FloodDefender: Protecting Data and Control Plane Resources under SDN-aimed DoS Attacks

GAO Shang*, PENG Zhe*, XIAO Bin*, HU Aiqun†, REN Kui‡
*Department of Computing, The Hong Kong Polytechnic University
†School of Information Science and Engineering, Southeast University
‡Department of Computer Science and Engineering, University at Buffalo, State University of New York
{cssgao, cszpeng, csbxiao}@comp.polyu.edu.hk, {aqhu}@seu.edu.cn, {kuiren}@buffalo.edu

***Abstract*—The separated control and data planes in software-defined networking (SDN) with high programmability introduce a more flexible way to manage and control network traffic. However, SDN will experience long packet delay and high packet loss rate when the communication link between two planes is jammed by SDN-aimed DoS attacks with massive table-miss packets. In this paper, we propose FloodDefender, an efficient and protocol-independent defense framework for SDN/OpenFlow networks to mitigate DoS attacks. It stands between the controller platform and other controller apps, and can protect both the data and control plane resources by leveraging three new techniques: table-miss engineering to prevent the communication bandwidth from being exhausted; packet filter to identify 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. All designs of FloodDefender conform to the OpenFlow policy, requiring no additional devices. We implement a prototype of FloodDefender and evaluate its performance in both software and hardware environments. Experimental results show that FloodDefender can efficiently mitigate the SDN-aimed DoS attacks, incurring less than 0.5% CPU computation to handle attack traffic, only 18ms packet delay and 5% packet loss rate under attacks.**

I. Introduction

Software-defined networking (SDN) is a new network paradigm that separates the control and data planes in a network [1]. The control plane of SDN dictates the whole network behavior. This logical centralization introduces a simpler and more flexible way to manage and control network traffic by a “southbound” protocol (OpenFlow). OpenFlow protocol allows the control plane to install flow rules on the data plane. The data plane will follow these flow rules to handle network flows. When a new table-miss flow comes, it does not match any existing flow rules. The data plane will send a `packet_in` message to the control plane for instructions.

The limited communication bandwidth between the control and data planes could be a bottleneck of the whole network, and lead to security problems. Today’s commercial OpenFlow switches [2] only support cable connection to the controller. The practical connection bandwidth is tested to be less than 10Mbps [3], [4]. An attacker can leverage this costly communication to launch SDN-aimed DoS attacks (data-to-control plane saturation attacks) [3], [5]. Specifically, the attacker can first generate table-miss packets by randomly forging some or all fields, making them hard to match any existing flow rules on a victim switch. Then, he can launch SDN-aimed DoS attacks to flood the network by sending a large amount of table-miss packets. These table-miss packets will trigger massive `packet_in` messages from the switch to the controller, and consume their communication bandwidth, CPU computation, and memory in both control and data planes.

To protect OpenFlow networks against the SDN-aimed DoS attacks, we face the following two challenges:

- How to efficiently handle table-miss packets while maintaining short packet delay, low packet loss rate and normal packet forwarding operation?
- How to precisely distinguish attack traffic from benign traffic without straining computational resources?

These two challenges are not easy to solve. For the first challenge to mitigate attack traffic, we cannot simply drop table-miss packets because all new flows from benign hosts will be dropped as well. We should find a way to let the control plane receive these `packet_in` messages without consuming much bandwidth between the controller and victim switch. Since some table-miss packets are generated by benign hosts, we have the second challenge to precisely identify attack traffic and filter out them accordingly. To deal with these two challenges, AvantGuard [3] introduces a connection migration module on the data plane to identify attack traffic by verifying the TCP handshake of each new flow. To handle other flow traffic, e.g., UDP and ICMP traffic, FloodGuard [4] adopts proactive flow rules to forward table-miss packets to the data plane cache. However, both approaches need to use additional devices to accommodate table-miss traffic, which are not compatible to the OpenFlow protocol.

In this paper, we propose FloodDefender, a scalable and protocol-independent defense system to protect OpenFlow networks against SDN-aimed DoS attacks. FloodDefender stands between the controller platform and other controller apps, and is protocol-independent against different types of attack traffic (e.g. TCP-based attacks or UDP-based attacks). All designs conform to the OpenFlow policy and need no additional devices, which makes FloodDefender scalable. When attacks occur, FloodDefender detours table-miss packets to neighbor switches with wildcard flow rules to protect the communication link from being jammed, filters out attack packets from the received `packet_in` messages to save the computational resources, and 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.

