

Detection and Mitigation of DoS Attacks in Software Defined Networks

Shang Gao¹, Zhe Peng¹, Bin Xiao¹, *Senior Member, IEEE, Member, ACM*, Aiqun Hu²,
Yubo Song, and Kui Ren, *Fellow, IEEE, Member, ACM*

Abstract—The introduction of software-defined networking (SDN) has emerged as a new network paradigm for network innovations. By decoupling the control plane from the data plane in traditional networks, SDN provides high programmability to control and manage networks. However, the communication between the two planes can be a bottleneck of the whole network. SDN-aimed DoS attacks can cause long packet delay and high packet loss rate by using massive table-miss packets to jam links between the two planes. To detect and mitigate SDN-aimed DoS attacks, this paper presents FloodDefender, an efficient and protocol-independent defense framework for SDN/OpenFlow networks. FloodDefender stands between the controller platform and other controller apps, and conforms to the OpenFlow policy without additional devices. The detection module in FloodDefender utilizes new frequency features to precisely identify SDN-aimed DoS attacks. The mitigation module uses three new techniques to efficiently mitigate attack traffic: table-miss engineering to prevent the communication bandwidth from being exhausted; packet filter to filter out attack traffic and save computational resources of the control plane; and flow rule management to eliminate most of useless flow entries in the switch flow table. Our evaluation on a prototype implementation of FloodDefender shows that the defense framework can precisely identify and efficiently mitigate the SDN-aimed DoS attacks with very little overhead.

Index Terms—SDN, SDN-aimed DoS attacks, attack detection, table-miss engineering, packet filter, flow rule management.

I. INTRODUCTION

SOFTWARE-defined networking (SDN) has speeded up network innovations for the ossified network infrastructure. By separating the traditional network architecture into control and data planes, SDN introduces a more flexible way to manage and control network traffic with high programmability [1]. This logical centralized control plane dictates the whole network behavior through a “southbound” protocol. Among all implementations of SDN (the “southbound” protocol), the OpenFlow [2] framework is the leading embodiment of SDN concept. The control plane installs flow rules on the data plane via OpenFlow

protocol. The data plane then follows these flow rules to handle network flows. When a packet that does not match any existing flow rules (table-miss packet) comes, the data plane encapsulates this packet into a `packet_in` message and reports it to the control plane for instructions.

The communication between the control and data planes causes considerable overhead, and could become a bottleneck of the whole network [3], [4]. Today’s commercial OpenFlow switches [5] only support cable connection to the controller. The practical connection bandwidth was tested to be less than 10Mbps [6], [7]. This costly communication can be leveraged by an attacker to launch SDN-aimed DoS attacks (e.g., data-to-control plane saturation attacks) [6], [8]. Specifically, the attacker randomly forges some or all fields of a packet, making it hard to match with any existing flow rules on a victim switch. Then, the attacker sends a large amount of these table-miss packets to flood the network by SDN-aimed DoS attacks. These table-miss packets will trigger massive `packet_in` messages from the victim switch to the controller, and consume their communication bandwidth, CPU computation, and memory in both control and data planes.

We face the following three challenges to protect OpenFlow networks against the SDN-aimed DoS attacks:

- How to precisely detect SDN-aimed attacks and timely notify the defense system when attacks occur?
- How to efficiently handle table-miss packets while maintaining short delay, low loss rate and forwarding operation for normal packets?
- How to precisely distinguish attack traffic from benign traffic without straining computational resources?

These three challenges are not easy to solve. The attack detection requires a low false-positive rate to respond timely to attacks. However, detection accuracy and timely response are two factors that conflict with each other. For the second challenge to handle table-miss packets, we cannot simply drop all table-miss packets, since the new flows from benign hosts will be dropped as well. We should figure out a way to let the control plane receive `packet_in` messages (triggered by table-miss packets) without consuming much bandwidth. Because some table-miss packets are generated by benign hosts, we have the third challenge to precisely identify attack traffic and filter them out accordingly. To deal with these three challenges, several solutions have been proposed, such as AvantGuard [6] and FloodGuard [7]. However, both approaches focus on the mitigation of the SDN-aimed DoS attacks. The attack detection, which is equally important to mitigation, is not deeply discussed. Besides, additional devices are used in both approaches, which are not compatible to the standard OpenFlow protocol.

Manuscript received April 11, 2017; revised January 9, 2018, August 28, 2018, and December 10, 2019; accepted March 22, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Guan. Date of publication April 15, 2020; date of current version June 18, 2020. This work was supported in part by the HK RGC GRF under Grant PolyU 152124/19E. (Corresponding author: Bin Xiao.)

Shang Gao and Bin Xiao are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong (e-mail: cssgao@comp.polyu.edu.hk; csbxiao@comp.polyu.edu.hk).

Zhe Peng is with the Department of Computer Science, Hong Kong Baptist University, Hong Kong (e-mail: pengzhe@comp.hkbu.edu.hk).

Aiqun Hu and Yubo Song are with the School of Cyber Science and Engineering, Southeast University, Nanjing 210096, China (e-mail: aqhu@seu.edu.cn; songyubo@seu.edu.cn).

Kui Ren is with the School of Computer Science and Technology, Zhejiang University, Hangzhou 310007, China (e-mail: kuiren@zju.edu.cn).

Digital Object Identifier 10.1109/TNET.2020.2983976

1063-6692 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.

See <https://www.ieee.org/publications/rights/index.html> for more information.

In this paper, we study the SDN-aimed DoS attacks, and propose FloodDefender, a scalable and protocol-independent defense system in OpenFlow networks. FloodDefender stands between the controller platform and other controller apps, and is protocol-independent against all kinds of attack traffic (e.g. TCP-based attacks or UDP-based attacks). All designs in FloodDefender conform to the OpenFlow policy and need no additional devices.

FloodDefender has two modules: detection module and mitigation module. The detection module utilizes new frequency features for attack detection. Frequency features can significantly reduce false-alerts in previous detection solutions. The mitigation module contains three components: table-miss engineering, packet filter, and flow rule management. The table-miss engineering component detours table-miss packets to neighbor switches with wildcard flow rules to protect the communication link between the control and data planes from being jammed; the packet filter component filters out attack packets from the received `packet_in` messages to save computational resources of the controller; and the flow rule management component constructs a robust flow table in the data plane by separating the flow table into “flow table region” and “cache region” to save the Ternary Content Addressable Memory (TCAM) of OpenFlow switches.

This paper also theoretically analyzes the impact of neighbor switches in the table-miss engineering by using an average queueing delay model. The analytical result shows that FloodDefender can keep the average delay of communication links within 0.3s by evenly distributing attack traffic to 3 neighbor switches.

Finally, we implement a prototype of FloodDefender and evaluate its performance in both software and hardware environments. Experimental results show that FloodDefender can reduce the false-alerts and ensure the accuracy in attack detection, save more than 70% and 20% bandwidth in the software and hardware tests respectively, and consume only 0.5% CPU computation to handle attack traffic. Meanwhile, it precisely filters out more than 96% attack traffic, and incurs only 18ms delay and 5% packet loss rate for benign traffic under attacks.

The rest of the paper is organized as follows. Section II introduces some background knowledge and the security problem of SDN-aimed DoS attacks in OpenFlow networks. In Section III, we present the overview of FloodDefender system. Section IV and Section V show the detailed designs in both detection module and mitigation module respectively. In Section VI, we theoretically analyze how many neighbor switches should be involved in the table-miss engineering. The implementation and experimental evaluation of FloodDefender are shown in Section VII. We summarize the related work in Section VIII. In Section IX, we discuss the limitations of FloodDefender and future improvements. Finally, we conclude this paper in Section X.

II. PROBLEM STATEMENT

In this section, we first introduce the workflow of handling normal traffic in OpenFlow networks. Then we present the adversary model of SDN-aimed DoS attacks. Finally, we state the problem and challenges of mitigating the DoS attacks in OpenFlow networks.

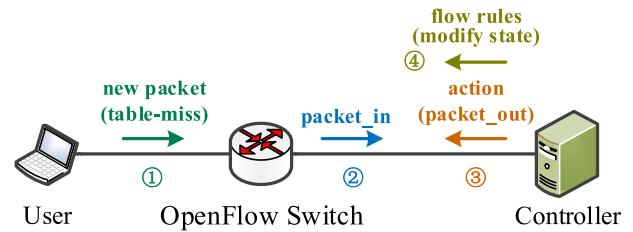


Fig. 1. The workflow of SDN.

A. SDN Workflow

In OpenFlow networks, the controller in the control plane dictates the behaviors of the whole network by installing flow rules on the data plane via two approaches: proactive flow installation and reactive flow installation. In the proactive approach, the control plane pre-installs flow rules on the data plane to process network traffic. The data plane then follows these rules to handle incoming packets. In the reactive approach, when an OpenFlow switch receives several packets, it will queue them in an input queue, and follow the following four steps to process each packet in a FIFO (first input first output) manner, as depicted in Fig. 1.

- 1) The OpenFlow switch looks up its flow table to find flow rules that match with the header of received packet. If a match is found, the switch processes the packet based on the action field of the flow rule. Otherwise, the switch regards the packet as a table-miss packet, buffers the packet, encapsulates its header in a `packet_in` message,¹ and reports to the controller.
- 2) When the controller receives the `packet_in` message, it decides how to process the packet based on the logic of control apps. The “action” will be sent back to the switch in a `packet_out` message.
- 3) The OpenFlow switch then processes the buffered table-miss packet based on the “action” from the controller.
- 4) The controller can further install flow rules (via “modify state” messages) with “match” and “action” fields to indicate the switch to directly process packets of the same flow when received again.

This reactive flow installation approach enables a more flexible way to manage and control network traffic, and has been widely used in most OpenFlow applications.

B. Adversary Model

In reactive OpenFlow networks, the communication overhead between the control plane and data plane could be leveraged by an adversary. An attacker first randomly generates forged packets with forged fields, making them hard to match existing flow rules in a switch. Then, the attacker sends massive table-miss traffic mixed with normal traffic to an OpenFlow switch and launches SDN-aimed DoS attacks. To process each table-miss packet, the victim switch has to buffer it and send out a `packet_in` message with its header, as depicted in Fig. 2. Even worse, the OpenFlow Specification v1.4 [9] requires that a `packet_in` message should contain the whole packet when the memory of a switch is full. This feature could be further exploited by an attacker to flood the network with less resource.

¹An OpenFlow switch will encapsulate the whole packet when its buffer is full.

