2020 IEEE 17th International Conference on Mobile Ad Hoc and Sensor Systems (MASS) | 978-1-7281-9866-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/MASS50613.2020.00048

FSDM: Fast Recovery Saturation Attack Detection and Mitigation Framework in SDN

Xuanbo Huang*, Kaiping Xue*†§, Yitao Xing†, Dingwen Hu†, Ruidong Li‡, Qibin Sun*
* School of Cyber Security, University of Science and Technology of China, Hefei, Anhui 230027, China
† Department of EEIS, University of Science and Technology of China, Hefei, Anhui 230027, China
‡ National Institute of Information and Communications Technology (NICT), Tokyo 184-8795, Japan
§Corresponding author, kpxue@ustc.edu.cn

***Abstract*—The whole Software-Defined Networking (SDN) system might be out of service when the control plane is overloaded by control plane saturation attacks. In this attack, a malicious host can manipulate massive table-miss packets to exhaust the control plane resources. Even though many studies have focused on this problem, systems still suffer from more influenced switches because of centralized mitigation policies, and long recovery delay because of the remaining attack flows. To solve these problems, we propose FSDM, a Fast recovery Saturation attack Detection and Mitigation framework. For detection, FSDM extracts the distribution of Control Channel Occupation Rate (CCOR) to detect the attack and locates the port that attackers come from. For mitigation, with the attacker's location and distributed Mitigation Agents, FSDM adopts different policies to migrate or block attack flows, which influences fewer switches and protects the control plane from resource exhaustion. Besides, to reduce the system recovery delay, FSDM equips a novel functional module called Force_Checking, which enables the whole system to quickly clean up the remaining attack flows and recovery faster. Finally, we conducted extensive experiments, which show that, with the increasing of attack PPS (Packets Per Second), FSDM only suffers a minor recovery delay increase. Compared with traditional methods without cleaning up remaining flows, FSDM saves more than 81% of ping RTT under attack rate ranged from 1000 to 4000 PPS, and successfully reduced the delay of 87% of HTTP requests time under large attack rate ranged from 5000 to 30000 PPS.**

***Index Terms*—Software Defined Networks, Saturation Attack, Mitigation Mechanism, Fast Recovery**

I. Introduction

Software Defined Network (SDN) has been becoming the protagonist on the stage of next generation network. Taking advantage of decoupling the control and data planes, it's flexible function deployment ability and reduced operational overhead are very attractive to data center and cloud network providers, thereby enjoys a great popularity [1]. SDN centralized the control logic by using an open southbound interface to allow controllers and switches to interact. One of the most popular southbound interface protocol is OpenFlow [2]

OpenFlow introduces the concept of flows and flow tables, which is used to identify and manage the network traffics. Each OpenFlow switch maintains a pipeline to deal with network flows under the guidance of a set of flow tables, produced by the controller. The network operation and maintenance personnel can easily define and change the flow process logic by writing applications to deliver the flow tables either proactively or reactively. It is worth mentioning that when in the reactive mode, the switches will ask for instructions from controller for all flows that do not have a match in current flow tables. However, while this gives SDN more elasticity and visibility, a potential bottleneck vulnerability comes ensuing.

A notable attack that exploits the communication of controller and switches to overload the control plane, are called the control plane saturation attacks. Saturation attack is a kind of Denial-of-Service attack and can be easily performed by a host in or just connecting to SDN networks. From previous studies, it is mainly implemented by SYN flood [3]–[5]. While, UDP flood, ICMP flood, IP spoofing and their combinations also holds a candle to the saturation attackers [6]. Recently, with more advanced SDN reconnaissance methods [7]–[10] being proposed, saturation attacks have been even more destructive and stealthy [11], [12].

Therefore, how to detect and mitigate control plane saturation attack is a very challenging problem. Previous studies have proposed several methods to detect and mitigate saturation attacks. In terms of detection, [13], [14] use the rate of PACKET_IN messages to determine whether there is an attack. [6], [15]–[17] build upon the studies of flow entropy or self-similarity to detect the attacks. [18] adopts flow rate and flow duration for anomaly detection. These methods can determine that the whole network is under attack but cannot locate the port that malicious hosts come from, thus may result in complex mitigation policies with unnecessary filtration, and more involved switches. If we can locate the attacker, we can deal with the attack flows more precisely and reduce the impact on benign traffics. And on the other hand, in terms of mitigation. [19] introduces mechanism to reduce interaction between controllers and switches. [6], [13], [15] introduce extra centralized network entity and migrate attack flows to it for further filtration. However, centralized mitigation method based on migration may influence all the switches on the migration path. [14] detours the attack flows to the victim's neighbor switches to protect their bandwidth, and filter the attack flows in the controller, but the efficiency is related to number of switches that involved in the detour process. Besides, [15] suffers from high system recovery delay. From discussion above, we summarized two challenges:

- How to accurately detect the attack while locating the port that attackers come from?

2155-6814/20/$31.00 ©2020 IEEE
DOI 10.1109/MASS50613.2020.00048

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

- How to efficiently mitigate the attack flows, reduce the impact switches number, while making the system recover faster?

