On Denial of Service Attacks in Software Defined Networks

Peng Zhang, Huanzhao Wang, Chengchen Hu, and Chuang Lin

Abstract

Software defined networking greatly simplifies network management by decoupling control functions from the network data plane. However, such a decoupling also opens SDN to various denial of service attacks: an adversary can easily exhaust network resources by flooding short-lived spoofed flows. Toward this issue, we present a comprehensive study of DoS attacks in SDN, and propose multi-layer fair queueing (MLFQ), a simple but effective DoS mitigation method. MLFQ enforces fair sharing of an SDN controller's resources with multiple layers of queues, which can dynamically expand and aggregate according to controller load. Both testbed-based and emulation-based experiments demonstrate the effectiveness of MLFQ in mitigating DoS attacks targeted at SDN controllers.

Introduction

Software defined networking (SDN) is a promising architecture for computer networks [1]. In SDN, control functions are decoupled from switches and centralized at a controller. The controller allows operators to express high-level policies, and breaks down these policies into low-level flow rules that are installed by switches. Switches are only responsible for forwarding packets according to these rules. Such a decoupling of the control plane and data plane makes SDN a desirable approach to design, build, and manage large networks.

Generally, there are two modes in which rules can be installed at switches: *proactive* and *reactive*. For the proactive mode, the controller first breaks down network policies into flow rules, and installs them at switches when the network bootstraps. For the reactive mode, the controller computes and installs rules only when a switch explicitly requests them. Clearly, reactive mode enables switches to quickly adapt to network dynamics, and does not require switches to have large flow tables.

However, the reactive rule installation also makes SDN controllers and switches vulnerable to denial of service (DoS) threats. Specifically, an attacker can compromise hosts and flood short-lived spoofed flows such that switches have to send a large number of requests to the controller. The consequences include:

1. The software components of switches become overloaded.
2. The channels between switches and the controller become congested.
3. The controller's resource becomes saturated.
4. The switches' hardware flow tables overflow.

All these consequences can seriously downgrade the performance of a portion of or even all of the network.

There have been some studies on DoS attacks in SDN [2]. For example, Avant-Guard [3] and FloodGuard [4] target the controller resource saturation threat; Wang *et al.* [5] focus on the threat of switch software overloading; Kandoi *et al.* [6] discuss the control channel congestion and flow table overflow threats. However, they only consider one or two attacks, and a more systematic treatment of DoS threats in SDN is needed. To this end, the first goal of this article is to give a systematic study of DoS threats in SDN. Our approach is to break down the reactive rule installation into multiple stages, show the bottlenecks of each stage, and explain how an adversary can exploit the bottlenecks for DoS attack.

Controller resource saturation is perhaps the most concerning form of DoS attack in SDN, while existing approaches like Avant-Guard [3] and FloodGuard [4] need modifications of SDN switches. A better approach is to mitigate the attack at the controller side, while leaving the switches untouched. Thus, our second goal is a new countermeasure to controller resource saturation attacks in SDN.

We propose to enforce fair sharing of a controller's resources among switches and hosts in the network. The challenge is that when there are a large number of switches and hosts, we need to maintain a large number of queues. To address this challenge, we propose multi-layer fair queueing (MLFQ), a novel queue management method that allows dynamic queue expansion and aggregation. Using MLFQ, the controller normally only needs to maintain a small number of queues, and when attacks take place, the controller expands the corresponding queues into multiple lower-level queues to isolate flooded requests.

In sum, our contribution is three-fold:

- We present a systematic review of DoS threats in SDN, and discuss the limitations of existing countermeasures.
- We propose MLFQ, a new approach to mitigate controller resource saturation attacks in SDN. MLFQ is simple but effective, and does not require switch modifications.
- We demonstrate the effectiveness of MLFQ with both a physical testbed and an emulated network.

Denial of Service in SDN

This section first introduces the multiple stages of reactive rule installation in SDN, and show how the bottlenecks of each stage can be exploited for DoS attacks.