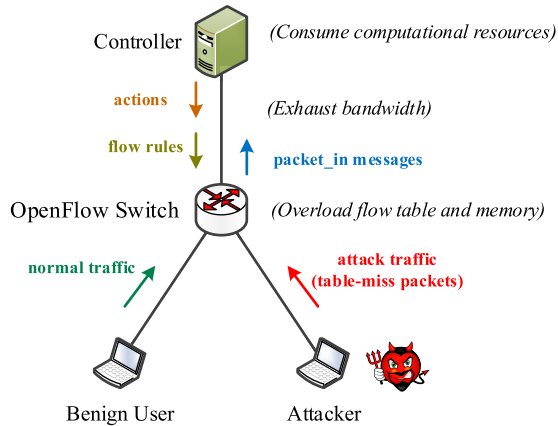
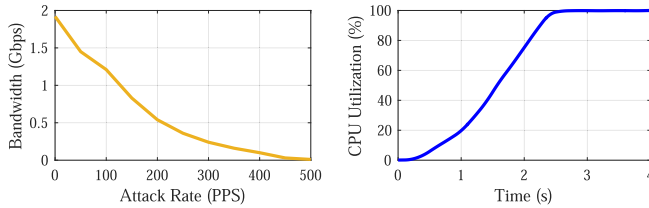


Fig. 2. SDN-aimed DoS attacks in OpenFlow networks.



(a) Victim switch available bandwidth. (b) Control plane CPU utilization.

Fig. 3. Bandwidth and computational resource consumption under SDN-aimed DoS attacks. The data are collected from our experiments.

The DoS attacks can jam the bandwidth between the controller and a switch by generating massive table-miss packets, overload a switch's flow table by installing useless rules, and consume controller's computational resources when processing `packet_in` messages, as depicted in Fig. 3. The result is much worse when the memory of the switch is full. For benign traffic, the throughput of both packet forwarding and packet processing will be significantly degraded. For new flows (benign table-miss traffic), since the switch-controller bandwidth is jammed and the controller is overwhelmed, the switch can hardly receive the `packet_out` message to handle the table-miss traffic (these packets can hardly be processed). Besides, for matched packets (matching with existing rules in the flow table), since they are queued in the switch due to the attack traffic, they have to wait for a long time before the switch process all attacking packets in front of them.²

SDN-aimed DoS attacks are different from DoS attacks in legacy networks. Since switches can process each incoming packet by themselves in legacy networks (i.e. there are no "table-miss" packets in legacy networks), SDN-aimed DoS attacks cannot affect legacy networks. Besides, legacy network switches are designed with high performance in processing packets. Attackers will use great resource to dysfunction a switch in legacy networks. OpenFlow switches have to ask the control plane for instructions to handle table-miss traffic. Attackers can dysfunction OpenFlow switches via flooding table-miss packets with little resource. Lastly, switches can ignore switch-aimed attack traffic (e.g. ignoring TCP and UDP packets with destination IP = target switch's IP) to mitigate

²Processing table-miss packets also consumes a longer time than forwarding matched packets, since processing table-miss packets requires encapsulation and output (to the controller) operations, while forwarding matched packets only requires output (to an output port(s)) operation.

switch-aimed DoS attacks in legacy networks. However, this solution is impractical for OpenFlow switches under SDN-aimed DoS attacks, since attackers can use host-to-host traffic to launch the attacks (e.g. table-miss packets with destination IPs = new hosts' IPs). Ignoring switch-aimed traffic cannot avoid the attacks, and dropping all table-miss packets will affect benign traffic. Therefore, solutions to traditional DoS attacks are not suitable for SDN-aimed DoS attacks.

C. Problem and Challenge

The problem studied in this paper is how to detect and mitigate the SDN-aimed DoS attacks in OpenFlow networks. The attack detection should be fast without causing many false-alerts. FloodGuard [7] uses several features such as `packet_in` rate and utilization of infrastructure to calculate current usage percentage of the network capacity. However, when a network starts up, these features have similar patterns to SDN-aimed DoS attack scenarios, which makes false-alerts. We should use new features to reduce the false-alerts in FloodGuard. To protect the communication bandwidth between the controller and victim switch, a good solution should be able to handle table-miss packets efficiently and maintain the functionality of forwarding benign traffic. Meanwhile, before the control plane processes received `packet_in` messages, the protecting system should filter out attack traffic (from received `packet_in` messages) both efficiently and precisely to save computational resources of the control plane.

In the design of a defense system, we also face two challenges. First, we should be able to handle all kinds of attack traffic (e.g. TCP-based attacks and UDP-based attacks). Solutions such as AvantGuard [6] may not be practical since attackers can use UDP/ICMP-based attack traffic to bypass them. Second, the defense system should be scalable and conform to the OpenFlow policy without employing additional devices. The hardware modification (TCP proxy) in AvantGuard [6] and additional devices (data plane cache) in FloodGuard [7] will increase the cost when deploying in OpenFlow networks.

III. SYSTEM OVERVIEW

We design a system named FloodDefender, which can precisely identify SDN-aimed attacks, as well as save resources like bandwidth, computation and flow table space when SDN-aimed DoS attacks occur. We describe the design of the FloodDefender system, including its architecture and workflow below.

A. FloodDefender Architecture

FloodDefender stands between the controller platform and other controller apps, as depicted in Fig. 4. It consists of two functional modules: detection module and mitigation module. The mitigation module is composed of three components to protect OpenFlow networks against SDN-aimed attacks: table-miss engineering, packet filter, and flow rule management.

FloodDefender works in three states: alert, active, and block, as depicted in Fig. 5. When no attacks are detected, FloodDefender remains in the alert state to monitor network

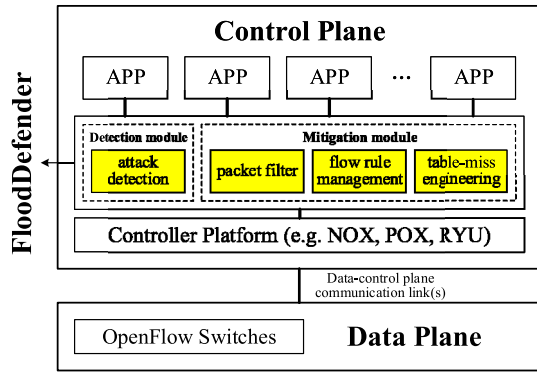


Fig. 4. The architecture of FloodDefender. Apps indicate the OpenFlow control apps on the control plane for network traffic management (e.g. `I2_forwarding` and `firewall`).

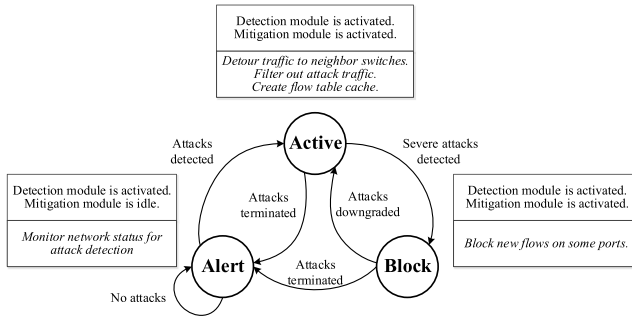


Fig. 5. States of FloodDefender.

status for attack detection, and delivers the `packet_in` messages, action messages, and flow rules between the controller platform and controller apps. When attacks are detected, FloodDefender switches to the active state to mitigate attacks. It filters `packet_in` messages and forwards them to control apps through the packet filter component. It also manages the flow rule installation through the flow rule management component. In some extreme cases, the network is under severe attacks,³ FloodDefender comes to the block state and blocks all table-miss packets from the victim switch's input port which has a high table-miss rate. When attacks are detected to be terminated, FloodDefender switches back to the alert state again.

B. FloodDefender Workflow

Initially, the detection module monitors the network status and the mitigation module remains idle. When the detection module detects SDN-aimed DoS attacks, the mitigation module is activated for attack mitigation in the following six steps:

- 1) The detection module identifies victim's neighbor switches (not under attacks) that directly connect to the controller. The flow rule management component logically separates the flow table into flow table region and cache region;
- 2) The table-miss engineering component installs protecting rules on the victim switch to detour some table-miss packets to neighbor switches. When neighbor switches receive the detoured table-miss packets, they will send `packet_in` messages to the controller;

³The attack traffic could exhaust both the victim switch bandwidth and its neighbor switch bandwidth.

- 3) When the controller receives `packet_in` messages, the packet filter stores them and roughly filters out attack traffic from these messages. The filtered traffic will then be delivered to the control apps;
- 4) The control apps process these packets and then send out action messages and flow rules. Action messages will be sent to the switch that reports `packet_in` messages. However, flow rules will be intercepted by the flow rule management component;
- 5) The flow rule management component decides the monitoring rules based on intercepted processing rules. Intercepted processing rules will be installed on the cache region of the victim switch. Monitoring rules will be installed on the flow table region instead;
- 6) The victim switch can move a rule from its cache region to the flow table region if the rule is regarded as legal by the packet filter component. Cache region will then be flushed to save the space of flow table.

IV. DETECTION MODULE

The detection module continues monitoring the network status for attack detection. When attacks occur, it triggers the mitigation module to work and FloodDefender enters the active state. The detection module also provides important information to dynamically adjust protecting rules for evenly splitting table-miss traffic to neighbor switches. When attacks are detected to be over, mitigation module stops working and FloodDefender goes back to the alert state.

Though FloodGuard [7] utilizes `packet_in` message rate, buffer memory, controller memory, and CPU to identify potential attacks, its detection may cause false alerts. For instance, when an OpenFlow network adopts a reactive approach, all switch flow tables are empty initially. Both `packet_in` message rate and utilization of the infrastructure (buffer memory, controller memory, and CPU) are high and false alerts may be triggered. This is more serious for large networks. Even though FloodGuard can automatically switch to the idle state (similar to the alert state of FloodDefender), false alerts can downgrade the network performance.

To precisely identify attack traffic, we introduce another feature: flow entry frequency. The flow entry frequency describes the number of packets of each flow received by the switch. For attack traffic, the frequency will be very low since the attackers try to generate as much attack traffic as possible. The frequency of normal flows will be much higher. Specifically, we separate flow entries into three parts⁴: low-frequency flows (the frequency is lower than 10, first 4 flows in Fig. 6), mid-frequency flows (the frequency is between 10 and 100, 5th flow in Fig. 6), high-frequency flows (the frequency is higher than 100, 6th flow in Fig. 6). For each part, we calculate the number of flows, and the average frequency of these flows ($\frac{\text{Total number of packets of flows in this part}}{\text{Total number of flow entries in this part}}$).