To defend against SDN-aimed DoS attacks, our main technical contributions are as follows:

- *Novel Techniques.* We propose three new techniques: table-miss engineering, packet filter, and flow rule management. These techniques designed in SDN can mitigate the bandwidth jamming, reduce computational resource consumption, and save the memory space of OpenFlow switches.
- *Framework Design.* Based on the mentioned novel techniques, we design FloodDefender to protect OpenFlow networks against variant attack traffic. FloodDefender has four modules: attack detection, table-miss engineering, packet filter, and flow rule management. All modules conform to the OpenFlow policy and no additional specific devices are needed.
- *Theoretical Analysis.* We use an average queueing delay model to analyze how many neighbor switches should be involved in the table-miss engineering. The analytical result shows that FloodDefender can keep the average delay of each link within 0.3s with 3 neighbor switches.
- *Performance Evaluation.* We evaluate the performance of FloodDefender in both software and hardware environments. Experimental results show that FloodDefender can save more than 70% software bandwidth and 20% hardware bandwidth, and consume only 0.5% CPU computation to handle attack traffic. Meanwhile, it precisely identifies more than 96% attack traffic, and incurs only 18ms delay and 5% packet loss rate for normal 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 detailed designs of FloodDefender system. Section IV analyzes how many neighbor switches should be involved theoretically. The implementation and experimental evaluation of FloodDefender are shown in Section V. We summarize the related work in Section VI. Finally, we conclude this paper in Section VII.

II. Problem Statement

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. After receiving incoming packets, an OpenFlow switch processes them based on the flow rules in its flow table. When a table-miss occurs, the OpenFlow switch sends a `packet_in` message which contains the packet header to the controller. The controller then decides how to process the new packet and responses with action and flow rule(s). 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.

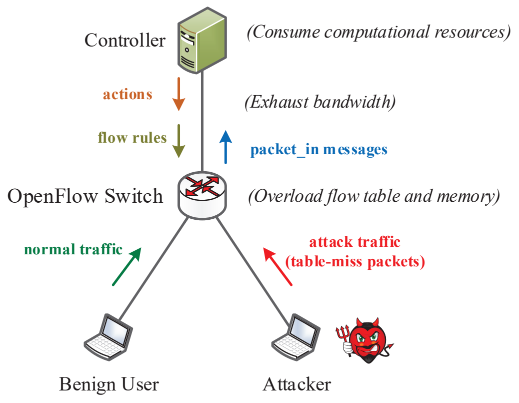
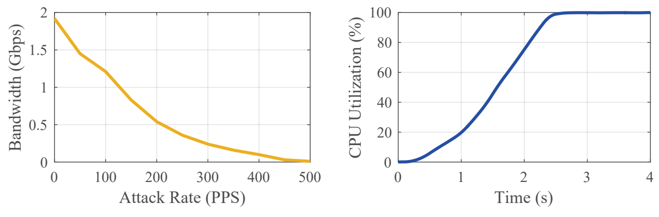


Fig. 1. SDN-aimed DoS attacks in OpenFlow networks.



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

Fig. 2. Bandwidth and computational resource consumption under SDN-aimed DoS attacks. The data are collected from our experiments.

B. Adversary Model

The reactive flow installation approach of OpenFlow networks could be leveraged by an adversary. An attacker first randomly forges some or all fields of each packet, making it hard to match any existing flow rules in a switch. Then, the attacker sends massive table-miss traffic mixed with normal traffic to the OpenFlow switch and launches SDN-aimed DoS attacks. To process each table-miss packet, the victim switch has to buffer it and send out `packet_in` message with its header, as depicted in Fig. 1. Even worse, the OpenFlow Specification v1.4 requires that the `packet_in` message should contain the whole packet when the memory of a switch is full. This feature could be further exploited by the attacker to flood the network with less resources.

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 computational resources by processing `packet_in` messages, as depicted in Fig. 2. The result is much worse when the memory of the switch is full. The throughput of both packet forwarding and packet processing will be significantly degraded.

C. Problem and Challenge

The problem studied in this paper is how to mitigate the SDN-aimed DoS attacks in OpenFlow networks. 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, it should distinguish attack traffic from benign traffic 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). Second, the defense system should be scalable and conform to the OpenFlow policy without additional devices.