Peng Zhang is with Xi'an Jiaotong University, and the Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory.

Huanzhao Wang, and Chengchen Hu are with Xi'an Jiaotong University.

Chuang Lin is with Tsinghua University.

Digital Object Identifier:
10.1109/MNET.2016.1600109NM

Background on Reactive Rule Installation

We consider a network that deploys OpenFlow [7], the de facto standard for SDN. An OpenFlow switch consists of a *switching application-specific integrated circuit* (ASIC) and an OpenFlow Agent (OFA). The switching ASIC is a piece of hardware holding one or multiple *flow tables*, and each flow table consists of a set of *flow rules*, or simply *rules*, that indicate how to process packets. In this article, we assume a switch has only one flow table. The OFA is a software agent that talks with the controller using the OpenFlow protocol. The connection between an OFA and the controller is termed the *control channel*.

The process of reactive rule installation is illustrated in Fig. 1. When a packet comes to a switch, the switch first looks up in its flow table using the packet header and input port. If there are no matching rules, the switch treats the packet as the first packet of a new flow, and triggers a `table_miss` event to the OFA. Then the OFA stores the packet in the *packet buffer*, encapsulates a fraction of the packet header into a flow request, and sends it to the controller via the control channel. Since the flow request is defined by the `packet_in` message in OpenFlow, we use flow request and `packet_in` interchangeably in the rest of this article. When the packet buffer becomes full, the OFA sends the entire packet to the controller without buffering it.

Upon receiving a `packet_in` message, the controller computes a set of rules and installs these rules at the switches along the path of the flow. Each rule is assigned a *hard-timeout* and a *soft-timeout* that jointly determine how long the rule can exist. On receiving a rule, a switch's OFA checks whether its flow table is full. If not, the rule is inserted into the flow table; otherwise, the rule is dropped and an error message will be sent to the controller. From version 1.4 on, the OpenFlow specification has allowed switches themselves to evict rules without consulting the controller. This means that switches can locally delete rules in order to make space for new flows.

Flow Request Flooding in SDN

As shown above, OpenFlow switches can install rules reactively by sending flow requests. Consider a compromised host that injects a large number of short-lived flows (either TCP or UDP) into the network. If crafted well, each flow forces a switch to send a flow request to the controller. We term this attack *flow request flooding*, and the switch that is forced to send flow requests the *victim switch*. In the following, we show how flow request flooding attacks can exploit multiple stages of reactive rule installation for DoS attacks, and discuss existing countermeasures. For convenience, we summarize these countermeasures in Table 1.

Threat 1: Switch Software Overloading

Threat: As mentioned above, the OFA of the victim switch should generate a flow request and send it to the controller. Since OFAs generally run on relatively low-end CPUs, they can only generate a limited number of flow requests per unit time. Wang *et al.* show that hardware switches like HP Procurve and Pica8 Pronto can only generate less than 1000 requests per second [5]. Thus, the OFA is indeed a bottleneck and can be overloaded, and as a result, flows from benign hosts may be delayed or dropped. Note that this threat has local impact: only those hosts directly connected with the victim switch are affected.