The frequency of matched flows can be easily collected from the packet count field of the flow rules. While for new flows (table-miss packets), their frequency cannot be obtained directly. Here we use a heuristic method to calculate the new

⁴We use the (0, 10), [10, 100], (100, +∞) splitting since it performs best in our experiments. We also encourage users to adjust the splitting based on their network environment

Flow Rules		
Match	Action	Count
IP_Dst = A	To S1	1
IP_Dst = B	To S2	3
IP_Dst = C	To S1	1
IP_Dst = D	To S2	7
IP_Dst = E	To S1	11
IP_Dst = F	To S2	105

Low-frequency flows: IP_Dst = A, B, C, D
 Mid-frequency flows: IP_Dst = E
 High-frequency flows: IP_Dst = F

Low-frequency flows:
 Number of flows = 4
 Average frequency = 3

Fig. 6. Frequency of flow entries (the number of flows and the average frequency of these flows). Count field represents the number of packets per flow.

flow frequency. Specifically, each table-miss packet of a new flow will be regarded as a 1-frequency flow and grouped based on the destination IP and MAC. Once the corresponding flow rule is installed, all 1-frequency records in this group will be removed, and use the frequency of this flow to calculate frequency features. Otherwise, when the rule is not installed, frequency features are calculated based on the 1-frequency records. For instance, in a layer 2 forwarding OpenFlow network with an existing host A and a new host B (B just joins in). When A sends several packets to B (table-miss packets since “to-B” rule is not installed), each A-to-B packet will be regarded as a 1-frequent flow first and grouped in “to-B” group (“to-B” group just counts the number of 1-frequency records). Then, after “to-B” rule is installed (the installation can be triggered by table-miss packets generated by B), records in “to-B” group will be removed. The frequency of the “to-B” flow will be collected from the packet count field. When A sends packets to X or forges packets from Y to X (X and Y are non-existing hosts), each packet will be regarded as a 1-frequent flow to calculate the frequency features, since the “to-X” rule will not be installed.⁵

The values of frequency feature differ a lot under attacks and in network startups scenarios, especially for low-frequency flows. When networks first start up, the number of low-frequency flows increases slowly, and will drop down quickly due to the increment of frequency (most low-frequency flows become mid-frequency or high-frequency flows). Meanwhile, the average frequency of low-frequency flows will quickly increase to the upper bound (10 in FloodDefender). On the other hand, when networks suffer from SDN-aimed DoS attacks, the number of low-frequency flows increases dramatically till the flow table of the victim switch is full. The average frequency of low-frequency flows will remain in a very low value (1 in our experiment). By employing the frequency features, the two scenarios can be easily identified. Based on frequency features and other mentioned features (`packet_in` message rate, buffer memory, controller memory, and CPU), the attack detection module adopts a certain anomaly threshold to monitor the network status and trigger state transitions of FloodDefender.

Besides attack detection, the detection module also provides important information to the table-miss engineering component in mitigation module when FloodDefender is in active state. This information will help the table-miss engineering component dynamically adjust protecting rules to

⁵A benign host may keep sending packets to a non-existing host (e.g. a client may try to keep connecting to a server until the server is online), which can result in low-frequency rate in our mechanism. We regard such traffic legal as long as the rate is acceptable. To reduce the false-alerts in such scenarios, we combine frequency features as well as other features for detection.

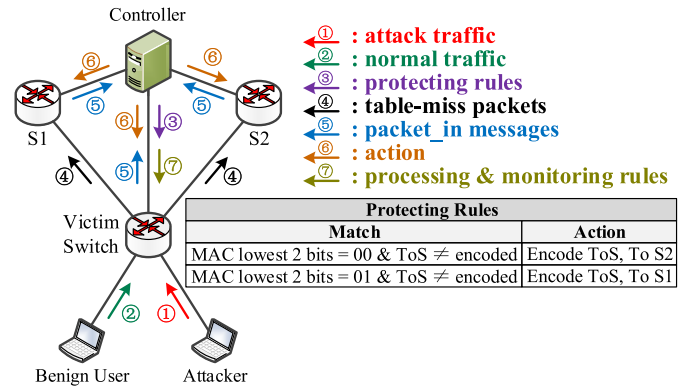


Fig. 7. Detouring table-miss traffic to neighbor switches when attacks occur.

evenly split table-miss traffic to neighbor switches. When the detection module finds that some links between the controller and switches (including both victim switch and neighbor switches) are jammed, or some links have more available bandwidth, the detection module sends the information of these switches to allow the table-miss engineering component offload more/less table-miss packets to them by adjusting protecting rules.

V. MITIGATION MODULE

FloodDefender utilizes three components in the mitigation module to protect OpenFlow networks against SDN-aimed DoS attacks: table-miss engineering, packet filter, and flow rule management.

A. Table-Miss Engineering Component

The table-miss engineering component works when the SDN-aimed DoS attacks are detected. In SDN-aimed DoS attacks, massive table-miss packets (attack traffic and benign traffic, which are represented by step 1 and 2 in Fig. 7 respectively) will be sent to the target switch to exhaust the switch-controller bandwidth. Therefore, the table-miss engineering component will install some flow rules (i.e. protecting rules, respected by step 3 in Fig. 7) to offload some table-miss packets to neighbor switches to save the bandwidth of the victim switch. Specifically, the table-miss engineering component issues protecting rules to forward some table-miss traffic to neighbor switches⁶ (step 4). The corresponding `packet_in` messages will be delivered to the controller via different links (step 5). In this way, the bandwidth between the target switch and controller can be saved. The controller can further process these `packet_in` messages and respond with actions (step 6) and flow rules (i.e. processing rules, step 7). Since the offloaded packets will only be delivered to the controller once by neighbor switches, table-miss engineering component will not amplify the attack traffic between the controller and switches.

Protecting rules are wildcard flow entries with the lowest priority to split the table-miss traffic into several parts to

⁶Though the same “offloading traffic” mechanism is also applied on neighbor switches, we do not allow neighbor switches to use this mechanism when they are involved as neighbors (i.e. the mechanism is not allowed when switches are not victims). This could be done by selecting switches that are not under attacks as neighbors, and involving more neighbors/switching to Block state when attack rate increases.

different neighbor switches. The match fields of protecting rules are adjusted dynamically by a traffic balancer to ensure the load balance of each neighbor switch. When neighbor switches are flooded by attack traffic, table-miss engineering will use more protecting rules to involve more neighbor switches. The maximum number of protecting rules depends on the number of neighbor switches that directly connect to the controller, which is obtained from the network topology at the first place. Protecting rules will not use much TCAM space in an OpenFlow switch. Normally, the bandwidth can be saved with less than 5 protecting rules (5 neighbor switches).

When two victims offload their traffic to one neighbor switch, additional information should be added to identify each victim before the neighbor switch sends `packet_in` to the controller. Hence, in our design we only consider that each neighbor switch is only responsible for one victim. The controller maintains a different set of neighbor switches for each victim switch and adjusts the sets dynamically based on the network status.

There are three challenges in the design of protecting rules: INPORT loss, detoured traffic identification, and packet bouncing problems.

INPORT Loss Problem: In OpenFlow specification, INPORT information indicates the controller's input port, and is contained in a `packet_in` message. Therefore, the `packet_in` message generated by a neighbor switch will replace the original INPORT information with its own. In the design of protecting rules, we should ensure the original INPORT information not to be lost. To solve this problem, we utilize some reserved fields in packet header (e.g. ToS field) to preserve the original INPORT information and denote detoured traffic. Specifically, we encode the ToS field with the INPORT information, and set "modify ToS field" in the protecting rules, as depicted in Fig. 7.

Detoured Traffic Identification Problem: After receiving and processing the detoured `packet_in` messages, the controller will send actions to neighbor switches and flow rules to victim switch respectively, which is different from the procedures in processing regular `packet_in` messages (sending the actions and flow rules to the same switch). Therefore, the controller should be able to identify these detoured `packet_in` messages. Our solution is to identify these detoured `packet_in` messages based on the encoded reserved fields, and associate detoured messages with the datapath of the victim switch to install flow rules. For instance, suppose we use 2-bit reserved ToS field, and the original value is "00". After encoding, this value becomes either "01", "10", or "11". Therefore, the controller regard "00"-messages as regular messages, and other messages as detoured messages. Clearly, we can identify at most $2^n - 1$ different INPORT values with n bits in the reserved field.

Packet Bouncing Problem: Since the neighbor switch may have some flow rules to process the detoured table-miss traffic, some table-miss packet could bounce between the neighbor and victim switches. For instance, the victim switch in Fig. 8 regards packet A as a table-miss, and forwards it to S1 based on the protecting rule. S1 accidentally has a flow rule to process packet A, and the action is "to Victim Switch". Therefore, packet A will bounce between the two switches. To avoid this problem, we only apply the protecting rules on

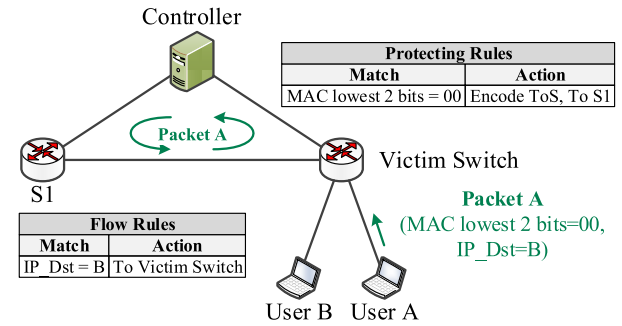


Fig. 8. Packet bouncing problem.

```
# Install protecting rule (to S1)
actions = [dp.ofproto_parser.OFPActionSetField(ip_dscp=
entos), dp.ofproto_parser.OFPActionOutput(1)]
match = datapath.ofproto_parser.OFPMatch(
eth_dst=('00:00:00:00:00:00', '00:00:00:00:00:03'),
ip_dscp=('00', 'C0'))
inst = [parser.OFPInstructionActions(ofproto.
OFPIT_APPLY_ACTIONS, actions)]
mod = parser.OFPFlowMod(datapath=victim_datapath, priority
=0, match=match, instructions=inst)

# Install protecting rule (to S2)
actions = [dp.ofproto_parser.OFPActionSetField(ip_dscp=
entos), dp.ofproto_parser.OFPActionOutput(2)]
match = datapath.ofproto_parser.OFPMatch(
eth_dst=('00:00:00:00:00:01', '00:00:00:00:00:03'),
ip_dscp=('00', 'C0'))
inst = [parser.OFPInstructionActions(ofproto.
OFPIT_APPLY_ACTIONS, actions)]
mod = parser.OFPFlowMod(datapath=victim_datapath, priority
=0, match=match, instructions=inst)
```

Fig. 9. Python codes to generate protecting rules.

non-detoured traffic. Therefore, these packets will bounce only once between the neighbor and victim switches. Specifically, the table-miss engineering adds "ToS is not encoded" into the match field of the protecting rule, as depicted in Fig. 7. When the victim switch receives the bounced-back packets, it delivers them to the controller since these packets do not match the protecting rules.

Based on the descriptions above, we present an example of generating protecting rules. Suppose the victim switch has two neighbor switches (S1 and S2), the protecting rules split table-miss packets based on the lowest 2 bits of source MAC address (MAC lowest 2 bits = 00, to S1; MAC lowest 2 bits = 01, to S2), and we use the reserved 2 bits in IP DSCP (6 bits in ToS field) to encode INPORT information, the protecting rules can be created with codes in Fig. 9.