To solve these challenges, in this paper, we propose FSDM, a Fast recovery Saturation attack Detection and Mitigation framework, which is light-weight and scalable. FSDM contains an Attack Detector, a Mitigation Manager, and a novel functional module Force_Checking for different designed objectives. Specifically, Attack Detector is designed to tackle the first challenge while Mitigation Manager and Force_Checking aim at the second one. Attack Detector deploys a novel, light-weight detection scheme that extracts the distribution of Control Channel Occupation Rate to detect the attack, while locating the attackers from the distributions. Mitigation Manager contains a Mitigation Server in the controller, and distributed Mitigation Agents in each edge gateway. Once the attack is detected, Mitigation Server migrates attack flows to the nearest Mitigation Agents or blocks attack flows directly depending on the attacker's location. Thus, it can reduce the influenced switches and protect control plane. In the meantime, to reduce the system recovery delay, FSDM deploys a novel functional module Force_Checking in the Event Dispatcher module of controller platforms. The Force_Checking function quickly cleans up buffered attack flows and enables the system to recover in a short time. FSDM is easy to be implemented in physical networks and there is no need for changing OpenFlow protocol. Besides, we have evaluated our framework in SDN simulation environment with Mininet [20] and RyuController [21]. The experimental results show that FSDM can timely detect the attack, and enable the system to recover in a short time. The main contributions of our work can be summarized as follows:

1) We propose FSDM, a Fast recovery Saturation attack Detection and Mitigation framework. By adopting novel detection mechanism that can locate attackers, and introducing distributed Mitigation Agents, FSDM reduces the influenced switches and avoid unnecessary filtration, while protecting the control plane from resource exhaustion.
2) We make analysis of the system recovery model, and proof that, some attack flows that still remained in the system after mitigation are the culprits of long system recovery delay. To solve this problem, we propose a novel functional module Force_Checking, which can quickly clean up remained attack flows and enable the system to recover in a short time.
3) We conduct extensive experiments in a simulation SDN platform implemented a prototype system of FSDM. The results show that FSDM can timely detect the attack and saves more than 81% of the ping RTT (Round Trip Time) compared with traditional method without cleaning up buffered packets, and reduces averagely 87% of the HTTP requests time under large number of attacks.

The rest of this paper is structured as follows: Section II-A elaborates some necessary background knowledge on SDN, OpenFlow, and control plane saturation attacks. Section II-B discuss the related work and the difference from ours. In Section III-A we will illustrate the threat model, and a theoretical analysis on system recovery delay in Section III-B. Then, our detection and mitigation mechanism framework design is depicted in Section IV, and is evaluated in Section V. We draw our conclusion in Section VI.

II. Background and RelatedWork

A. SDN and OpenFlow

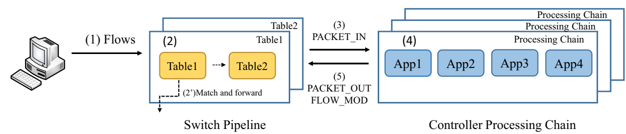


Fig. 1: Flow processing in reactive method

SDN separates the control plane and the data plane of the network, making the data plane more generic. The network's control logic is managed by centralized controller applications. Taking advantages of programmability, efficiency, and flexibility, SDN plays a role in more and more scenarios.

In SDN networks, the interface between the control plane and the data plane is called the southbound interface. The SDN southbound interface is the core embodiment of the programmability of the entire network, so an open, unified, and more programmable interface is required. In 2008, Nick McKeown *et al.* proposed the OpenFlow [2]. Currently, OpenFlow is the most popular SDN southbound interface protocol. According to the version of OpenFlow1.5 [22], firstly, controller and switches will establish an OpenFlow channel over TCP or TLS. The controller installs flow rules to the switches to instruct them to handle different network traffics. Each flow rule contains a match field, an action field, a timeout field, *etc*. Flow rules can be installed proactively (preinstalled) or reactively. OpenFlow switches maintain a pipeline with a set of flow tables contains of flow rules to deal with network traffics. In the reactive method shown in Fig. 1, the lifecycle of a flow can be divided into several steps. At Step (1), a certain flow arrives at a switch. Then in Step (2), the switch checks whether there is a matching flow rule in flow tables. If there is a match, it will execute Step (2') to follow the corresponding flow rule's action, such as forward to a certain port, set some fields, or go to other tables. If no flow rule matches the current incoming flow, switch executes Step (3), encapsulating the first packet of the flow as a PACKET_IN message to the controller. Controller applications are responsible for Step (4), in which they extract the header and process their functions. In addition, a forwarding application or link discovery module may execute Step (5), to generate PACKET_OUT messages send it back with a set of actions, or generate a FLOW_MOD message to instruct the switch to process the same kind of flows.

In the controller platforms, different applications make up multiple processing chains waiting for dealing with events.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

When controller receives a message, for example, a PACKET_IN message, the Event Dispatcher module transforms it to a PACKET_IN event, and dispatches the event to corresponding applications that subscribed the PACKET_IN event (e.g., a forwarding application). Each the of applications have an event queue, to temporarily store the events haven't been processed.

For attackers connected to SDN networks, they can manipulate large number of no match flows to trigger PACKET_IN floods and PACKET_OUT floods to occupy the control channel and controller's resources, and trigger FLOW_MOD floods to exhaust the memories of switches. This kind of attacks are called the control plane saturation attacks.

B. Detection and Mitigation of Saturation Attack

Shin *et al.* [23]firstly introduced the concept of SDN saturation attacks, which trigger new flow rules to exhaust the switches' memory. To mitigate the attack, AVANT-GUARD [3] and LineSwitch [24] extend the data plane, making the switches act as SYN proxy to reduce interaction in controller and switches. They are applicable to TCP-based attacks only. However, saturation attacks can be launched with a various methods [6] as long as the malicious host can manipulate considerable table miss flows to trigger PACKET_IN floods.