III. System Design

A. FloodDefender Architecture

FloodDefender strands between the controller platform and other controller apps, as depicted in Fig. 3. It consists of four functional modules: attack detection, table-miss engineering, packet filter, and flow rule management. When no attacks are detected, FloodDefender forwards the `packet_in` messages, actions, and flow rules between the controller platform and controller apps. When attacks occur, FloodDefender filters `packet_in` messages and forwards them to the apps through the packet filter module, and manages the flow rule installation through the flow rule management module.

Initially, the attack detection module monitors the status of the network, and other modules remain idle. When DoS attacks are detected, the other three modules are activated for attack mitigation in the following six steps:

1) The attack detection module identifies victim's neighbor switches that directly connect to the controller. The flow rule management module logically separates the flow table into flow table region and cache region;
2) The table-miss engineering module installs protecting rules to detour some table-miss packets to the neighbor switches. When these neighbor switches receive the detoured table-miss packets, they will send `packet_in` messages to the controller;
3) When the controller receives `packet_in` messages, the packet filter stores and roughly filters out attack traffic from these messages. The filtered traffic will then be delivered to the apps;
4) The apps process these packets and decide actions and flow rules. Actions will be sent to the switches directly, but the flow rules will be intercepted by the flow rule management module;
5) The flow rule management module decides the monitoring rules based on intercepted processing rules. Intercepted processing rules will be installed on the cache region, and monitoring rules on the flow table region;
6) The packet filter module uses the traffic information of these intercepted and monitoring rules to precisely identify normal traffic. If one processing flow entry is regarded legal, it will be moved to the flow table region. Monitoring rules and cache region will then be flushed to save the space of flow table.

B. Attack Detection Module

The attack detection module continues monitoring the network status (`packet_in` message rate, memory, and CPU) for attack detection, as discussed in [4]. When attacks occur, it triggers other modules into the active state. The attack 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, the attack detection module stops other modules.

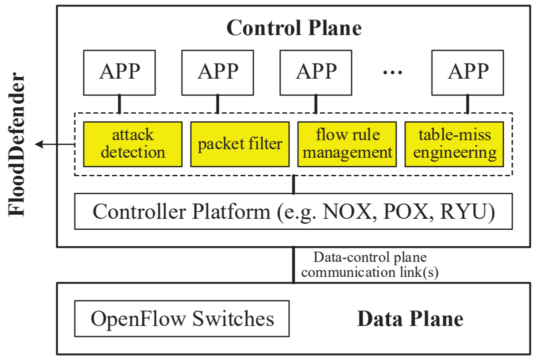


Fig. 3. The architecture of FloodDefender.

C. Table-miss Engineering Module

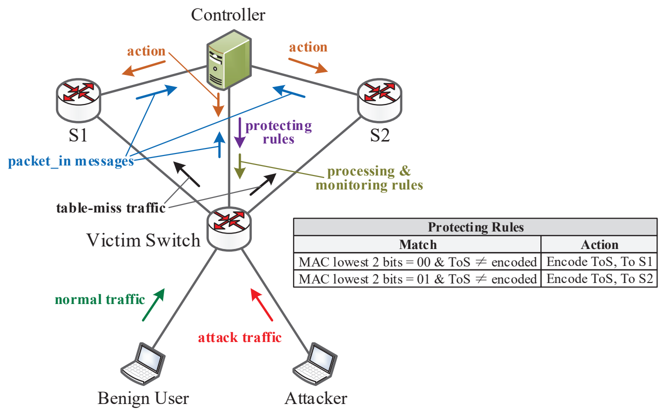
The table-miss engineering module works when the SDN-aimed DoS attacks are detected. In SDN-aimed DoS attacks, massive table-miss packets will be triggered to exhaust the available bandwidth between the controller and a victim switch. Therefore, the table-miss engineering module offloads some table-miss packets to neighbor switches to save the bandwidth of the victim switch. Specifically, the table-miss engineering module issues protecting rules to forward some table-miss traffic to neighbor switches.

Protecting rules are wildcard flow entries with the lowest priority to split the table-miss traffic into several parts to 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. Each victim switch maintains a different set of neighbor switches.