In conclusion, the table-miss engineering component saves the bandwidth between the victim switch and the controller. It also improves the processing performance of benign flows by (1) reducing the link delay of `packet_in` and `packet_out` messages (with more available bandwidth) for benign table-miss traffic; and (2) reducing the queueing delay of benign matched packets (i.e. more efficiently processing attacking table-miss traffic).

B. Packet Filter Component

Packet filter component can identify attack traffic and filter them out to save the computational resources of the control plane. It works as a low-level app between the controller and other apps to preprocess the `packet_in` messages. It contains two components, `packet_in` buffer to store

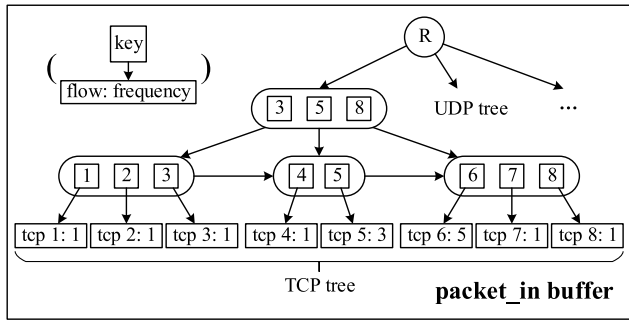


Fig. 10. B+ trees in packet_in buffer component.

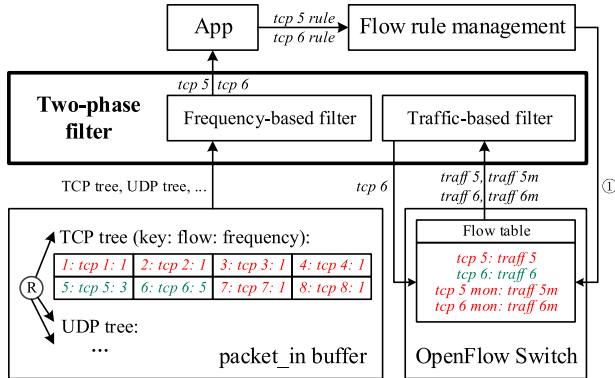


Fig. 11. Two-phase filtering design in the two-phase filter component. Flow rule management inserts tcp 5 monitoring rule, tcp 5 rule, tcp 6 monitoring rule, and tcp 6 rule (represented by ①) to the flow table.

packet_in messages, and two-phase filter to identify attack traffic.

1) *packet_in Buffer*: packet_in buffer classifies detoured packet_in messages based on protocols, and uses a B+ tree to efficiently store and index the packet_in messages of each protocol. The key of each node is the flow entry, and value of a leaf node is the packet_in message and frequency of this flow. For transport layer protocols (TCP and UDP), the key is the combination of source and destination MAC, IP and port; for network layer protocols (e.g. ICMP), the key is the combination of source and destination MAC and IP addresses; and for other protocols (e.g. ARP and RARP), the key is the combination of source and destination MAC addresses. Though SDN apps could use different fields in packet header to define a flow, the most significant ones are those mentioned source and destination fields. All B+ trees are connected by a root pointer R , as depicted in Fig. 10.

packet_in buffer stores packet_in messages in a time period, and flushes all B+ trees after the two-phase filtering to save space. We allow users to set the time of collecting packet_in messages based on their demands. Generally speaking, a longer period will save more computational resources, but will cause longer delay for new benign flows, and will cost more memory of the controller. We also give a suggested time of 5 seconds.

2) *Two-Phase Filter*: Two-phase filter applies two filtering functions to efficiently and precisely identify attack traffic. It first roughly filters out attack traffic based on the frequency in packet_in buffer, and then precisely filters them based on the monitored traffic information, as depicted in Fig. 11.

The frequency of new flows is the most significant feature of SDN-aided DoS attacks. Since the packets belonging to

existing flows will not trigger packet_in messages in most cases, packets of the same flow will downgrade the performance of SDN-aided DoS attacks. Therefore, the attacker tries to generate massive new flows to flood the network, and the frequency of attack flows will be very low.

At the first phase, frequency-based filter utilizes the frequency feature to efficiently filter out attack traffic. It will search the leaf nodes of each protocol's tree, and get the flow records whose frequency is higher than a threshold. This threshold changes dynamically, and is initially set to 1. A bigger threshold will filter out more messages, but may sacrifice some normal traffic. The threshold will be updated based on the result of traffic-based filter. To reduce the false-positives, we adopt a smaller threshold which only filters out a portion of attack traffic (the threshold ensures the recall rate bigger than 60%, and is normally set to 1 or 2 in our experiment), the accuracy will be improved by the traffic-based filtering. For instance, in Fig. 11, the packet filter component searches tcp 1 to tcp 8 in packet_in buffer with $threshold = 1$, and gets tcp 5 and tcp 6. These two messages will be forwarded to apps to generate processing rules. Other TCP flows will be regarded as attack traffic.

At the second phase, traffic-based filter needs to precisely identify normal flows from the filtered flows. It monitors the traffic of each flow with processing rules and extracts features for classification. To precisely identify attack packets even considering that attackers are smart enough to resend these packets to increase the frequency of each flow, we use traffic rate asymmetry features in the classification. Asymmetry features can be extracted by monitoring the traffic of "reverse flows" (response packets of one flow). For example, in Fig. 11, a layer 2 learning switch has a processing flow entry " $eth_dst = 00:00:00:00:01, action = outport:01$ " for tcp 5, that forwards packets from port 01 when its destination MAC is 00:00:00:00:00:01. The reverse flow is the packets with source MAC 00:00:00:00:00:01 and input port 01. If this flow entry is installed maliciously by an attacker with forged source MAC address, there will not be much reverse traffic for tcp 5, since no one can establish a connection with 00:00:00:00:00:01 on port 01. By adopting the asymmetry features (*traff 5m*), the traffic-based filter can precisely classify tcp 5 as attack traffic. We use monitoring flow rules to monitor reverse traffic. In this case, the match field of the monitoring rule is " $eth_src = 00:00:00:00:00:01 \&\& in_port = 01$ ".

Though asymmetric features can be applied to most flows, they can also lead to incorrect results in some cases, and cause the asymmetric feature problem:

Asymmetric Feature Problem: Asymmetric features can lead to incorrect classification results for some asymmetric flows, such as flow entries with the "drop" action or multi-path routing flows, since the reverse traffic of these flows can be hardly observed on the victim switch. For instance, a firewall app blocks all packets with source IP 0.0.0.2 and destination IP 0.0.0.1 (" $ipv4_src = 0.0.0.2 \&\& ipv4_dst = 0.0.0.1, action = []$ "). The monitoring flow rule of the blocked packets is " $ipv4_src = 0.0.0.1 \&\& ipv4_dst = 0.0.0.2$ ". Since the connection is not established, there will not be reverse traffic for this flow. Using asymmetric features for these asymmetric flow rules could lead to incorrect classification. To solve

this problem, we will not use asymmetric features for the classification of these asymmetric flows.

Specifically, we use the following features for traffic-based filtering classification:

- 1) *Packet Count (P)*: describe the total number of packets of one flow entry in an interval;
- 2) *Byte Count (B)*: describe the total number of bytes of one flow entry in an interval;
- 3) *Asymmetric Packet Count (AP)*: describe the total number of packets of one reverse flow entry in an interval;
- 4) *Asymmetric Byte Count (AB)*: describe the total number of bytes of one reverse flow entry in an interval.

After extracting the features above, we employ Support Vector Machine (SVM) [10], a supervised learning model as our classifier. This classification algorithm is robust even with noisy training data. The detailed implementation can be referred to [10], and we skip this part due to space constraints.

We summarize frequency-based and traffic-based filtering algorithms in Algorithm 1 and Algorithm 2:

Algorithm 1 Frequency-Based Filtering ($R, freq$)

Input: R : set of leaf nodes; $freq$: frequency threshold

Output: F : set of filtered flows

- 1: $F \leftarrow \emptyset$
 - 2: **for** each $p \in R$ **do**
 - 3: **if** $p.freq > freq$ **then**
 - 4: $F.add(p)$
 - 5: **end if**
 - 6: **end for**
 - 7: **return** F
-

In conclusion, the packet filtering component saves the computational resources of the control plane. It also improves the performance of processing benign table-miss traffic by dropping illegal `packet_in` messages.

C. Flow Table Management Component

The flow table management component installs monitoring rules on the victim switch's flow table, and manages the flow rule installing on the victim switch. Monitoring rules are generated to monitor the traffic of "reverse flows" to extract asymmetric features. Since monitoring rules and useless rules (i.e. flow rules triggered by attack traffic) cost space in the flow table, the flow table management component enables a dynamic way to manage flow rules.

Monitoring rules are generated based on the logic of processing rules, as we discussed in Section III-D. They monitor reverse traffic and help the packet filter component to generate asymmetric features. Each monitoring rule is assigned with an expire time in its timeout field to save the space of flow table (the expire time should be the same with the time of collecting `packet_in` messages in the `packet_in` buffer, which is set to 5 seconds initially).

The management of flow table stems from the multiple flow tables in OpenFlow Specification v1.3 [11]. Specifically, the flow table management uses the first k tables (table 0 to $k-1$) and the last table (table n) as "flow table region", and other tables (table k to $n-1$) as "cache region". Notice that

Algorithm 2 Traffic-Based Filtering (s, F)

Input: s : switch; F : set of filtered flows

Output: $freq$: frequency threshold

- 1: $N \leftarrow \emptyset, t \leftarrow \emptyset, freq = 1, rst = \text{NORMAL}$
 - 2: $P = 0, B = 0, AP = 0, AB = 0$
 - 3: $ASet = \{\text{DROP, MULTIPATH, ...}\}$ // asymmetric flow set
 - 4: `Forward_To_Apps(F)` // process messages
 - 5: $t = \text{Get_Monitored_Traffic}(s)$
 - 6: **for** each $p \in F$ **do**
 - 7: **if** $p \notin ASet$ **then**
 - 8: $(P, B, AP, AB) = \text{Extract_Feature}(t.flow_traff(p))$
 - 9: $rst = \text{Classifier}(P, B, AP, AB)$
 - 10: **else**
 - 11: $(P, B) = \text{Extract_Feature}(t.flow_traff(p))$
 - 12: $rst = \text{Classifier}(P, B)$
 - 13: **end if**
 - 14: **if** $rst = \text{NORMAL}$ **then**
 - 15: $N.add(p)$
 - 16: **end if**
 - 17: $P = 0, B = 0, AP = 0, AB = 0$
 - 18: **end for**
 - 19: `s.flush_cache_region()`
 - 20: `s.del_monitor_flow()`
 - 21: `s.add_flow(N)`
 - 22: $freq = \text{Calculate_New_Frequency}(N)$
 - 23: **return** $freq$
-

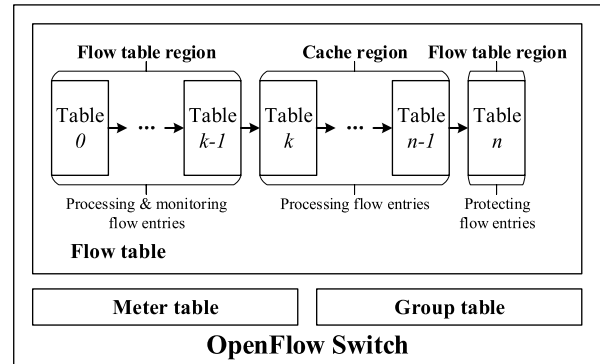


Fig. 12. The flow table is logically separated into flow table region and cache region by the flow table management component.

