of China National Aeronautical Radio Electronics Research Institute (PM-12210-2016-001), the National Natural Science Foundation (61572397, U1766215, U1736205, 61502383, 61672425, and 61702407), and State Grid Corporation of China (DZ71-16-030).

References

[1] W. Li, W. Meng, and L. F. Kwok, "A survey on OpenFlow-based Software Defined Networks: Security challenges and countermeasures," *Journal of Network and Computer Applications*, vol. 68, pp. 126–139, 2016.

[2] M. Brooks and B. Yang, "A Man-in-the-Middle attack against OpenDayLight SDN controller," in *Proceedings of the 4th Annual Conference on Research in Information Technology*, Association for Computing Machinery, Chicago, IL, USA, September 2015.

[3] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks, HotSDN 2012*, pp. 121–126, Association for Computing Machinery, Helsinki, Finland, August 2012.

[4] N. Gude, T. Koponen, and J. Pettit, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[5] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, ACM SIGCOMM 2012*, pp. 467–472, Finland, August 2012.

[6] G. Zhao, L. Huang, Z. Yu, H. Xu, and P. Wang, "On the effect of flow table size and controller capacity on SDN network throughput," in *Proceedings of the 2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, Paris, France, May 2017.

[7] A. Roy, H. Zengy, J. Baggay, G. Porter, and A. C. Snoeren, "Inside the social network's (Datacenter) network," in *Proceedings of the ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, pp. 123–137, UK, August 2015.

[8] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 151–156, Chicago, Illinois, USA, August 2014.

[9] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "CAB: A reactive wildcard rule caching system for software-defined networks," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 163–168, Chicago, Illinois, USA, August 2014.

[10] Katta Naga, Jennifer Rexford, and David Walker, "Infinite cacheflow in software-defined networks," Princeton School of Engineering and Applied Science, 2013.

[11] Yanwen Wang, Hainan Chen, Hainan Chen, and Lei Shu, "An energy-efficient SDN based sleep scheduling algorithm for WSNs," *Journal of Network and Computer Applications*, vol. 59, pp. 39–45, 2016, http://dx.doi.org/10.1016/j.jnca.2015.05.002.

[12] S.-N. Yang, S.-W. Ho, Y.-B. Lin, and C.-H. Gan, "A multi-RAT bandwidth aggregation mechanism with software-defined networking," *Journal of Network and Computer Applications*, vol. 61, pp. 189–198, 2016, http://dx.doi.org/10.1016/j.jnca.2015.11.003.

[13] M. Dehghan, L. Massoulie, D. Towsley, D. Menasche, and Y. C. Tay, "A utility optimization approach to network cache design," in *Proceedings of the 35th Annual IEEE International Conference on Computer Communications, IEEE INFOCOM 2016*, San Francisco, CA, USA, April 2016.

[14] "Mininet" http://mininet.org/.

[15] "POX Controller" http://www.noxrepo.org/pox/about-pox/.

[16] "libnet-dev" https://github.com/sam-github/libnet.

[17] "libpcap" http://www.tcpdump.org/.

[18] X. Zhao, Y. Liu, L. Wang, and B. Zhang, "On the aggregatability of router forwarding tables," in *Proceedings of IEEE INFOCOM 2010*, Institute of Electrical and Electronics Engineers, San Diego, CA, USA, March 2010.

[19] "OpenFlow Switch Specification 1.4.0" https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[20] S. Sezer, S. Scott-Hayward, P. Chouhan et al., "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[21] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[22] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Proceedings of the 2013 Workshop on Software Defined Networks for Future Networks and Services, SDN4FNS 2013*, Trento, Italy, November 2013.

[23] A. Akhunzada, E. Ahmed, A. Gani, M. K. Khan, M. Imran, and S. Guizani, "Securing software defined networks: Taxonomy, requirements, and open issues," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 36–44, 2015.

[24] M. Tsugawa, A. Matsunaga, and J. A. B. Fortes, "Cloud computing security: What changes with software-defined networking?" *Secure Cloud Computing*, pp. 77–93, 2014.

[25] R. Klöti, V. Kotronis, and P. Smith, "OpenFlow: A security analysis," in *Proceedings of the 2013 21st IEEE International Conference on Network Protocols, ICNP 2013*, Institute of Electrical and Electronics Engineers, Goettingen, Germany, October 2013.

[26] X. Gong, N. Borisov, N. Kiyavash, and N. Schear, "Website Detection Using Remote Traffic Analysis," in *Privacy Enhancing Technologies*, vol. 7384 of *Lecture Notes in Computer Science*, pp. 58–78, Springer, Berlin, Heidelberg, 2012.

[27] S. Shin and G. Gu, "Attacking software-defined networks: a first feasibility study," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*, pp. 165-166, Association for Computing Machinery, Hong Kong, China, August 2013.

[28] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 663–668, 2014.

[29] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite CacheFlow in software-defined networks," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 175–180, Association for Computing Machinery, Chicago, Illinois, USA, August 2014.

[30] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proceedings of the 14th International Conference on Distributed Computing and Networking*, vol. 7730 of *Lecture Notes in Computer Science*, pp. 439–444, Springer, Mumbai, India, 2013.

[31] E.-D. Kim, S.-I. Lee, Y. Choi, M.-K. Shin, and H.-J. Kim, "A flow entry management scheme for reducing controller overhead," in *Proceedings of the 16th International Conference on Advanced Communication Technology: Content Centric Network Innovation!, ICACT 2014*, pp. 754–757, Institute of Electrical and Electronics Engineers, Pyeongchang, South Korea, February 2014.