There are two challenges in the design of protecting rules: INPORT loss, 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



Protecting Rules	
Match	Action
MAC lowest 2 bits = 00 & ToS $\neq$ encoded	Encode ToS, To S1
MAC lowest 2 bits = 01 & ToS $\neq$ encoded	Encode ToS, To S2

Fig. 4. Detouring table-miss traffic to neighbor switches when attacks occur.

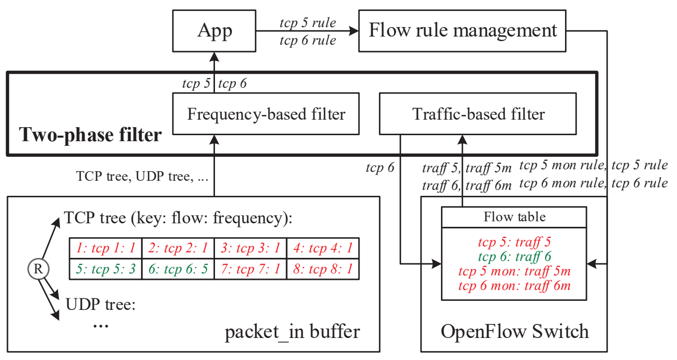


Fig. 5. Two-phase filtering design in the two-phase filter component.

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. 4. Furthermore, the encoded ToS field also allows us to distinguish normal packets with detoured packets. The identified detoured packets will be associated with the datapath of the victim switch to install flow rules.

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. 4 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 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. 4. When the victim switch receives the bounced-back packets, it delivers them to the controller since these packets do not match the protecting rules.

D. Packet Filter Module

Packet filter module 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 `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.

`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. 5.

The frequency of new flows is the most significant feature of SDN-aimed 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-aimed 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. 5, 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. 5, a layer 2 learning switch has a processing flow entry "eth_dst=00:00:00:00:00:01, $action$=$outport$: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 flow entries with the "drop" action, since these flows do not have reverse traffic. 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 drop-action flow rules could lead to incorrect classification. To solve this problem, we will not use asymmetric features for the classification of drop-action 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) [6], a supervised learning model as our classifier. This classification algorithm is robust even with noisy training data. The detailed implementation can be referred to [6], and we skip this part due to space constraints.

E. Flow Table Management Module

The flow table management module 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 module introduces a dynamic way to manage flow rules.

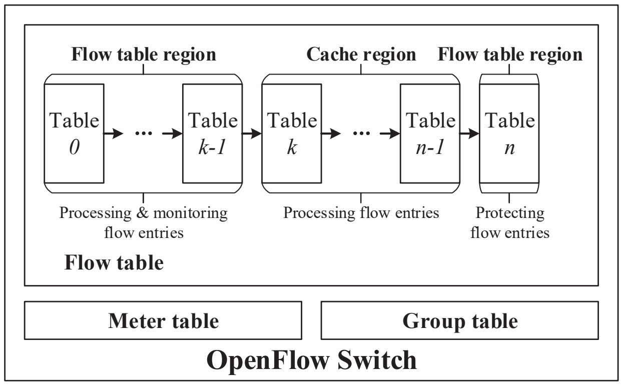


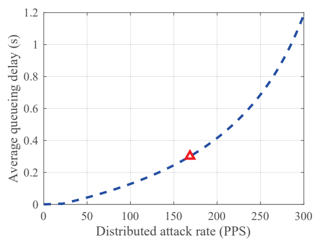
Fig. 6. The flow table is logically separated into flow table region and cache region by the flow table management module.

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 module to generate asymmetric features.

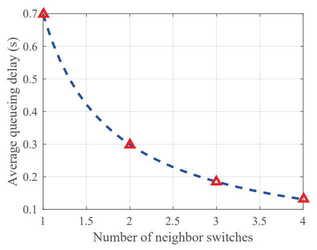
The management of flow table stems from the multiple flow tables in OpenFlow Specification v1.3. Specifically, the flow table management uses the first k tables (table 0 to $k-1$) and the last table (table n) as "flow table region", and other tables (table k to $n-1$) as "cache region". Notice that OpenFlow 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. 6. 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 module 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 all monitoring flow rules will be flushed after traffic-based filtering to save the space of the flow table.

The flow table management module 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 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



(a) Average queueing delay of a switch under different attack rates.



(b) Number of neighbor switches involved in the table-miss engineering.[2]

Fig. 7. Average queueing delay of switches.

one flow table. Though the efficiency of indexing is affected, flow table management module 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.