OpenFlow Specification v1.3 indicates that a flow entry can only direct a packet to a flow table with a bigger flow table number. Therefore, we install processing and monitoring flow rules (flow entries to process normal traffic and monitor reverse traffic) in the first k tables of the flow table region, intercepted processing rules in the cache region (newly generated flow rules to process table-miss traffic), and protecting rules in the last table of the flow table region, as depicted in Fig. 12. The larger size of cache region (larger k) can improve the efficiency of traffic-based filtering, but will use more space of the flow table. The flow table management component sets the value of k based on the free space of the flow table and adjusts it dynamically. Processing flow rules in the cache region and will be flushed after traffic-based filtering to save the space of the flow table.

The flow table management component ensures the timely responses of old benign flows when attacks occur. Since a packet can only be directed to a flow table with a bigger flow table number, old flows will not index cache region, and will be processed efficiently. To activate protecting rules in the last flow table, the default table-miss instructions of all but the last flow table should be set to “Goto Table n ”.

Though OpenFlow Specification v1.3 [9] encourages multiple flow tables, an OpenFlow switch with a single flow table is also allowed. In this scenario, the flow table will not be separated into two regions, and all rules are mixed together in one flow table. Though the efficiency of indexing is affected, flow table management component can still protect the flow table by removing attack flow entries. Processing flow rules which are regarded as normal flows will be kept in the flow table without flushing.

In summary, the flow table management component avoid the flow table being overloaded, which will improve the performance of processing benign packets by allowing legal flow rules installed in the flow table.

VI. NEIGHBOR SWITCH ANALYSIS

The number of neighbor switches will greatly affect the performance of FloodDefender. We first use an average queueing delay model to analyze how to distribute attack traffic, and then analyze how many neighbor switches should be involved in the table-miss engineering.

A. Traffic Distribution

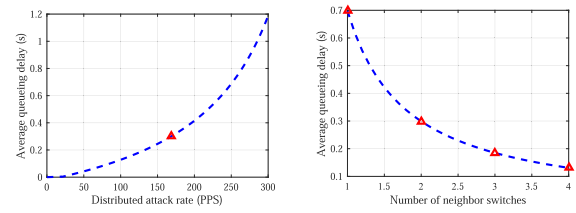
We consider a set of switches $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ involved in the table-miss engineering, and a set of attack traffic rates $\mathcal{A} = \{a_1, a_2, \dots, a_n\}$ distributed to each switch ($\sum_{i=1}^n a_i = a$). For each s_i , let a_{s_i} be its maximum ability to process attack messages without buffering them. Let L_h be the payload of a header information, and L_p be the average payload of an attack packet. For each link between s_i and the controller, let R_i be its maximum bandwidth, and \tilde{R}_i be the allocated bandwidth to process other packets.

We use average queueing delay (D_i) to evaluate the performance on each link. It is not easy to get the formula of D_i , since the calculation is related to the distribution of incoming packets, which is determined by the attacker. Therefore, we use an empirical formula [12] to roughly describe the relationship between D_i and the utilization of this link (ρ_i):

$$D_i = \frac{1}{2\mu} \times \frac{\rho_i}{1 - \rho_i}. \quad (1)$$

In Equation (1), μ is a coefficient of delay, and ρ_i describes link utilization ($\rho_i = \frac{\text{Total Payload}}{\text{Transmission Ability}}$, and $0 \leq \rho_i < 1$). The calculation of ρ_i could be separated into two scenarios: when the incoming packets rate is within the processing ability of s_i ($a_i \leq a_{s_i}$), s_i only sends the header of each attack packet to the controller; otherwise, the buffer of s_i will be overloaded eventually, and s_i needs to send the whole packet. Therefore, ρ_i can be calculated as follows:

$$\rho_i = \begin{cases} \frac{\tilde{R}_i + a_i \times L_h}{R_i}, & a_i \leq a_{s_i} \\ \frac{\tilde{R}_i + a_{s_i} \times L_h + (a_i - a_{s_i}) \times L_p}{R_i}, & \text{else.} \end{cases} \quad (2)$$



(a) Average queueing delay of a switch under different attack rates. (b) Number of neighbor switches involved in the table-miss engineering.²

Fig. 13. Average queueing delay of switches.

Fig. 13-a shows that the average queueing delay goes up quickly when the distributed attack rate increases. The configuration adopts 20PPS (packet per second) a_{s_i} , 750bit L_h , 5Kb L_p , 2Mbps R_i , and 0 \tilde{R} when $\mu = 1$. In this scenario, we could maintain the average queueing delay within 0.3s with less than 168.5PPS distributed attack rate.

We further analyze the scenario with multiple switches. The traffic balancer will distribute attack traffic to each switch. The optimal distributing strategy can be obtained by minimizing the average queueing delays of all packets (D) based on Equation (2):

$$D = \frac{\sum_{i=1}^n (D_i \times a_i)}{a} = \frac{1}{2\mu a} \times \sum_{i=1}^n \frac{\rho_i}{1 - \rho_i} a_i, \quad (3)$$

$$\text{s.t. } \sum_{i=1}^n a_i = a.$$

In Equation (3), ρ_i and a_i could be roughly regard as a linear relationship ($\rho_i = ua_i + v$), since the incoming packets rate is higher than the processing ability of s_i ($a_i > a_{s_i}$) for most cases under SDN-aimed DoS attacks. The sum of ρ_i (normalized attack traffic) could also be regarded as a constant C when n is given ($\sum_{i=1}^n \rho_i = ua + vn = C$). Suppose each switch has the same processing ability ($a_{s_i} = a_s$), maximum bandwidth ($R_i = R$), and allocated bandwidth ($\tilde{R}_i = \tilde{R}$), Equation (3) could be further simplified as follows:

$$D = \frac{1}{2\mu a} \times \sum_{i=1}^n \frac{\rho_i \times \frac{\rho_i - v}{u}}{1 - \rho_i} = k \times \sum_{i=1}^n \frac{\rho_i(\rho_i + v)}{1 - \rho_i}, \quad (4)$$

$$\text{s.t. } \sum_{i=1}^n \rho_i = C.$$

In Equation (4), the positive real number k represents the coefficient of the system ($k = \frac{1}{2\mu a u}$). We introduce the Lagrange multiplier λ , and the objective function of Equation (4) can be constructed as a Lagrange function $\mathcal{L}(\boldsymbol{\rho}, \lambda)$ ($\boldsymbol{\rho} = (\rho_1, \rho_2, \dots, \rho_n)$).

$$\mathcal{L}(\boldsymbol{\rho}, \lambda) = k \times \sum_{i=1}^n \frac{\rho_i(\rho_i + v)}{1 - \rho_i} + \lambda \left(\sum_{i=1}^n \rho_i - C \right). \quad (5)$$

It follows from the saddle point condition that the partial derivatives of $\mathcal{L}(\boldsymbol{\rho}, \lambda)$ with respect to the primal variables

²When $n = 1$ (0 neighbor switch), the victim switch is overloaded, and $\rho > 1$. The average queueing delay will be infinite.

(ρ, λ) have to vanish for optimality.

$$\partial_{\rho_i} \mathcal{L}(\rho, \lambda) = k \frac{-\rho_i^2 + 2\rho_i + v}{(1 - \rho_i)^2} + \lambda = 0 \quad (6)$$

$$\sum_{i=1}^n \rho_i = C. \quad (7)$$

The minimized $\mathcal{L}(\rho, \lambda)$ will be obtained when:

$$\rho_1 = \rho_2 = \dots = \rho_n = C/n. \quad (8)$$

Therefore, the best strategy to minimize D for the whole system is to evenly distribute the attack traffic ($\rho_1 = \rho_2 = \dots = \rho_n = C/n = \rho$).

B. Number of Neighbor Switches

Suppose the traffic balancer could precisely follow the best strategy and distribute attack traffic to each switch evenly. In this scenario, similar to Equation (2), the calculation of ρ is also separated into two scenarios:

$$\rho = \begin{cases} \frac{\tilde{R} + a \times L_h}{nR}, & a \leq na_s \\ \frac{\tilde{R} + na_s \times L_h + (a - na_s) \times L_p}{nR}, & \text{else.} \end{cases} \quad (9)$$

We could find out how many neighbor switches ($n - 1$) should be involved based on Equation (1) and Equation (9). The result is depicted in Fig. 13-b. The configuration adopts 20PPS a_s , 750bit L_h , 5Kb L_p , 2Mbps R , 500PPS a , and 0 \tilde{R} when $\mu = 1$. With 2 neighbor switches ($n = 3$), D can be less than 0.3s and $\rho = 0.38$. D nearly decreases to 0.1s with 4 neighbor switches ($\rho = 0.22$). Generally speaking, FloodDefender can preserve the major functionality with 4 or fewer neighbor switches. When 4 neighbors are involved, each switch (victim switch or neighbor switch) has 1.65Mbps available bandwidth. However, without this protecting scheme, the victim switch has no available bandwidth (the switch becomes dysfunctional).

VII. EXPERIMENT

We first introduce our implementation of FloodDefender system, and then describe the experiment setups in both software and hardware environments. Finally, we discuss the experimental results.

A. Implementation

We implement FloodDefender system, including the detection module and mitigation module. All of them are implemented as applications on RYU controller [13] in Python. Meanwhile, we install RYU controller the on a computer equipped with i7 CPU and 8GB memory, and run a layer 2 learning switch app (l2_learning) which can discover the network topology and provide basic forwarding service (a little modification is made in the last experiment). In the *software environment*, we use Mininet [14] to create virtual OpenFlow switches, and in the *hardware environment*, we use commercial OpenFlow switches, Polaris xSwitch X10-24S2Q [5], to build the test environment, as depicted in Fig. 14. Each hardware switch can store 2000 flow entries, and has 8MB

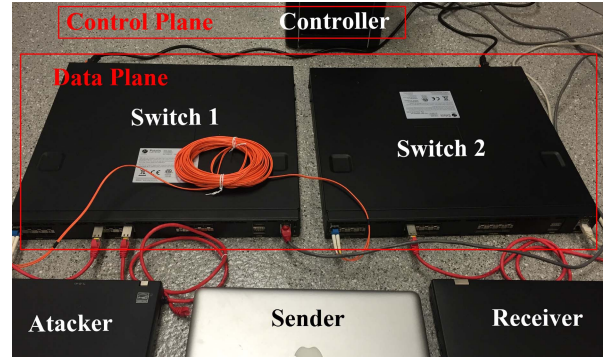


Fig. 14. Hardware environment.