Then, FloodGuard [13], FADM [15], and FloodDefender [14] introduced the protocol-independent methods to deal with the various attack flows. Designed as an application, FloodGuard consists of two new functional modules: flow rule analyzer module, and packets migration module. The flow rule analyzer dynamically collect the flow rules in the network, and install all of them when the attack is detected to reduce the PACKET_IN messages. Migration module combines the real time PACKET_IN rate and the CPU utilization of the controller to detect the attack. When the attack is detected, FloodGuard installs a flow rule with the lowest priority to migrate table-miss flows to a middle layer cache and limits the PACKET_IN rate of the flows, using a round-robin scheduler to send packets to controller. However, this kind of mitigation policies may cause high packet loss rate in some cases [14]. FADM consists of two modules: DDoS detection module and DDoS mitigation module. The detection module employs SVM (Support Vector Machine) [25] to detect the attack, and the mitigation module installs a wildcard rule with high priority to migrate similar attack flows with same protocol and IP destination to the Mitigation Agent for further filtration. The high priority migration rule results in all benign flows need to trigger a new PACKET_IN events, intensifying the control plane's pressure. Besides, FADM does not clean up buffered attack flows, thus suffers from long system recovery delay under large number of attacks. FloodDefender lied between controller platforms and applications as a middleware to better filter the flows. Different from the works mentioned above, FloodDefender doesn't introduce a cache to store temporary attack flows, but installs a flow rule to detour attack flows to the victim's neighbor switches when the attack occurs. In this method, there is no need for extra network entity and no need to change OpenFlow protocol. However, FloodDefender's mitigation efficiency is relevant to the number of switches that involved in the detour process, which means it might not be efficient when a numerous attack occurs in a small-scale network that contains only several switches. Resources might still be exhausted in the detour process. In the meantime, as the controller affords most of the filtration work, the control channel will still be occupied by attack flows.

By contrast, deploy a Mitigation Agent as an entity is more stable and practice. The main idea of the Mitigation Agent is to take the place of control plane's work to deal with attack flows. However, traditional centralized mitigation agent may suffer from more influenced switches. For example, if the agent is far from the victim switch, the attack flows will influence all the switches on the migration path. To solve this problem, we deploy distributed Mitigation Agents in each edge gateway as virtual functionality module to deal with the attack flows. Compared with the centralized Mitigation Agent, the distributed method is more scalable and robust. In the meantime, the attack flows coming outside the network will be migrated with no more than one hop, thus reduce the influenced switches.

III. Threat Model And System Recovery Model

A. Threat Model

Control plane saturation attack is a kind of resource exhaustion attack that can be launched in a malicious host inside or outside the layer two SDN network, to paralyze the control plane, data plane, or both. With the technology of time-based reconnaissance [26] and flow rule reconstruction [7] of flow rules in SDN, the malicious host can know most of the matching fields in the flow tables, thus easily manipulates new flows to trigger PACKET_IN flood to deplete the resources. In this paper, we assume that the attacker can reconnaissance some matching fields of flow rules to manipulate new flows. Also, he can launch attacks from botnet outside the network, or compromise a single host inside the network, which can be easily achieved (*e.g.,* to rent a virtual machine in SDN based cloud network).

B. System Recovery Model

Generally, in the reactive SDN environment, packets that cannot match a flow rule will experience higher delay as it will be sent to controller as a PACKET_IN message. In the controller, there are multiple applications waiting to deal with these flows, (*e.g.,* ARP Proxy, ACL, Load Balancer, *etc.*) The packets need time to queue and experience all the applications pipeline. When a saturation attack happens, numerous malicious packets will be injected into control plane. Before the attack is detected and mitigated, all the attack flows injected into controller will be buffered, queued, and processed normally in the controller's application pipeline. Traditional mitigation policies [13]–[15] were generally installing a flow rule to block or migrate the attack flows in the switches, while ignoring the buffered attack flows. However, the buffered attack flows are also in a large number, though the attack is

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

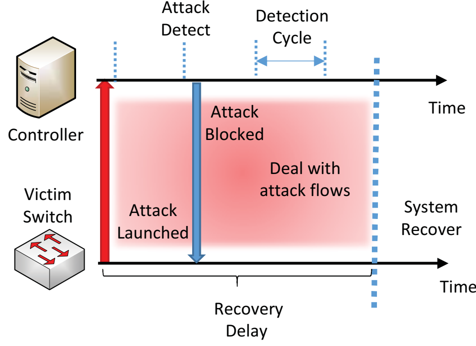


Fig. 2: Recovery time model without cleaning up remaining attack flows

timely blocked, other applications in the network still regard these flows as benign traffics, thus process them normally. That is because these applications do not have a secure attribute to distinguish and drop attack flows. These buffered attack flows can occupy the system resources for a long time, force the system to deal with useless flows. Thus, benign traffics can not be timely forwarded until the attack flows being dealt, that is to say, the system need a long time to recover.

Then, we estimate the time period that control plane might be influenced under an attack. We assume that the attacker launches an attack at time t_a with the rate K_{pps} and average size of b_a byte, and the detector need a period of time t_w (i.e., a detection cycle) to collect information and determine whether there is an attack. Additionally, B_{cs} denotes the bandwidth between switches and controller.

Fig. 2 shows the recovery time model in traditional methods without cleaning up buffered packets. In the worst case, the attack is launched exactly at the beginning of a new detection cycle, or slightly before a cycle start, that is to say the detector needs at least one t_w or more to detect the attack. We use t_d to depict the time period that controller need to detect the attack. During the period of t_d there are $t_d \cdot K_{pps}$ packets will be sent to switch before the attack being detected. If the bandwidth between controller and switches is sufficient, all the packets will be forwarded to controller. If the bandwidth is limited, the packets that are sent to controller can be calculated as $t_d \cdot \frac{B_{cs}}{b_a}$. Thus, the numbers of packets that injected to control plane N_a is about:

$$N_a = min\{t_d \cdot \frac{B_{cs}}{b_a},\ t_d \cdot K_{pps}\}. \tag{1}$$

Generally, in a traditional method, once the attack is detected, the controller installs a defensive flow rule to switch to block or migrate the attack flows. However, at that time the controller's send queue may be filled with triggered PACKET_OUT and FLOW_MOD messages thus the installation of defensive rule need to wait for a while. Once the defensive rule is installed, the switches stop injecting attack flows to control plane. Notice that though the attack flows are blocked from switches, the injected N_a flows have not been processed completely. Some of them are still remaining in the control plane, buffered in the event handler, controller modules or applications event queues. Then, when a normal flow needs to be forwarded, it has to wait for a long time to queue. To measure the time, we firstly assume that applications need time t_c to deal with each flow. Then the number of remaining packets N_r can be calculated as:

$$N_r = N_a - \frac{t_d}{t_c}. \tag{2}$$

Then the time period that controller needs to completely process all these attack flows from when the attack is detected can be denoted as

$$t_m = t_c \cdot N_r. \tag{3}$$

We define the whole system recovery time as t_r, and we have

$$\begin{aligned} t_r = t_d + t_m =&\ t_d + t_c \cdot N_r \\ =&\ t_d \cdot (1 + t_c \cdot max\{\frac{B_{cs}}{b_a}, K_{pps}\}). \end{aligned} \tag{4}$$

In a system, the process time t_c depends on its resources and hardware, and the attack rate K_{pps} depends on the attacker's ability. Thus, from the Equation 4, system recovery time t_r is mainly relevant to the buffered packets number N_r, and the detection time t_d. To reduce the system recovery time, intuitively we should reduce the t_d or N_r. However, reducing t_d is not a generic way as it is relevant to the detection algorithm and the time window length t_w, which is important on information collection. The longer t_w it is, the more information can be gained by the detector to make the decision more precisely. Thus, we cannot reduce the t_w to make the system recover faster as it may influence the accuracy of detection module. By contrast, using a method to reduce N_r is more generic and stable. To solve the problem, we proposed novel module Force_Checking to effectively reduce the N_r. We will verify the correctness of our analysis in practice in Section V, and show the efficiency of Force_Checking on reducing the system recovery delay.

IV. FSDM Framework Overview

In this section we introduce our Detection and Mitigation framework: FSDM. Fig. 3 illustrates the system overview and the basic working process. FSDM contains an Attack Detector, Mitigation Manager and the novel functional module Force_Checking. Mitigation Manager contains of a Mitigation Server in controller and distributed Mitigation Agents deployed in each gateway. When network is established, Attack Detector calculates the detection threshold and starts monitoring the control channel for anomaly detection. And the Mitigation Server starts collecting white-list in each detection cycle and sending them to the Mitigation Agents. Once the attack is detected, Mitigation Server stops white-list collection and uses the location information to block or migrate flows that come from the same port as attack flows. Migration happens when attack flows are mixed with benign traffics, and is implemented by installing a flow rule with the lowest priority to migrate these flows to the nearest Mitigation Agent

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

for filtration. Benign traffics in white-list will be forwarded to Mitigation Server, which will generate a PACKET_IN event to other applications, making the protection process transparent.

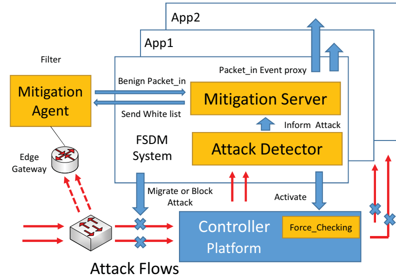


Fig. 3: System overview and working process for FSDM

A. Attack Detector

1) Attack Detection: To detect the attack, we extract the difference between the benign traffics and the attack flows of the Control Channel Occupation Rate (CCOR). The main idea is that, on the one hand, in each time window, the number of benign table-miss flows are different from the malicious one, which always comes in a large number. And on the other hand, the attack flows will change the distribution of CCOR of each port. Intuitively, when we observe a new flow from a certain port, we can observe the corresponding pair flow from another one. However, when numerous attack flows come, their pair flows can hardly be observed as they exceed the number that system can afford, and the benign flows from other port will also be congested and queued for a while. That is to say, in a short time the monitored CCOR of each port would be much more uneven, which cause a very low entropy value. For example, the CCOR of benign traffics for a certain port can hardly exceed 50%, if all the flows from that port are in pair. If the occupation rate exceeds 70%, that means there are at least 40% flows have no pair flows, they might be queued or just dropped. It is a signal that there are too many new flows for the system to afford, thus is determined as an attack. From the discussion above, we take the total number of packets and the entropy of CCOR of each port to detect the attack. For the total number of packets, we define threshold S_T. And for the distribution of CCOR, we define threshold E_T. Both of the threshold are calculated from the network parameters itself automatically after network is established. When the network PACKET_IN messages exceed the threshold S_T and the CCOR entropy lower than threshold E_T, we determine there is an attack.

2) Setup: To monitor the CCOR for the whole network, Attack Detector firstly call the link discovery module to find all the active ports that connected to active 'hosts' (real hosts or legacy routers). We do not monitor on the ports that between two switches, as the PACKET_IN messages from that port is not the newest (the last hop have sent it to the controller). For these ports, their *(dpid, port_no)* are used as the unique identifications, which can not be forged by attackers unless they compromise a switch. Considering there are n such ports exist in the data plane, we establish a monitor vector as: $\boldsymbol{M_t} = (h_1, h_2 \cdots h_n)$, where t represents the time window, and the initial value of $\boldsymbol{M_0} = (1, 1 \cdots 1)$ for the sake of entropy calculation. Assume in a time window, there comes k PACKET_IN messages from host h_i. Then the corresponding elements in monitor vector will be set as $h_i = k$. In each time window, we calculate the sum of $\boldsymbol{M_t}$ as $Sum_t = \sum_{i=1}^{n} h_i$. And the CCOR in this time window can be calculated as:

$$\boldsymbol{M_t^*} = \frac{\boldsymbol{M_t}}{Sum_t}. \tag{5}$$

Next, we use entropy to measure the distribution:

$$H(\boldsymbol{M_t^*}) = -\sum_{i=1}^{n} h_i^* \cdot \log_2(h_i^*). \tag{6}$$

In each time window, if the Sum_t exceeds threshold S_T and the $H(\boldsymbol{M_t^*})$ less than threshold E_T, the Attack Detector determines that there is an attack.