IV. 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 $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ involved in the table-miss engineering, and a set of attack traffic rates $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ 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 $\widetilde{R}_i$ 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 [7] to roughly describe the relationship between D_i and the utilization of this link (ρ_i):

$$D_i = \frac{1}{2\mu} \times \frac{\rho_i}{1-\rho_i}. \tag{1}$$

In Equation (1), μ is a coefficient of delay, and ρ_i describes link utilization ($\rho_i = \frac{\textit{Total Payload}}{\textit{Transmission Ability}}$, and $0 \leqslant \rho_i < 1$). The calculation of ρ_i could be separated into two scenarios: when the incoming packets rate is within the processing ability of s_i ($a_i \leqslant a_{s_i}$), 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:

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

Fig. 7-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 $\widetilde{R}_i$ 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):

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

In Equation (3), ρ_i and a_i could be roughly regard as a linear relationship ($\rho_i = ua_i + v$), since the table-miss engineering adopts $a_i > a_{s_i}$ for most cases. Suppose each switch has the same processing ability ($a_{s_i} = a_s$), maximum bandwidth ($R_i = R$), and allocated bandwidth ($\widetilde{R}_i = R_i$), Equation (3) could be further simplified as follows:

$$D = k \times \sum_{i=1}^{n} \frac{\rho_i(\rho_i + t)}{1-\rho_i}. \tag{4}$$

In Equation (4), the positive real number k represents the coefficient of the system, and t describes the constant of the linear relationship between ρ_i and a_i. Considering two switches s_p and s_q, ρ_q can be calculated based on ρ_p ($\rho_q = C - \sum_{i=1, i\neq q}^{n} \rho_i$, where C is the normalized attack traffic). When $i \neq p$ and $i \neq q$, $\frac{\partial \rho_i}{\partial \rho_p} = 0$. Since Equation (4) is a convex function, the minimized D can be obtained by solving the differentiation of Equation (4):

$$\begin{aligned} \frac{\partial D}{\partial \rho_p} = k(\frac{2\rho_p - \rho_p^2 + t}{(1-\rho_p)^2} + \frac{2\rho_q - \rho_q^2 + t}{(1-\rho_q)^2} \times \frac{\partial \rho_q}{\partial \rho_p}) &= 0 \\ k(\frac{2\rho_p - \rho_p^2 + t}{(1-\rho_p)^2} - \frac{2\rho_q - \rho_q^2 + t}{(1-\rho_q)^2}) &= 0. \end{aligned} \tag{5}$$

The minimized D will be obtained when $\rho_p = \rho_q$. Therefore, the best strategy to minimize D for the whole system is to evenly distribute the attack traffic ($\rho_1 = \rho_2 = ... = \rho_n = \rho$).

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 R is also separated into two scenarios:

$$\rho = \begin{cases} \frac{\widetilde{R} + a \times L_h}{nR}, & a \leqslant na_s \\ \frac{\widetilde{R} + na_s \times L_h + (a - na_s) \times L_p}{nR}, & else \end{cases}. \tag{6}$$

We could find out how many neighbor switches ($n-1$) should be involved based on Equation (1) and Equation (6). The result is depicted in Fig. 7-b. The configuration adopts 20PPS a_s, 750bit L_h, 5Kb L_p, 2Mbps R, 500PPS a, and

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

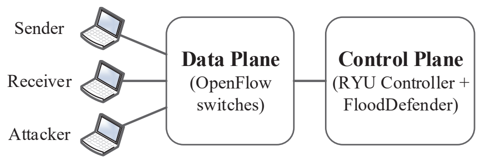


Fig. 8. Test network topology.

0 $\widetilde{R}$ 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 less neighbor switches.

V. Experiment

A. Implementation

We implement FloodDefender system, including the attack detection, table-miss engineering, packet filter, and flow rule management modules. All of them are implemented as applications on RYU controller [8] in Python. Meanwhile, we install RYU controller on a computer equipped with i7 CPU and 8GB memory. In the *software environment*, we use Mininet [9] to create virtual OpenFlow switches, and in the *hardware environment*, we use commercial OpenFlow switches, Polaris xSwitch X10-24S2Q [2], to build the test environment. Each hardware switch can store 2000 flow entries, and has 8MB buffer memory. We employ three hosts (sender, receiver, and attacker) in our test environment. The test network topology is depicted in Fig. 8.