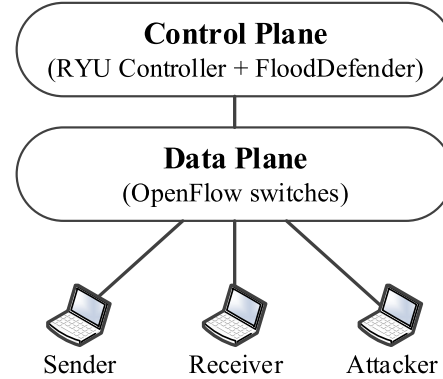


Fig. 15. Test network topology.

buffer memory. We employ three hosts (sender, receiver, and attacker) in our test environment, as depicted in Fig. 15.

To compare FloodDefender with previous work, we launch the SDN-aimed attacks in three scenarios: (i) an OpenFlow network without protecting system, (ii) an OpenFlow network with FloodGuard [7], and (iii) an OpenFlow network with our FloodDefender.

B. Setup

First, we show the importance of applying flow entry frequency in attack detection. We design a software OpenFlow network with 50 hosts and 5 switches, and compare the differences of six features (CPU, controller memory, switch memory, `packet_in` rate, number of flows in low-frequency flows, and average frequency in low-frequency flows) in two scenarios: when SDN-aimed attacks occur and when the network starts up. The attack will use *scapy* to keep flooding TCP packets with randomly forged *eth_src*, *eth_dst*, *tcp_src*, and *tcp_dst* fields under 500PPS attack rate. We also adjust our splitting settings in detection module and compare the performance with FloodGuard. Specifically, we set (0, 2), [2, 10], (10, +∞) for FloodDefender-s1, (0, 100), [100, 1000], (1000, +∞) for FloodDefender-s2, and our original settings (0, 10), [10, 100], (100, +∞) for FloodDefender. The accuracy of attack detection is evaluated by recall rate ($\frac{\text{Identified Attacks}}{\text{Total Attacks}}$) and false-positive rate ($\frac{\text{Startups Regarded as Attacks}}{\text{Total Startups}}$) in 10 times of attacks and 10 times of startups.

Second, we place the sender under the victim switch and test the available bandwidth rate in both software environment (with 4 neighbor switches) and hardware environment (with 1 neighbor switch). The attacker will keep flooding UDP

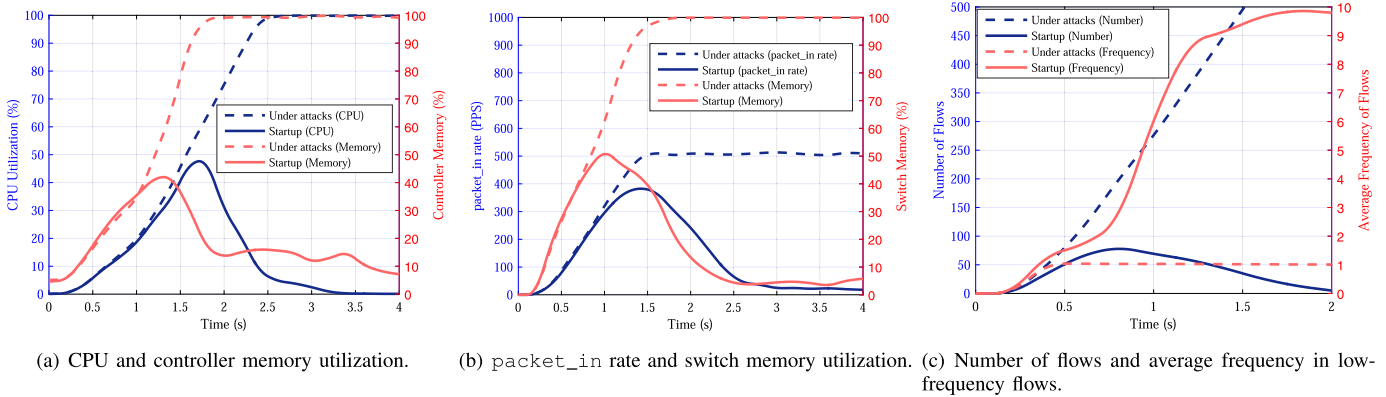


Fig. 16. Different features in attack detection.

packets (with forged *eth_src*, *eth_dst*, *udp_src*, and *udp_dst* fields) under different rates. We use *iperf* to measure the available bandwidth between the sender and receiver, and set the bandwidth threshold to 30% ($\rho = 0.7$) to ensure less than 1.2s average queuing delay.

Third, we place the sender under the each neighbor switch and measure the available bandwidth rate in software environment. We test FloodDefender system under a fully connected network with 5 switches, and FloodDefender will detour attack traffic to 1 to 4 neighbor switches. The UDP attack rate will be 500PPS.

Fourth, we measure the CPU utilization of the controller under UDP-based attacks to the computational resource consumption of the control plane.

Fifth, we compare the flow table utilization of the victim switch under OpenFlow, FloodGuard [7], and FloodDefender. The attacker generates TCP packets with randomly forged sender IP to flood the network and overload the flow table.

Sixth, we evaluate the performance of attack identification. We use recall rate ($\frac{\text{Identified Attack Packets}}{\text{Total Attack Packets}}$) and false-positive rate ($\frac{\text{Normal Packets Regarded as Attack Packets}}{\text{Total Normal Packets}}$) to measure the performance of two-phase filter under different attack rates.

Seventh, we measure the time delay of normal traffic under OpenFlow, FloodGuard [7], and FloodDefender. Here we measure the delay of all kinds of protocols under UDP-based DoS attacks. Since FloodGuard utilizes rate control to handle *packet_in* messages, its performance can be greatly affected by the attack rate. In this experiment, we use two different attack rate, 100PPS and 500PPS to evaluate the time delay. The maximum time delay usually occurs when the first packet in each flow arrives.

Finally, we compare the packet loss rate of new TCP flows in OpenFlow, FloodGuard [7], and FloodDefender under TCP-based DoS attacks. To generate new flows efficiently, we modify *l2_learning* app, and use *eth_src* && *tcp_src* (if *tcp_src* is available) instead of *eth_src* as the match field to generate flow rules. The first handshake packet of a new TCP connection is regarded as a new flow (table-miss), and triggers *packet_in* message. The packet loss rate shows the effectiveness of each system in processing new flows.

C. Experimental Result

Attack Detection: The CPU and controller memory utilization rates under attacks and network startups are depicted in Fig. 16-a; and *packet_in* rate and switch memory utilization

TABLE I

PERFORMANCE OF ATTACK DETECTION. *a/b* IN RECALL RATE INDICATES *IDENTIFIED ATTACKS/TOTAL ATTACKS*, AND IN FALSE-POSITIVE RATE INDICATES *STARTUPS REGARDED AS ATTACKS/TOTAL STARTUPS*

	Recall Rate	False-positive Rate
FloodGuard	100% (10/10)	10% (1/10)
FloodDefender-s1	100% (10/10)	30% (3/10)
FloodDefender-s2	100% (10/10)	10% (1/10)
FloodDefender	100% (10/10)	0% (0/10)

in Fig. 16-b. The switch memory utilization rate may be not very precise due to the communication delay between the switch and controller. All of them in the two scenarios are very similar before the 1 seconds. Even though they become much different afterwards, the attack detection may lead to false-alerts by only considering these features (they can be more similar when measured in larger scale networks). The number of flows and average frequency in low-frequency flows are depicted in Fig. 16-c. Though the communication delay may also affect the accuracy of these curves, we can find that there will be many differences after 0.4s in the two scenarios. The attack detection can precisely identify SDN-aimed attacks and reduce false-alerts by adopting these two features to increase sensitivity of the defense system.

The performance of attack detection is presented in Table I. Both FloodGuard [7] (4 features detection) and FloodDefender (6 features detection) can precisely identify SDN-aimed DoS attacks. However, as we analyzed before, FloodGuard can lead to some false-alerts in some cases due to the similarity in network startups and under attacks. By properly setting flow entry frequencies ((0, 10), [10, 100], (100, +∞) splitting), FloodDefender can outperform FloodGuard. Furthermore, since we have a 100% recall rate under different settings, the flow entry frequency features mainly contribute to reducing false-alerts.

Victim Switch Bandwidth: The results in software and hardware environments are depicted in Fig. 17 and Fig. 18. In this test, we do not show the result from FloodGuard [7], because it takes a designated extra link to a specific device, the data plane cache. The maximum bandwidth is 1.92Gbps in software environment, and 9.3Mbps in hardware environment. On one hand, the bandwidth in OpenFlow network without protecting systems is almost exhausted, only 3% left in software environment and 24% left in hardware environment. On the other hand, FloodDefender maintains the major functionality of the network, and saves 70% software bandwidth and nearly

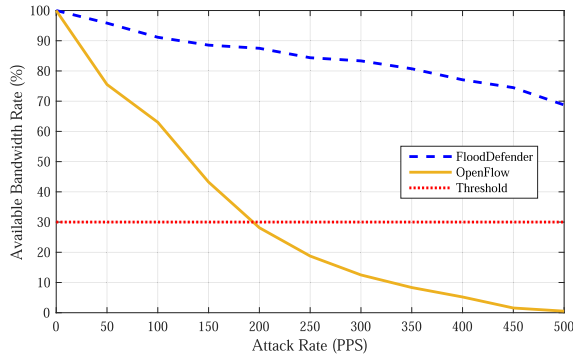


Fig. 17. Victim-controller bandwidth in software environment.

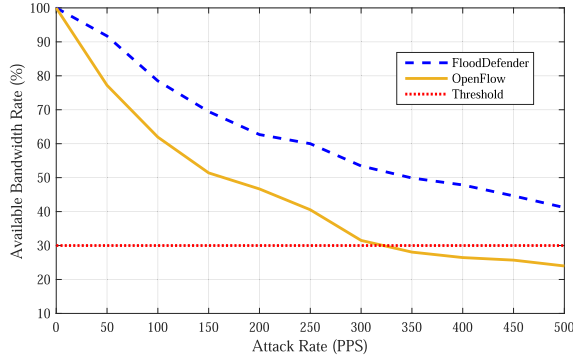


Fig. 18. Victim-controller bandwidth in hardware environment.