3) Threshold Calculation: The first threshold S_T should be set depend on the bottleneck feature in the SDN environment, thus when exceeded, we know there might be an attack. The bottleneck might be the MAX size of flow entries in switches, bandwidth, memory size, *etc.* In the simulated SDN environment, the switch flow entries and memory are sufficient compared with the computational resources of controller. Besides, the switch features can not be timely updated under attacks as the channel might be broken. Thus, we use the MAX number of packets that controller can handle per second (denoted as n_d) to measure the capacity of the whole control plane, which is used to calculate threshold S_T. To estimate the value of n_d, Attack Detector calculates average packets process time t_c in the first several time windows (assume there are no attack). The process time is calculated from when the event is dispatched to the certain application, to the time that being processed totally. And we have $n_d = \frac{1}{t_c}$. Note that, as the Attack Detector doesn't need to send PACKET_OUT or FLOW_MOD, the process time t_c is less than other applications, thus the estimated n_d is relatively larger. Besides, when under attacks, the computational resources will be distributed to all the applications, then the number of packets that Attack Detector monitored would be much less. Thus, the n_d should be multiplied by a correction parameter β to properly represents the 'bottleneck' value. Then we got $S_T = \beta \cdot n_d.(0 < \beta < 1)$After that, the question is how to find the proper value of β to accurately detect the attack. Intuitively, β is relevant to the PACKET_IN message subscriber applications number. Assume that there are large numbers of PACKET_IN floods inject to control plane, and there are multiple of applications asking for resources to process these flows. In a round-robin operating system, time resource are distributed evenly for each application. Assume there are m PACKET_IN subscriber applications in the network, we define $\beta = \frac{1}{m}$. The S_T now represents the 'max ability' that the system can handle in a time window. When the packets number

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

exceeds the value S_T, we further extract the distribution character to determine whether there is an attack.

The second threshold E_T is calculated from the network scale. We define the malicious single port occupation rate as α ($50\% < \alpha < 100\%$), and calculate the entropy of a monitor vector with malicious occupation port in Algorithm 1 as E_T. In our experiments, we set $\alpha = 70\%$.

Algorithm 1: Calculate Threshold E_T

Input: Number of hosts n, Occupation Rate α
Output: Threshold E_T
$x_1 = \alpha$;
$e_1 = -x_1 \cdot \log_2(x_1)$;
for $i \in range(2, n)$ **do**
  $x_i = \frac{1-x_1}{n-1}$;
  $e_i = e_{i-1} - x_i \cdot \log_2(x_i)$
end
return $E_T = e_n$

4) Locate the attackers: In order to efficiently deal with the attack flows, reducing the overhead in mitigation process, merely detect the attack is not enough. Thus, we propose a novel method to locate the port that attackers come from. As is discussed above, we monitor on each port of switches in each time window. In t time windows, if the network state is normal, Attack Detector collects a series of monitor vector as $< M_1, M_2 \cdots M_t >$. When the Attack Detector determines that there is an attack, it checks if there exists a single port that has the malicious occupation rate of control channels. If so, return the port index as the attacker's port. If not, it calculates $M_{atk} - M_{atk-1}$ to find the port that have the highest growth as the attacker's port.

B. Mitigation Manager

Mitigation Manager consists of Mitigation Server deployed in controller and distributed Mitigation Agents deployed in edge gateways, and they communicate with each other through TCP or TLS. As shown in Fig. 3, when the network is in normal state, the Mitigation Server collects white-list for benign traffics periodically and sends them to each Mitigation Agent. We adopt the collecting policy for white-list that discussed in [15], Mitigation Server updates white-list each benign time window according to the following rules:

- The flow is in pair;
- In a period of time, the number of flows with the same source IP exceeds a threshold value.

In each time windows, the source IP addresses that matched these requirements are added into white-list. We store white-list in a Bloom filter [27] data structure for each Mitigation Agents and the Mitigation Server to reduce the overhead in communication, and save the memories of each entity. Notice that even though the Bloom filter has a misjudgment for false negative samples, it would not influence the system as we only need to filter most of the attack flows. For example, for a Bloom filter with false rate 0.1%, it is possible that the Mitigation Agents forward 10 attack flows in each 10000 attack flows. However, few attack flows can hardly cause obvious impact on the whole system. Once the Attack Detector determines an attack takes place, Mitigation Server processes the attack flows in three steps:

- First, Mitigation Server installs a flow rule with the lowest priority on victim switch to migrate or block all the table-miss flows from attacker's port depending on the attacker's location. For attacks from single-host port inside the network, Mitigation Server installs a flow rule to block the abnormal traffics directly. For attacks from multiple-hosts port (e.g., the edge gateway's port), as the traffics are hybrid with benign traffics, Mitigation Server migrates these flows to the nearest Mitigation Agent for further filtration.
- If migration happens, Mitigation Server activates the corresponding Mitigation Agent, which uses *pypcap* to capture all the packets pass through. When activated, the agent extracts all the packets headers, and checks if the source IP address is in the white-list. If so, agent sends the data to the Mitigation Server.
- When Mitigation Server receives a packet from Mitigation Agent, it encapsulates the packet as a PACKET_IN event and sends it to all the PACKET_IN message subscriber applications. With all the OpenFlow fields set a proper value. For example, the *dpid, in_port* fields are set the same as corresponding victim switch, which is not hard to get. With this step, the whole protection process is transparent to other controller applications.