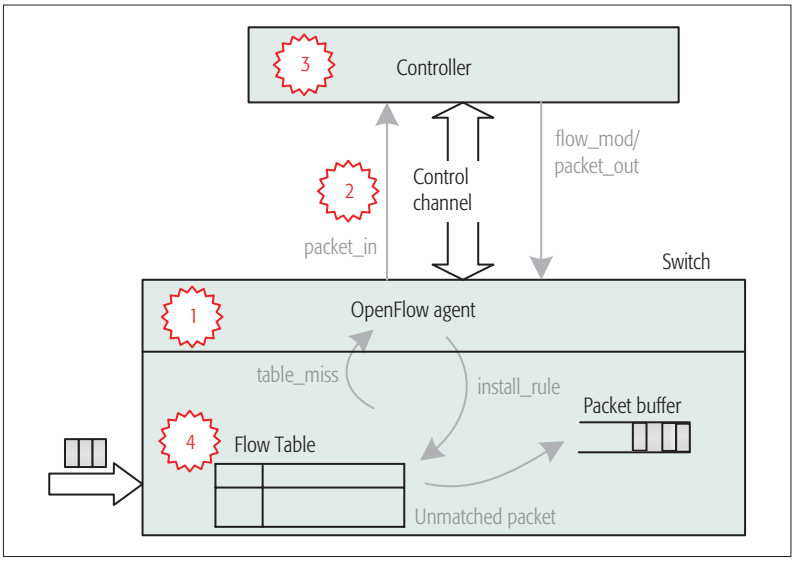

FIGURE 1. Illustration of the reactive rule installation process in OpenFlow SDN.

Countermeasures: To mitigate such a DoS threat, Scotch [5] uses an overlay network of software switches (e.g., Open vSwitches [8]), as a complement to hardware switches. Since software switches can run on more powerful CPUs, they can generate many more flow requests compared to hardware switches. New flows received by hardware switches would be redirected to software switches, which are responsible for generating flow requests. Data plane traffic can still be forwarded by hardware switches for large throughput. Indeed, Scotch can increase the number of new flows that a switch can handle in benign settings, while it may not be enough in adversarial settings, where the attacker can flood at a rate even higher than a software switch can handle.

Threat 2: Control Channel Congestion

Threat: Even though switches individually maintain their control channels with the controller, these logically separate channels may share common physical links. Thus, flow requests flooded by the victim switch may overwhelm common bottleneck links, and normal flow requests using those links experience congestion. In addition, if the packet buffer of the victim switch becomes full, the switch sends entire packets (instead of just packet headers) to the controller, resulting in even higher bandwidth consumption. Compared to the switch software overloading threat, which only affects those hosts connected to the victim switch, control channel congestion affects all hosts with flow requests traversing the congested links.

Countermeasures: Floodlight [9] uses a simple port throttling method to prevent switches from requesting too fast. Specifically, the controller keeps monitoring the `packet_in` rate for each switch, and once it exceeds a threshold, the controller begins to sample every *x* `packet_in`

DoS threats	Related works				
	Avant-Guard [3]	Scotch [5]	FloodGuard [4]	Kandoi [6]	MLFQ
Switch software overloading		✓			
Control channel congestion			✓		
Controller resource saturation	✓		✓	✓	✓
Flow table overflow				✓	

TABLE 1. An overview of DoS threats and countermeasures in SDN.

sent by that switch. If two `packet_in`s from the same medium access control (MAC) address/switch port are sampled, the controller installs a rule at the switch to block flows from that MAC address/switch port for 5 s. This simple throttling method has two problems. First, the threshold is a fixed value (currently set to 1000/s) and cannot adapt to network dynamics. Second, blocking all flows from certain MAC addresses or ports may be too aggressive, since there may be multiple hosts connected to the same port, and blocking that port can punish benign hosts too.

We think a better way is to dynamically adjust the requesting rates of switches, instead of temporary blocking. Once the controller detects that it is receiving more flow requests than it can handle, it can instruct switches to reduce the requesting rates. When the controller's load becomes low, it instructs switches to increase the requesting rates again. Yau *et al.* [10] proposed such a rate adjustment method to mitigate distributed DoS (DDoS) attacks in traditional IP networks. In their method, when the victim server notices that it is overloaded, it multicasts a rate-limit signal with parameter r_S to routers. Each router adjusts their sending rates to be strictly less than r_S. If the server is still overloaded, r_S will be cut in half and multicasted again. On the other hand, if the server load is smaller than a threshold, the server increases r_S additively. The authors proved that this additive increase multiplicative decrease (AIMD)-style adjusting method could converge. This method can be adapted to the SDN setting, where the controller can use it to adjust the requesting rates of switches.