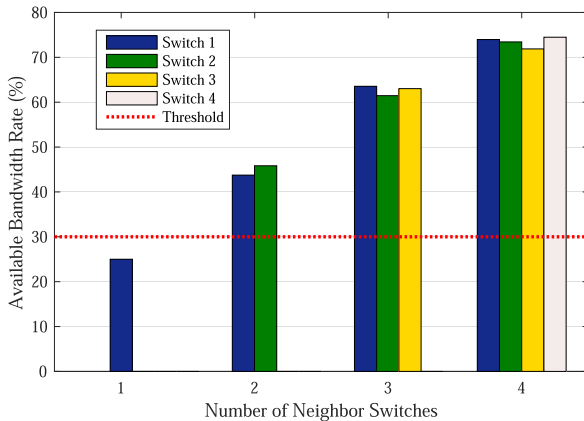


Fig. 19. Available bandwidth rates of neighbor switches.

20% hardware bandwidth (the performance can be improved by involving more neighbor switches).

Neighbor Switch Bandwidth: The attack traffic will affect the bandwidths of neighbor switches in FloodDefender, as depicted in Fig. 19. The bandwidths of neighbor switches will fluctuate when the mitigation module is activated first but remains steady after the traffic balancer component collects enough information (within 30s in our experiment). When only one neighbor switch is involved, the available bandwidth rate is within 30% (FloodDefender will avoid this scenario by involving more switches, but we block this function in this experiment). The network becomes functional with more neighbor switches. Specifically, the SDN-aimed DoS attacks can hardly affect the network when 4 neighbor switches are involved. Besides, the result also shows that the traffic balancer component can efficiently balance the traffic among neighbor switches.

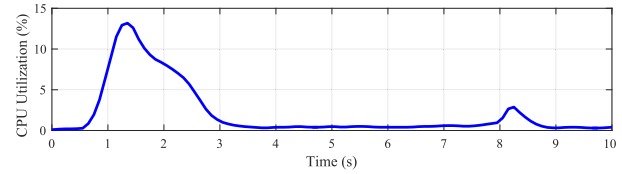


Fig. 20. CPU utilization under UDP-based attacks.

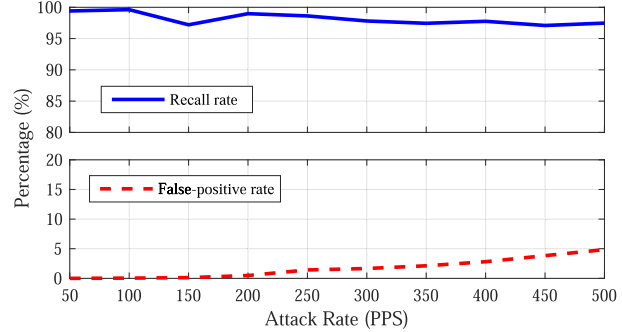


Fig. 21. Attack detection performance: recall rate and false-positive rate.

Computational Resource Consumption: We can get the computational resource protection performance of FloodDefender in Fig. 20. When attacks occur, the CPU utilization quickly reaches a peak (around 14%) in less than 1.5s. Then it goes down slowly because the table-miss engineering and `packet_in` buffer start to detour and store attack traffic. After about 1.5s, the CPU utilization remains steady. At this stage, the `packet_in` buffer efficiently stores the `packet_in` messages, and only consumes about 0.5% CPU utilization. In about 8s, there is a little spur: the CPU utilization reaches about 3%, and quickly goes down in 1s. This is caused by the two-phase filtering in packet filter. The result shows that FloodDefender can efficiently save the computational resources of the control plane, and the overhead of the packet filter is very little.

Flow Table Utilization: The flow table utilization rate in depicted in Table II. We can find that both FloodGuard [7] and FloodDefender will not incur overload into the network when there is no attack. Though FloodGuard uses rate control to protect the victim switch when attacks occur, the attack traffic still consumes about 30% flow table space. The flow table utilization rate fluctuates in FloodDefender, since monitoring rules will be expired and the flow table management component will flush cache region periodically. FloodDefender consumes less than 15% flow table space. Its performance is much better than FloodGuard.

Attack Identification: The attack detection performance of the two-phase filter is depicted in Fig. 21. We can find that the false-positive rate goes up with attack rate. It is because in a time interval, the frequency of the same flow will be higher with a higher attack rate. Therefore, the frequency-based filtering will use a bigger threshold to filter out attack traffic, and sacrifice some benign traffic. Though more attack packets are classified as normal flow when attack rate increases, the percentage of these packets remains the same, and the recall rate is more stable. Generally speaking, the two-phase filtering can precisely identify more than 96% attack traffic with less than 5% false-positive rate.

Time Delay: The time delays of normal flows are depicted in Table III and Table IV. Since FloodGuard [7] utilizes

TABLE II
FLOW TABLE UTILIZATION UNDER TCP-BASED ATTACKS

	OpenFlow	FloodGuard	FloodDefender
No attack	4%	4%	4%
Under attacks	100%	34%	6% ~ 19%

TABLE III
TIME DELAY OF NORMAL FLOWS UNDER 100PPS UDP-BASED ATTACKS

	OpenFlow	FloodGuard	FloodDefender
Max Delay	timeout	timeout	4913ms
Min Delay	10.9ms	0.3ms	0.3ms
Average Delay	1843ms	15.1ms	17.5ms

TABLE IV
TIME DELAY OF NORMAL FLOWS UNDER 500PPS UDP-BASED ATTACKS

	OpenFlow	FloodGuard	FloodDefender
Max Delay	timeout	timeout	4891ms
Min Delay	10.7ms	0.4ms	0.3ms
Average Delay	2038ms	29.2ms	18.7ms

rate control to save the computational resources, the delay of normal flows increases with the attack rate. When the attack rate is low (100PPS), the average time delay of FloodGuard is better than that of FloodDefender; but when the attack rate increases to 500PPS, FloodDefender has shorter delay than FloodGuard. The maximum time delays in both FloodGuard and OpenFlow become infinite (timeout), which is different from the results presented in [7]. Besides the attack rate, another reason is that [7] only measures the delay of TCP packets under UDP-based DoS attacks. In our experiment, we also measure the delay of UDP packets, and find out many of them are lost in FloodGuard. Though these UDP packets in FloodDefender suffer from long time delay, they are processed and received eventually. Both FloodGuard and FloodDefender are superior to OpenFlow in average and minimum time delays, and the performance of FloodDefender is better than that of FloodGuard when attack rate is high.

Packet Loss Rate: Finally, we compare the packet loss rate of new TCP flows under TCP-based DoS attacks. The result is depicted in Fig. 22. In this scenario, both FloodGuard and OpenFlow do not filter out attack traffic, and inevitably sacrifice benign TCP packets. We can find that FloodGuard is even worse than OpenFlow. It is because the round-robin scheduling in the data plane cache treats each protocol evenly, and only picks the header packet of each protocol. Therefore, it has a very low probability to pick the benign TCP packet (even lower than that of OpenFlow, which treats each packet evenly). The performance of FloodDefender is much better, the packet filter component can filter out attack traffic both efficiently and precisely, and the packet loss rate of new TCP flows remains within 5%.

VIII. RELATED WORK

The security of SDN has become a hot research area ever since it was proposed. On one hand, the attributes of centralized control and programmability in SDN can be exploited to enhance network security with a highly reactive security system [17]–[25]. On the other hand, the same centralized structure is considered vulnerable, which can cause severe network security problems [6]–[8], [15], [16], [26]–[35].

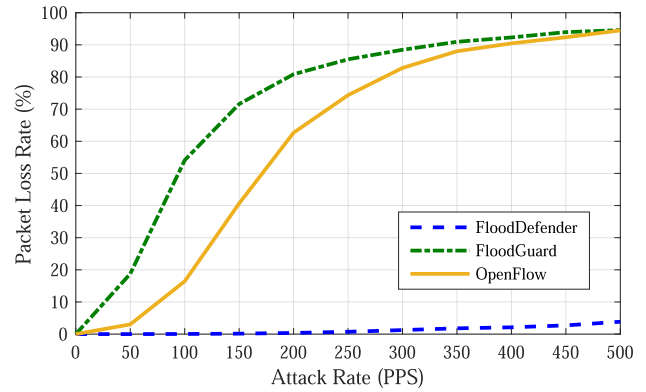


Fig. 22. Packet loss rate of new TCP flows under TCP-based DoS attacks.

SDN-Supported Security: SDN-supported security uses new techniques in SDN to solve traditional network security challenges. FloodGuard [23] introduces a comprehensive framework to facilitate not only accurate detection but also effective resolution of firewall policy violations in dynamic OpenFlow-based networks. Hu *et al.* combine SDN with inference techniques to derive a hybrid network monitoring scheme, which can strike a balance between measurement overhead and accuracy [17]. Xu *et al.* introduce new DDoS detection methods based on the flow monitoring capability [18]. These methods can balance the coverage and granularity, and quickly locate potential victims and attackers. Taylor *et al.* introduce a contextual and flow-based access control to improve enterprise security with flow-based monitoring [19]. It provides both detailed host-based context and fine-grained control of network flows by shifting the SDN agent functionality from the network infrastructures into the end-hosts.

SDN-Self Security: SDN-self security aims to identify new attacks against SDN and enhance the security of SDN-enabled devices. The SDN-aimed DoS attacks utilize the reactions of table-miss packets in SDN to exhaust control-data plane bandwidth [6], [8]. To mitigate the DoS attacks, AvantGuard uses an SYN proxy based module to verify the legality of each flow based on TCP handshake of each new packet [6]. Another approach, FloodGuard, mitigates the DoS attacks by installing proactive flow rules and sending table-miss packets to a specific data plane cache module [7]. FloodGuard breaks the protocol limitation in AvantGuard, but may suffer from long delay and high packet loss rate for some flows. FlowKeeper [4] adopts two components, traffic agent and global view agent, to filter table-miss traffic on both data plane and control plane. However, there will be a high delay for benign flows under severe attacks. Besides, all approaches use additional devices, and ignore the importance of attack detection. Another attack is poisoning the network visibility of the control plane by unauthorized LLDP packets [15]. TopoGuard uses the incoming port information to verify LLDP packets against network topology poisoning attacks [15]. To avoid malicious apps on the control plane, SDNShield expresses and enforces the minimum required privileges to individual apps [16].

IX. DISCUSSION

We have shown that FloodDefender can protect OpenFlow networks against SDN-aimed DoS attacks. However, there still remain hurdles in our current prototype implementation. In this section, we discuss these limitations as well as possible improvements for FloodDefender.

TABLE V
NEW ATTACKS AND COUNTERMEASURES IN SDN

SDN Attacks	DoS				Topology Poisoning	Malicious Apps
Countermeasures	AvantGuard [6]	FloodGuard [7]	FlowKeeper [4]	FloodDefender	TopoGuard [15]	SDNShield [16]
Controller Application		✓	✓	✓	✓	✓
Data Plane Extension	✓					
Additional Device		✓	✓			
Support Protocols	TCP	All	All	All	ARP, LLDP	-
Benign Traffic Delay	Medium	Mostly low, some high	High	Low	Low	Low

False-Positive Rate: In Fig. 21, we show that the false-positive rate increases with attack rate since the frequency of each attack flow will be higher with a higher attack rate. FloodDefender may drop benign packets when attack rate is high. However, for most benign traffic, retransmission schemes (e.g. Go-Back-N and selectively repeat for TCP connections) are used to provide reliable communications. When adopting these retransmission schemes, the frequency of the dropped benign flows will increase, which makes a higher probability of these flows regarded as benign ones. Furthermore, we use SVM to show the feasibility of FloodDefender. The classifier can be improved with more training data, or a more advanced classification algorithm (e.g. deep learning) to reduce false-positives.