By using these methods, we can timely protect the control plane from resources exhaustion while ensuring the communication between legitimate users are not affected.

C. Force_Checking Module

After the corresponding policy is implemented by the Mitigation Server, there will be no more attack flows injected into control plane. However, there are still lots of buffered attack flows remaining in the control plane. From the discussion in Section III-B, we propose novel functional module Force_Checking to clean up all the buffered attack packets immediately after the attack is detected. However, this is hard to achieve. For one thing, the attack flows might be mixed with benign flows, the module must have the ability to distinguish attack flows from all the flows. For another thing, there are multiple places for these flows to hide, which makes it hard to clean up them all. To solve these problems, the Force_Checking module uses white-list to distinguish attack flows, and cleans up most of buffered packets in the Event Dispatcher module of controller platforms. The basic working process of Force_Checking is: It does nothing when the network state is normal thus will not bring overhead to systems. But when the attack is detected, the Force_Checking module is evoked. It extracts the packets headers and checks whether the source IP addresses are in the white-list every time controller's Event Dispatcher wants to generate a PACKET_IN event. If not, drop that event to avoid adding them to the

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

subscriber applications event queue. In that way, the buffered attack flows can be efficiently reduced, thus enabling the system to release resources for benign traffics.

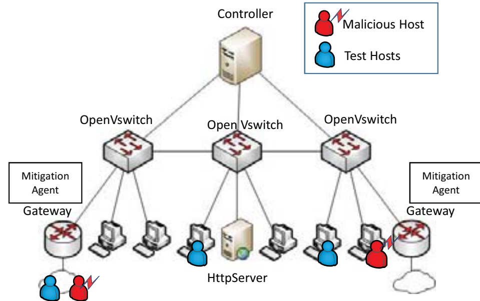


Fig. 4: Experimental topology with prototype system FSDM

V. Evaluation

A. Implementation

We implemented a prototype FSDM system in simulation SDN platform. The Attack Detector and Mitigation Server are implemented as an application in the Ryu controller, deployed in a computer with i5-9400 CPU and 6 GB memory. In the meantime, we use Mininet [20] to create virtual hosts and OpenVswitch [28]. From the *iperf* test, the whole bandwidth in our system is 25.6Gbps. As is simulated, all the links share the same bandwidth resources and it is much more sufficient for experiments. The experimental topology is shown in Fig. 4. There are three switches, two legacy routers as gateways deployed mitigation agents, and several hosts in the system. Besides, there are several Test Hosts in our topology, these hosts are used to send ping messages and record RTT, or download a certain page periodically from the web server and log the request time.

To simulate the realistic background traffics, we set that all the hosts will communication with random target in a random time, using TCP or UDP. In the Ryu controller, we implemented other two applications. They are *ArpHub* and *EasyACL*, both of them are written by ourselves and tested functional well. The *ArpHub* takes responsibility for all the arp reply to avoid flood and installs l2 forwarding rules, *EasyACL* provides a basic access control for some internal hosts on UDP and TCP messages based on IP address. In addition, we set that all IP address can access the web server. These two applications' working process can be regarded as a l3 forwarding function with access control, based on field *ip_src && ip_dst*. The whole network's flow tables are set as follows: In table 0, the traffics having access in ACL will be set *goto=table 2*, and the no-match flows are set *goto=table 1*. In table 1, the no-match arp, tcp, and udp messages are forwarded to controller. In table 2, l2 forwarding rules are installed and forward the traffics to certain port.

As mentioned above, we assume that the attacker have the ability to reconnaissance the flow rules in our system, which means the attacker knows that all the traffics with random value of *ip_src* and *ip_dst = Web Server* will not be dropped and can trigger PACKET_IN floods. The malicious hosts may outside the network or inside the network, and use *hping3* to create attack flows with different fields and protocols.

B. Setup

First, to verify the correctness of the system recovery model in practice, we measured the buffered attack flows number N_r under different attack rate and evaluated the influence on applications. We measured the maximum buffered packets to evaluate the worst case that the attacker can cause, and the average buffered packets to reflect more generic situations. The maximum buffered packets is obtained with the longest detection time t_d (to attack slightly before a detection cycle starts), and the average one is obtained from random detection time. In our system, the *ArpHub* and *EasyACL* start counting attack flow's number once the attack is detected and blocked. The attack rate ranged from 500 to 2500 PPS.

Second, we measured the system recovery delay. We define the system recovery delay as the time period that a new flow need to be forwarded as soon as the attack start, as mentioned above. To measure that, we use Test Hosts in our topology to send *ping* messages slightly behind the attack start (to make sure the attack flows are injecting). The *ping* messages and attack flows will both trigger PACKET_IN messages, and mixed with each other to be processed by the controller. The RTT is mainly relevant to the time that controller needs to reply for benign traffics under attacks, thus can be used to measure the system recovery delay. We conduct experiments with traditional methods and with Force_Checking function, and the attack rate increasing from 500 to 4000 PPS.

Third, we made a comparison between our work and the FADM [15] system. FADM defined the system recovery delay as the first HTTP request time under attacks. In their system, several hosts download pages from web server every 3 seconds and record the time, and the first recorded time under attacks is used to measure the recovery delay. The system recovery delay measured in this method is slightly lower than the method using *ping* RTT, as the *ping* messages are always sent almost simultaneously with attack flows. But both of them can reflect the time period that the system need to recover. Thus, we use the same method with them for evaluation and the same attack rate ranged from 5000 to 30000 PPS.

Fourth, we evaluated the detection efficiency under attack rate ranged from 500 to 5000 PPS.