Threat 3: Controller Resource Saturation

Threat: If the flooded flow requests arrive at the controller, they will consume the controller's resource (CPU, memory, bandwidth, etc.) for rule computation and installation. Without any protection, the controller's resource can be saturated by the flooded requests, and legitimate requests may be dropped. Similar to the control channel congestion, controller resource saturation has a network-wide consequence, as all switches connected to the controller will be affected.

Countermeasures: Avant-Guard [3] considers the controller resource saturation threat caused by TCP SYN flooding, where the attacker initiates a large number of TCP connections to a victim server. This will cause the victim switch to send a large number of `packet_in` messages to the controller. Avant-Guard lets switches proxy all TCP connections, and only send a flow request once a TCP connection finishes a handshake. A problem with Avant-Guard is that it only works for SYN flooding, but cannot defend against UDP or ICMP flooding. In addition, Avant-Guard needs to modify switches, which is undesirable for real deployment.

In contrast, FloodGuard [4] can defend against general flow request flooding attacks. Once the controller detects an attack, it installs a default rule at the victim switch so as to redirect all new flows to a *data plane cache*. The data plane cache is responsible for generating flow requests to the controller. At the same time, the controller proactively generates rules by symbolically executing controller applications, and installs these rules at the victim switch to suppress future flow requests. One possible problem with FloodGuard is that symbolic execution may not exhaust all possible execution paths for complicated controller applications. In addition, FloodGuard needs to deploy extra devices on the data plane.

Threat 4: Flow Table Overflow

Threat: An important feature of SDN is the fine-grained flow-based control, and this feature requires more matching fields than IP networks. For example, the latest OpenFlow standard defines 45 matching fields, of which 14 fields are mandatory [7]. Since TCAM is generally a scarce resource, typical commercial OpenFlow switches only support a small number of flow rules. For example, the Pronto-Pica8 3290 switch can only hold 2000 rules [11]. Thus, given that the controller has successfully processed the flooded requests and installed the rules, the victim switch's flow table fills up quickly and eventually overflows. As a result, rules for legitimate flows are dropped. Similar to switch software overloading, this threat also has a local impact as it only reduces the throughput of victim switches.

Countermeasures: Kandoi *et al.* [6] suggest that the controller should configure optimal timeout values to prevent flow table overflow. However, they have not shown how to obtain such optimal values. In addition, deleting only timed-out rules can hardly be enough during attacks, since rules can be installed much faster than they timeout. We suggest the following two approaches. First, the controller can proactively install aggregating rules at switches so as to reduce the number of rules at switches. Second, the controller can also compress rules already in the flow table. Previous packet classifier compression methods such as TCAM Razor [12] may be adapted for this purpose.

Mitigating the Controller Resource Saturation Attack

This section presents a new method to mitigate controller resource saturation, one DoS threat introduced earlier. Different from previous approaches like Avant-Guard [3] and FloodGuard [4], our method works at the controller side, and does not need any modifications of switches or any extra data plane devices. The basic idea is to let the controller schedule among switches or hosts in a fair way so as to isolate legitimate flow

requests from flooded ones. We first present the single-layer fair queueing (SLFQ) method as a baseline, and then present MLFQ, which addresses several limitations of SLFQ.

Single-Layer Fair Queueing

In SLFQ, the controller maintains a queue for each switch. When a request comes, the controller assigns it to the queue of the corresponding switch. Then the controller uses weighted round-robin (WRR) to poll requests from these queues for processing. Specifically, let there be n switches S_1, S_2, ..., S_n, with weights w_1, w_2, ..., w_n, respectively. Let W_{max} and W_{div} be the maximum and the greatest common divisor of the weights, respectively. The controller maintains a parameter W, which is initialized to W_{max}. In each round of scheduling, the controller iterates over all switches' queues, and polls one packet from S_i's queue if $w_i \geq W$. After a round, W is decreased by W_{div}, and if W becomes less than 0, it is set to W_{max} again. For example, suppose there are three switches A, B, and C, with weights 3, 2, and 1, respectively. Then the controller polls queues in the order $ABCABA$.