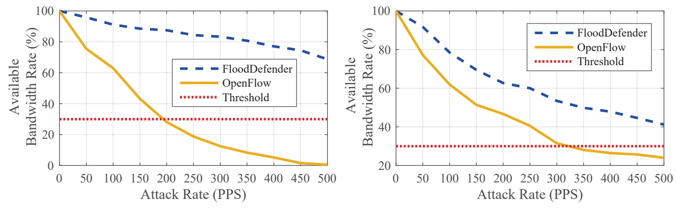
To compare FloodDefender with previous work, we launch the SDN-aimed attacks in three scenarios: (i) an OpenFlow network without protecting system, (ii) an OpenFlow network with FloodGuard [4], and (iii) an OpenFlow network with our FloodDefender.

B. Setup

First, 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). We install a layer 2 learning switch app (l2_learning) on the network, which can discover the network topology and provide basic forwarding service. The attacker will use *scapy* to keep flooding UDP packets with randomly forged 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 queueing delay.

Second, 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.

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



(a) Software environment. (b) Hardware environment.

Fig. 9. Available bandwidth rate of the victim switch.

Fourth, we compare the flow table utilization of the victim switch under OpenFlow, FloodGuard [4], and FloodDefender. We also use l2_learning as the app in the experiment. The attacker generates TCP packets with randomly forged sender IP to flood the network and overload the flow table.

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

Sixth, we measure the time delay of normal traffic under OpenFlow, FloodGuard [4], and FloodDefender. Here we measure the delay of all kinds of protocols under UDP-based DoS attacks, and the attack rate is 500PPS. 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 [4], and FloodDefender under TCP-based DoS attacks. To generate new flows efficiently, we modify l2_learning app, and use eth_src && tcp_src 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

Victim switch bandwidth. The results in software and hardware environments are depicted in Fig. 9-a and Fig. 9-b. In this test, we do not show the result from FloodGuard [4], 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 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. 10. 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,

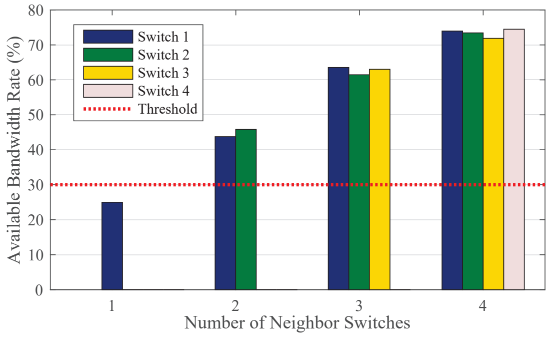


Fig. 10. Available bandwidth rates of neighbor switches.

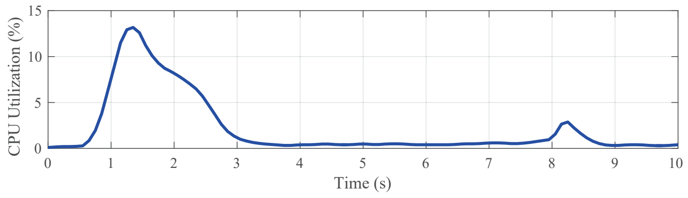


Fig. 11. CPU utilization under UDP-based attacks.

TABLE I. Flow Table Utilization under TCP-based Attacks

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

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.

Computational resource consumption. We can get the computational resource protection performance of FloodDefender in Fig. 11. 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 I. We can find that both FloodGuard [4] 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 the flow table management module will flush monitoring rules and 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

TABLE II. Time Delay of Normal Flows under 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

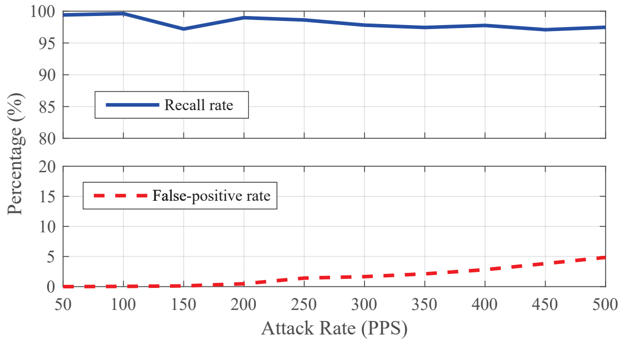


Fig. 12. Attack detection performance: recall rate and false-positive rate.