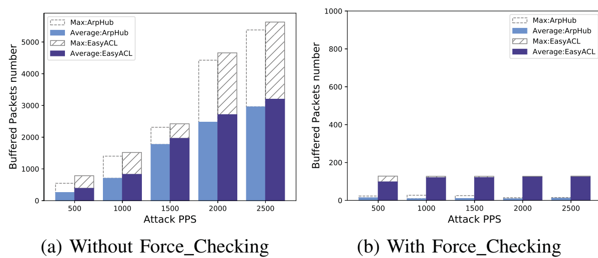
C. Evaluation

1) Correctness of recovery model: Fig. 5a shows the buffered packets number in traditional mitigation cases that ignored buffered attack flows. As we can see, with the attack rate increasing, the buffered packets number increases quickly. Besides, we measured the queue delay of each packet for each application in TABLE I. When under attacks, the queue delay increases because of the resource exhaustion. Obviously, if we do nothing about these buffered attack flows, these two applications need a large amount of time to process them all.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

For example, under the worst case with attack rate 2500PPS, there are more than 5628 packets injected into *EasyACL* and 5380 packets injected into *ArpHub*. The average queue time for each packet is 30.03ms in *ArpHub*, and 33.23ms in *EasyACL*. These packets will be regarded as normal traffics by these two applications to process. We can roughly evaluate the time that a benign traffic need to wait at this time. For *EasyACL*, it needs roughly 187s, and for *ArpHub*, it needs about 162s. Worse still, these buffered packets will trigger lots of PACKET_OUT and FLOW_MOD messages. And the physical OpenFlow switches have limited flow entries, which can be exhausted by these buffered packets easily. In an SDN environment with multiple applications, things could be even worse.

Then we measured the buffered packets number with our novel functional module Force_Checking. As shown in Fig. 5b, Force_Checking effectively reduce the buffered packets under attacks. With the attack rate increasing, buffered packets number remains a low value and does not increase. Under the worst cases with attack rate 2500 PPS, there are only 128 buffered packets in *EasyACL*, and only 20 in *ArpHub*. Averagely, Force_Checking reduces 98.43% of buffered packets for *ArpHub*. And 89.16% of buffered packets for *EasyACL*. Less buffered packets means that the Force_Checking function has effectively released system memory resources and the computational resources under attacks, thus make a big contribution to reduce the system recovery delay.



(a) Without Force_Checking (b) With Force_Checking

Fig. 5: Buffered Packets Number Under Attacks

TABLE I: Average queue time for each application

Average Queue Time	ArpHub	EasyACL
No attack	4.13 ms	7.26 ms
Under attacks	30.03 ms	33.23 ms

2) System recovery delay: We measured the system recovery delay in the traditional method without cleaning up buffered flows and our Force_Checking function with *ping* RTT. Fig. 6 shows the average *ping* RTT under different attack rates. Without the Force_Checking module, the recovery time increases fast, especially in the attack rates from 2000 to 2500 PPS. Under larger attack rates, the system needs more than 40 seconds to recovery. That is because the whole network needs a time to release the computational resources and memory that occupied by the attack flows. But with the Force_Checking module, the system can quickly clean up buffered attack flows to release computational and memory resources, thus quickly responds and deal with the benign flows. Under attack rate within 3000 PPS, Force_Checking enables the system to recover in less than 5 seconds. Under larger attack rate 3000 to 4000 PPS, system recovery time is no more than 8 seconds. Compared with the traditional mitigation policies that ignore the remaining traffics, we successfully reduced more than 81% of the system recovery delay with attack rate 2500 to 4000 PPS.

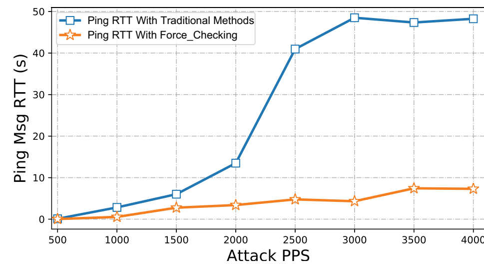


Fig. 6: System Recovery Delay with *ping* RTT

3) Comparison with FADM: We made a comparison with FADM system under larger attack rates. To keep consistency, we used the same method defined in FADM to measure the recovery delay, and we launched the same TCP-based attacks. Fig. 7 shows the first HTTP request time between two systems, under attack rate ranged from 5000 to 30000 PPS. After attack rate exceeds 5000 PPS, FADM system needs more than 30 seconds to recover from those large number attack flows. While, in our system, the HTTP requests time is less than 6 seconds. Averagely, we reduced 87.17% of the HTTP requests time under large number of attacks, which indicates that FSDM is robust and practice.

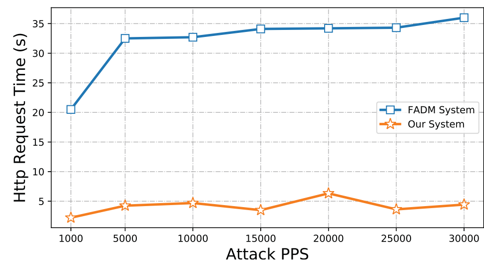


Fig. 7: Recovery Time compared with FADM system

4) Detection efficiency: We use Detection Rate (DR) to measure the performance of the Attack Detector, which is defined as:

$$DR = \frac{TP}{TP + FN},$$

where TP represents the sample quantity that attack state being detected in time (no more than two time windows), and FN denotes the quantity of attack state that regarded as normal. We conducted 139 experiments under attack rate ranged from

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.

500 to 5000 PPS, and the average Detection Rate is 84.93%. Some attacks cannot be timely detected mainly because that, in the first detection cycle, the injected attack flows do not exceed the threshold S_T. But in the second detection cycle, the system resources are too limited for Attack Detector to monitor all the attack flows. Thus, the attack is determined as a normal state. And on the other hand, in all the TP samples, Attack Detector located the malicious host accurately, which indicates that our method is effective on the malicious host tracking.