Even though SLFQ can ensure max-min fairness among switches, it has limitations. Consider a data center that consists of tens of thousands of switches. Maintaining a queue for each switch and scheduling among them can incur a large overhead. Furthermore, even if we can afford to maintain a queue for each switch, hosts under the same switch as the attacker may be punished unfairly since their flow requests share the same queue. Again, maintaining a queue for each port may solve this problem, but will end up with even more queues.

Multi-Layer Fair Queueing

To address the above challenges, we present MLFQ. The basic idea is to maintain just a small number of queues at the controller when there are no attacks, and dynamically "expand" a queue into multiple sub-queues when its size exceeds a threshold (indicating the existence of flooded requests in this queue). If the sizes of these expanded sub-queues all drop below another threshold, they are "aggregated" into a single queue again. For example, initially each queue corresponds to a group of switches. When the size of a queue is beyond a threshold, it is expanded into per-switch queues. If the size of a per-switch queue is still beyond the threshold, it is further expanded to per-port queues. Finally, the queues in the controller are organized in multiple layers, and the layer a queue is at depends on how many switches or hosts it aggregates.

Without loss of generality, in the following we only consider MLFQ with two layers: the first layer corresponds to per-switch queues, and the second layer corresponds to per-port queues, as shown in Fig. 2. For simplicity, we refer to them as switch queues and port queues.

Algorithm 1 specifies how a received flow request, denoted as *pktin*, is enqueued. If the switch queue is expanded into port queues, pktin will be put the corresponding port queue (lines 1–4); otherwise, pktin will be put in the switch queue (lines 5–6). If the switch queue is not expanded, the controller checks whether the queue length is larger than a threshold Th_{high}. If so, the controller creates port queues for this switch (lines 7–9).

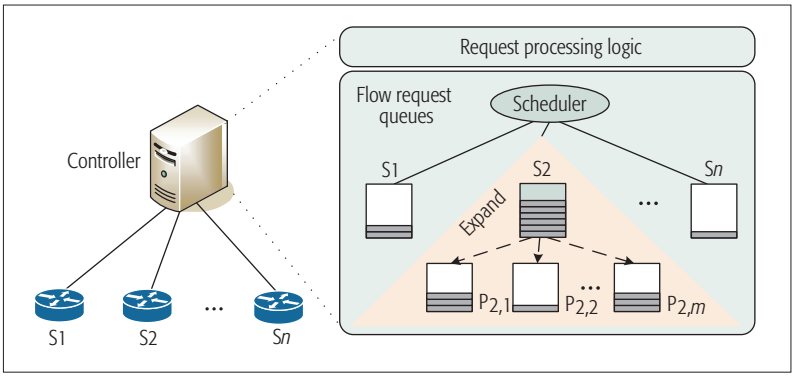

FIGURE 2. Illustration of the MLFQ method.

```
1. i ← getSwitchID(pktin);
2. if Expanded(i) then
3.     j ← getPortID(pktin);
4.     PortQueue(i, j).enqueue(pktin);
5. else
6.     SwitchQueue(i).enqueue(pktin);
7.     if SwitchQueue(i).length > Th_high then
8.         Create a queue PortQueue(i, j) for each port j of switch i;
9.         Expanded(i) ← true;
```

ALGORITHM 1. `OnReceiveRequest(pktin)`.

```
1. i ← 1; W ← W_max;
2. while true do
3.     if w_i ≥ W then
4.         if SwitchQueue(i).length > 0
5.             Process a request from SwitchQueue(i);
6.         else if Expanded(i)
7.             Schedule among port queues PortQueue(i, j) with WRR;
8.             if ∀j, PortQueue(i, j).length < Th_low then
9.                 Put packets in port queues back into SwitchQueue(i);
10.                Delete all port queues PortQueue(i, j);
11.                Expanded(i) ← false;
12.    if i = n then
13.        i ← 1; W ← W − W_div;
14.        if W < 0
15.            W ← W_max;
16.    else
17.        i ← i + 1;
```

ALGORITHM 2. `ScheduleQueue()`.