Switch Buffer: In our packet filtering design, we simply filter out (ignore) illegal `packet_in` messages. However, the corresponding table-miss packets may be stored in the switch's buffer. Even though OpenFlow switches can still operate even when their buffer is full and buffered packets can automatically be expired after some time, the throughput can be downgraded due to encapsulating whole packets when the buffer is full. To save the switch's buffer, an improvement could allow the packet filtering to send `packet_out` messages to drop buffered packets when `packet_in` messages (only containing the headers) are classified as illegal. However, such operations can increase the bandwidth and computational resources consumption. Considering buffered packets can automatically be expired after some time, the packet filtering can only send a part of `packet_out` messages to balance the cost and improvement.

Low-Rate Attacks: Attackers can send packets at a low rate to avoid being detected. In this scenario, even though the attacker cannot dysfunctional switches and control plane, the low-rate attacks can still consume much network resource. Here we discuss a possible countermeasure to detect low-rate attacks with host behaviors.

One observation is that a benign host normally sends DHCP packets (to a DHCP server) and ARP packets first when it connects to SDN, and sends `PORT_DOWN` signals when it is offline. Besides, a forged source is mostly used in SDN-aimed DoS attacks to hide the identity of an attacker. Therefore, the detection module can first monitor DHCP and ARP packets as well as `PORT_DOWN` signals. When abnormal packets with non-existing sources (no DHCP/ARP packet is received from the source before), or abnormal packets with existing sources (`in_port` is different from the previous one and no `PORT_DOWN` signal is received from the previous port), detection module can alert the mitigation module even when the attack rate is low (we also suggest to set a threshold of abnormal packet rate to avoid false-alerts).

X. CONCLUSION

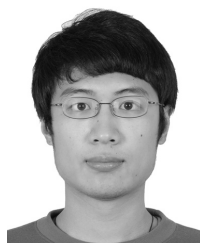
SDN-aimed DoS attacks can paralyze OpenFlow networks by exhausting the bandwidth, computational resources, and

flow table space. We propose FloodDefender, a scalable and protocol-independent system to protect OpenFlow networks against SDN-aimed DoS attacks based on new features in attack detection, and three novel techniques in attack mitigation: table-miss engineering, packet filter, and flow table management. FloodDefender can precisely detect SDN-aimed attacks, efficiently process table-miss packets, as well as precisely identify attack traffic. We use a queueing delay model to analyze how many neighbor switches should be used in the table-miss engineering, and implement a prototype to evaluate the performance of FloodDefender in both software and hardware environments. Compared with previous work, FloodDefender reduces the false-alerts in attack detection, significantly improves the flow table utilization, time delay, and packet loss rate, and is more scalable and easier to deploy without employing additional devices.

REFERENCES

- [1] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
- [2] OpenNetworking. *OpenFlow*. [Online]. Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-switch-v1.5.0.noipr.pdf>
- [3] Q. Yan and F. R. Yu, "Distributed denial of service attacks in software-defined networking with cloud computing," *IEEE Commun. Mag.*, vol. 53, no. 4, pp. 52–59, Apr. 2015.
- [4] S. Gao, Z. Li, B. Xiao, and G. Wei, "Security threats in the data plane of software-defined networks," *IEEE Netw.*, vol. 32, no. 4, pp. 108–113, Jul. 2018.
- [5] Polaris networks Co. Ltd. *Polaris X Switch*. [Online]. Available: <http://www.polarisnfv.com/en/product/html/?80.html>
- [6] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 413–424.
- [7] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS attack prevention extension in software-defined networks," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 239–250.
- [8] S. Song, S. Hong, X. Guan, B.-Y. Choi, and C. Choi, "NEOD: Network embedded on-line disaster management framework for software defined networking," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2013, pp. 492–498.
- [9] OpenNetworking. *OpenFlow Switch Specification V1.4.0*. [Online]. Available: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf>
- [10] V. N. Vapnik and V. Vapnik, "Statistical learning theory," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 988–999, 1998.
- [11] OpenNetworking. *OpenFlow Switch Specification V1.3.0*. [Online]. Available: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf>
- [12] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 4, pp. 1–12, Aug. 1989.
- [13] RYU SDN Framework Community. *RYU Controller*. [Online]. Available: <https://osrg.github.io/ryu/>
- [14] W. Buck, A. Grammer, M. Moerike, and B. Muehlemeier, "MININET," *Das Rechenzentrum*, vol. 2, no. 3, pp. 137–141, Jan. 1979.
- [15] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.
- [16] X. Wen *et al.*, "SDNShield: Reconciling configurable application permissions for SDN app markets," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2016, pp. 121–132.

- [17] Z. Hu and J. Luo, "Cracking network monitoring in DCNs with SDN," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 199–207.
- [18] Y. Xu and Y. Liu, "DDoS attack detection under SDN context," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.
- [19] C. R. Taylor, D. C. MacFarland, D. R. Smestad, and C. A. Shue, "Contextual, flow-based access control with scalable host-based SDN techniques," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.
- [20] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Enabling practical software-defined network security applications with OFX," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.
- [21] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. (NDSS)*, 2013, pp. 1–16.
- [22] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu, "Towards SDN-defined programmable BYOD (bring your own device) security," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.
- [23] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: Building robust firewalls for software-defined networks," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 97–102.
- [24] J. Li, S. Berg, M. Zhang, P. Reiher, and T. Wei, "Drawbridge: Software-defined DDoS-resistant traffic engineering," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 591–592, Aug. 2014.
- [25] S. Gao, Z. Li, Y. Yao, B. Xiao, S. Guo, and Y. Yang, "Software-defined firewall: Enabling malware traffic detection and programmable security control," in *Proc. Asia Conf. Comput. Commun. Secur. (ASIACCS)*. New York, NY, USA: ACM, 2018, pp. 413–424.
- [26] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect SDN forwarding with RuleScope," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.
- [27] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust SDN control plane for transactional network updates," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 190–198.
- [28] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software defined network control layer," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.
- [29] H. Cui, G. O. Karame, F. Klaedtke, and R. Bifulco, "On the fingerprinting of software-defined networks," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 10, pp. 2160–2173, Oct. 2016.
- [30] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2015, pp. 8–11.
- [31] R. Kandoi and M. Antikainen, "Denial-of-service attacks in OpenFlow SDN networks," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2015, pp. 1322–1326.
- [32] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran, "LineSwitch: Tackling control plane saturation attacks in software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1206–1219, Apr. 2017.
- [33] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN control plane," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*. Berkeley, CA, USA: USENIX, 2017, pp. 451–468.
- [34] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*. Berkeley, CA, USA: USENIX, 2017, pp. 415–432.
- [35] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "FloodDefender: Protecting data and control plane resources under SDN-aimed DoS attacks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.



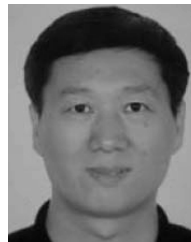
Shang Gao received the B.S. degree from Hangzhou Dianzi University, China, in 2010, the M.E. degree from Southeast University, China, in 2014, and the Ph.D. degree from the Hong Kong Polytechnic University, Hong Kong, in 2019. He is currently a Researcher with Microsoft, China. His research interests include information security, network security, software-defined networks, blockchain security, and applied cryptography.



Zhe Peng received the Ph.D. degree in computer science from The Hong Kong Polytechnic University in 2018. He is currently a Research Assistant Professor with the Department of Computer Science, Hong Kong Baptist University. He publishes regularly at top venues of mobile computing and networking, such as TMC, TON, TASE, CCS, and INFOCOM. His research interests include blockchain system, mobile computing, and data security and privacy.



Bin Xiao (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in electronics engineering from Fudan University, China, and the Ph.D. degree in computer science from The University of Texas at Dallas, USA. He is currently an Associate Professor with the Department of Computing, The Hong Kong Polytechnic University. He has published more than 100 technical articles in international top journals and conferences. His research interests include distributed wireless systems, network security, software-defined networks (SDN), and blockchain. He is an ACM member. He is a recipient of the Best Paper Award of the international conference IEEE/IFIP EUC-2011. He is currently an Associate Editor of the *Journal of Parallel and Distributed Computing (JPDC)* and *Security and Communication Networks (SCN)*.



Aiqun Hu received the B.Sc. (Eng.), M.Eng.Sc., and Ph.D. degrees from Southeast University in 1987, 1990, and 1993 respectively. He was invited as a Post-Doctoral Research Fellow with The University of Hong Kong from 1997 to 1998 and a TCT Fellow with Nanyang Technological University in 2006. He is currently a Professor with the School of Cyber Science and Engineering, Southeast University. He has published two books and over 100 technical articles in wireless communications field. His research interests include data transmission and secure communication technology.



Yubo Song received the B.Sc. degree in electric engineering, the M.Sc. degree in communication and information system, and the Ph.D. degree in signal and information processing from Southeast University, China. He joined the School of Cyber Science and Engineering, Southeast University, as a Lecture after graduation. He is currently an Associate Professor and the Vice Director of the School of Cyber Science and Engineering, Southeast University. He has published more than 30 technical articles in international journals and conferences. His research interests include mobile security, wireless network security, and security protocol analysis.



Kui Ren (Fellow, IEEE) received the B.E. and M.E. degrees from Zhejiang University, Hangzhou, China, in 1998 and 2001, respectively, and the Ph.D. degree from the Worcester Polytechnic Institute, Worcester, MA, USA, in 2007. He is currently a Professor of computer science and the Director of the Institute of Cyberspace Research, Zhejiang University. His research interests include spans cloud and outsourcing security, wireless and wearable systems security, and mobile sensing and crowdsourcing. He has published 200 peer-review journal articles and conference papers. His research has been supported by NSF, DoE, AFRL, MSR, and Amazon. He is a Distinguished Lecturer of the IEEE, an ACM member, and a past Board Member of the Internet Privacy Task Force and the State of Illinois. He was a recipient of the SEAS Senior Researcher of the Year in 2015, the Sigma Xi/IIT Research Excellence Award in 2012, and the NSF CAREER Award in 2011. He received several best paper awards including the IEEE ICNP 2011. He currently serves as an Associate Editor for the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE WIRELESS COMMUNICATIONS, the IEEE INTERNET OF THINGS JOURNAL, and the IEEE TRANSACTIONS ON SMART GRID.