VI. Conclusion

In this paper, we proposed FSDM, a Fast recovery Saturation attack Detection and Mitigation framework. First, FSDM deploys a novel method that extracts the distribution characteristic of Control Channel Occupation Rate to timely detect the attack while locating the port that attackers come from. Second, FSDM introduces distributed Mitigation Agents and different mitigation policies depending on the locations of attackers to reduce the influenced switches, while increasing the robustness and promising benign communication. Third, we made a systematical analysis on the system recovery model, and proofed that the remaining attack flows are the culprit of long system recovery delay. To reduce the system recovery delay, we proposed a novel functional module Force_Cheking, which can quickly clean up the buffered attack flows and release controller resources, thus enabling the system to recover in a short time. The experimental results show that FSDM shortens *ping* RTT by 81% compared with traditional methods without cleaning buffered packets under attacks, and reduces 87% of the average HTTP requests time under large number of attacks.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China under Grant No. 61972371 and No. U19B2023, Youth Innovation Promotion Association of the Chinese Academy of Sciences (CAS) under Grant No. 2016394, and JSPS KAKENHI under Grant No. JP19H04105.

References

[1] D. Kreutz, F. Ramos, P. Verssimo *et al.*, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "Openflow: Enabling innovation in campus networks," in *ACM SIGCOMM Computer Communication Review*, vol. 38, 2008, p. 6974.

[3] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 413–424.

[4] S. Fichera, L. Galluccio, S. Grancagnolo *et al.*, "OPERETTA: An openflow-based remedy to mitigate TCP SYNFLOOD attacks against web servers," *Computer Networks*, vol. 92, no. 9, pp. 89–100, 2015.

[5] R. Mohammadi, R. Javidan, and M. Conti, "SLICOTS: An SDN-based lightweight countermeasure for tcp syn flooding attacks," *IEEE Transactions on Network and Service Management*, vol. 14, no. 2, pp. 487–497, 2017.

[6] Z. Li, W. Xing, S. Khamaiseh, and D. Xu, "Detecting saturation attacks based on self-similarity of openflow traffic," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 607–621, 2020.

[7] S. Achleitner, T. L. Porta, T. Jaeger, and P. McDaniel, "Adversarial network forensics in software defined networking," in *Proceedings of 2017 Symposium on SDN Research (SOSR)*. ACM, 2017, p. 820.

[8] S. Liu, M. K. Reiter, and V. Sekar, "Flow reconnaissance via timing attacks on SDN switches," in *Proceedings of the 2017 International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 196–206.

[9] J. Cao, Q. Li, R. Xie *et al.*, "The crosspath attack: Disrupting the SDN control channel via shared links," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2019, pp. 19–36.

[10] A. Patwardhan, D. Jayarama, N. Limaye *et al.*, "SDN Security: Information disclosure and flow table overflow attacks," in *Proceedings of the 2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–6.

[11] M. Zhang, G. Li, L. Xu *et al.*, "Control plane reflection attacks in SDNs: New attacks and countermeasures," in *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2018, pp. 161–183.

[12] J. Cao, M. Xu, Q. Li *et al.*, "Disrupting sdn via the data plane: A low-rate flow table overflow attack," in *Proceedings of the 2018 Security and Privacy in Communication Networks (Securecomm)*. Springer, 2018, pp. 356–376.

[13] H. Wang, L. Xu, and G. Gu, "FloodGuard: A dos attack prevention extension in software-defined networks," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015, pp. 239–250.

[14] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "FloodDefender: Protecting data and control plane resources under SDN-aimed DoS attacks," in *Proceedings of the 36th IEEE International Conference on Computer Communications (INFOCOM)*, 2017, pp. 1–9.

[15] D. Hu, P. Hong, and Y. Chen, "FADM: DDoS flooding attack detection and mitigation system in software-defined networking," in *Proceedings of the 2017 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2017, pp. 1–7.

[16] K. Giotis, C. Argyropoulos, G. Androulidakis *et al.*, "Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments," *Computer Networks*, vol. 62, p. 122136, 2014.

[17] A. Silva, J. Wickboldt, L. Granville, and A. Schaeffer-Filho, "ATLANTIC: A framework for anomaly traffic detection, classification, and mitigation in SDN," in *Proceedings of the 2016 IEEE/IFIP Conference on Network Operations and Management Symposium (NOMS)*, 2016, pp. 27–35.

[18] C. Buragohain and N. Medhi, "FlowTrApp: An SDN based architecture for DDoS attack detection and mitigation in data centers," in *Proceedings of the 2016 3rd International Conference on Signal Processing and Integrated Networks (SPIN)*. IEEE, 2016, pp. 519–524.

[19] D. Kotani and Y. Okabe, "A packet-in message filtering mechanism for protection of control plane in openflow networks," in *Proceedings of the 2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014, pp. 29–40.

[20] "Mininet," [Online], 2020, available: http://mininet.org/.

[21] "Ryu controller," [Online], 2020, available:https://osrg.github.io/ryu/.

[22] "Openflow v1.5 specification," [Online], 2020, available:https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[23] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, 2013, p. 165166.

[24] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran, "Lineswitch: Tackling control plane saturation attacks in software-defined networking," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1–14, 2016.

[25] C. Cortes, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, p. 273297, 1995.

[26] J. Sonchack, A. Dubey, A. J. Aviv *et al.*, "Timing-based reconnaissance and defense in software-defined networks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. ACM, 2016, p. 89100.

[27] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Ipsj Magazine*, vol. 12, no. 7, pp. 422–426, 1970.

[28] "Openvswitch," [Online], 2020, available:https://www.openvswitch.org/.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 01,2021 at 00:10:34 UTC from IEEE Xplore. Restrictions apply.