Algorithm 2 specifies how queues are scheduled at the controller. First, the controller iterates over all switches using WRR (lines 1–3). If a switch queue is nonempty, the controller polls a request from it for processing (lines 4–5). Otherwise, if the switch queue is expanded, the controller schedules among port queues with WRR (lines 6–7). After that, the controller checks whether the length of each port queue is below a threshold Th_{low}. If so, the controller puts all unprocessed messages in the port queues back into the switch queue, and deletes the port queues (lines 8–11). Finally, parameters used for WRR are updated, as explained earlier (lines 12–17).

Parameters. The queue length used in the

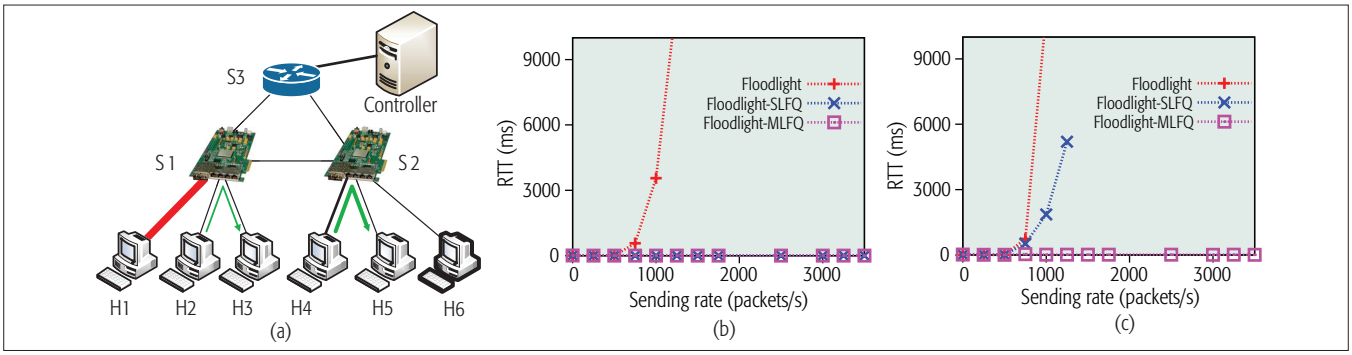

FIGURE 3. Testbed-based experiments. The topology consists of one controller, two physical OpenFlow switches, and one Ethernet switch: a) topology; b) RTT when H4 pings H5; c) RTT when H2 pings H3.

algorithms is an exponential weighted moving average (EWMA): $L_{avg} \leftarrow (1 - \alpha)_{avg} + \alpha L_{ins}$, where L_{avg} and L_{ins} are the average and instant queue length, and $0 < \alpha < 1$ is a constant value. Second, Th_{high} should not be too small in order to prevent burst of flow requests from triggering unnecessary queue expansions. Here, we calculate Th_{high} using the method proposed in RED [13]:

$$L + 1 + \frac{(1-\alpha)^{L+1} - 1}{\alpha} < Th_{high}, \quad (1)$$

where L is the number of burst requests allowed.

Experiment

We implement SLFQ and MLFQ by modifying the Floodlight controller v. 0.9 [9], and evaluate them with both hardware testbed and emulated networks.

Setup and Methodology

We run `hping3` on one host to generate UDP packets with spoofed random source addresses, each of which triggers the victim switch to send a flow request to the controller. At the same time, we let normal hosts ping each other to check the availability of the controller. To ensure each ping triggers a flow request toward the controller, we set the idle-timeout to 1 s and the ping interval to 2 s. In our experiment, we set the queue size at the controller to 1000. For SLFQ and MLFQ, we set $\alpha = 0.01$, $Th_{high} = 10$, and $Th_{low} = 2$. Then a maximum burst of 48 requests is allowed according to Eq. 1. The controller runs on a laptop with a 2.40 GHz Intel Core i3 CPU and 4 GB RAM.