the two-phase filter is depicted in Fig. 12. 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 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 II. Since FloodGuard [4] utilizes rate control to save the computational resources, the delay of normal flows increases with the attack rate. When the attack rate is set to 500PPS, the maximum time delays in both FloodGuard and OpenFlow become infinite (timeout), which is different from the results presented in [4]. Besides the attack rate, another reason is that [4] 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. 13. 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,

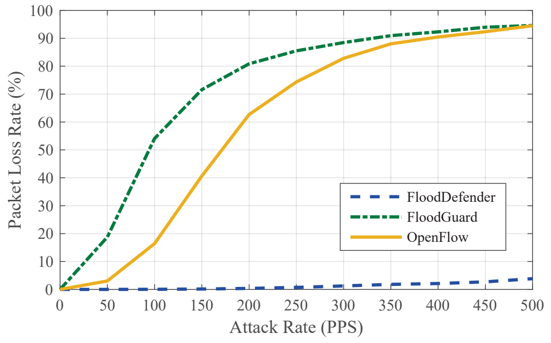


Fig. 13. Packet loss rate of new TCP flows under TCP-based DoS attacks.

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%.

VI. 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 [10], [11], [12], [13]. On the other hand, the same centralized structure is considered vulnerable, which can cause severe network security problems [3], [4], [5], [14], [15], [16].

SDN-supported security: SDN-supported security uses new techniques in SDN to solve traditional network security challenges. 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. Xu *et al.* introduce new DDoS detection methods based on the flow monitoring capability [11]. 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 [12]. 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 [3], [5]. To mitigate the DoS attacks, AvantGuard introduces an SYN proxy based module to verify the legality of each flow based on TCP handshake [3]. Another approach, FloodGuard, mitigates the DoS attacks by installing proactive flow rules and sending table-miss packets to a specific data plane cache module [4]. FloodGuard breaks the protocol limitation in AvantGuard, but may suffer from long delay and high packet loss rate for some flows. Another attack is poisoning the network visibility of the control plane by unauthorized LLDP packets [14]. TopoGuard uses the incoming port information to verify LLDP packets against network topology poisoning attacks [14]. To avoid malicious apps on the control plane, SDNShield expresses and enforces the minimum required privileges to individual apps [15].

VII. 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 three novel techniques: table-miss engineering, packet filter, and flow table management. FloodDefender can 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 significantly improves the flow table utilization, time delay, and packet loss rate, and is more scalable and easier to deploy without introducing additional devices.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling Innovation in Campus Networks,” in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 69–74, 2008.

[2] Polaris networks Co. ltd., “Polaris xSwitch.” http://www.polarisdn.com/en/product/html/?80.html.

[3] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, “AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks,” in *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2013.

[4] H. Wang, L. Xu, and G. Gu, “FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks,” in *Proc. of the IEEE/IFIP Dependable Systems and Networks (DSN)*, 2015.

[5] 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. of the IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2013.

[6] V. N. Vapnik and V. Vapnik, “Statistical Learning Theory,” vol. 1, 1998.

[7] A. Demers, S. Keshav, and S. Shenker, “Analysis and Simulation of a Fair Queueing Algorithm,” in *ACM SIGCOMM Computer Communication Review*, vol. 19, pp. 1–12, 1989.

[8] RYU SDN Framework Community, “RYU Controller.” https://osrg.github.io/ryu/.

[9] Mininet Team, “Mininet.” http://mininet.org/.

[10] Z. Hu and J. Luo, “Cracking Network Monitoring in DCNs with SDN,” in *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2015.

[11] Y. Xu and Y. Liu, “DDoS Attack Detection Under SDN Context,” in *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[12] 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. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[13] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, “Enabling Practical Software-defined Networking Security Applications with OFX,” in *Proc. of the Network and Distributed System Security (NDSS)*, 2016.

[14] S. Hong, L. Xu, H. Wang, and G. Gu, “Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures,” in *Proc. of the Network and Distributed System Security (NDSS)*, 2015.

[15] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, and X. Chen, “SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets,” in *Proc. of the IEEE/IFIP Dependable Systems and Networks (DSN)*, 2016.

[16] 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. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2016.