Hardware Testbed

First, we set up a testbed with three physical switches and six hosts, as shown in Fig. 3a. Here, S1 and S2 are FPGA-based OpenFlow switches developed by us [14], and S3 is a standard Ethernet switch. Each OpenFlow switch has four 1 Gb/s ports for data forwarding, and one 1 Gb/s port for controller connection. The OpenFlow agent of the switch runs on a dual-core 800 MHz ARM processor.

We let H1 be the attacking host, and let H2 ping H3, and H4 ping H5. Figure 3b shows the round-trip times (RTTs) between H4 and H5. We can see that using the native Floodlight, the RTTs become extremely large when the flooding rate is above 1000/s, while the RTTs are almost unchanged when using either SLFQ or MLFQ. This demonstrates the effectiveness of SLFQ and MLFQ in maintaining fairness among multiple switches. Figure 3c shows the RTTs between H2 and H3. We can see that SLFQ fails to respond to flow requests when the flooding rate is above 1250/s, since these requests and flooded ones are placed in the same queue (i.e., the switch queue for S1). However, MLFQ still maintains small RTTs due to the expansion of switch queues into port queues. This demonstrates that MLFQ can effectively ensure fairness among multiple ports of the same switch.

Mininet-Based Emulation

For a larger topology, we emulate a $k = 4$ fat tree with Mininet [15]. As shown in Fig. 4a, the topology consists of 20 Open vSwitches (OVS [8]) and 16 hosts.

Similar to the previous experiment, we let H1 be the attacking host, and let H2 ping H3, and H4 ping H5. The results shown in Figs. 4b and 4c acknowledge the testbed-based experiment. Two differences should be noted. First, SLFQ performs a little better when H2 pings H3 than in the testbed-based experiment. The reason is that in the testbed-based experiment, both the ICMP requests and responses trigger S1 to send flow requests, and they compete with flooded requests at S1's queue. On the other hand, in this experiment the ICMP responses trigger another switch to send flow requests, which does not compete at S1's queue. Second, we observe that when the flooding rate increases to 1500/s, the OVS (i.e., S1) becomes the bottleneck and begins to drop `packet_ins`; thus, we stop increasing the rate.

Conclusion

This article presents a systematic review of various DoS threats in SDN. Then we focus on one such attack, controller resource saturation, and propose a queueing-based countermeasure. To address the challenges of maintaining too many queues, we present MLFQ, which allows queues to be dynamically expanded and aggregated according to controller load. With MLFQ, the controller only needs to maintain a small number of queues under normal situations, while it can effectively isolate flooded requests when attacks take place. Both the testbed-based and emulation-based experiments demonstrate the feasibility and effectiveness of MLFQ.

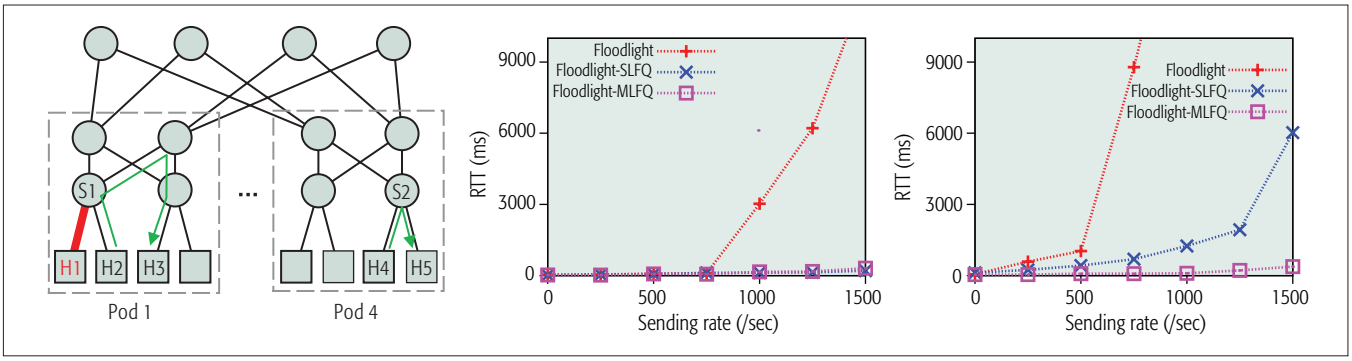

FIGURE 4. Emulation-based experiments. The topology is a k=4 fat tree, part of which is shown in a) topology; b) RTT when H4 pings H5; c) RTT when H2 pings H3.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (No. 61402357, 61672425), and the open project of the Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory (ITD-U15004/KX152600013).

References

[1] N. McKeown *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Comp. Commun. Rev.*, vol. 38, no. 2, 2008, pp. 69–74.

[2] Q. Yan *et al.*, "Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges," *IEEE Commun. Surveys & Tutorials*, vol. 18, no. 1, 2016, pp. 602—22.

[3] S. Shin *et al.*, "AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks," *Proc. ACM CCS*, 2013.

[4] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks," *Proc. IEEE/IFIP Int'l. Conf. Dependable Systems and Networks*, 2015.

[5] A. Wang *et al.*, "Scotch: Elastically Scaling Up SDN Control-Plane Using Vswitch Based Overlay," *Proc. ACM CoNEXT*, 2014.

[6] R. Kandoi and M. Antikainen, "Denial-of-Service Attacks in OpenFlow SDN Networks," *Proc. IFIP/IEEE Int'l. Symp. Integrated Network Management*, 2015.

[7] "OpenFlow Switch Specification Version 1.5.1," https://www.opennetworking.org/sdn-resources/technical-library.

[8] Open vSwitch, http://openvswitch.org/.

[9] "Floodlight Project," http://www.projectfloodlight.org/floodlight/.

[10] D. K. Yau *et al.*, "Defending Against Distributed Denial-of-Service Attacks with Max-Min Fair Server-Centric Router throttles," *IEEE/ACM Trans. Net.*, vol. 13, no. 1, 2005, pp. 29–42.

[11] N. Katta *et al.*, "Cacheflow: Dependency-Aware Rule-Caching for Software-Defined Networks," *Proc. ACM Symp. SDN Research*, 2016.

[12] C. R. Meiners, A. X. Liu, and E. Torng, "TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs," *Proc. IEEE Int'l. Conf. Network Protocols*, 2007.

[13] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Trans. Net.*, vol. 1, no. 4, 1993, pp. 397–413.

[14] C. Hu *et al.*, "Design of All Programmable Innovation Platform for Software Defined Networking," *Proc. Open Networking Summit*, 2014.

[15] Mininet, http://mininet.org/.

Biographies

Peng Zhang (p-zhang@xjtu.edu.cn) received his Ph.D. degree in computer science from Tsinghua University, China, in 2013. He is now an associate professor in the Department of Computer Science and Technology, Xi'an Jiaotong University, China. His research interests include network security, privacy, and software-defined networks.

Huanzhao Wang (hzhwang@xjtu.edu.cn) received her Ph.D. degree in computer science from Xi'an Jiaotong University, China, in 2009. She is now an associate professor in the Department of Computer Science and Technology, Xi'an Jiaotong University. Her research interests include wireless networks and software-defined networks.

Chengchen Hu (chengchenhu@xjtu.edu.cn) received his Ph.D. degree from Tsinghua University in 2008. He is currently a professor with the Department of Computer Science and Technology, Xi'an Jiaotong University. His research interests include network measurement, data center networking, software defined networking, and so on. He is a recipient of a fellowship from the European Research Consortium for Informatics and Mathematics (ERCIM), New Century Excellent Talents in University awarded by Ministry of Education, China.

Chuang Lin (clin@csnet1.cs.tsinghua.edu.cn) received his Ph.D. degree in computer science from Tsinghua University in 1994. He is now a professor in the Department of Computer Science and Technology, Tsinghua University. He is an Honorary Visiting Professor at the University of Bradford, United Kingdom. His current research interests include computer networks, performance evaluation, network security analysis, and Petri net theory and its applications.